January 1992

Report No. STAN-CS-92-1401

Also numbered CSL-TR-92-503

The Performance Impact of Data Reuse in Parallel Dense Cholesky Factorization

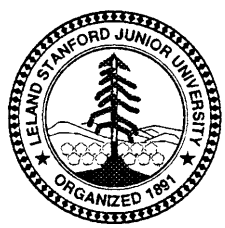
by

Edward Rothberg and Anoop Gupta

Department of Computer Science

Stanford University

Stanford, California 94305



REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1, 1992	

4. TITLE AND SUBTITLE
The Performance Impact of Data Reuse in Parallel Dense Cholesky Factorization

5. FUNDING NUMBERS
N00039-91-C-0138

6. AUTHOR(S)
Ed Rothberg and Anoop Gupta

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Science Department
Stanford University
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER
STAN-CS-92-1401
CSL-TR-92-503

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA
Arlington, V-4

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Abstract

This paper explores performance issues for several prominent approaches to parallel dense Cholesky factorization. The primary focus is on issues that arise when blocking techniques are integrated into parallel factorization approaches to improve data reuse in the memory hierarchy. We first consider panel-oriented approaches, where sets of contiguous columns are manipulated as single units. These methods represent natural extensions of the column-oriented methods that have been widely used previously. On machines with memory hierarchies, panel-oriented methods significantly increase the achieved performance over column-oriented methods. However, we find that panel-oriented methods do not expose enough concurrency for problems that one might reasonably expect to solve on moderately parallel machines, thus significantly limiting their performance. We then explore block-oriented approaches, where square submatrices are manipulated instead of sets of columns. These methods greatly increase the amount of available concurrency, thus alleviating the problems encountered with panel-oriented methods. However, a number of issues, including scheduling choices and block-placement issues, complicate their implementation. We discuss these issues and consider approaches that solve the resulting problems. The resulting block-oriented implementation yields high processor utilization levels over a wide range of problem sizes.

14. SUBJECT TERMS
hierarchical-memory machines, Cholesky factorization

15. NUMBER OF PAGES
28

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRAC
unclassified	unclassified	unclassified	

The Performance Impact of Data Reuse in Parallel Dense Cholesky Factorization

Edward Rothberg and Anoop Gupta
Department of Computer Science
Stanford University
Stanford, CA 94305

January 14, 1992

Abstract

This paper explores performance issues for several prominent approaches to parallel dense Cholesky factorization. The primary focus is on issues that arise when blocking techniques are integrated into parallel factorization approaches to improve data reuse in the memory hierarchy. We first consider panel-oriented approaches, where sets of contiguous columns are manipulated as single units. These methods represent natural extensions of the column-oriented methods that have been widely used previously. On machines with memory hierarchies, panel-oriented methods significantly increase the achieved performance over column-oriented methods. However, we find that panel-oriented methods do not expose enough concurrency for problems that one might reasonably expect to solve on moderately parallel machines, thus significantly limiting their performance. We then explore block-oriented approaches, where square submatrices are manipulated instead of sets of columns. These methods greatly increase the amount of available concurrency, thus alleviating the problems encountered with panel-oriented methods. However, a number of issues, including scheduling choices and block-placement issues, complicate their implementation. We discuss these issues and consider approaches that solve the resulting problems. The resulting block-oriented implementation yields high processor utilization levels over a wide range of problem sizes.

1 Introduction

This paper studies dense Cholesky factorization on multiprocessors with memory hierarchies. The dense Cholesky factorization problem arises in a number of problem domains, including linear programming, radios@ computations, and boundary element methods, and it also in many ways comprises the computational kernel of the important *sparse* Cholesky factorization problem. The primary difference between this work and the wealth of existing work on parallel dense Cholesky factorization (for example, [1, 7, 8, 11]) is that we consider the impact of memory hierarchies on parallel performance. We study parallel machines in which each processor has a small high-speed cache and a portion of the global main memory. Such machines offer the potential for extremely high performance and extremely cost-effective performance, and consequently they are becoming increasingly more common (e.g., Intel Touchstone, Stanford DASH multiprocessor [lo]).

These machines, however, raise new issues for the efficient implementation of parallel dense Cholesky factorization. In particular, in the presence of a memory hierarchy the computation must reuse significant amounts of data in the faster levels of the hierarchy (i.e., the caches) to achieve high performance. This data reuse can be accomplished by using blocking techniques [2, 4, 6], where sub-blocks of the matrix are retained in the cache across a number of uses. However, as the block size is increased to provide greater data reuse, the available concurrency decreases, thus limiting the number of processors that can be used effectively. Understanding the resulting tradeoff is the primary focus of this paper. We study this tradeoff by first proposing a performance model that takes the effect of data reuse into account and then simulating alternative parallel approaches using this model.

This paper is organized as follows. We begin in Section 2 by describing existing work on parallel dense Cholesky factorization. In particular, we first describe details of a method that manipulates columns of the

matrix [7], and then extend it so that it manipulates panels, or sets of contiguous columns, to increase data reuse. Section 3 presents our performance model, which captures the benefits of data reuse for dense Cholesky factorization in a precise way.

In Section 4, we use this performance model to study the parallel performance of panel-oriented methods (Such methods are becoming increasingly popular; for example, the parallel version of the LAPACK dense linear algebra library is currently being written using such an approach [5]). We simulate panel-oriented methods, considering a wide range of problem sizes, machine sizes, and panel sizes. We find that performance varies quite dramatically as the panel size is changed. However, even for relatively large problem sizes the maximum speedups obtained are substantially less than linear in the number of processors. We explain the less-than-perfect speedups by looking at two factors that bound speedups, the *critical path* and *load balance* of the computation. Section 5 then looks at a modified panel-oriented approach that attempts to improve these performance bounds by splitting panel operations into smaller pieces. We show that overall performance can only be slightly increased

Section 6 then considers factorization approaches that abandon the notion of panels entirely and instead distribute square sub-blocks of the matrix among the processors. Such approaches have been shown to possess a number of important advantages. For one, they reduce interprocessor communication volume significantly [1, 11, 15]. For this paper, however, our interest in these approaches is not on communication volume, but rather on a second advantage, the enormous increase in concurrency that such approaches generate. With more available concurrency, the tradeoff between large blocks and reduced concurrency becomes much less of a limitation. We find, however, that block-oriented approaches bring up a number of important considerations involving the mapping of blocks to processors and the scheduling of block tasks. We show that these considerations must be effectively addressed before a block-oriented scheme can realize its full potential. A brief discussion follows in Section 7 and conclusions are presented in Section 8.

The contributions of this paper do not come from new algorithms for dense parallel factorization; the algorithms we consider are quite well known. Instead the contributions come from the paper's formalization and quantification of widespread notions about the tradeoffs between data reuse and problem concurrency. We develop a simple model for understanding the performance of dense factorization approaches, By studying this model, it becomes clear that panel-oriented approaches have serious scalability problems. We also present a thorough study of block-oriented methods, including an examination of the more practical scheduling and block distributions issues that arise for these methods. Another contribution is in the area of performance prediction. We present detailed results of extensive simulation, predicting parallel performance on problems much larger than those that could reasonably be solved on current machines.

2 Background

2.1 Parallel Dense Cholesky Factorization

The dense Cholesky computation factors a symmetric positive definite matrix A into the form $A = L L^T$, where L is lower triangular. We begin our discussion by describing an existing approach [7] for performing dense Cholesky factorization on a multiprocessor. The method can be expressed in terms of rows or columns, but the results of Geist and Heath [7] indicate that column-oriented methods are to be preferred. We therefore concentrate on column-oriented methods, although we occasionally comment about how our results would apply to a row-oriented approach.

The column-oriented computation is accomplished through the use of two simple primitives. The first primitive, the $cmod(\ )$ or column modify operation, adds a multiple of one column into another column to zero an entry in the upper triangle of the factor. The second primitive, the $cdiv(\ )$ or column division operation, divides a column by the square-root of its diagonal. The method of Geist and Heath [7] assigns each column of the matrix A to some processor. The owner processor performs all modifications to the column. In a distributed-memory environment, the non-zeroes of a column are placed in the local memory of the owner processor. The columns are assigned in a *wrap-mapped* fashion, where column i is assigned to processor i mod P, in order to balance the computational load among the processors.

To illustrate, consider a simple example. As a first step of the parallel computation, the processor that owns column 0 of L performs a $cdiv(0)$ operation on that column, and broadcasts the result to all other processors.

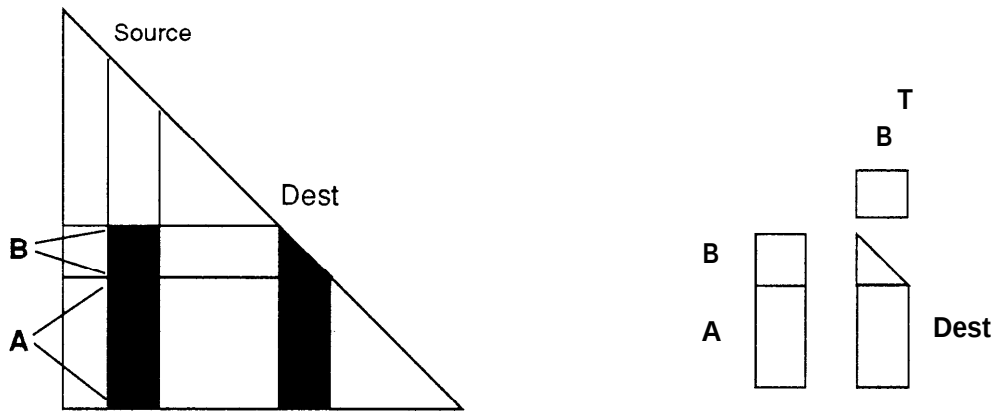


Figure 1: The $pmod(\ )$ operation.

A processor receiving a column performs all $cmod(j, k)$ operations from the received column k to all columns j that it owns. When a column j has received all modifications, the owner performs a $cdiv(j)$ on that column and broadcasts it. In rough pseudo-code, each processor executes the following:

```
1.  Set L = A
2.  while I own a column that is not complete
3.        if some j that I own has received all modification
4.           cdiv(j)
5.           broadcast j
6.        else receive some k
7.              for each j > k that I own
8.                   cmod(j,k)
```

Experiments with this column-oriented method [7] have shown it to be an effective approach for a variety of parallel machines. However, this method achieves little data reuse, and will therefore achieve extremely low performance on machines that rely on such data reuse to achieve high performance. We now consider the performance improvements that are possible when this approach is modified to manipulate larger portions of the matrix, thus increasing the potential for data reuse.

2.2 Panel-Oriented Parallel Dense Cholesky Factorization

The above parallel dense Cholesky factorization approach can be extended in a natural way to make better use of a memory hierarchy. Instead of distributing individual columns to processors, the approach can instead distribute sets of adjacent columns, or *panels* [5, 12, 13]. For example, a panel-oriented factorization method might divide an n x n matrix into $n/4$ panels, each containing 4 contiguous columns. Each operation in the column-oriented parallel factorization algorithm has a close analogue in a panel-oriented approach. The $cmod(\ )$ operation is replaced by a more complicated operation, which we refer to as $pmod(\ )$, for panel-modify. Similarly, the $cdiv(\ )$ operation is replaced by a $pdiv(\ )$ operation. Finally, the broadcast of a column is replaced by the broadcast of a panel. Note that the column-oriented approach is a special case of the panel-oriented approach.

The actual implementations of panel-oriented primitives are quite straightforward. We now provide a brief description, beginning with the implementation of the $pmod(\ )$ primitive. In Figure 1, we show a pictorial representation of the this operation. A destination panel is modified by some source panel by performing a matrix-matrix multiplication using submatrices from the source panel and subtracting the result from the destination. The first matrix in the matrix-multiply operation is the source panel, below the diagonal of the destination (the A and B submatrices in Figure 1). The second matrix is the transpose of the portion of the source panel in the rows corresponding to the triangular diagonal block of the destination (the *B* matrix in the figure). The result is subtracted from the destination. Since the result of multiplying the matrix *B* by its own

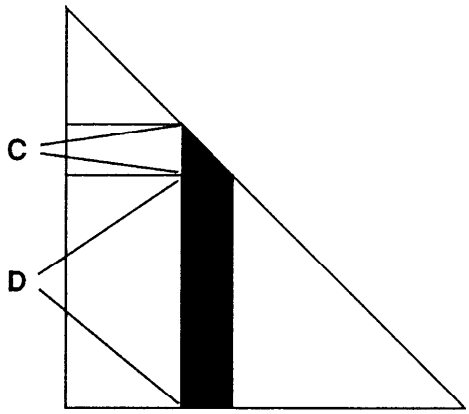


Figure 2: The $pdiv()$ operation.

transpose is symmetric, the primitive is actually implemented as two steps. In the first step, *B* is multiplied by its transpose, and only the lower triangle of the result is computed. In the second step, A is multiplied by the transpose of *B*.

Moving on to the implementation of the $pdiv()$ operation (see Figure 2), we note that this primitive involves two distinct steps. In the first step, the diagonal block C at the top of the panel is factored. In the second step, the portion *D* of the panel below the diagonal block is multiplied on the right by the inverse of the diagonal block (i.e., $D \leftarrow DC^{-1}$).

Recall that the point of moving to these higher-level primitives is to allow the computation to be *blocked* [2, 4, 6] to increase the amount of data reuse and consequently improve performance. This blocking is accomplished by reading some block of data into the high-speed processor cache and reusing it a number of times. A reasonable blocking approach is apparent from Figure 1. The B^T matrix can be retained in the cache and used repetitively to multiply rows of the source panel to produce updates to rows of the destination. We will discuss the specific benefits of this blocking in the next section.

3 A Performance Model for Hierarchical-Memory Multiprocessors

The results presented in this paper are based on a parallel performance model coupled with a multiprocessor simulator, In this section, we describe our performance model. The model is meant to include all essential factors that affect parallel performance on hierarchical-memory multiprocessors, while at the same time abstracting away many of the inessential details.

3.1 Parallel Machine Organization

In this study, we focus on multiprocessors with hierarchical memory systems, as shown in Figure 3. The two most important features of these machines are per-processor caches and distributed main memory. In these machines, cache accesses are much faster than local memory accesses, and local memory accesses are much faster than remote memory accesses. As a result, if an algorithm can make effective use of this memory hierarchy, it can significantly reduce the average memory access time seen by the processor as well as reducing the bandwidth requirements both between a processor and its local memory and also between processors on the global interconnect. These factors make this class of machines very attractive both from scalability and cost-performance points of view (especially in contrast to vector supercomputers that rely on brute-force bandwidth). The downside, of course, is the algorithmic complexity and programming effort needed to exploit the memory hierarchy, which is what we reflect on in this paper. Finally, note that a distributed main memory does not imply a message-passing memory model; shared-memory machines can also be built with distributed memory (the Stanford DASH machine [10] is one example).

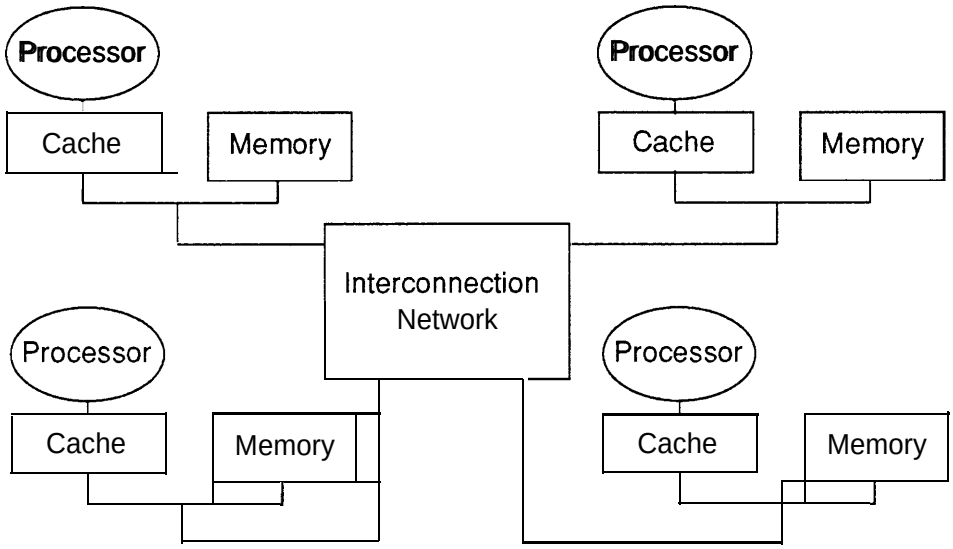


Figure 3 : Parallel machine.

3.2 Basic Model Assumptions

We now describe the assumptions that we make in order to develop a performance model. The goal is that the model be simple enough to be easily analyzable while at the same time accurate enough to provide meaningful results. This subsection presents these assumptions in a qualitative manner; the quantitative implications of these assumptions are described in the following subsection.

In our model, all costs related to the factorization program are attributed to floating-point operations (including the costs of fetching data from memory and the costs of integer operations). Thus, we model the runtime of a program in terms of the number of floating-point operations performed and the cost of each operation. We will be more precise about these costs in the next subsection.

Regarding interprocessor communication, we assume that such communication can be thought of as a series of messages between the processors. Note that we use the same model for both message-passing and shared-memory programming models, even though in the shared-memory model such messages would be exchanged using ordinary load and store instructions, We also assume that the only cost associated with sending a message is a latency cost. Thus, neither the sending nor receiving processors expend any processing cycles in transferring a message; the message is simply unavailable at the destination until some amount of time after it is sent. We believe that this assumption is reasonable, since many modem multiprocessors contain dedicated hardware to handle message transmission. For example, the Stanford DASH machine [10] contains a prefetch instruction to fetch data from a distant memory node while the fetching processor continues to operate on available data. Similarly, many message-passing machines possess hardware to send, forward, and receive messages with little or no help from the associated processors. In both cases, the processors expend some number of cycles to handle a message even with this dedicated hardware, but in our context these costs are small in comparison to the amount of work that is done in response to a message.

Regarding communication latency, we assume that it has three components: a fixed latency associated with sending the message, a transfer latency that depends on the length of the message, and a small random component. The fixed latency accounts for message setup and buffer allocation costs, as well as latencies in the hierarchical memory system. The transfer latency accounts for the fixed bandwidth available on the interconnect between the sending and receiving processors. The small random component accounts for a number of unpredictable delays within the processor and the interconnection network that are inevitable in any asynchronous multiprocessor.

Our final assumption regarding interprocessor communication is that arriving messages are placed in the *local memory* of the receiving processor, as opposed to being placed in the receiving processor's cache. To read the contents of the message, the receiving processor must therefore pay the costs associated with accessing its local memory. While one could conceive of a communication scheme where messages would be placed closer

to the processor in the memory hierarchy, such a design would complicate machine design and it could easily lead to the displacement of actively used data.

3.3 Domain-Specific Model

33.1 Uniprocessor Performance Model

One of the most important costs associated with performing parallel dense Cholesky factorization is certainly the cost that each processor incurs in performing floating-point operations. Recall that this cost is not a constant for a particular problem: different panel sizes lead to different processor performance levels. To model the effect of the panel size on overall runtime, we define a quantity, T_{op} (*B*), to be the average cost of performing a single floating-point operation within a panel operation that manipulates panels of size *B.*

In order to assign an actual value to T_{op} *(B*), we now describe a simple model for the performance of a hierarchical-memory processor on blocked matrix-matrix multiplication, the most prevalent operation in the panel-oriented factorization. While the relatively infrequent $pdiv(\,)$ operation is not a matrix-matrix multiplication and thus is not directly addressed by this analysis, we note that the results hold for this operation as well.

Our uniprocessor performance model follows the model of Gallivan et. al. [6] closely. As was mentioned earlier, we assume that in executing a program, the processor incurs a certain cost for performing machine instructions, and a certain cost for fetching data from memory. The total runtime is the sum of these two components. To obtain an estimate for the magnitude of each of these two components, we consider the execution of an entire $pmod(\,)$ operation. This operation requires the multiplication of an r x s matrix by an s x t matrix. The number of floating-point operations is $2rst$, and the number of floating-point reads is also $2rst$.

We assume in this paper that the goal of parallel dense factorization is to solve very large problems. Based on this assumption, we further assume that no panel of the dense matrix would fit in the processor cache. In other words, we assume that even single column of the dense matrix is too large to fit in a cache. The miss rate on the memory references for an unblocked matrix-matrix multiplication would therefore be expected to be 100%. In other words, one cache miss would be generated for every floating-point operation.

By blocking the matrix-matrix multiply, the number of misses can be reduced by a factor of *B,* where *B* is the block size, thus reducing the miss rate to one miss every *B* floating-point operations [6]. The benefits of blocking the computation do not increase without bound, however. They are limited by two factors. First, the block must fit in the processor cache. Second, the block size can be no larger than the minimum of r, s, and t, the dimensions of the matrices. Rather than considering the size of the processor caches in our performance analysis, we instead assume that memory reference costs become a trivial fraction of total runtime once some relatively large block size is reached, and that further increases in the block size beyond this point have little effect on performance. We also assume that this practical maximum block size is small enough to fit in any reasonable cache. A 32 by 32 block appears to satisfy both of these criteria reasonably well.

Returning to our cost model, we define *0* to be the cost of performing a single floating-point operation without memory system costs and we define M to be the cost of fetching a single floating-point datum from main memory. The units for each of these quantities are arbitrary, since it will actually be the ratio of these quantities that is important. Recall that a block size of *B* results in one cache miss every B floating-point operations. In terms of the quantities just defined, we therefore have

$$T_{op}(B) = 0 + M/B.$$

This paper will measure parallel performance in terms of the improvement that is obtained over a single processor solving the same problem. Parallel performance will be expressed either in terms of parallel speedups ($\frac{T_{seq}}{T_{parallel}}$) or parallel processor efficiencies ($\frac{T_{seq}}{pT_{parallel}}$). We assume that a single processor can always work with a block size of 32, thus achieving its highest possible performance. This assumption leads to a frequent need to compare the highest possible performance of a single processor with the performance obtained with a particular panel size *B.* In particular, the T_{op} *(B*) term will always appear in a ratio

$$T_{op}(B)/T_{op}(32)$$

in our analysis. To simplify our presentation, we define T_{op} (32) to be equal to one time unit, and express all costs in the parallel performance model in terms of the cost of performing one floating-point operation in a sequential program.

If T_{op} (B) is expressed in this way, then the terms 0 and M that represented the costs of machine instructions and cache misses in the original expression for T_{op} (B) are never actually needed. In terms of our original expressions, we find that

$$T_{,,,,}(B) = T_{op}(B)/T_{op}(32) = \frac{0 + M/B}{O + M/32} = \frac{1 + \frac{M/O}{B}}{1 + \frac{M/O}{32}}.$$

The value of T_{op} (B) depends solely on the block size B and on the ratio of the cost of a cache miss to the cost of the instructions executed to perform a floating-point operation.

We can determine a reasonable estimate for this ratio by looking at the ratio of the performance of a fully blocked code to the performance of an unblocked code on current sequential hierarchical-memory machines, We have looked at a number of current machines, including machines based on the MIPS R3000 and the IBM RS/6000, and have found a surprising degree of consistency. On a variety of machines, a matrix multiply code that uses a block size of 32 to reduce cache misses is roughly 5 times as fast as a code that uses a block size of 1 and generates a 100% cache miss rate. In other words, T_{op} (1) = 5. The resulting ratio of the cost of a miss to the cost of executing the appropriate machine instructions is roughly 4.75. We will use the value T_{op} (1) = 5 throughout this paper.

The uniprocessor performance model that we have just described makes a number of simplifying assumptions, For one, it ignores the impact of floating-point registers and cache-to-register traffic. It also ignores the fact that some of the latency of cache misses can be hidden, since the processor can often perform other operations while misses are being serviced. We note that the model is quite reasonable for todays microprocessors, which have high cache miss costs and few provisions for hiding cache miss latency. Future microprocessors may be built with more latency hiding mechanisms, but they will most likely have higher miss latencies as well, so their ability to hide latency may be limited. In either case, a different model than the one we use here may be appropriate for some processors. We note that most of the analysis in this section is independent of the specifics of the operation cost model. The main change with a different uniprocessor performance model would be in the specific performance and panel size numbers that we derive, not in the conclusions that we draw.

3.3.2 Interprocessor Communication Model

The other parallel performance component that requires quantification is the latency cost of sending a message. We model this cost as

$$T_{comm}(L) = \alpha + \beta L,$$

where α is the fixed cost of sending a message, β is the additional cost of each word of data, and L is the length of the message.

To arrive at reasonable estimates for α and 3, we consider the values for these parameters that have been achieved in recent multiprocessors. Recall that these values will be expressed relative to T_{op} (32), so we must compare the communication parameters of these machines with their floating-point performance. To estimate the value of T_{op} (32) in absolute terms, we use uniprocessor LINPACK 1000 benchmark results [3], which typically come from a fully blocked version of the LINPACK code. The two example machines we consider are the Stanford DASH machine and the Intel Touchstone machine. The Stanford DASH machine is made up of a network of 33 MHz MIPS R3000 processors. The LINPACK 1000 number for a single processor of the DASH machine is 8.8 MFLOPS. The Touchstone machine is made up of a network of 33 MHz Intel i860 processors. The LINPACK number for a single processor of the Touchstone is 25 MFLOPS. To obtain an approximate value for 3, we also need an estimate of interprocessor communication bandwidths of these machines. The DASH machine provides a realistic communication bandwidth of 16 MBytes/s, while the Touchstone provides roughly 25 MBytes/s. We therefore find that the ratios of computation to communication bandwidth on the DASH and Touchstone machines are 4 and 8, respectively. We use a ratio of β = 4 in this paper. For the parameter α, we believe that α = 100 is a reasonable, although possibly optimistic estimate for the latency of a message. In other words, we assume that 100 floating-point operations can be performed in the fixed time required to

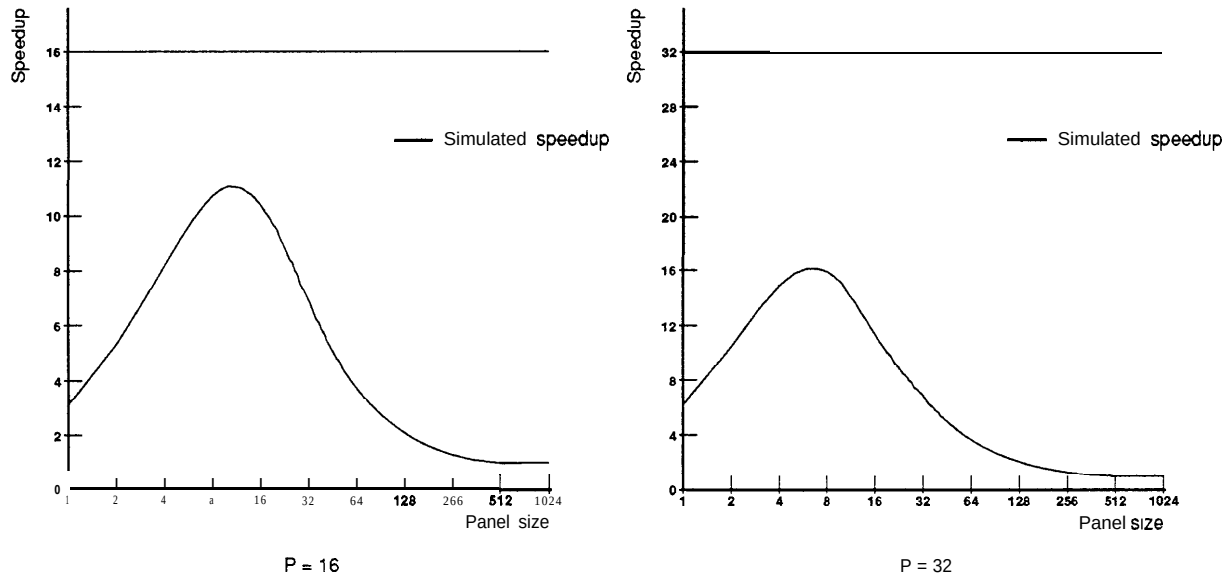


Figure 4: Simulated speedups for parallel dense Cholesky factorization, n = 1000. T_{op} (1) = 5. β = 4.

transmit a message. We also add a small random number into α (between 0 and 1) to reflect the small random component of message latency. While this small random component may appear insignificant, a later section will show that this component can have a substantial performance impact.

3.3.3 Simulation

The final component of our performance model is a means to estimate overall parallel performance given the above defined costs for individual factorization tasks. We use a simple event-driven simulator to predict performance. Simulated processors act on messages placed in their input queues. The time at which a message arrives at a processor is determined by the time that the message is sent and the latency incurred in transmitting the message, as described in our performance model. In acting on a message, a processor performs some set of panel operations. The time taken for each of these panel operations is again determined by our performance model.

4 Parallel Performance and Performance Bounds for Panel-Oriented Methods

We now use the performance model of the previous section to estimate parallel speedups for the previously described panel-oriented parallel dense Cholesky factorization algorithm. Our goal is to understand the general considerations that underly the tradeoff between the increased processor efficiencies that result when the panel size is increased and the reduction in concurrency that goes along with it.

4.1 Parallel Cholesky Factorization Simulation Results

We begin our discussion by presenting parallel speedups for the Cholesky factorization of 1000 by 1000 dense matrices on parallel machines with 16 and 32 processors. Using the previously derived machine parameters, we obtain the results shown in Figure 4. A 1000 by 1000 problem would most likely be considered large enough to yield high processor utilizations on 16 or 32 processors. The graphs in Figure 4 indicate that this is not the case. On 16 processors, the maximum speedup over all panel sizes is roughly 11-fold, yielding roughly 70% processor efficiencies, On 32 processors, the maximum speedup is roughly 16-fold, yielding 50% processor efficiencies. In both cases, the processor efficiencies are surprisingly low.

In order to better explain the results in this figure, we now look more closely at the factors that limit speedups. We can intuitively identify two competing concerns. On the one hand, we wish to use large panels in order to achieve high computation rates on each processor. Recall that we have assumed that performance when using a panel size of 1 is one-fifth of that achieved when using a panel size of 32. On the other hand, we wish to have as many panels as possible, so as to maximize the amount of available concurrency in the problem. Another reason to keep the number of panels high is to better balance the computational load among the processors. We now attempt to explain the observed speedups in terms of these two limiting factors.

4.2 Critical Path

We begin by considering the amount of concurrency available in the problem, given a particular choice of panel size. We measure the concurrency by looking at the critical path of the computation. Clearly the parallel computation can not be completed in less time than the time required to complete the critical path.

The critical path for parallel panel-oriented dense Cholesky factorization is computed by determining the earliest time at which each panel can be completed, assuming all dependencies between panels are observed. The most important dependency in this case is that a panel be modified by all previous panels before a $pdiv()$ can be performed on the panel and the panel used to modify other panels. For the fixed panel size case that we are studying, a simple analysis shows that this condition is equivalent to the condition that a panel cannot be completed until it has been modified by the previous panel. Thus, the critical path includes a $pdiv(0)$ operation on the first panel, a $psend(0)$ to send the panel to the processor that owns the next panel, a $pmod(1.0)$ to modify panel 1 by panel 0, a $pdiv(1)$ on panel 1, a $psend(1)$ to send panel 1 to the processor that owns panel 2, and so on. Simple calculations reveal that these operations have the following costs:

- $pdiv(k)$: $((N-k-2/3)B^3)T_{op}(B)$
- $pmod(k+1.\ k)$: $(2(N-k+1/2)B^3)T_{op}(B)$
- $psend(k)$: $T_{comm}((N-k)B^2) = \alpha + (N-k)B^2\beta$

In the above expressions, the problem size is $n = NB$, where B is the panel size and N is the number of panels.

To determine the critical path of the entire computation, we sum the costs of each task on the critical path over all k. Dropping low-order terms, the result is:

$$(\frac{3}{2}N^2B^3)T_{op}(B) + N\alpha + N^2/2B^2\beta.$$

We have assumed that the computation can be blocked on a single processor, so the sequential computation requires $\frac{N^3B^3}{3}T_{op}$ (32) time. Dividing the sequential time by the best-case parallel time gives an upper bound on the possible speedup from the problem.

$$\frac{N^3B^3/3T_{op}(32)}{\frac{3}{2}(N^2B^3)T_{op}(B) + N\alpha + N^2/2B^2\beta} = \frac{NBT_{op}(32)}{(\frac{9}{2}B)T_{op}(B) + \frac{3}{N^2B^2}\alpha + \frac{3}{2}\beta}.$$

At this point, we observe that the term involving α, the fixed latency of sending a message, is a trivial component of the overall denominator. We drop this term, giving (after simplification):

$$\text{speedup} \leq \frac{n}{(\frac{9}{2}B)T_{op}(B) + \frac{3}{2}\beta}.$$

This expression shows the maximum parallel speedup that can be obtained for the panel-oriented parallel dense Cholesky computation with a panel size of B.

An obvious question at this point is what panel size yields the maximum potential speedup overall. A small panel size places many small tasks that execute relatively inefficiently on the critical path. On the other hand, a larger panel size places more floating-point operations on the critical path, but these operations are performed

more quickly. A simple calculation reveals that the choice of B that maximizes potential speedup overall is $B = 1$. The maximum obtainable speedup in a dense Cholesky factorization problem is therefore

$$\text{speedup} \leq \frac{n}{\frac{9}{2}T_{op}(1) + \frac{3}{2}\beta}.$$

As a simple example, consider a 1000 by 1000 problem. Using the same machine parameters from before, we find that the maximum obtainable speedup on such a machine is $1000/28.5 = 35$, no matter how many processors are used. Furthermore, this speedup would require more than 35 processors, since this bound is obtained for the rather inefficient choice of a one column panel size. More detailed results from our critical path analysis will be presented later in this section.

One thing that is clear about the critical path bound is that it is quite limiting. One possible reason for the severity of this bound is that it assumes that the panels are of a fixed size. It may be more efficient to use small panels at the beginning and end of the matrix, where few processors are active and concurrency is limited, and use larger panels in the middle to increase the efficiency of the bulk of the computation. Unfortunately, such added flexibility does not significantly increase the amount of available concurrency. We implemented a simple dynamic program to determine a bound on the optimal critical path, given the freedom to choose the size of each individual panel, and found the resulting bounds to be less than 2% better.

Our analysis has so far only considered a column-oriented approach to the computation. We briefly note without presenting the derivation that the critical path bound on speedup would be identical for a row-oriented approach.

4.3 Load Balance

A second important factor that bounds parallel performance is the balance of computational load. The load balance bound simply states that the parallel computation cannot be completed in less time than the maximum time required by any processor to complete the work assigned to it. Load balance has a somewhat non-traditional meaning in this problem. Typically, one would think of the total load to be distributed among the processors as fixed; the load balance is then a measure of how evenly this work is distributed. In the context of hierarchical-memory machines, the total amount of work to be done varies with the panel size. The load balance therefore presents two different constraints on speedup. For small panel sizes, speedups are limited because each processor is performing its tasks at low efficiency. For large panel sizes, speedups are limited because the number of panels is reduced, thus making it more difficult to distribute the work evenly among a number of processors.

Given an assignment of panels to processors, the load balance is easily computed. Each processor has a set of tasks assigned to it, and the runtimes of these tasks are easily computed using our performance model. While it is possible to derive an analytic estimate of the load balance bound, a computed bound is adequate for our purposes.

We briefly note that it is shown in [7] that a row-oriented approach to parallel dense Cholesky factorization has worse load balancing properties than the column-oriented approach.

4.4 Simulated Speedups Versus Speedup Bounds

We now consider how simulated speedups are affected by the critical path and load balance bounds that have been described. We return to the example of the previous section. In Figure 5 we show the bounds that result, both from the critical path and from the load balance, using the parameters from the previous example. It is clear from these figures that the simulated speedups are almost entirely determined by the upper bounds. The speedups are below these bounds only near the point where the two upper bounds are equal. This fact can be easily understood as follows. The critical path bound (the dotted curve) gives the performance that would be obtained if every task on the critical path were executed as soon as the tasks it depends on were completed. The load balance bound gives the performance that would be obtained if the processor with the most work assigned to it started executing tasks as soon as the computation began, and never had to wait for dependencies to be satisfied to execute another task. If this processor is always executing some task, then it is clearly unlikely that a task along the critical path would be executed as soon as it is ready. While we have only demonstrated that

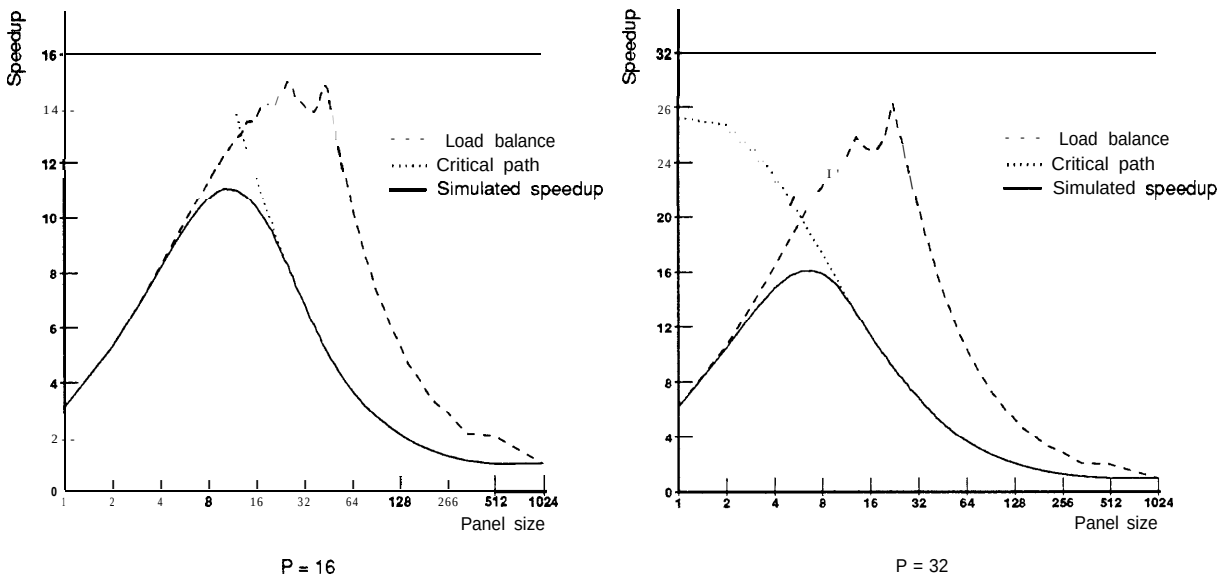


Figure 5: Performance and performance bounds for panel-oriented parallel dense Cholesky factorization, n = 1000. $T_{op}(1) = 5.3 = 4$.

simulated speedups are determined by the upper bounds for two examples, we note that we have found this to be the case for a wide range of other examples as well.

4.5 Implications of Speedup Bounds

While it would be interesting at this point to derive an expression for the panel size that yields the largest speedups for a given problem size, number of processors, and machine parameter set, our goal in this section is instead to examine the general implications of the performance bounds derived in the previous section. We now study these bounds in more detail, and derive a number of conclusions.

We begin by noting that simulated speedups for the panel-oriented methods are almost always nearly equal the upper bounds. The performance limitations that we have observed come from the upper bounds, not from the parallel algorithm.

Another item to note is that the upper bound and thus the maximum parallel speedup in a panel-oriented dense factorization code is proportional to the number of columns in the matrix. While this fact in and of itself is quite obvious, a less obvious fact is that the constant of proportionality for machines with hierarchical memories is quite small. For example, a parallel machine similar to the Stanford DASH machine can factor an n x 71 dense matrix at most n/28.5 times faster than a single processor of the machine. We expect comparable results for other parallel hierarchical-memory machines.

Another conclusions that can be drawn from the results presented so far is that maximizing concurrency is not the only goal in achieving high performance. Recall that the panel size that maximizes available concurrency, a panel size of 1, yields such low per-task performance that the resulting parallel speedups would necessarily be extremely low. It appears to be the true, both from previous figures and from intuition, that performance is maximized when concurrency is only as large as it needs to be. In other words, the objective appears to be to increase the panel size until the point at which the critical path would limit the achievable performance. Unfortunately, the critical path is quite constraining, as was recently mentioned. With a panel size of one, concurrency is limited, and concurrency decreases rapidly as the panel size is increased. It is therefore the case that unless the matrix is extremely large, the panel size that optimizes performance is quite small.

In order to better illustrate these results, we show in Figure 6 the problem sizes required in order to achieve given levels of processor utilization. The machine parameters are the same as those used so far. In general, the problem sizes necessary to yield high processor utilization levels are quite large. For example, to achieve 90% utilization on a 256 processor machine requires a roughly 86000 x 86000 problem. This problem requires

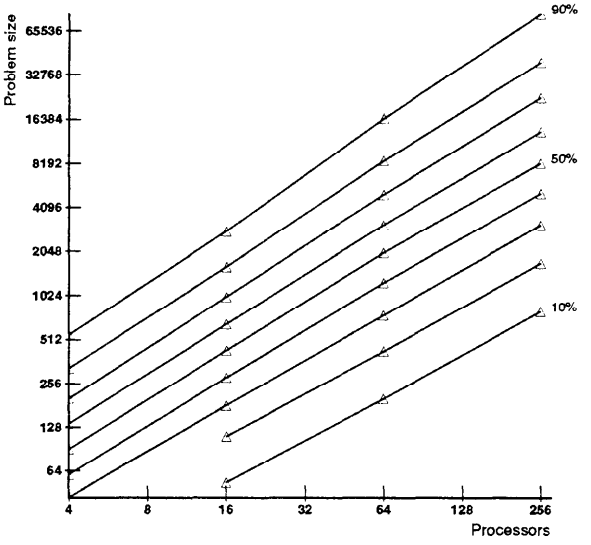


Figure 6: Problem sizes necessary to achieve different processor utilization levels. T_{op} (1) = 5. β = 4.

roughly 30 GBytes of storage space, and roughly 200 trillion floating-point operations. To put this number in perspective, note that each processor would require at least 117 MBytes of memory, and if each processor achieved 25 MFLOPS on blocked code then the problem would require more than 9 hours to complete. A 256 processor machine would therefore be expected to achieve much less than 90% utilization for the vast majority of dense factorization problems.

An interesting trend can be seen in these processor utilization curves. The slopes of the individual utilization curves are quite constant, and while it is not clear from the figure, the slopes are roughly equal to one. Thus, in order to maintain a constant level of processor efficiency, the problem size must grow at roughly the same rate that the number of processors grows.

We have also noticed another interesting trend, related to the panel sizes that achieve the highest performance for a given problem size and number of processors. The panel size that yields the highest processor utilization depends only on the utilization level that is achieved. In other words, if a problem is only large enough to achieve at most, for example, 40% utilization on some number of processors, then the panel size that achieves that maximum utilization level does not depend on the number of processors. The panel sizes that achieve these maximum utilization levels are plotted in Figure 7. This figure allows one to obtain a better feel for the tradeoff between individual processor efficiency and concurrency. We see that when the problem is too small to allow for both large panels and a large amount of concurrency, then the better choice in general is to favor concurrency. For example, whenever the problem size is sufficiently small that less than 70% processor efficiencies are possible, the best choice is a panel size of 10 or less. The reason is simply that performance gains from increasing the panel size experience a diminishing return. The amount of concurrency available in the problem, on the other hand, decreases quite quickly as the panel size is increased.

Our next conclusions relate to the critical path expression of the previous section, which we now repeat.

$$\text{speedup} \leq \frac{n}{(\frac{9}{2}B)T_{op}(B) + \frac{3}{2}\beta}.$$

Consider the impact of communication costs. The fixed latency of sending a message has little or no effect on performance. The corresponding term was dropped since it was insignificant in comparison to the other terms. Also note that the term involving β, the communication bandwidth, is independent of the panel size. Communication therefore reduces concurrency by some constant amount that depends only on the characteristics of the communication system.

Another thing to note about the above expression is the relationship between communication costs and optimal panel sizes. If the cost of communication is increased, then the amount of available concurrency decreases and the optimal panel size for a given problem decreases as well. This is a somewhat counter-intuitive result. Simple

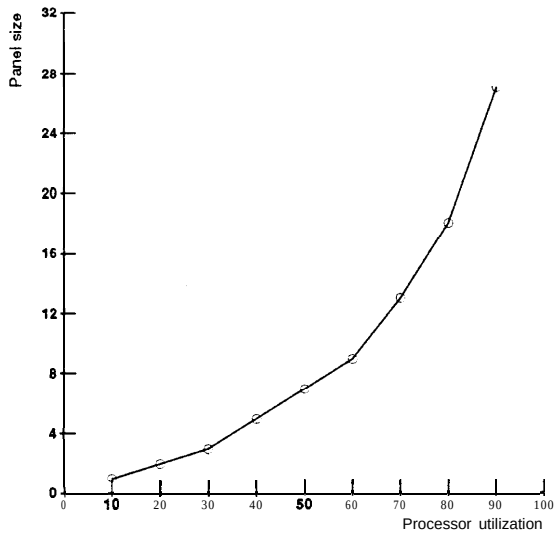


Figure 7: Optimal panel sizes at various processor utilization levels.

intuition would indicate that higher communication latencies would favor larger panels.

Further note that communication bandwidth seriously reduces concurrency only when it is significantly less than computational bandwidth. On a machine where cache misses are expensive, the computation term in the denominator of the critical path expression is much more important than the communication term.

Taking a high-level view of concurrency considerations, it is clear that even if available concurrency were a significantly larger fraction of n, a panel-oriented approach would still have serious scalability limitations. The number of operations in the problem grows as the cube of the number of columns, and the amount of space required to store the matrix grows as the square of the number of columns. If the concurrency grows as n, then the time and space demands of the problem obviously grow faster than the processor resources that can be employed to solve it. The interesting thing to note about the results of this section is that scalability limitations come into play with much smaller machines than one might have expected.

We therefore conclude from this section that a panel-oriented parallel dense Cholesky factorization code requires very large problems before it can achieve high processor efficiencies on moderately parallel hierarchical-memory multiprocessors. The required problem sizes are so large that they would almost certainly exceed either the available memory or the acceptable runtime. In other words, we would expect these machines to achieve low utilization levels for the problems that would be encountered in practice.

5 Modified Panel-Oriented Approach

In this section, we consider a simple modification to the panel-oriented factorization approach of the previous section to increase the amount of available concurrency. The modified method still works with panels, but it now treats a panel as a logical column of rectangular sub-blocks. The most important consequence of this logical division is that a processor no longer needs to wait for an entire panel to complete before it can send it to another processor. The processor can instead send an individual sub-block immediately after it has completed.

Recall that the performance of the panel-oriented method of the previous section was mostly determined by the load balance and critical path upper bounds. We now consider how these bounds are affected by the modification that we have described. If we first consider the load balance upper bound, we find that this bound is unaffected by the modification. The matrix is still distributed in a panel-wise fashion among the processors, and although the panel modification work now happens in smaller pieces, the total work performed by a processor remains unchanged.

Moving to the critical path upper bound, we note that this bound is quite dramatically changed. The most

important change relates to the time at which a panel modification can begin. In the unmodified method, a destination panel is modified by some source panel only once the entire source panel is complete and is sent to the owner of the destination panel. In the modified approach, the modification can begin as soon as the first sub-block of the source panel is completed and sent.

Providing a precise estimate of the critical path upper bound for the modified approach is somewhat difficult. If we look at the task dependence graph alone, we ignore the fact that a large number of these tasks must be performed on the same processor. For example, the modification of one panel by another is accomplished with a series of sub-block tasks. From the point of view of the critical path, these tasks can be performed in parallel, but in fact all of these tasks must be performed on the same processor. The critical path has become intertwined with the scheduling of the computation.

In order to obtain some idea of the amount of concurrency available with this approach, we take a slightly different approach. We compute an effective critical path by simulating the computation assuming that an infinite number of processors are available. In this context, one processor per panel of the matrix suffices. The length of the effective critical path has proved difficult to estimate analytically, so we simply describe our observations about the amount of available concurrency in the modified method in comparison to the concurrency for the unmodified method.

The first thing we note about the concurrency of the modified method is that it is much less dependent on communication latencies than is the concurrency of the unmodified method. In fact, for all but the smallest panel sizes and the highest communication costs, the concurrency in the modified method is essentially independent of the cost of communication. The reason is the pipelining of communication that can be achieved. One sub-block can be communicated from one processor to another while the modification from the previous sub-block is going on.

Even if the communication cost differences between the methods are ignored, the modified method still contains significantly more concurrency. We have observed a factor of nearly two difference. The source of this difference is again the pipelining that is made possible by the sub-block modification. In the unmodified approach, a panel modification cannot begin until the entire source panel is complete. In the modified approach, the modification can begin as soon as the first sub-block in the source is complete. The source processor can continue to complete sub-blocks and broadcast them while the destination processor performs modifications with earlier blocks.

Unfortunately, we have found that while the factor of roughly two increase in concurrency does increase processor utilization levels, the increase is relatively small. The main reason is that the scheduling of the modified method is somewhat less effective than that of the unmodified method. Whereas in the unmodified method the simulated speedups are always nearly equal the upper bound, the simulated speedups for the modified method are usually somewhat below the upper bounds. We therefore find that while the simulated speedups for the modified method are larger than those of the unmodified method, overall the difference is not large enough to overcome the deficiencies of panel-oriented methods.

6 Block-Oriented Parallel Dense Factorization

It is clear from the previous discussion that panel-oriented dense factorization has a number of important limitations, In this section, an alternate means of performing dense Cholesky factorization on a parallel machine is considered. This approach divides the matrix into square submatrices and distributes these submatrices among the processors. Such an approach, which we refer to as a block-oriented approach, is a natural alternative to an approach that manipulates rows or columns of the matrix, and it has been explored in a number of papers [1, 11, 12, 15, 16]. A block-oriented approach has two main appeals. First, a dense matrix clearly contains as many as $O(n^2)$ blocks, while it only contains $O(n)$ rows or columns. Thus, a block-oriented approach can potentially increase concurrency. Second, the communication volume in a block-oriented approach can be shown to grow as $O(\sqrt{P})$ in the number of processors when the block size is chosen appropriately, versus $O(P)$ communication growth for a panel-oriented approach. This section investigates the benefits and complications of a block-oriented approach.

Table 1: Primitives for block factorization.

Primitive	Description	Step	Uses	Modifies	FP Ops
$bfactor(k,k)$	Factor diagonal block	2	-	L_{kk}	$B(B+1)(2B+1)/6$
$bsolve(i,k)$	Solve an off-diagonal block	4	L_{kk}	L_{ik}	B^3
$bmoddiag(j,j,k)$	Modify a diagonal block	6	L_{jk}	L_{jj}	$B^2(B+1)$
$bmod(i,j,k)$	Modify an off-diagonal block	8	L_{ik}, L_{jk}	L_{ij}	$2B^3$
$addin(i,j)$	Add an update into a block	8	-	L_{ij}	B^2

6.1 Block-Oriented Method Background

We begin this section by describing the implementation of a block-oriented method. In such a method, the matrix is divided into a set of square blocks. The factorization computation in terms of blocks is quite similar to the computation in terms of individual elements. If each element of the matrix L_{ij} is thought of as a square block instead of a single element, then the following code would factor the resulting matrix of blocks:

```
1.  for k = 0 to N do
2.      L_kk = Factor(L_kk)
3.      for i = k + 1 to N do
4.          L_ik = L_ik L_kk^-1
5.      for j = k + 1 to N do
6.          L_jj = Ljj - L_jk L_jk^T
7.          for i = j + 1 to N do
8.              L_ij = L_ij - L_ik L_jk^T
```

For conciseness, we define a number of primitives to express the block-oriented computation. We describe these primitives in Table 1. The descriptions include a list of the blocks that must be accessed to perform the primitive, a count of the number of floating-point operations that are necessary to perform the primitive when using a block size of ***B,*** and the line number in the above pseudo-code where the corresponding primitive is performed.

The parallel implementation will also use two other primitives, $bsend()$ and $bsenddiag()$, to communicate blocks and block updates between processors. The communication cost model of the previous section is used to model the costs of these primitives. We note that the $T_{op}(B)$ expression of this model is somewhat less accurate for the block-oriented primitives than it was for the panel-oriented ones, but the inaccuracies are small and not easily corrected

If the complexity of implementing a block-oriented code is compared with that of a panel-oriented code, one major difference is apparent. In the panel-oriented method, all primitives have two or fewer operands. In the most complicated of panel-oriented primitives, a panel is modified by another panel. To perform such a panel modification, a message from the processor that owns one panel to the processor that owns the other is sufficient. In a block-oriented parallel code, the $bmod()$ primitive involves three different blocks, and thus it potentially involves three different processors. The organization of a parallel code that performs these operations is therefore significantly more complicated.

We begin our discussion of the implementation of a parallel block-oriented factorization code by considering the implementation of a $bmod()$ primitive, the most frequently executed and most complicated of the primitives. There are three processors on which this computation can reasonably be performed, these being the processors that own the three blocks involved. Let us briefly consider the block communication that would be necessary for each of these three possibilities, assuming that the owner processor is responsible for all $bmod()$ operations involving the block. In Figure 8, we show the blocks that are needed to perform all of the $bmod()$ operations related to a particular block. The arrows indicate the block communication necessary for a single $bmod()$ operation. In the case where the L_{ik} block computes the updates, a $bmod()$ operation would require a block from above L_{ik} in the same column, and it would produce an update to a block to the right in the same row. If the L_{jk} block were to compute the update, it would need to receive a block from below it in the same block column, and it would produce an update to a block in a later column. If the L_{ij} block were to compute the

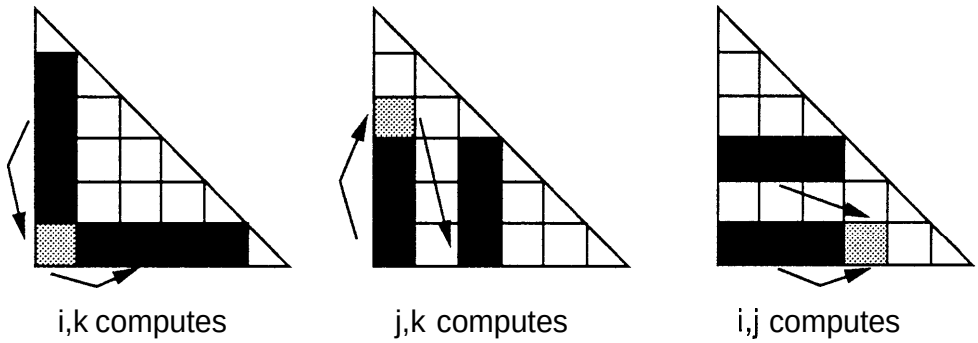


Figure 8: Blocks used for $bmod(\ )$ operations.

update, then it would need to receive two blocks, one from row i and one from row j, and it would produce an update for itself.

Of these three approaches, the L_{ik} approach appears to be the cleanest. It has the advantage that a block receives other blocks from above it in the same column, and produces updates to blocks to the right in the same row. The L_{jk} approach is similar, the primary difference being that the updates are sent in a less regular pattern. The L_{ij} approach has the slight disadvantage that two incoming blocks must be matched before a $bmod(\ )$ primitive can be performed. We now describe an implementation of the $\boldsymbol{L}_{ik}$ approach. Note that the differences that we have pointed out between the three approaches are not extremely important, and that the other approaches are not obviously inferior. We have simply chosen one to investigate.

Before beginning a description of the functioning of a parallel block-oriented method, we make a brief note about terminology. When we say that something is "sent to a block" of the matrix, we mean that a message is sent to the processor that owns the block.

6.1.1 Parallel Block-Oriented Program Flow

We now describe the operation of a parallel block-oriented code. The method is best understood by describing the sequence of tasks executed during the computation. The overall parallel factorization begins with the factorization of L_{00}. Once this diagonal block has been factored, it is immediately broadcast to the processors that own blocks in the same block column. These processors perform $bsolve(\ i.0)$ operations on the appropriate blocks once the diagonal block arrives. Once a block has been solved, it is immediately broadcast to all blocks below it in the same column.

When a processor receives a block L_{jk}, it performs a $bmod(i.\mathrm{j}.k)$ operation using the received $\boldsymbol{L}_{jk}$ and any L_{ik} with $i > j$ that it owns. In order to perform a $bmod(\ )$ operation, the appropriate $\boldsymbol{L}_{ik}$ block must already have been solved. If it has not, then the received block is queued with the $\boldsymbol{L}_{ik}$ block to indicate that a block modification should be performed as soon as the block is solved When the $bmod(\ )$ operation is performed, an update to block L_{ij} is computed. This update must somehow be added into its destination. One option is to send the update directly to the processor that owns $\boldsymbol{L}_{ij}$. That processor would then add the update directly into the destination block. Another option is to send the update to the block immediately to the right of $\boldsymbol{L}_{jk}$. Recall that this block produces an update to $\boldsymbol{L}_{ij}$ as well, so the two updates can be combined before they are sent off to the next block to the right. The first of these two approaches has the advantage that it does not need to combine updates. The second has the advantage that messages are only sent to immediate neighbor blocks. If the mapping of blocks to processors matches the topology of the multiprocessor interconnect, then the second approach improves the locality of communication. We will consider methods that use both approaches.

All blocks keep track of how many updates must be added into them. Once the diagonal block has received all updates, then it is factored and broadcast to all blocks below it. Similarly, once an off-diagonal block has received all updates, and once it receives the appropriate diagonal block, then that block is solved and broadcast to all blocks below it. It is clear that this process will continue until the entire matrix is factored. While the above description omits a number of important details, we believe that it provides a thorough enough explanation

to allow us to proceed with an analysis of the method. Details will be added in the next few subsections.

6.2 Parallel Performance Bounds

The parallel performance of the panel-oriented methods studied in the last section was limited by two upper bounds, a critical path bound and a load balance bound. We now look at these bounds in the context of the block-oriented parallel factorization computation.

The critical path is computed by determining the earliest possible time that each block of the matrix can complete. This computation is simplified by noting that all blocks in a column, with the exception of the diagonal block, complete at the same time in the best case. This is easily seen for the first column. These blocks depend only on the diagonal block. It can be seen for the rest of the blocks by noting that the earliest time a block can complete depends on the time that a pair of blocks in the previous column completed (as well as the time the diagonal block of that column completes). Since all blocks in the previous column are assumed to complete at the same time, then all blocks in the current column must complete at the same time as well. The result that all blocks in the same column complete at the time follows by a simple induction.

The critical path for the entire computation, can therefore be determined by finding the critical path along any block row of the matrix. Since the matrix is factored when the bottom right block is complete, the critical path along the last row is most appropriate. As was mentioned before, a block cannot be solved until it has received an update from the previous column. Thus, the critical path from one column to the next involves a $bmod(\,)$ operation to compute the update from the previous column, a $bsend(\,)$ operation to communicate that update, and a $bsolve(\,)$ operation to solve the block in the current column. A second set of dependencies between one column and the next also limits concurrency. This second path, which involves the diagonal block, includes a $bmoddiag(\,)$ operation to update the diagonal block by a block of the previous column, a $bsenddiag(\,)$ operation to communicate the update to the diagonal block, a $bfactor(\,)$ operation to factor the diagonal, a $bsenddiag(\,)$ operation to send the diagonal block to the block in question, and then a $bsolve(\,)$ operation to solve the block. The critical path is clearly determined by the longer of these two dependency chains.

The critical path computation naturally requires cost estimates for the tasks along the critical path. These costs are determined by augmenting the floating-point operation counts for these tasks given earlier with estimates of the costs of these operations, given the block size.

- $bfactor(k)$: $(B^3/3)T_{op}(B)$
- $bsolve(i,k)$: $(B^3)T_{op}(B)$
- $bmod(i,k,k+1)$: $(2B^3)T_{op}(B)$
- $bmoddiag(i,k)$: $(B^2(B+1))T_{op}(B)$
- $bsend(i,k)$: $T_{comm}(B^2) = \alpha + B^2\beta$
- $bsenddiag(i,k)$: $T_{comm}(B^2/2) = \alpha + B^2/2\beta$

The above-described critical path task dependencies, combined with the costs of these tasks, give the time costs of moving from one block column to the next. Since the matrix has N block columns, then the cost of overall critical path is therefore roughly:

$$\sum_{k=0} bmod(N-1,k,k-1) + bsend(N-1,k) + bsolve(N-1,k)$$

by the first set of dependencies, and

$$\sum_{k=0}^{N} bmoddiag(k,k,k-1) + bsenddiag(k-1,k) + bfactor(k,k) + bsenddiag(k,k) + bsolve(N-1,k)$$

by the second. The true critical path is the larger of these two expressions. The first of the two is larger if communication costs are low, so we use it to determine a bound on parallel speedup.

If an n x n matrix is factored using a block size of *B,* where $n = NB$, then the speedup is bounded from above by the sequential time divided by the critical path time. Recall that the sequential time is $\frac{n^3}{3}T_{op}(32)$. The bound is therefore:

$$\frac{\frac{n^3}{3}T_{op}(32)}{3NB^3T_{op}(B) + N\alpha + NB^2\beta} = \frac{n^2}{9B^2T_{op}(B) + \frac{3}{B}\alpha + 3B\beta}.$$

We can again determine the choice of block size that maximizes potential speedup, and the answer is a very small *B.* However, in this case the maximum is less meaningful. In the panel-oriented code, a panel size of 1 led to an inefficient but reasonable approach, where columns were passed among processors. In the block-oriented approach, a very small block size leads to a method that manipulates small sets of matrix entries. Our assumptions about the costs that could be ignored would certainly not hold, and the resulting approach would certainly achieve extremely poor performance due to overheads. We therefore assume that the block size must be reasonably large (at least 8) to obtain reasonable performance.

Note that the critical path bound for the block-oriented approach is much larger than the same quantity for the panel-oriented approach. Here, available concurrency is proportional to the square of the problem size, while in the panel-oriented method it was proportional to the problem size. To get some idea of how constraining the above bound is, we consider the example we looked at for a panel-oriented approach. Recall that the example bounded the speedup of a panel-oriented approach for the factorization of a 1000 x 1000 matrix on a machine with T_{op} (1) = 5 and β = 4. The upper bound on speedup was 28.5, and many more than 28.5 processors would be required to achieve this speedup. For the block-oriented bound above, if a block size of 32 were used, thus achieving full floating-point performance, the critical path bound would limit speedup to 104.

Another important factor that bounds performance is the balance of computational load among the processors. Note that once the assignment of blocks to processors has been performed, it is a simple matter to compute the amount of work that must be done by each processor, Examples of the actual bound that results will be shown in the next subsection.

A final factor that bounds parallel performance is the cost of performing interprocessor communication. A simple calculation reveals that the total volume of interprocessor communication for a block-oriented method is $\frac{n^3}{3B} + O(n^2)$, where n is the matrix size and *B* is the block size. Thus, a larger block size leads to a reduction in total communication volume. The performance impact of communication depends on more than the total message volume, however. Message locality is also important. Recall that the block-oriented method broadcasts blocks down a column of the matrix and sends updates to the right. If the logically local blocks that participate in these message exchanges are not mapped to physically local processors, then the messages are significantly more likely to experience contention on the processor interconnect. The costs of message contention are note modeled in the results we now present. Instead, we will make qualitative statements about the communication behavior of the various approaches.

6.3 Scheduling the Computation

Having described a block-oriented parallel factorization approach and the factors that bound its performance, we now look at the performance obtained with such an approach. Before presenting simulation results, a few more implementation details must be discussed. The first relates to the order in which parallel tasks are performed. At any one time, a processor may have a number of tasks to choose from. The order in which a processor executes these tasks can have a significant impact on the performance of the parallel computation. This subsection looks at the effectiveness of two different approaches to the scheduling. The second implementation detail relates to the strategy that is used for mapping blocks to processors. We consider a number of alternative mapping strategies in the next subsection. Similar mapping and scheduling issues were investigated by O'Leary and Stewart [12]. The primary way in which our work differs from this earlier work is that O'Leary and Stewart address asymptotic parallel performance on highly-parallel machines, whereas this paper is more concerned with practical performance issues on moderately-parallel machines.

6.3.1 First-Come, First-Served

The first approach to task scheduling we consider is probably the most obvious and the most analogous to the simple approach used for the panel-oriented method. This approach acts on messages on a first-come, first-served

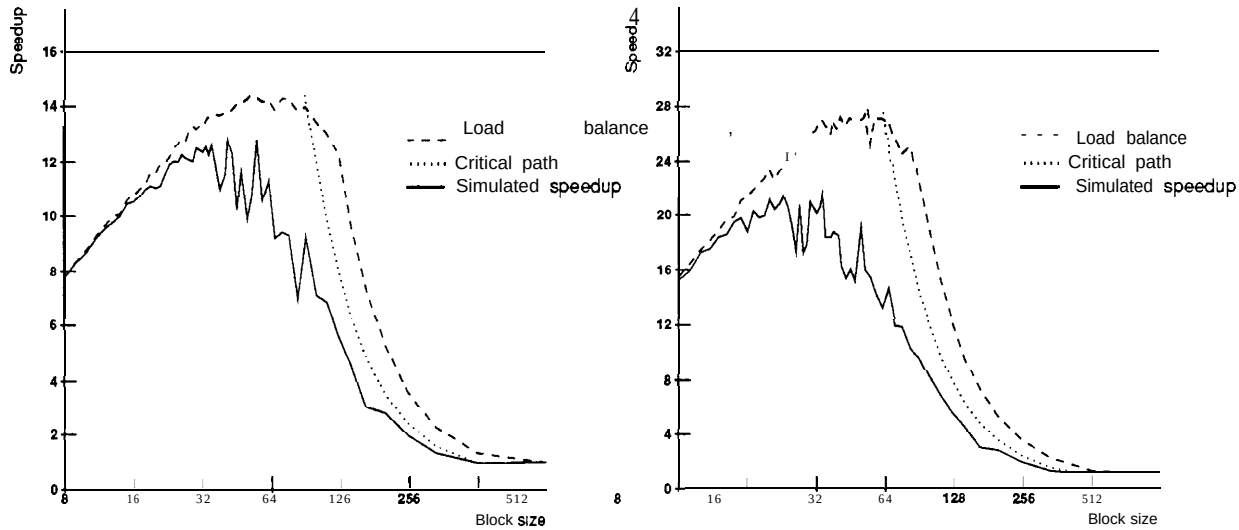


Figure 9: Performance bound for parallel dense Cholesky factorization, $n = 1000$. $T_{op}(1) = 5.3 = 4$.

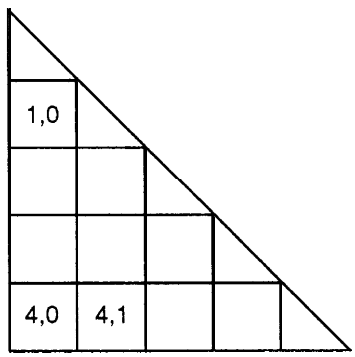


Figure 10: Two adjacent blocks.

basis. Simulation results for this approach are presented in Figure 9. The curves shown in this figure give the performance of this method on the example of the previous section (a 1000 x 1000 matrix factored on 16 and 32 processors), along with critical path and load balance upper bounds. A number of important things can be noted from these results. First, note that the simulated speedups are somewhat better than those of the panel-oriented method. Maximum speedups are roughly 13-fold on 16 processors and 21-fold on 32 processors. This is compared with 1 l-fold and 16-fold for the panel-oriented approach. Note also that the simulated speedups behave quite erratically. It would therefore be extremely difficult to choose a block size that yields consistently high performance. Another thing to note is that the simulated speedups are well below the critical path and load balance upper bounds. Recall that the speedups for the panel-oriented method were easily explained in terms of the two upper bounds. We now briefly consider the reasons for the behavior of this block-oriented approach.

The observed behavior can best be understood by considering a simple example. Consider the first few steps of the block-oriented parallel factorization of the matrix of Figure 10. The overall factorization begins with the factorization of the L_{00} block. Once factored, this block is then broadcast to the processors owning blocks in the first column. When the diagonal block arrives at the receiving processor, that processor performs a $bsolve(k,0)$ operation on the appropriate block. Each sub-diagonal block in the first column is completed at roughly the same time. Each of these blocks L_{k0} is then broadcast to the processors owning blocks $L_{i0}, i > k$ (i.e., the blocks below it in the same column).

Now consider the actions taken in response to these messages. In particular, consider the messages arriving at the processor that owns block L_{40}. This processor receives messages for each of the blocks above it in column 0. Since these blocks complete simultaneously, the corresponding messages will also arrive simultaneously. More accurately, since we have assumed that message latencies have some small random component, the messages will arrive in some random order. Since the messages are acted on in the order in which they are received, the random arrival order implies that the updates to blocks to the right of L_{40} are computed in a random order as well.

Now consider the consequences of this random order for the processor that owns block L_{41}. Recall that a processor cannot compute the updates from a block until that block is complete. Recall also that a block cannot be completed until it has received updates from all blocks to its left. The processor that owns block L_{41} must therefore sit idle until it receives an update from block L_{40}. Ideally this update would be the first one computed from L_{40}. However, as we noted in the previous paragraph, the updates are computed in random order, so this particular update is just as likely to be the last one computed from L_{40} as it is to be the first one. Thus, the processor that owns block L_{41} will incur a significant amount of idle time waiting for this update. Similarly, all of the processors that own blocks in column 1 would incur idle time waiting for the appropriate updates from blocks in column 0. Indeed, if one were to continue to step through the parallel execution, one would find that significant processor time is spent waiting for updates that are ready to be computed, but are waiting in queues behind updates that are much less important. The random nature of the update computation substantially disrupts the efficient scheduling of the parallel computation.

If the schedule of the parallel computation is so disrupted by this random component, an interesting question is why the performance obtained with this approach is not worse than what is observed in Figure 9. We have observed two factors that mitigate the problems associated with the inefficient task scheduling resulting from a first-come, tit-served task execution order. The first relates to the amount of concurrency in the problem. If the amount of concurrency is much larger than the number of processors, then the problem contains some *slack.* In the presence of slack, the scheduling of the computation is less crucial. Thus, the simulated speedups in Figure 9 are close to the upper bound for small block sizes, where the concurrency is greatest, and they fall away from the upper bound as the amount of concurrency approaches the number of processors.

The second factor that allows a first-come, first-served schedule to achieve reasonable performance is the inaccuracy of our conclusion that the blocks above a block L_{ij} in the same column arrive at L_{ij} in a random order. Recall that this assumption was based on the assumption that the blocks in a column complete simultaneously. While this assumption is true of the first column, it becomes less and less true as the computation proceeds. In fact, the completion times of blocks in a later column tend to attain a nearly sorted order, with the blocks near the top of column being completed before those near the bottom. This order of block arrival yields a much better task execution order than a random arrival order would.

The reason for this roughly sorted order of block completion times is clear if we consider the simple example of Figure 10. Recall that a block cannot complete until it has received updates from all blocks to its left. Consider the completion times of blocks L_{21} and L_{41}. Block L_{41} requires an update from block L_{40}. Since block L_{40} produces updates to all blocks to its right in a random order, the update to block L_{41} would be one of four updates, and would on average be produced second or third. In contrast, block L_{20} must produce updates to only two blocks. The update to block L_{21} would therefore be expected to arrive before the update to L_{41}, implying that L_{21} would be expected to complete before L_{41}. The later completion time of block L_{41} *also* delays the computation of updates to blocks in the same row, thus delaying their completion times as well. This delay, combined with the same effect of higher blocks producing fewer updates, leads to the near-sorted order that we have observed. Note, however, that the order is only nearly-sorted, and significant scheduling problems can still arise.

In summary, a block-oriented method with a first-come, first-served task schedule yields higher performance than a panel-oriented method. This method, however, appears to contain significant room for improvement. The obtained speedups vary quite erratically with the block size, and they are well below the load balance and critical path upper bounds. We next consider the possibility of improving the performance of the block-oriented method by choosing a different scheduling strategy.

6.3.2 Prioritized Block-Oriented Method

The scheduling problem that was encountered in the first-come, first-served approach to task ordering is not unique to Cholesky factorization. It is quite a common problem that arises whenever a processor in a parallel computation must choose from among a set of tasks with varying degrees of urgency. Some tasks are on the critical path and will prolong the execution of the entire program if they are not performed as soon as they are ready. Others are far off the critical path and can be performed at a later time without affecting the overall runtime.

In the dense block-oriented Cholesky factorization computation, each task has some destination block that is affected by it. To some rough approximation, the urgency of a task in this computation is determined by the column number of the block that is affected by that task. The more urgent of two tasks is the one that modifies the leftmost block. Note that a program that always chooses the highest priority task will not necessarily perform the computation in the minimum amount of time. The optimal schedule is determined by more than the relative urgencies of the various tasks. However, an approach that takes the priority of tasks into account will certainly lead to a better schedule than one that ignores these priorities entirely.

One possible approach to task scheduling would be to always choose the available task with the highest priority. One disadvantage of such an approach is that it may miss high priority tasks that arrive soon after a lower priority task is started. We investigate an approach that goes one step further in explicitly managing the order in which tasks are executed. Recall that the tasks that are performed on behalf of a block are known before the computation begins. Each block will produce updates to all of the blocks to its right. It is a simple matter for a processor to simply perform the tasks assigned to it in a strictly decreasing priority order. With such an approach, each processor would wait for the outstanding task with highest priority. If a task other than the one of highest priority arrives, it is simply delayed until higher priority tasks have been executed.

Unfortunately, our earlier notion of task priority does not mesh well with this prioritization approach. The problem is that the task priority scheme discussed earlier does not consider the source block: a task's priority depend only on its destination. Consider the case where a processor owns two blocks, one towards the left of the matrix and the other towards the right. The block towards the left would be expected to produce tasks before the one to the right, even though some of these tasks may have lower priority than those of the right block. Waiting for the highest priority task could therefore result in significant inefficiencies. To avoid this possibility, we modify our notion of task priority by prioritizing based first on the source block, and then on the destination block. Thus, update tasks from source blocks to the left have higher priority. Among the tasks from a particular source block, the task that modifies the leftmost block has the highest priority.

This approach to task scheduling also allows the method to communicate block updates in a different manner. Recall that an update can either be sent directly to its destination or it can be sent to the block immediately to the right, to be combined with the corresponding update from that block and further sent to the right. The first-come, first-served method sent updates directly to their destinations to avoid a potential explosion in the storage requirements of the algorithm. If this method were to send the somewhat random stream of updates produced by a block to the right, the block to the right would have to store received updates until its corresponding updates were computed. Since this block produces updates in a somewhat random order as well, a received update would be expected to have to wait quite a while.

The extremely orderly stream of updates produced by a block in the prioritized method avoids the possibility of a storage explosion, since the updates can be combined quickly thus reducing the number that must be stored. Sending blocks to the right also produces two important advantages. First, the locality of communication is potentially increased (depending on how blocks are mapped to processors), since messages travel between immediate neighbor blocks. Also, for reasons having to do with the costs of adding updates into the destination, the amount of work each processor must perform is decreased. We therefore study a prioritized method that sends updates to the immediate right.

In Figure 11 we present simulation results for this prioritized block-oriented parallel method. One interesting thing to note from these figures is that the erratic behavior that was seen in the previous method has disappeared. This is to be expected, since the computation is not nearly as sensitive to small perturbations. In the first-come, first-served method, a small delay in the arrival time of an urgent task could result in that task being placed at the end of a task queue. In the prioritized approach, the same delay would simply cause the receiving processor to wait a little longer for the task to arrive. Also note that the simulated speedups are much closer to the load

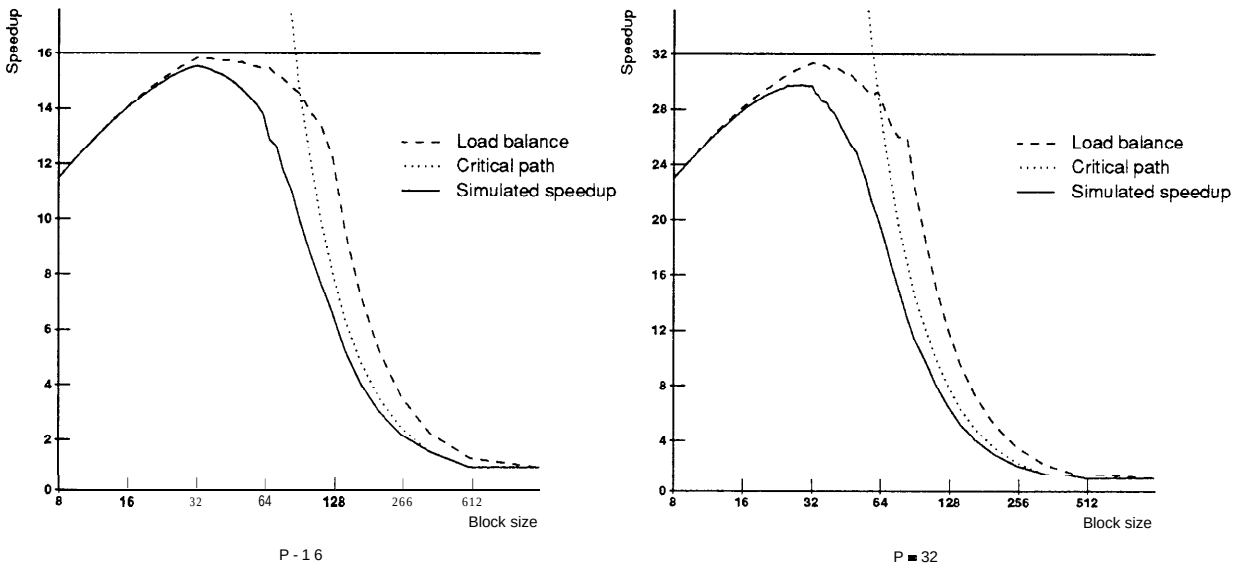


Figure 11: Performance bound for parallel dense Cholesky factorization, n = 1000. $T_{op}(1)$ = 5. β = 4.

balance and critical path upper bounds.

6.4 Block Mapping Strategies

One implementation detail that has not yet been described is the strategy used for mapping blocks to processors. In fact, the previous subsection used an undescribed block mapping strategy. A range of possible strategies are now considered, including the one that was used in the previous subsection.

A strategy for mapping tasks to processors will in general have two goals. As a first goal, it will assign the blocks to the processors so as to minimize processor idle time. The blocks need to be distributed so as to balance the computational load evenly among the processors, while at the same time avoiding the situation where at some point in the computation some processors have many available tasks while others have none. Another goal of a block mapping strategy is to achieve locality of communication. To avoid contention in the processor interconnect, it is best if messages travel between processors that are near each other. The mapping strategies that are considered will address these two goals to varying degrees.

6.4.1 Embedded Mapping

Recall that the blocks in a dense factorization computation communicate with blocks below them and blocks to their right in the matrix. One obvious mapping strategy for the block-oriented computation that addresses the goal of communication locality is one that places adjacent blocks in the matrix onto adjacent processors in the interconnect. We consider *an embedded mapping* strategy in which we assume that a 2-dimensional torus can be embedded in the processor network. This embedded torus is then used in a cookie-cutter fashion to determine the processor mapping (see Figure 12). Note that this mapping strategy does not deal with the load balancing question particularly well. The two primary problems with the resulting load balance are the dimensions of the matrix of blocks, which usually do not align with the dimensions of the embedded torus, and the fact that the matrix of blocks is triangular while the torus is rectangular. Performance results for this mapping strategy will be presented shortly.

The embedded mapping strategy that is investigated here is one of potentially many ways of mapping a triangular array of blocks onto a network of processors while maintaining locality of communication. While other approaches might improve load balance, we do not further investigate this area, and instead we now turn to mapping strategies that abandon some or all of the communication locality obtained with an embedding. One consequence of the decision to investigate approaches with less message locality is that messages may now

experience contention on the interconnect. Recall that our performance model does not consider the impact of such contention. While this is indeed a limitation of our model, we hope that its impact will be minimized by the fact that block-oriented methods are able to use large blocks and thus require very little inter-processor communication.

6.4.2 Column Round-robin Mapping

The second block mapping strategy that we consider is a *column round-robin* strategy. Simply stated, this strategy traverses the grid of blocks one column at a time, moving down a column and then on to the next column, assigning blocks to processors in a round-robin fashion (see Figure 12). The intuition behind this mapping strategy is as follows. Since much of the communication in a block-oriented method involves blocks being broadcast to blocks below them in the same column, a column round-robin mapping keeps this communication localized. Also, since the blocks in a single column are all at the same point in the critical path, they are expected to complete at roughly the same time. Since the blocks begin producing updates as soon as they complete, it is potentially desirable to assign them to different processors so that these updates can be computed simultaneously. Such a mapping is therefore expected to produce a reasonable parallel schedule. The load balance of the computation will hopefully be reasonable since each processor receives a mix of blocks, some requiring a large amount of work and others requiring less work.

6.4.3 Row Round-robin Mapping

Another block mapping strategy that we consider is a *row round-robin* strategy. This strategy is almost identical to the column round-robin strategy, with the difference being that the row round-robin strategy traverses the blocks row-by-row, The communication locality is similar to that obtained with the column round-robin strategy. The scheduling argument used for the column round-robin strategy does not hold, however. Our hope is that the somewhat arbitrary mapping of blocks to processors will avoid pathological scheduling problems.

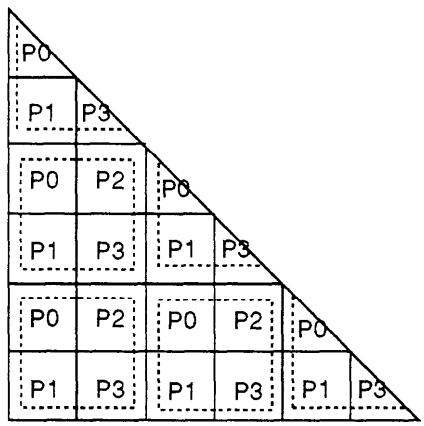
6.4.4 Diagonal Round-robin Mapping

Upon further investigation, we have found that the three mapping strategies described so far all yield relatively poor processor load balances. Our hope that the assignment of multiple blocks to each processor would avoid the possibility of one processor receiving much more work than any other is simply not realized. We therefore add a fourth strategy, the *diagonal round-robin* strategy. This strategy functions as follows. The vast majority of the work related to a block involves the computation of updates for blocks to the right. Therefore, the amount of work associated with a block is determined by its distance from the rightmost block in its row. Equivalently, all of the blocks along a 45 degree diagonal line in the matrix require the same amount of work. A much better load balance can therefore be achieved if these sets of blocks are assigned in a round-robin manner. In other words, this strategy traverses diagonals in the matrix, one at a time, and performs a round-robin assignment of the blocks as they are reached (see Figure 12). This strategy has abandoned message locality entirely, as well as ignoring the possibility of scheduling glitches, but it is expected to produce a greatly improved load balance.

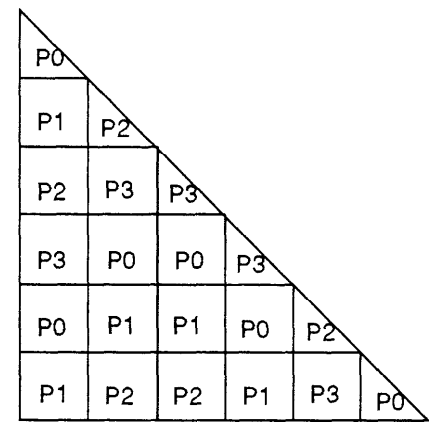
6.4.5 Pre-Simulated Mapping

The fifth block mapping strategy that we consider we term the *pre-simulated* strategy. This strategy performs a rough symbolic simulation of the parallel computation. When it comes time to assign a block to a processor, the state of the simulated computation is consulted to determine the processor that is first available to perform the tasks associated with a block. The block is assigned to that processor. While this strategy might appear to be extremely complicated, the description that follows will show that it is in fact quite simple, due primarily to the very regular manner in which the tasks are handled in a block-oriented method that uses a prioritized schedule.

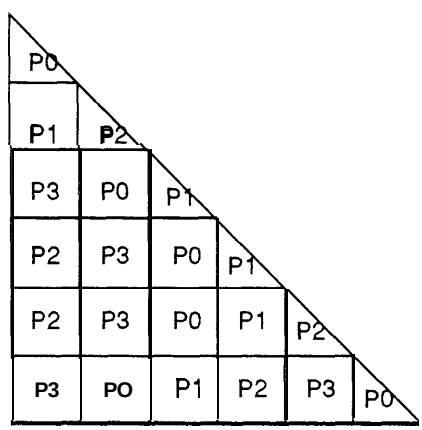
Consider the tasks associated with a single block of the matrix. This block will receive messages containing the blocks above it in the same column, and each of these messages will result in the computation of an update to a block to the right of the receiving block. The block will also receive updates from blocks to its left. Our symbolic simulation will traverse the blocks of the matrix in an order such that all blocks that affect a



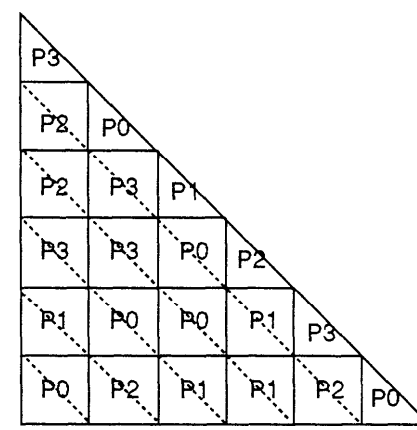
Em bedded



Column Round-Robin



Row Round-Robin



Diagonal Round-Robin

Figure 12: Four block mapping strategies for a four processor machine.

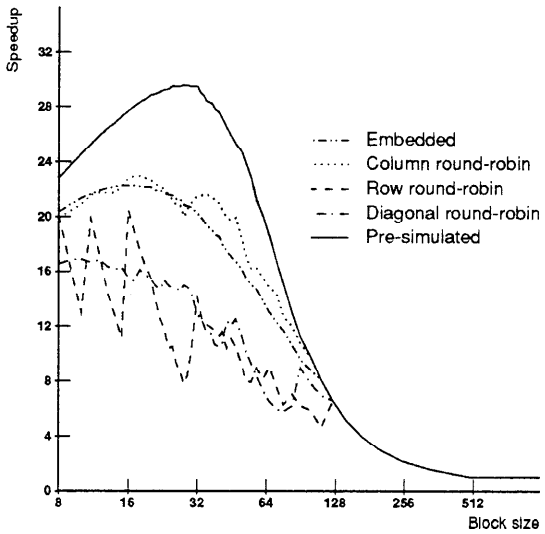


Figure 13: Performance results for five block mapping strategies. n = 1000. T_{op}(1) = 5. 3 = 4. P = 32.

particular block will have been visited by the time that block is visited. The specific order that we choose is column-by-column, top-to-bottom.

Now consider what should be done when a block is reached in this traversal Our goal upon reaching a block is to complete that block and perform all updates emanating from that block as soon as possible. The block of course depends on updates from blocks to the left, but we can assume that these blocks were treated with the same goal in mind, to produce their updates as soon as possible. The block also depends on blocks above it to produce its own updates, and again we can assume that these blocks were completed as soon as possible. The current block is therefore assigned to the processor that is done with its current set of tasks the soonest. Once a block is assigned to a processor, the simulated time of the processor is updated to reflect the fact that this processor must perform the tasks associated with this block before it can handle the tasks of another block The updating of the simulated time requires a means of estimating the time required to perform the various tasks. We use the same estimates that we have been using for our simulations. This pre-simulated strategy has the advantage that it manages the load balance of the computation while at the same time avoiding scheduling glitches by assigning tasks to the processor that is idle the soonest.

One possible disadvantage of a pre-simulated approach is its cost. The block-oriented factorization performs $O(n^3/B^3)$ block operations, and a simulation would be expected to perform some amount of work for each of these block operations. While the resulting simulation cost may be prohibitive, we have found that a full simulation is not necessary. By maintaining summary information about the completion times of the blocks in a column as the simulation proceeds, an $O(n^2/B^2)$ approximate simulation gives results that are almost indistinguishable from the results of a true simulation.

6.4.6 Results

In Figure 13 simulated speedups are shown for the five block mapping strategies we have described. The first thing to note in this figure is that the pre-simulated strategy achieves significantly better results than the other four. The careful placement of each block onto a processor that is free to handle it when it is ready allows this strategy to balance the load well and avoid scheduling glitches.

Looking at the other strategies, the embedded and column round-robin strategies achieve significantly lower performance than the pre-simulated strategy, primarily because of the relatively poor load balance that they provide. The diagonal round-robin and row round-robin strategies achieve the worst performance of all the mapping strategies. This result is especially interesting since the row round-robin strategy produces a load balance that is comparable to the column round-robin strategy, and the diagonal round-robin strategy produces a

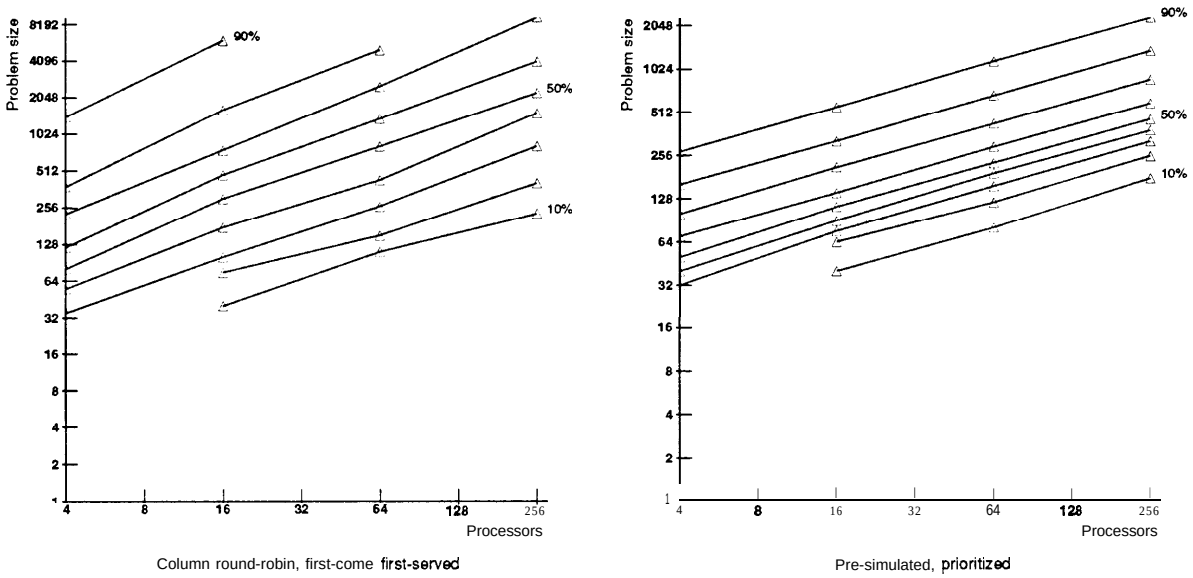


Figure 14: Problem sizes necessary to achieve different processor utilization levels.

load balance that is significantly better than either the column or row round-robin strategies. Further investigation has revealed that the diagonal and row round-robin methods achieve poor performance because these two strategies ignore scheduling issues entirely. The result is a substantial number of scheduling glitches, where some processors are idle while others have numerous tasks to perform.

6.5 Summary

For a more global picture of the performance of block-oriented methods (specifically the simple column round-robin, first-come first-served approach and the more sophisticated pre-simulated prioritized approach), Figure 14 shows the problem sizes that are necessary to achieve various levels of processor utilization. Unfortunately, simulation costs proved prohibitive for many of the larger problems in the graph on the left. The trends are clear, however. By comparing these graphs with each other and with the earlier graph that presented equivalent information for panel-oriented methods (Figure 6), two things become clear. First, the simple block-oriented method gives only slightly better results than the panel-oriented method. Second, the more sophisticated block-oriented method gives much better performance than either the panel-oriented method or the simpler block-oriented method. We therefore conclude that block-oriented methods are capable of achieving high processor utilization levels for moderate problem sizes, but to achieve these levels the related task scheduling and block placement issues must be handled effectively.

7 Discussion

While the data in Section 6 appears to indicate that block-oriented methods are always preferable to panel-oriented methods, we note that there are several exceptions. For example, panel-oriented methods are quite appropriate for small-scale multiprocessors, say with 16 or fewer processors (e.g. SGI 4D/380). They are also appropriate for parallel machines that do not rely on data reuse to achieve high performance (e.g. CRAY Y-MP). In both cases, panel-oriented methods would achieve very high performance for a wide range of problem sizes, leaving little room for improvement. In addition, they are much simpler to implement, and they work with what we consider to be a more natural representation of the matrix. The results of this paper, however, indicate that block-oriented methods are essential for larger parallel machines that rely on significant data reuse.

We now briefly comment on some of the limitations of the study performed in this paper. One potential source of inaccuracy in our results is contention in the processor interconnection network. Our belief is that the communication volumes are sufficiently low and the multiprocessors we have considered are sufficiently

small-scale that network contention would not pose a problem for any of the schemes we have considered. Another obvious limitation of our study is that the results have not been validated using a real parallel machine. We are in the process of performing such validation studies; initial experiments on the Stanford DASH machine indicate that the qualitative predictions are quite accurate.

We also note that the results presented for the panel-oriented method could be improved somewhat by incorporating an optimization. When a processor is ready to perform the modifications from a particular panel, some number of subsequent panels may also be available on that same processor. The processor could perform the modifications from all available source panels, rather than just those from a single one. This optimization could reduce cache misses by as much as a factor of two for small panel sizes. The benefit would depend on the number of source panels available at once, which would depend on the amount of excess concurrency available in the problem.

One practical implementation issue that has not been touched on in this study concerns our assumption that the processor caches act like ideal caches. Specifically, we have assumed that a block of data that is smaller than the processor cache remains in the cache across many uses. Unfortunately, real caches do not necessarily have this property, due primarily to cache conflicts between items in the reused block (see [9]). It is possible to achieve near-ideal cache performance, however, using techniques such as block copying or alternative matrix representations, and these techniques may be necessary for efficient practical implementations.

A natural extension of the work in this paper would be to use the proposed performance models to evaluate the performance of sparse Cholesky factorization. Sparse factorization is much more common in practice, and there is greater interest in solving sparse problems quickly on parallel machines. We consider sparse problems in a related paper [14].

8 Conclusions

In this paper we have examined several methods for performing dense Cholesky factorization on parallel machines. In particular, we have considered the tradeoff between the amount of data reuse exploited by individual processors and the amount of available concurrency. Through the use of performance models and multiprocessor simulation, we have found that this rather complicated tradeoff can be better understood in terms of two upper bounds, a concurrency upper bound and a load balance upper bound.

We first considered panel-oriented factorization methods. We found that the bounds on parallel performance for these methods were quite restrictive. When large panels were used, data reuse levels were high enough to achieve high performance on individual tasks, but concurrency was a bottleneck. When smaller panels were used, overall performance was limited by the lack of data reuse. However, we did find that panel-oriented were able to achieve performance levels that were nearly equal to the performance bounds. Thus, in cases where the bounds are not constraining, panel-oriented methods are quite appropriate.

We then considered block-oriented approaches to the factorization. While offering significantly higher concurrency, these approaches also introduced several complications. A straightforward block-oriented approach, where processors acted on messages in the order in which they arrived, led to erratic and relatively poor performance, This behavior was improved by prioritizing the tasks of the computation. Performance also depended heavily on the mapping of blocks to processors. We symbolically simulated the parallel factorization computation in order to determine an effective mapping. After the above modifications, the block-oriented method achieved extremely high performance, achieving full processor utilizations for reasonable problem sizes. We conclude that a block-oriented approach has significant performance advantages over a panel-oriented approach, but that careful scheduling and data placement are required to achieve these benefits.

Acknowledgments

This research is supported by DARPA contract N00039-91-C-0138. Anoop Gupta is also supported by an NSF Presidential Young Investigator award.

References

[1] Ashcraft, *C., A taxonomy of distributed dense LU factorization methods,* Boeing Tech Report ECA-TR- 16 1, March, 1991.

[2] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D., "LAPACK: A portable linear algebra library for high-performance computers", *Proceedings of Supercomputing '90,* November, 1990.

[3] Dongarra, J., *Performance of various computers using standard linear equations software,* Technical Report CS-89-85, University of Tennessee, October, 1989.

[4] Dongarra, J., Du Croz, J., Hammarling, S., and Duff, I., "A set of level 3 basic linear algebra subprograms", *ACM Transactions on Mathematical Software, 16:* 1-17, *1990.*

[5] Dongarra, J., and Ostrouchov, *S., LAPACK block factorization algorithms on the Intel iPSC/860,* LAPACK Working Note 24, Technical Report CS-90-115, University of Tennessee, October, 1990.

[6] Gallivan, K., Jalby, W., Meier, U., and Sameh, A.H., "Impact of hierarchical memory systems on linear algebra algorithm design", *International Journal of Supercomputer Applications, 2: 12-48, 1988.*

[7] Geist, *G.,* and Heath, M., *Parallel Cholesky factorization on a hypercube multiprocessor, Oak* Ridge National Laboratory Technical Report ORNL-6190, August, 1985.

[8] George, A., Heath, M., and Liu, J., "Parallel Cholesky factorization on a shared-memory multiprocessor", *Linear Algebra and its Applications,* 77:165-187, *1986.*

[9] Lam, M., Rothberg, E., and Wolf, M., "The Cache Performance and Optimizations of Blocked Algorithms", *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* April, 1990.

[10] Lenoski, D., Gharachorloo, K., Laudon, J., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M., "Design of scalable shared-memory multiprocessors: The DASH approach", *Proceedings of COMPCON '90,* 1990.

[11] Naik, V.K., and Patrick, M.L., *Data traffic reduction schemes for Cholesky factorization on asynchronous multiprocessor systems,* IBM Research Report No. 64785, 1989.

[12] O'Leary, D., and Stewart, G., "Assignment and scheduling in parallel matrix factorization", *Linear Algebra and its Applications,* 77:275-299, *1986.*

[13] Rothberg, E., and Gupta, A., "Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations", *Proceedings of Supercomputing '90,* November, 1990.

[14] Rothberg, E., and Gupta, A., "The performance impact of data reuse in parallel sparse Cholesky factorization", in preparation.

[15] Saad, Y., "Communication complexity of the Gaussian elimination algorithm on multiprocessors", *Linear Algebra and its Applications,* 77:315-341, *1986.*

[16] Van De Geijn, R., *Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems,* Technical Report CS-91-28, University of Texas at Austin, August, 1991.