February 1990

Report No. STAN-CS-90-1305

Also Numbered CSL-TR-90-416

A Comparative Evaluation of Nodal and Supernodal Parallel Sparse Matrix Factorization: Detailed Simulation Results

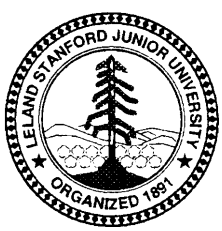
by

Edward Rothberg and Anoop Gupta

Department of Computer Science

Stanford University

Stanford, California 94305



SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

form Approved
OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION	1 b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION /AVAILABILITY OF REPORT
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: Computer Science Department Stanford University; 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Stanford, CA 94305	7b ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING /SPONSORING ORGANIZATION: DARPA; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-87-K-0828
8c. ADDRESS (City, State, and ZIP Code): Arlington, VA	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
A Comparative Evaluation of Nodal and Supernodal Parallel Sparse Matrix Factorization: Detailed Simulation Results

12 PERSONAL AUTHOR(S)

13a TYPE OF REPORT	13b TIME COVERED FROM _____ TO _____	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

Abstract

In this paper we consider the problem of factoring a large sparse system of equations on a modestly parallel shared-memory multiprocessor with a non-trivial memory hierarchy. Using detailed multiprocessor simulation, we study the behavior of the parallel sparse factorization scheme developed at the Oak Ridge National Laboratory. We then extend the Oak Ridge scheme to incorporate the notion of supernodal elimination. We present detailed analyses of the sources of performance degradation for each of these schemes. We measure the impact Of interprocessor communication costs, processor load imbalance, overheads introduced in order to distribute work, and cache behavior on overall parallel performance. For the three benchmark matrices which we study, we find that the supernodal scheme gives a factor of 1.7 to 2.2 performance advantage for 8 processors and a factor of 0.9 to 1.6 for 32 processors. The supernodal scheme exhibits higher performance due mainly to the fact that it executes many fewer memory operations and produces fewer cache misses. However, the natural task grain size for the supernodal scheme is much larger than that of the Oak Ridge scheme, making effective distribution of work more difficult, especially when the number of processors is large.

20 DISTRIBUTION /AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code) 22c OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603

A Comparative Evaluation of Nodal and Supernodal Parallel Sparse Matrix Factorization: Detailed Simulation Results

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

February 28, 1990

Abstract

In this paper we consider the problem of factoring a large sparse system of equations on a modestly parallel shared-memory multiprocessor with a non-trivial memory hierarchy. Using detailed multiprocessor simulation, we study the behavior of the parallel sparse factorization scheme developed at the Oak Ridge National Laboratory. We then extend the Oak Ridge scheme to incorporate the notion of supemodal elimination. We present detailed analyses of the sources of performance degradation for each of these schemes. We measure the impact of interprocessor communication costs, processor load imbalance, overheads introduced in order to distribute work, and cache behavior on overall parallel performance. For the three benchmark matrices which we study, we find that the supemodal scheme gives a factor of 1.7 to 2.2 performance advantage for 8 processors and a factor of 0.9 to 1.6 for 32 processors. The supemodal scheme exhibits higher performance due mainly to the fact that it executes many fewer memory operations and produces fewer cache misses. However, the natural task grain size for the supernodal scheme is much larger than that of the Oak Ridge scheme, making effective distribution of work more difficult, especially when the number of processors is large.

1 Introduction

The factorization of a large sparse positive definite system of equations on a multiprocessor is an important problem. A number of parallel factorization schemes have been proposed, and a number of implementations have been developed [1, 4, 7, 12]. Unfortunately, the results of such implementations are typically presented as a single set of numbers, the parallel runtimes. Parallel runtimes are influenced by several important factors, including interprocessor communication costs, processor load imbalance, overheads introduced in order to distribute work, and processor cache behavior. In order to better understand how parallel factorization performance can be improved, we must obtain a better appreciation for the contributions of each of these factors to overall parallel runtime.

In this paper, we use a detailed multiprocessor simulator in order to measure the importance of the various factors which affect parallel runtime. We begin by studying the behavior of the parallel factorization scheme developed at the Oak Ridge National Laboratory. We present the speedups obtained on our parallel machine model, and investigate the reasons for the obtained results. We find that this scheme achieves good parallel speedups, as compared to the same scheme run on a single processor. The cost of communicating data between the processors is almost entirely offset by the reduction in the number of cache misses which occurs when more processors, and thus more cache memory, are available.

We then extend the Oak Ridge scheme by making use of the concept of supemodal elimination. In supemodal elimination, sets of matrix columns, called supemodes, are treated as single units, and the operations related to the columns in a set are performed as a group. We find that the consolidation of the work associated with a supemode provides substantial benefits for parallel factorization. Runtimes for single processor runs are decreased by a factor of two to three over the Oak Ridge scheme. The benefits are substantial for multiple-processor runs as well, although they decrease as the number of processors is increased. The benefits of the

consolidation of work that supemodes make possible must be traded against the load balancing problems which result from a larger task grain size. For larger numbers of processors, the supemodes must be broken into smaller pieces in order to keep the processors busy, thus decreasing the benefits.

To further improve the performance of supemodal factorization, we first examine how finely the supemodes should be split in order to keep multiple processors busy. We find that the benefits of decreasing processor idle time by splitting supemodes into finer pieces are offset by the overheads which come with the splitting. We also investigate a more dynamic scheme for assigning tasks to processors. In the Oak Ridge scheme, and our subsequent supemodal modification, all task assignment is done statically, before the factorization begins. We find that by assigning tasks more dynamically, the processor idle time is reduced, but the increased communication which results again offsets the benefits of the decreased idle time.

This paper is organized as follows: In section 2, we briefly discuss the problem of sparse Cholesky factorization. We assume the reader has some prior knowledge of sequential sparse Cholesky factorization methods. Then in section 3, we describe the parallel factorization scheme developed at the Oak Ridge National Laboratory. Section 4 describes the parallel machine model which we assume for our experiments, and in section 5 we present the results of factoring a number of benchmark matrices using the Oak Ridge scheme on our machine model. Section 6 describes how supernodal elimination can be incorporated into the Oak Ridge parallel factorization scheme, and in section 7 we examine the behavior of the resulting parallel factorization code. Section 8 discusses the load balancing problems which supemodes bring up, and considers a number of possible methods for alleviating these problems. Then in section 9, we examine in detail the behavior of parallel factorization schemes over time. That is, we look at how computation, processor utilization, and communication rates vary throughout the computation. In section 10 we discuss directions for future work and we conclude in section 11.

2 Sparse Cholesky Factorization

We now present a brief description of the problem of sparse Cholesky factorization, including a brief discussion of the concept of an elimination tree and the concept of supemodal elimination. The reader should consult [8] for further information on sparse factorization, [10] for further information on elimination trees, and [21] for further information on supemodal elimination.

Consider a system of equations, $Ax = b$, where A is an $n \times n$ sparse positive definite matrix, b is a column vector of length n, and x is an unknown column vector, also of length n. This sparse positive definite system of equations is typically solved using a sequence of four steps, in which the matrix A is factored into the form LL^T and the solution x is determined by performing a pair of triangular system solves. The first step, *ordering*, heuristically reorders the rows and columns of A in order to decrease the amount of fill in factor matrix *L. The* second step, *symbolic factorization,* determines the non-zero structure of *L,* and sets up storage for the factor. The third step, *numerical factorization,* determines the values of the non-zero elements in *L,* using the structure determined in the symbolic factorization step. *The* fourth step, *triangular solution,* solves two triangular systems, using the *L* matrix computed in the numerical factorization, to determine the value of

This paper only considers the most time-consuming of the four steps, the numerical factorization step. We now present pseudo-code for a column-oriented method for performing the numerical factorization:

```
1.  for k = 1 to n do
2.        l_kk ← √l_kk
3.        for each i s.t. l_ik ≠ 0 do
4.              l_ik ← l_ik / l_kk
5.        for each j s.t. l_jk ≠ 0 do
6.              l_*j ← l_*j − l_jk * l_*k
```

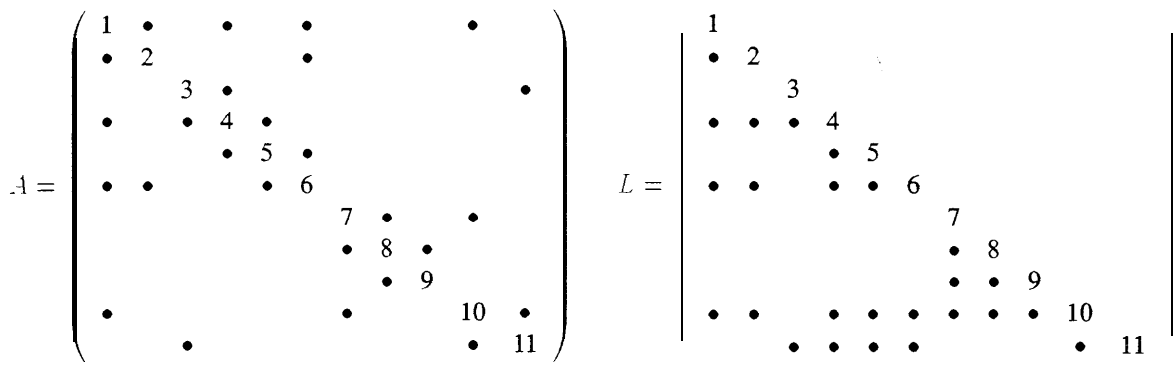


Figure 1: Non-zero structure of a matrix A and its factor L.

At each iteration, column k has been modified by every column which will modify it. In steps 2 through 4, column k is scaled by the square root of the diagonal, and it is then complete. This scaling operation is typically referred to as $cdiv(k)$. In steps 5 through 6, the newly completed column k is used to modify all columns j which are modified by it. The modification of a column j by column k is typically referred to as $cmod(j, k)$. The above pseudo-code can therefore be restated as:

```
1.  for k = 1 to n do
2.      cdiv(k)
5.      for each j s.t. l_jk ≠ 0 do
6.          cmod(j, k)
```

In the above method for numerical factorization, the loop of statement 1 defines an order in which the columns are completed. That is, column k will always be completed before all columns $k + c$, c > 0. This order is not necessarily the only order which will produce the correct result. Consider, for example, the case where column k does not modify column $k + 1$. This means that column $k + 1$ does not depend on column k, and therefore it can be completed before column k, with exactly the same result. Looking ahead to parallel factorization, notice that columns k and $k + 1$ could also be completed simultaneously, again with the same result.

The determination of which columns are dependent on which other columns is made simpler by looking at the *elimination tree* of matrix A [10]. Given a matrix A, with factor L, the parent of column j in the elimination tree of A is:

$$parent(j) = \min\{ i | l_{ij} \neq 0,\ i > j \}$$

In other words, column j is a child of column i if and only if the first sub-diagonal non-zero of column j in L is in row i. It can be shown that a column will only modify its ancestors in the elimination tree, and equivalently that a column will only be modified by its descendents. Given this fact, it can be shown that columns which do not have an ancestor/descendent relationship in the elimination tree are independent of each other, and thus can be completed in parallel.

Consider the example matrix of Figure 1, and its elimination tree in Figure 2. We now consider the dependencies among the columns of this matrix, and how the numerical factorization of this matrix would proceed We know that a column can only be modified by its descendents in the elimination tree. Since columns 1, 3, and 7 of the example matrix A are leaves in the elimination tree of A, they are not dependent on any other columns. They can thus be completed immediately, in any order or in parallel. Once complete, their contributions to other columns are propagated. By the nature of elimination trees, we know that a column can only modify its ancestors in the tree. Thus, for example, column 7 can only modify columns 8 through 11. Similarly, column 3 can only modify columns 4 through 6 and columns 10 through 11. Once these contributions have been propagated, then columns 2 and 8 have received all contributions which will be made to them. All of their descendents in the elimination, names columns 1 and 7, have modified them. They are thus ready to be completed. This process of determining columns all of whose descendents in the elimination tree have been completed, and then completing them, continues until all columns in the matrix have been completed.

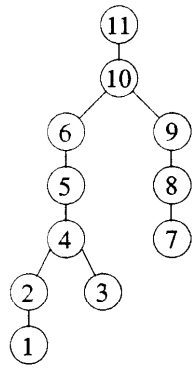


Figure 2: Elimination tree of A.

Another important concept in sparse factorization is the concept of the supernode. The term supemode is derived from the graph interpretation of a sparse matrix. In such an interpretation, columns of the matrix correspond to nodes in an undirected graphs, and a non-zero in matrix location (i. **j)** corresponds to an edge between the node representing column *i* and the node representing column **j.** A supemode is a set of nodes in the graph representation of a factor matrix *L* with the following three properties. First, the column numbers of the nodes in the set form a contiguous block. Second, all nodes within the set are adjacent to all other nodes within the set. Third, if one considers the set of all nodes with larger column numbers than those of the nodes in the supemode, then all nodes in the supemode are adjacent to the same subset of these nodes. In the matrix representation of *L*, a supemode appears as a set of adjacent columns with a dense triangular block on the diagonal, followed by identical non-zero structure for each of the columns below this block. For example, in the matrix of Figure 1, the supemodes are $\{1, 2\}$, $(3\}$, $(4, 5, 6\}$, $\{7, 8, 9\}$, and $\{10, 11)$. Supemodal elimination, as opposed to nodal elimination, is the action of modifying a column by an entire supemode instead of simply by a single column [2].

By making use of the fact that the columns within a supemode have such similar structure, supemodal elimination can provide substantial advantages over a nodal elimination approach. One important advantage is that supemodal elimination decreases the number of references to index vectors needed. Since all of the columns in a supemode have the same structure, the indices corresponding to the non-zeroes in the supemode need only be referenced once per supemode, instead of once per column, when performing a column modification. Another significant advantage comes from the fact that supernodal elimination permits a form of loop unrolling which decreases the number of loads and stores done from/to memory. This unrolling allows one to perform many modifications to a data item once it has been loaded into a register. The result is a substantial decrease in the total number of loads and stores, and thus a reduction in the amount of memory traffic. The loop unrolling available in non-supemodal factorization does not permit multiple modifications to a loaded data item, and thus only decreases loop overhead, a much less important component of factorization time. The third benefit of supemodal elimination is the ability to better utilize the processor cache. By making possible a number of cache-miss reducing techniques, supemodal elimination allows one to substantially decrease the number of cache misses which occur in factoring large matrices. The net effect of these three advantages was a threefold improvement in Cholesky factorization speed over a purely nodal code, SPARSPAK, on a DECStation 3100 as shown in [15].

3 Parallel Sparse Factorization

3.1 An Overview of the ORNL Scheme

The parallel factorization scheme developed at the Oak Ridge National Laboratory (ORNL scheme) performs sparse Cholesky factorization on a local-memory multiprocessor [7]. On such a machine, a processor can only access the memory local to it. All communication between processors is accomplished by passing messages

```
1. while I own a column which is not complete do
2.        if nmod[k] = 0 then --- k is ready to be completed
3.           cdiv(k)
4.           send k to processors which own columns {j|j > k. l_jk ≠ 0}
5.        else
6.              receive a column k
7.              for each j s.t. l_jk ≠ 0 which I own do
8.                   cmod(j, k)
```

Figure 3 : Oak Ridge parallel factorization algorithm.

on an interconnection network. In the ORNL scheme, each column of the matrix is assigned to a particular processor, which we refer to as the column's owner. The method for determining the distribution of columns to processors will be discussed later in this section. The storage for a column is allocated in the local memory of its owner, and all modifications to a column are performed by the owner processor.

Before beginning the numerical factorization process, the ORNL scheme first examines the structure of the matrix in order to determine the number of columns which will modify each column k. This number, which is referred to as $nmod[k]$, is decremented when a modification to column k occurs during the factorization process. Once this count reaches zero, then all modifications to column k have been performed and column k is ready to be completed.

The overall structure of the parallel factorization computation is as follows. A processor chooses a column k which it owns and which is ready to be completed, and performs a $cdiv(\ )$ operation on it. The processor then sends a message containing the completed numerical values of column k to each processor which owns a column which is modified by k. Once all columns which can be completed are completed, then the processor waits to receive a message containing the values of some newly completed column owned by another processor. The processor performs a $cmod(\ )$ on all such columns, which may result in more columns becoming ready to be completed. This process repeats until all columns have been completed. A pseudo-code description of this algorithm appears in Figure 3. Note that this code is executed by each processor.

Our implementation contains a slight optimization to the $cmod(\ )$ operation [1]. Usually, when modifying column j by column k, for each non-zero in column k we must search for the non-zero which appears in the corresponding row in column j. Our optimization simply checks whether the number of non-zeroes in column j is equal to the number of non-zeroes below row j in column k. If they are equal, then the columns have the same structure for the purposes of this $cmod(\ )$, and no search in column j is necessary. This modification improves performance by approximately 10% for relatively sparse matrices, and by approximately 40% for dense matrices.

3.2 Distribution of Columns to Processors

We now return to the issue of how columns are distributed to processors in the ORNL scheme. The column distribution must be performed with a number of considerations in mind. First, since sending a message between two processors is an expensive operation, the distribution should strive to keep the number of messages as low as possible. Clearly this is not the only objective, though, since assigning all columns to a single processor results in an ideal zero messages. In order to make effective use of a number of processors, the computational load must also be well balanced. Since each processor is responsible for performing all modifications to the columns which it owns, the columns should be distributed such that the processor workloads are as uniform as possible. Furthermore, the distribution must be made such that processors do not spend large amounts of time sitting idle, waiting for columns owned by other processors to be completed and sent.

One distribution technique which provides excellent load balancing characteristics is the *wrap-mapping*, where column k is assigned to processor k mod P. This interleaving of columns, with each processor receiving an approximately equal number of columns, makes it unlikely that any processor will have too much work to

[1] It is not clear from [7] whether this optimization was present in the original ORNL implementation.

do. Similarly, significant idle time is unlikely with such a mix of columns assigned to each processor. However, this scheme has been shown to require an enormous number of interprocessor messages [9].

An alternative to the wrap-mapping described above is the *subtree-to-subcube mapping* [9]. Subcube refers to an $n - 1$ dimensional subcube of an n dimensional hypercube, the topology which was employed in the interconnection network of the multiprocessor used in their experiments. This mapping uses the elimination tree of the matrix to determine how columns should be distributed. In our previous discussion of elimination trees, it was noted that columns with no ancestor/descendent relationship between them in the elimination tree are independent, and thus can be processed simultaneously. The columns of an elimination tree can therefore be divided in the following way. We start at the root of the elimination tree, and proceed down the tree until a node with more than one child is reached. The children of this node are root nodes of a set of subtrees. Since no node in any of these subtrees can be an ancestor or descendent of any node in the other subtrees, these subtrees are independent. The elimination tree can thus be divided into a set of subtasks based on these subtrees. The subtasks will be referred to as follows. The set of nodes on the path from the root to the first node with more than one child is called the *separator subtask*. The subtrees rooted at the children of that node are simply called *independent subtasks*. Note that while the independent subtasks are independent of each other, the separator subtask is dependent on all of the independent subtasks. A single separator subtask will typically generate only two independent subtasks. As an example of such a division, the elimination tree of Figure 2 would be divided as follows: nodes 10 and 11 would form the separator subtask, and the subtrees rooted at node 6 and node 9 would form the independent subtasks.

The subtree-to-subcube mapping takes a divide-and-conquer approach to parallel factorization, making use of the ability to split an elimination tree into a number of independent pieces. It recursively distributes the work of the factorization among P processors as follows: The factorization task is divided into three subtasks, as in the previous paragraph. For the subtree-to-subcube mapping, the hope is that the two resulting independent subtasks will require approximately the same amount of time to complete. These two subtasks are distributed to two disjoint subsets of the P available processors, each subset of size $P/2$ (i.e. subcubes in the hypercube). Once these two subtask have been completed, the separator subtask is computed by all P processors. This division of work continues recursively until $P = 1$, at which point all of the columns in the remaining subtask are assigned to one processor. Columns in a separator subtask are assigned to the processors responsible for that subtask in a wrap-mapping fashion.

The subtree-to-subcube mapping has proven to be an effective distribution technique for matrices where the nested dissection ordering heuristic [8] generates good orderings. The ORNL parallel factorization scheme requires substantially fewer messages with the subtree-to-subcube mapping than with the wrap-mapping. In general, nested dissection is an effective ordering heuristic for matrices which result from finite element problems. However, in the case where nested dissection does not generate a good ordering, the subtree-to-subcube mapping can be substantially less effective. One potential reason for a loss of effectiveness is the fact that a poor ordering can produce large amounts of fill in the factor and thus can result in much more work being required in order to compute the factor. In the sequential factorization case, this extra work can often be avoided by using a different ordering heuristic, such as the minimum degree ordering [8]. In this case, the speedups of the multiprocessor implementation which relies on the nested dissection ordering will suffer, due to the poor uniprocessor performance relative to a sequential code which uses a minimum degree ordering. Another possible reason for performance loss when using the subtree-to-subcube mapping is the potential for unbalanced subtasks. The load balancing of the subtree-to-subcube mapping is highly dependent on the relative amounts of time that each of the two independent subtasks require in order to complete. If one subtask completes significantly earlier than the other, then the processors responsible for the completed subtask will be idle until the other subtask completes.

Geist and Ng [6] have developed a column distribution scheme which addresses the case where it is either not possible or simply not desirable to divide a factorization task into two nearly-equal size subtasks. The basic goal of their scheme is to assign some set of subtasks to each processor such that all processors have roughly the same amount of work assigned to them. Their method can briefly be described as follows: A pool of subtasks is maintained, with all subtasks in the pool independent of each other. Associated with each subtask is an estimate of the amount of time which the subtask would require to complete. The estimate which they choose is simply the number of floating point operations performed when processing the subtask. The tasks in the current pool are distributed heuristically to the available processors, with the goal of making the difference between the largest and smallest amount of work assigned to any one processor as "small" as possible. If an acceptable

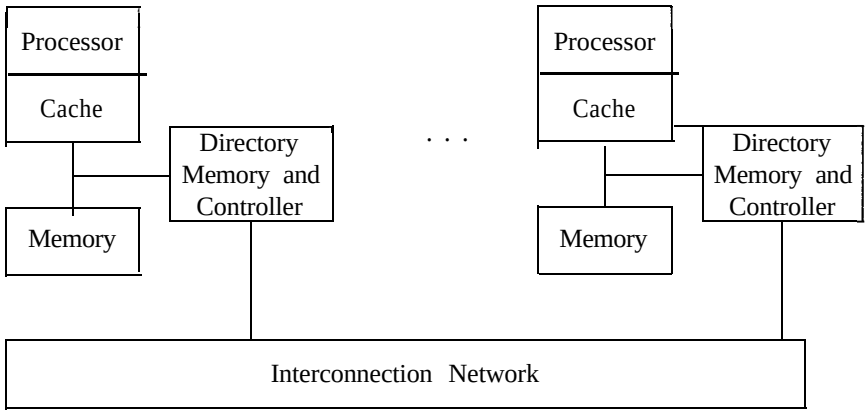


Figure 4: Organization of our multiprocessor model.

distribution of work is not found for the current pool of tasks, then the task with the largest estimated runtime is split into three subtasks, as in the subtree-to-subcube scheme. The two independent subtasks are placed into the task pool, and the separator subtask is distributed among all of the available processors. The task pool now contains a different mix of tasks, and an even distribution is again attempted. This process repeats until an acceptable balance in the amount of work assigned to each processor is achieved.

A great deal of work is being done (see, for example, [11]) on the problem of how to distribute the factorization work so as to achieve low communication, as was achieved in the subtree-to-subcube mapping approach, while also maintaining good load balancing for the full spectrum of matrices which are encountered in practice. However, at this point it is not clear that any general purpose approach to the problem has been developed, Since we are interested in studying general purpose parallel sparse factorization, the subtree-to-subcube mapping is not appropriate for our study. We will therefore study the subtask partitioning approach of Geist and Ng.

In this section we have summarized some of the work which has been done on parallel sparse Cholesky factorization. We have provided an overview of the scheme developed at the Oak Ridge National Laboratory, and have discussed some of the column assignment issues which this scheme brings up. We will now study the performance of the ORNL scheme using a detailed multiprocessor simulator, in order to learn more about the demands which the ORNL scheme places on multiprocessor hardware.

4 Multiprocessor Model

The machine model which we have chosen for our simulated multiprocessor is an attempt at extracting the most important characteristics of the Stanford DASH multiprocessor [14]. The DASH machine is a scalable shared-memory architecture with coherent caches currently being built at Stanford. A directory-based protocol is used to keep the caches coherent. We now present a brief overview of our machine model, and the tools which we use to simulate this multiprocessor model.

Our simulated shared-memory machine consists of a number of processing nodes, each containing a 64 Kilobyte data cache and a portion of global memory. The processing nodes communicate with each other through an interconnection network. The high-level organization of our machine model is shown in Figure 4. The base processor in our machine is identical to that used in the DASH prototype, the MIPS R3000/R3010 running at 25 MHz. In a typical system, this processor is nominally rated at 20 MIPS and 4 MFLOPS. The processor data cache can hold the contents of any location in global memory, not just the locations held in the memory local to the processor. A directory is kept with each location in memory, which keeps track of the set of processors which are currently caching that particular location. When a processor reads a location, a read request is sent to the node in which the requested memory location resides. The contents of that location are returned to the requesting processor, and a bit is set in the directory of that location, indicating that the location now resides in the cache of the requesting processor. When a write is done to a memory location, the directory

of that location is used in order to send invalidate messages to all caches which contain a copy of that location.

Naturally, reading a memory location which is non-local requires more time than reading a local memory location. In our machine model, we assume the following costs for memory operations. These costs are estimates of what these operations will actually cost on the DASH prototype. All writes are assumed to require a single processor cycle. While the write will obviously not complete in a single cycle, a write buffer enables the processor to continue its computation while the write is being serviced. A processor read which hits in the cache is serviced in a single processor cycle, and does not involve any memory module. A read which misses, on the other hand, must be serviced by some memory module. If the location accessed is stored in the memory module local to the processor which issued the access, then the cache miss can be serviced locally, requiring 20 processor cycles. This will be referred to as a *local cache miss* throughout this paper. If the location is stored in a remote memory module, then the request requires communication between two distinct nodes of the multiprocessor. The node from which the memory request was issued must send a message to the node in which the requested location is stored. That node will then reply with the contents of the requested location. This transaction requires two messages to be sent on the interconnection network, one to request to location and one to provide its contents, and thus will be referred to as a *two-hop cache miss* throughout the paper. A two-hop cache miss requires 60 cycles to service. The third possibility for a read miss is that the location is requested by one node, has its home in a second node, and is currently in a modified, or *dirty* state in the processor cache of a third node. In this case, one message is sent from the requester node to the home node to request the location, another message is sent from the home node to the node where the location is dirty, and a third message is sent from the node where the location is dirty to the node which requested the location providing the contents of that location. This transaction will be referred to *as* a *three-hop cache miss;* it requires 80 processor cycles to service.

The accurate simulation of a machine with the above characteristics is made possible by the Tango [3] architectural simulation tool developed at Stanford. Tango allows the user to take parallel code, annotated with parallel constructs from the ANL macro package [13], and simulate its multiple processor execution on a single processor. Note that the actual parallel code is executed during the simulation. The source code can be recompiled and run, unmodified, on a true multiprocessor. Tango allows the user to trap all memory references made by the program, thus allowing the monitoring of memory traffic and the simulation of any desired memory system. For the simulation results which follow, we implemented the memory model described in the previous paragraph by catching each memory reference and determining the amount of time which would be required to service it. In Tango, each simulated processor has a virtual time associated with it. The effect of the latency of a cache miss is modeled by simply incrementing the issuing processor's virtual time by the appropriate amount.

Note that our machine model says nothing about the interconnection between nodes in the multiprocessor. In the DASH prototype, a two-dimensional mesh interconnect is being used, but we ignore this aspect of the multiprocessor architecture here. While this omission may represent an optimistic view of multiprocessor performance, we believe that any inaccuracies which result from this simplification will be small. This is due to the following two assumptions. First, we assume that the non-local cache miss service time will be effectively independent of the number of intermediate nodes through which a message must travel. In the DASH prototype, the time it takes a message to propagate to its destination is a very small part of the overall cache miss latency. Therefore, a small increase in the propagation time has little effect on the overall latency. Our second assumption is that contention on the network will be small. Data traffic results which will be presented later in this paper will serve to back up this claim. The main reason for our omission of network considerations is simulation runtime. Without any network simulation, the ratio of simulation time to real time is approximately 1000 to 1. In other words, a run which would require 1 second on a real machine requires approximately 1000 seconds to simulate. With the added overhead of an accurate interconnection network model, this ratio would have increased to the point where this study would no longer have been feasible.

5 Simulation Results for the Oak Ridge Scheme

We now present the results of executing the Oak Ridge parallel factorization scheme on our simulated multiprocessor. Throughout the rest of this paper, we will study a set of three benchmark matrices. The first, LSHP3466, comes from a finite element discretization of an L-shaped region. We have chosen it in order to represent the class of matrices which result from the discretization of two-dimensional domains. The second

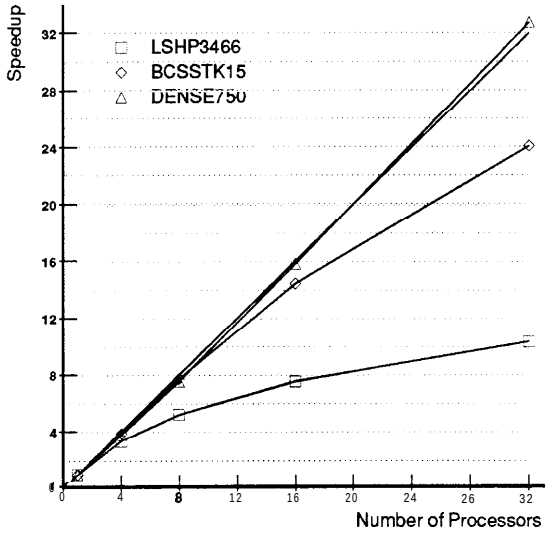


Figure 5: Speedups for ORNL scheme.

matrix, BCSSTK15, is the global stiffness matrix for a module of an offshore platform. This matrix was chosen as a representative of the three-dimensional problem domain. The third matrix, DENSE750, is simply a dense 750 by 750 matrix. The dense problem is included because it removes some of the complexities of the sparse problems, thus presenting an easier to understand problem, while still retaining aspects which are important to sparse problems. Matrices LSHP3466 and BCSSTK15 come from the Boeing-Harwell Sparse Matrix Test Collection [5].

Before we present multiprocessor speedups for our implementation of the ORNL scheme, we must first discuss how the factorization data structures are distributed among the local memories of the processors. The arrangement of data is an important issue which can have a dramatic effect on the amount of communication necessary, and consequently on the overall runtime. We have tried to keep data local to the processor which uses it. This is not difficult for the data structures which are read-only during the numerical factorization. The primary example of these is the set of data structures which keep track of where non-zero values are present in the factor matrix. The structure of the factor is known before the numerical factorization begins, since it is computed in the symbolic factorization phase. All read-only data structures are replicated, so that each processing node has its own copy in local memory. The data structures which are modified during the numerical factorization, the primary example being the structure containing the non-zero values in the factor matrix, are distributed among the memory modules based on column ownership. In other words, the storage for the non-zero values of a column is allocated in the memory module of the processor which owns that column. The only accesses which a processor makes to non-local memory modules occur when a column modification is done. If column j is to modify column k, then the processor which owns column k reads the possibly non-local values in column j. In order to quantify the relative sizes of the replicated and distributed data structures, we look at figures for 32 processor runs. Since 32 is the largest number of processors which we use in our simulations, these figures represent the largest percentage of overall storage that replicated data will require. We find that replicated data consumes 89%, 67%, and 20% of total storage for LSHP3466, BCSSTK15, and DENSE750, respectively. Some of the read-only data is read sufficiently infrequently that it could be distributed without a significant loss in performance, but we do not study this issue in this paper.

In Figure 5 we present the speedups obtained with the ORNL scheme on our three test matrices. Note that these speedups are relative to the performance of the parallel code on a single processor. A true sequential code would be somewhat faster, thus resulting in smaller speedups; we will return to this issue later in this paper.

In order to better display the behavior of the parallel code, we now present runtime breakdown graphs, where we attempt to break down the total runtime of the factorization into its most important and most easily identified components. The first component which *we* distinguish is simply *processor utilization,* the percentage of the total runtime which each processor spends executing useful program instructions. We define useful instructions

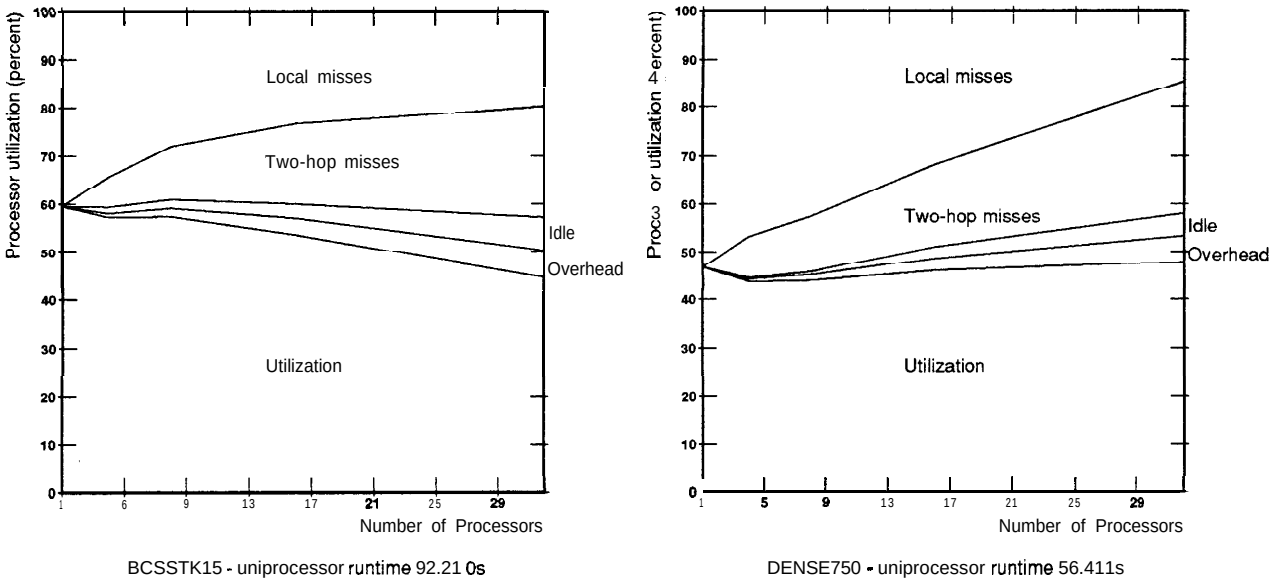


Figure 6: Processor utilization for ORNL scheme.

to be those instructions which would be executed when factoring the same matrix on a single processor. This portion is the main determinant of how effectively the processors are being used, and thus how quickly the matrix is factored. The next component in the breakdown is *multiprocessing overhead,* which gives the percent of time spent executing program instructions which are introduced strictly in order to allow parallelism in the program. Such overhead is generally unavoidable in writing parallel programs, due to the fact that it requires more work to coordinate a number of processors than it does to simply use a single processor. The next component, *idle time,* gives the percent of time each processor spends with nothing to do. Again, idle time is virtually unavoidable in parallel programs. Unless the computation can be perfectly load balanced, processors will sometimes be left with nothing to do. The next component, *two-hop cache misses,* gives the percent of time spent in fetching data from non-local memory. Two-hop cache misses are the method by which processors communicate in our multiprocessor machine model, and thus this component measures the cost of interprocessor communication when factoring the matrix. Finally, the *local cache misses* component gives the percent of time each processor spends fetching data items from its local memory.

In Figure 6 we give runtime breakdown graphs for the two matrices which achieve near-linear speedup, DENSE750 and BCSSTK15. This figure shows, for example, that in factoring matrix BCSSTK15 on a single processor, roughly 40% of the runtime is spent waiting for cache misses to be serviced, while the other 60% is spent executing the instructions of the factorization code. Similarly, when factoring BCSSTK15 on 32 processors, roughly 20% of the runtime is spent servicing local cache misses, roughly 25% is spent servicing non-local cache misses, and roughly 45% is spent executing program instructions.

Not surprisingly, as the number of processors is increased, the percentage of time spent in communicating information between processors (which is reflected in the *two-hop misses* section of the graphs) increases as well. For both matrices, however, this increase in communication cost is almost entirely offset by a decrease in the cost of servicing local cache misses. This decrease can be explained as follows, Each processor is statically assigned a portion of the matrix on which it will work. As the number of processors is increased, clearly the size of the portion assigned to each processor decreases. Since the amount of cache memory per processor remains constant (at 64 Kbytes), the cache becomes more effective when used to cache a smaller data set.

In figure 7, we give a runtime breakdown for the smaller matrix, LSHP3466. Note that a much smaller percentage of time is spent servicing cache misses for this matrix, as compared to the previous two. For the single processor run, cache misses comprise less than 15% of the runtime, as compared with the 40% or more which was spent here for the previous matrices. This superior cache performance is to be expected, since LSHP3466

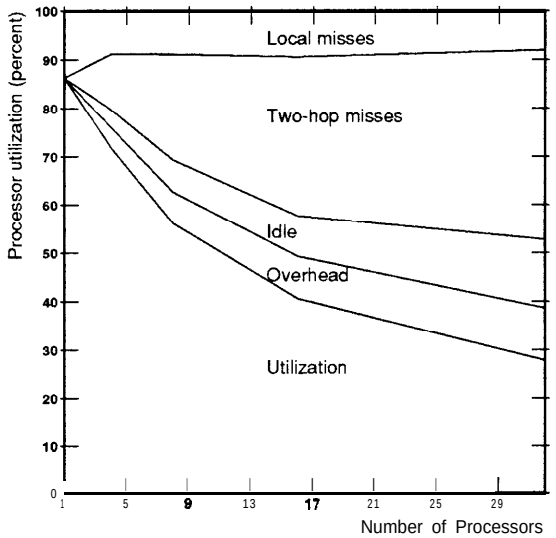


Figure 7: Processor utilization for ORNL scheme.

is a much smaller matrix than either BCSSTK15 or DENSE750 (LSHP3466 has 83,116 non-zeroes in its factor, while BCSSTK15 has 647,274 and DENSE750 has 280,875). As the number of processors is increased, the communication cost again increases, but for this matrix the cost of cache misses does not significantly decrease. The cache of a single processor is sufficiently large for this matrix that adding more cache provides little benefit. The net result is substantially lower speedups than for the larger matrices.

Thus we see that as the number of processors is increased, the increase in communication cost is traded against a decrease in the number of local cache misses due to a larger aggregate cache. For matrices where a larger cache provides significant cache miss benefits, the result is near-linear speedup. However, in the case where adding more cache does not result in substantially fewer cache misses, multiprocessor speedups are degraded by the cost of interprocessor communication.

6 Incorporating Supernodes into Parallel Sparse Factorization

One important assumption underlying the Oak Ridge parallel factorization scheme is that the natural task grain size for parallel factorization is cancellation by a single completed column. That is, they assume that since sequential codes perform the factorization by canceling by single columns, a parallel code which does the same will not necessarily introduce any substantial overheads. However, sequential factorization codes have recently been developed which realize substantial reductions in runtime by making use of the supernodal structure of a matrix [2, 15]. Rather than canceling by a single completed column, these codes cancel by a supemode, a potentially large set of columns. Thus, the natural task grain size for a parallel factorization code based on the more efficient supemodal elimination is cancellation by a set of columns, a much larger grain than that employed in the ORNL scheme. In this section, we modify the ORNL factorization scheme to make use of the supemodal structure of the factor, look at the issues which the increase in grain size bring up, and compare the resulting code to the original ORNL code.

In order to integrate supemodal elimination into the ORNL parallel factorization scheme, we make the following modification. Where in the ORNL scheme *columns were* distributed among the processors, we now distribute *supernodes.* In the ORNL scheme, once a column k has been completed, it is distributed to all processors which contain columns which are modified by it. Analogously, in our supemodal scheme, once an entire supemode has been completed, it is distributed to all processors which contain columns which are modified

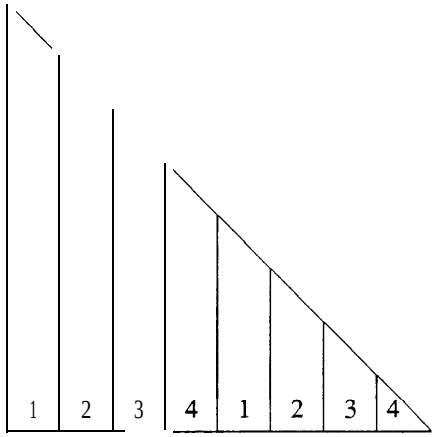


Figure 8: Vertical splitting of matrix DENSE750.

by it. Note that since all of the columns within a supemode modify the same set of columns, distributing the entire supemode never results in data being sent to a processor which does not need it.

The most obvious difficulty with such a modification is the load balancing problem which results from moving from a small grain workload division, namely by single columns, to a much larger unit, the supemode. In the most extreme case, the matrix DENSE750, the entire matrix is a single supemode. Consequently, if supemodes were treated as indivisible tasks, then this matrix would provide no opportunity for concurrency. Of course, sparse matrices contain many more than a single supemode and thus the reduction in available concurrency would not be quite so severe. The reduction is still substantial, however. Thus, in order to make use of a number of processors, the work of a supemode must be divided.

In splitting the work of a supemode, three important factors come into play. The first factor, which we have already discussed, is that if the work of a supemode is not split into sufficiently many pieces, then processors will go idle. Another important and competing factor, however, is due to the fact that the entire benefit of making use of supemodes comes from the consolidation of work which they allow. In splitting the work of supemodes, we must therefore attempt to retain to as great an extent as possible the benefits which supemodes provide. The third factor, common to many parallel processing problems, is the cost of communication relative to the cost of computation. That is, we must make sure that the benefit of having work done on another processor does not exceed the cost of communicating the data needed to do that work. Our goal in splitting the work of supemodes is thus to achieve the highest performance by striking a balance between processor utilization, the overhead of parallelization, and the cost of communication.

Because of these factors, we see that the question of how the work of a supemode should be divided is really two separate questions. The first question is, how can the work within a particular supemode be divided in order that some given number of processors can work on it at the same time? The second question is, how many processors will be available when a particular supemode is ready to be processed? Since there will be an overhead associated with splitting a supemode, we wish to avoid unnecessary splitting.

We first deal with the question of how to allow a number of processors to work within a single supemode. We have chosen to split the work contained within an individual supemode by doing a *vertical splitting* of the supemode. By *vertical*, we mean that the work of supemode is divided by drawing vertical lines through the supemode, and allowing each resulting section to be distributed to a different processor. In other words, our vertical splitting corresponds to dividing the original supemode into a set of smaller supernodes, each a subset of the original supemode. For example, the dense matrix of Figure 8, which would ordinarily be treated as a single supemode, is pictured as having been split into 8 smaller supemodes. In the limit, our vertical splitting scheme splits supemodes into vertical sections of size one, or single columns, and would thus be no different from the nodal factorization method.

In order to determine the extent to which a particular supemode should be split, we must arrive at an estimate of how many processors will be available when that supemode is to be processed. For example, we wish to avoid splitting a supemode into only two parts if four processors will be available. On the other hand, we wish to avoid splitting the work of a supemode into four parts if only two processors will be working on it at once.

In order to motivate our splitting strategy, we return to the example matrix of Figure 1. Consider supemode { 10, 11) of this matrix. The elimination tree of Figure 2 shows us that the two columns in this supemode are dependent on all other columns of the matrix, and similarly that column 11 is dependent on column 10. Thus neither of these columns can be completed until all other columns in the matrix are complete. This means that when this supemode is ready to be completed, no other factorization work is available to be distributed, and thus any processors which are not working on this supemode will be sitting idle. In general, all processors performing the factorization will be available to work on the supemode at the root of the elimination tree. Thus, we wish to divide this supemode so that all of these processors can work on it simultaneously.

The issue of how many processors will be available to work on a particular supemode becomes more difficult as we move down the elimination tree. The answer becomes dependent on the schedule chosen for the completion of the columns of the matrix. Consider, for example, the supemodes (4, 5, 6) and {7, 8, 9). If we wish to factor this matrix using two processors, then it is not clear how many processors will work on each of these supemodes. One processor might work on columns 1, 2, and 3, while the other works on the entire supemode (7, 8, 9}. Then supemode (4, 5, 6} would be worked on by both processors simultaneously. On the other hand, one processor could work on columns 1 and 2, while the other works on column 3, and then once these are complete, one processor could work alone on supemode (4, 5, 6) while the other works on (7, 8, 9}. Even for a matrix with only 11 columns, it is not clear how best to distribute columns to processors or how many processors will be available to work on a given supemode. Our benchmark matrices contain thousands of columns, which make studying the effects of small-scale scheduling decisions impossible.

Given the fact that the space of all possible schedules can not be exhaustively searched, we must use heuristics to determine how to split supemodes in order to allow for concurrent work while at the same time not introducing excessive overheads or communication costs. We have tried a number of different heuristic methods for determining the amount of splitting necessary, The most effective of these methods all had one property in common. In order to balance the overhead introduced when a supemode is split against the idle time which may result from having too few pieces, these methods split the supemodes near the root of the elimination tree into many small pieces, and decreased the amount of splitting as they moved further from the root.

In order to intuitively explain why the amount of splitting should decrease as we traverse further down into the elimination tree, we must consider two facts associated with the elimination tree and the factorization in general. The first is that, as we move down the elimination tree, the tree branches out into a number of subtrees. As was discussed earlier, each of these subtrees represents a piece of work which is independent of the work in all other subtrees. As the number of independent subtrees increases, we would expect that the number of processors working within a subtree will decrease, thus requiring less division of supemodes. Thus, it is not surprising that as the tree branches out, we can decrease the amount of supemode splitting.

The second and somewhat more surprising fact is that the splitting can be decreased even as we move down a single supemode. This is due to the nature of the parallelism which a vertical splitting of supemodes makes possible. Since the columns within a supemode are dependent on all previous columns within the same supemode, later columns cannot be completed until all earlier ones have been completed. Thus the main source of concurrency within a supemode is the pipelining of the work of dependent columns. That is, the completion of columns is overlapped with the propagation of contributions from previously completed columns. Consider, for example, the factorization of matrix DENSE750 with 4 processors (see Figure 8). Each piece of the matrix is assigned to the processor whose number is indicated. The factorization begins with processor 1 completing piece 1. This is done by first computing all contributions of columns within the piece to other columns within the piece, and then scaling the columns. Once this is done, then the contributions of piece 1 are propagated to all other pieces. Thus, for example, processor 4 will modify the columns of the pieces which it owns by the columns of the completed piece. Note that processor 2 does not immediately modify all of the pieces which it owns. It will modify piece 2 of the matrix, and then complete that piece, and only then will it modify piece 5. In this way, the completion of piece 2 on processor 2 will be overlapped with the propagation of the contributions of piece 1 on the other processors. Then the contributions of piece 2 to other pieces will be ready to be propagated by the time the other processors are finished with propagating the contributions of piece 1. This pipelining process continues until all pieces have been completed. Note that not much concurrency can be exploited during the completion of the final pieces of the matrix. For example, if piece 6 is in the process of being completed, then processor 1 has nothing left to do and must sit idle for the remainder of the factorization. Furthermore, as each subsequent piece is completed, one more processor runs out of work to do and becomes idle. This effect is typically known as pipeline draining.

The effect of the loss of concurrency due to pipeline draining can be kept small by making the pieces towards the end of the supemode small. As we move away from the end of the supemode, pipeline draining ceases to be an issue, and the main issue becomes the smooth overlap of the completion of one piece with the propagation of the contributions of previous pieces. Note that as we move further from the end, the amount of propagation work available to make up for the dependencies between pieces increases, and thus the size of the pieces can be increased. We observed that on the dense problem, performance can be improved by 10% by gradually increasing the size of the pieces instead of choosing a fixed size piece.

Due to the above-mentioned observations, we decided to divide supemodes based strictly on their depth in the elimination tree. Our splitting scheme can be described as follows. We being at the root of the elimination tree and traverse down the tree. We split the supemode at the root of the tree into pieces of four columns each as we traverse. Four column pieces retain many of the benefits of larger supemodes while at the same time not being so large that they would severely limit concurrency. We continue traversing and dividing until we have generated 3 x P pieces of size four, where P is the number of processors cooperating on the factorization. The number 3 was determined empirically. Once we reach this point, then we begin splitting supemodes into pieces of size eight. Similarly, once we have generated $3 \times P$ pieces of size eight, we then generate pieces of size twelve. When a branch in the elimination tree is reached, we continue down each branch, treating each as if it were still the only path. That is, if we were on our third piece of size eight and we reached a branch in the tree, then we would begin one side of branch with the fourth piece of size eight, and we would also begin the other side of the branch with the fourth piece of size eight.

We modify the above division scheme slightly in order to improve cache miss behavior. In [15], it was shown that the number of cache misses incurred in performing sparse factorization are substantially reduced if the non-zeroes in a supemode all fit in the processor cache. Thus, any piece resulting from the above splitting scheme which is larger than the cache is split into smaller pieces which do fit in the cache.

The above-described splitting strategy achieves our goal of gradually increasing the piece size within a supemode. It also achieves our goal of increasing the piece size as more and more subtrees are available. In our splitting scheme, supemodes are split into multiples of 4 columns because the loop unrolling in supemode cancellation is done in multiples of 4. In general, pieces of size less than 4 were not observed to improved performance.

The one remaining issue which must be resolved before the above supemode division scheme can be incorporated into a working parallel code is the issue of assigning the resulting pieces to specific processors. Recall that each column must be statically placed on a particular processor, and that processor will handle all work associated with that column. We have decided to take the same approach in our supernodal code which was taken in the ORNL scheme, a wrap-mapping of supemodes to processors. That is, we assign the nth supemode in the matrix to processor n mod P. In this way, each processor works on a number of threads of the factorization computation at once, thus reducing the likelihood that avoidable idle time will be incurred. A wrap-mapping also does a good job of balancing the computational load among the processors.

We have implemented a parallel supernodal factorization code which incorporates these design decisions. In the next section, we will look at the performance of this code, compare its performance with that of the ORNL code, and examine the issues which the supemodal code brings up.

7 Simulation Results for the Supernodal Scheme

In Figure 9 we give parallel speedups for our supemodal scheme, as compared with those of the ORNL scheme. In order to make the relative speeds of the two schemes more clear, Figure 10 shows the ratios for each of the three benchmark matrices of the runtime of the ORNL scheme to the runtime of the supemodal scheme when run on the same matrix. In looking at Figures 9 and 10, it is clear that the supemodal parallel scheme runs much faster than the ORNL scheme for all of the three test matrices. It executes between two and three times as fast when run on a single processor, and roughly 50% faster when run on thirty-two processors. This speed advantage can better be explained by comparing the graphs of Figure 11, where we give runtime breakdowns for our parallel supemodal scheme, to the corresponding graphs in Figures 6 and 7. We notice a number of substantial differences in the processor utilization patterns of these two factorization schemes.

The first and most important difference, which is not immediately obvious from the data presented so far,

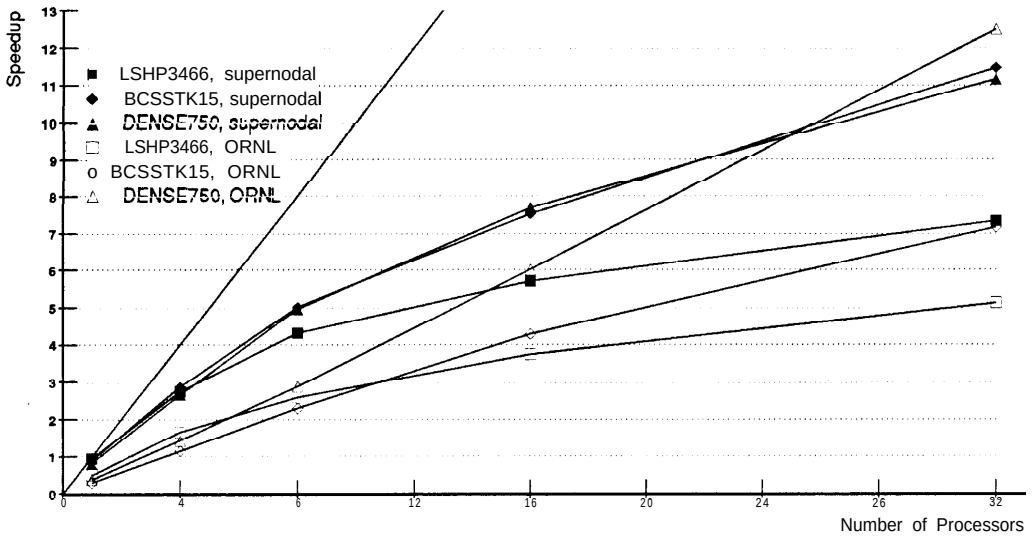


Figure 9: Speedups for ORNL and supemodal schemes, both relative to a highly efficient sequential code.

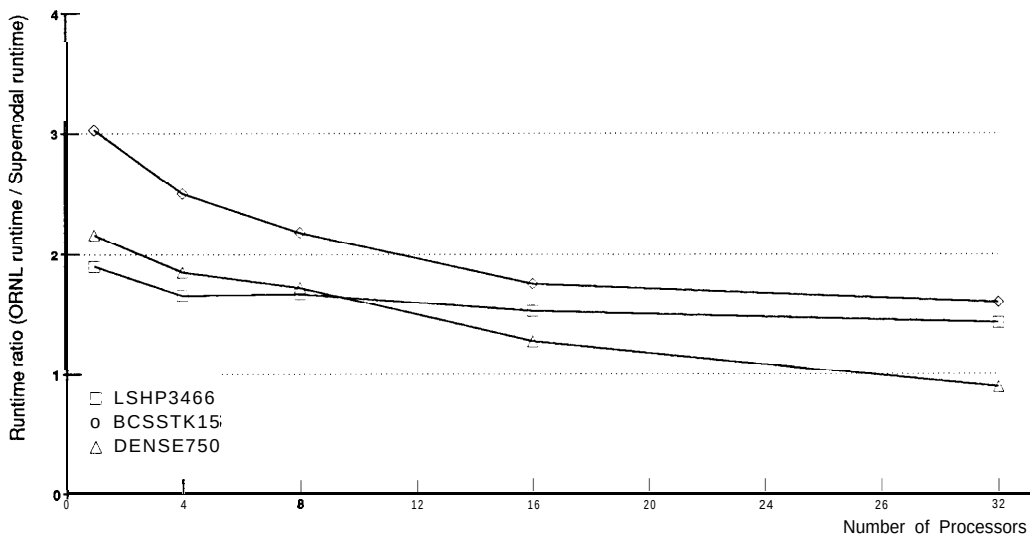
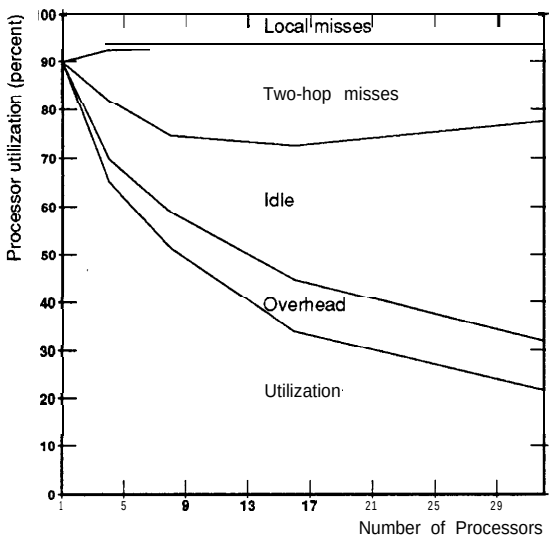
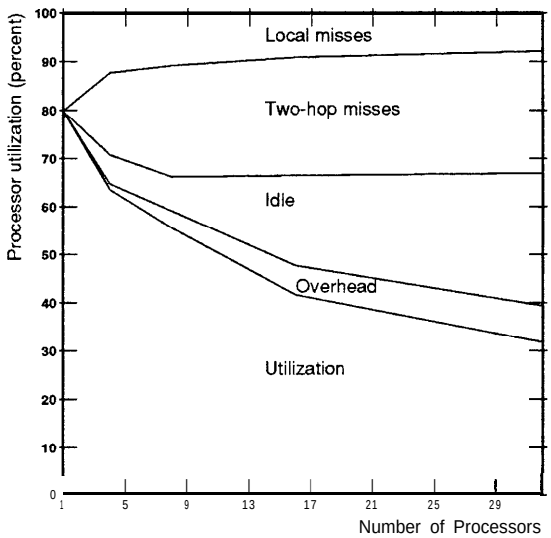


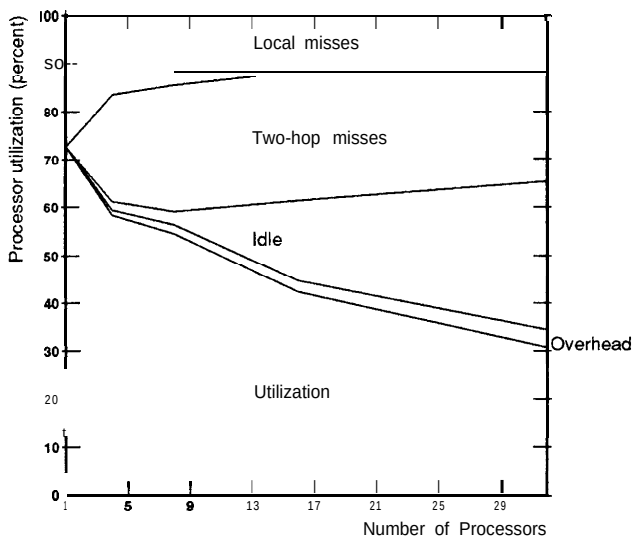
Figure 10: Ratio of ORNL runtime to supemodal runtime.



LSHP3466 - uniprocessor runtime 0.955s



BCSSTK15 - uniprocessor runtime 30.423s



DENSE750 - uniprocessor runtime 26.073s

Figure 11: Processor utilization for supemodal scheme.

is that the supemodal scheme executes roughly half as many instructions as the ORNL scheme in factoring the same sparse matrix. Thus we observe that, even though both schemes have nearly equal processor utilization when factoring matrix LSHP3466 on a single processor, the runtime for the supemodal scheme is roughly half that of the ORNL scheme. Similarly, the uniprocessor utilization figure for the supemodal scheme on matrix BCSSTK15 is less than 50% higher than the corresponding figure for the ORNL scheme, yet the runtime is almost 3 times smaller. This substantial reduction in the number of instructions executed is due to the loop unrolling and the decrease in index access which result from performing supemodal elimination, as was discussed earlier.

Another important observation is that for the larger matrices, the supemodal scheme spends a substantially smaller percentage of its time in servicing local cache misses when executed on a single processor. Furthermore, this is a smaller percentage of a smaller runtime, thus representing a much smaller absolute number of misses. This decrease is due to the cache miss reduction techniques which supemodes make possible.

We also observe that, in moving from a nodal parallel factorization scheme to a supemodal scheme, the bottleneck has shifted. In the ORNL scheme, the single largest component of runtime for the 32 processor runs was processor utilization. The rate at which the factorization proceeds was mainly limited by the speed at which the individual processors could execute instructions. In the supemodal scheme, processors spend roughly the same amount of time sitting idle as they spend executing useful instructions. The load balancing for the supemodal work distribution is much worse than that of the nodal work distribution. In the next section, we attempt to further improve factorization performance by decreasing the processor idle time which results from supemodal elimination.

8 Improving Load Balancing

In the previous section, we discovered that processor idle time is a substantial component of multiprocessor runtime for the supemodal factorization scheme. This idle time has two possible sources. It may be due to an insufficient quantity of available concurrency. That is, there might not be a sufficient amount of distributable work available to keep all of the processors busy. Alternatively, there may be a sufficient number of distributable tasks, but these tasks may be statically assigned to processors which are already busy. One processor might have a large number of tasks waiting to complete, while another processor sits idle. In this section we investigate the sources of processor idle time, and we try a number of modifications to the supemodal scheme in an attempt to improve parallel runtimes by reducing this idle time.

8.1 Varying the Supernode Division Strategy

In order to further examine the possibility that the idle time is due to a lack of distributable work, we now look at the effect of varying the number of pieces into which a supemode is divided. So far in our parallel code, we have divided a supemode based on its depth in the elimination tree. Whenever 3 x P pieces are available above a given supemode, then the size of the pieces into which the supemode is divided is increased. We say that the *supernode division rate* is 3. Figure 12 gives the runtimes, when using 16 processors, for a number of different choices of supemode division rates. The runtimes are expressed as a fraction of the runtime obtained when using a division rate of 3. In this figure, a larger division rate represents a finer division of supernodes. A very large rate, for example, would correspond to splitting all supemodes into four column pieces. At the other end of the spectrum, a division rate of 0 corresponds to almost no splitting of supemodes. Note that supemodes which are larger than the processor cache are still split. As can be seen from this figure, with the exception of the smallest division rates, the differences in runtimes between various splitting strategies are not extremely large. However, what is not clear from the figure is that the idle percentages are quite different for the various division rate. For example, when factoring matrix BCSSTK15 using a division rate of 1, 30% of the processor's time is spent idle. When using a division rate of 8, only 12% is spent idle. This is despite the fact that the runtimes are nearly identical In general, the decrease in idle time which comes with a finer division of supemodes does not translate into faster runtimes. Rather, this decrease is offset by the increase in overhead and communication which results when supemodes are divided into smaller pieces.

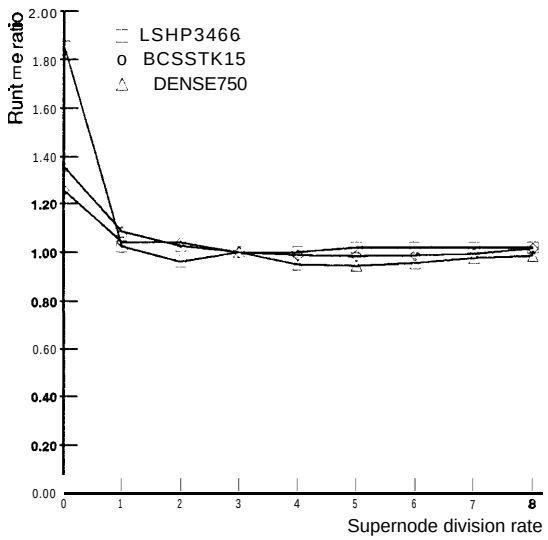


Figure 12: Runtime ratios for various supemode division rates, 16 processors.

8.2 Dynamic Supernodal Approach

We now look into the possibility that the idle time is the result of distributable tasks which are simply not distributed to idle processors. Since our simulated machine implements shared memory, we can easily modify our supemodal factorization code so that task assignment is more dynamic. That is, we can assign tasks to processors as they become available, as compared to deciding on a static assignment before the factorization begins. We now investigate whether, by using a more dynamic task assignment strategy, the idle time can be substantially reduced.

Our dynamic parallel factorization scheme differs only slightly from the static scheme. In the dynamic scheme, each processor has access to the task queues of all other processors as well as its own task queue. If a processor is ready to process a new task and has no tasks currently waiting on its queue, then it will look for an unprocessed task on the task queue of some other processor. Note that in this dynamic supemodal factorization scheme, a task obtained from the task queue of some other processor will require the modification of columns which the performing processor does not own. This increases the amount of interprocessor communication. Also, since it is now possible that a number of processors might wish to modify the same column at the same time, the dynamic scheme requires that a lock be associated with each column of the matrix. Before a processor may modify a column, it must first obtain the lock which corresponds to that column.

Our dynamic factorization scheme clearly has the advantage that if some task exists, then it will be available to any processor which is currently able to perform it. The main disadvantage of this scheme is that it generates more inter-processor communication than the static scheme. In Figure 13, we show the ratios of the runtimes for our dynamic supemodal scheme, as described above, to those of our static scheme for various numbers of processors. We also give runtime breakdowns for our dynamic supemodal scheme in Figure 14. We note from Figure 13 that the dynamic scheme is not substantially faster than the static scheme; in the best case, it is only 10 percent faster. This fact can be better understood by comparing the runtime breakdowns of the dynamic scheme, shown in Figure 14, to those of the static scheme, shown previously in Figure 11. From these figures, we see that the dynamic scheme decreases idle time somewhat. This decrease is moderate for matrix DENSE750, while it is much larger for matrices LSHP3466 and BCSSTK15. Idle time for a 32 processor run of the static scheme on matrix BCSSTK15 accounts for almost 30% of the overall runtime, while idle time accounts for less than 15% in the dynamic scheme. However, this decrease in idle time is offset by an increase in communication costs. The percentage of time spent in passing information between processors, represented in the dynamic case

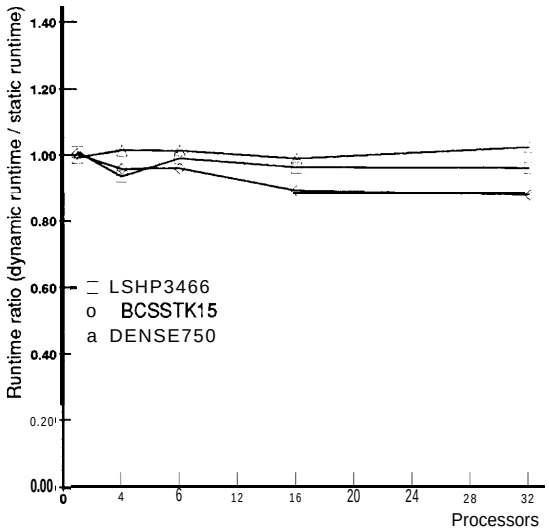


Figure 13: Runtime ratios for dynamic supemodal factorization scheme.

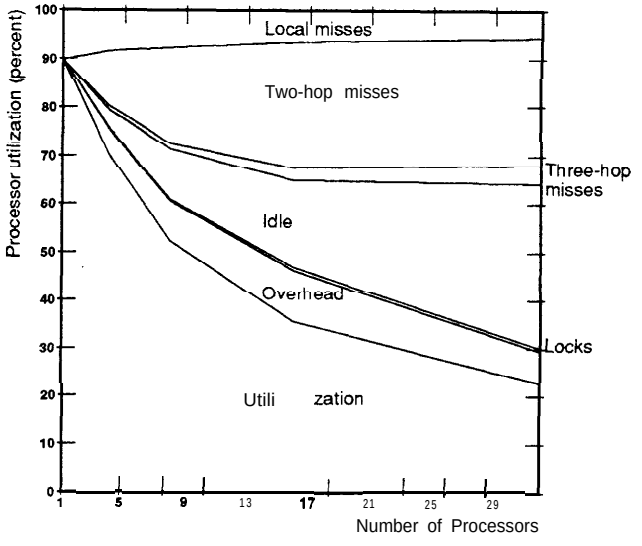
as 2-hop and 3-hop cache misses, has increased substantially. Looking again at the 32 processor executions on matrix BCSSTK15, we see that the static scheme spends around 25% of its runtime on cache misses, while the dynamic scheme spends around 33%. If we compare overall processor utilizations for the dynamic and static schemes, we see that they are nearly identical for all three matrices, for any number of processors.

To summarize, in this section we have looked at two possible sources of the idle time component of a parallel supernodal factorization code. We tried modifications of the code whose goal was to reduce the idle time resulting from each possible source. In both cases, we found that idle times were reduced by these modifications. However, in both cases the modifications had little effect on overall factorization runtime. By varying the supemode division rate, we were able to trade communication costs against idle time, but for the most part it was an even trade. Similarly, in going from a static task assignment scheme to a dynamic one, and also moving from a local-memory machine model to a shared-memory model, we were again able to trade communication costs against idle time, and again the result was an even trade. While both of these schemes changed the ratios of time spent in performing communication to time spent sitting idle, neither succeeded in decreasing the overall parallel runtime significantly.

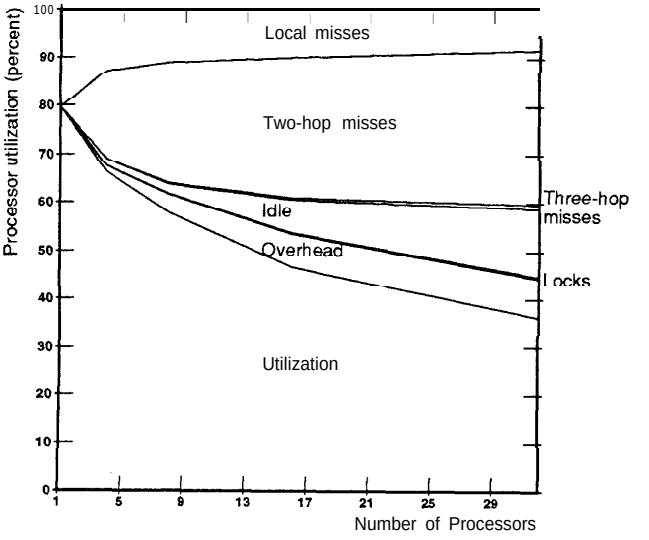
9 Parallel Factorization Behavior Over Time

Up until this point in the paper, all data which has been presented has been summary data. That is, we have given numbers such as multiprocessor speedup and processor utilization, all of which present only a summary of how the computation behaves over time. In this section, we will study in detail how the computation evolves over time. We will look at a number of questions concerning the dynamic nature of the computation. For example, in the previous section we showed that processor idle time is a large problem for our supemodal approach to parallel factorization. An interesting question is, is this idle time distributed uniformly throughout the course of the computation, or are there particular phases when idle time is particularly large? Similarly, how does interprocessor communication evolve? Does interprocessor communication take place at a relatively constant volume throughout the computation, or are there bursts of communication interspersed with periods of relative inactivity in the network?

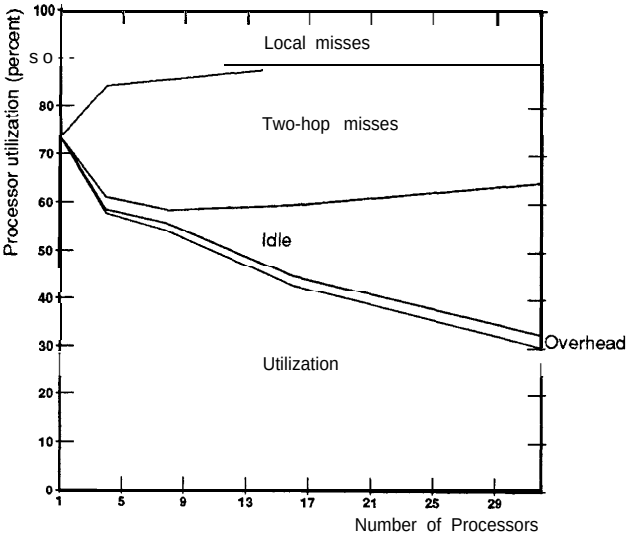
In Figures 15 through 17, we present graphs of the instantaneous computation rates, cache miss rates, and idle percentages, when factoring each of our benchmark matrices with 16 processors. The horizontal lines in



LSHP3466 - uniprocessor runtime 0.965s



BCSSTK15 - uniprocessor runtime 30.51 0s



DENSE750 - uniprocessor runtime 25.681s

Figure 14: Processor utilization for dynamic supemodal scheme.

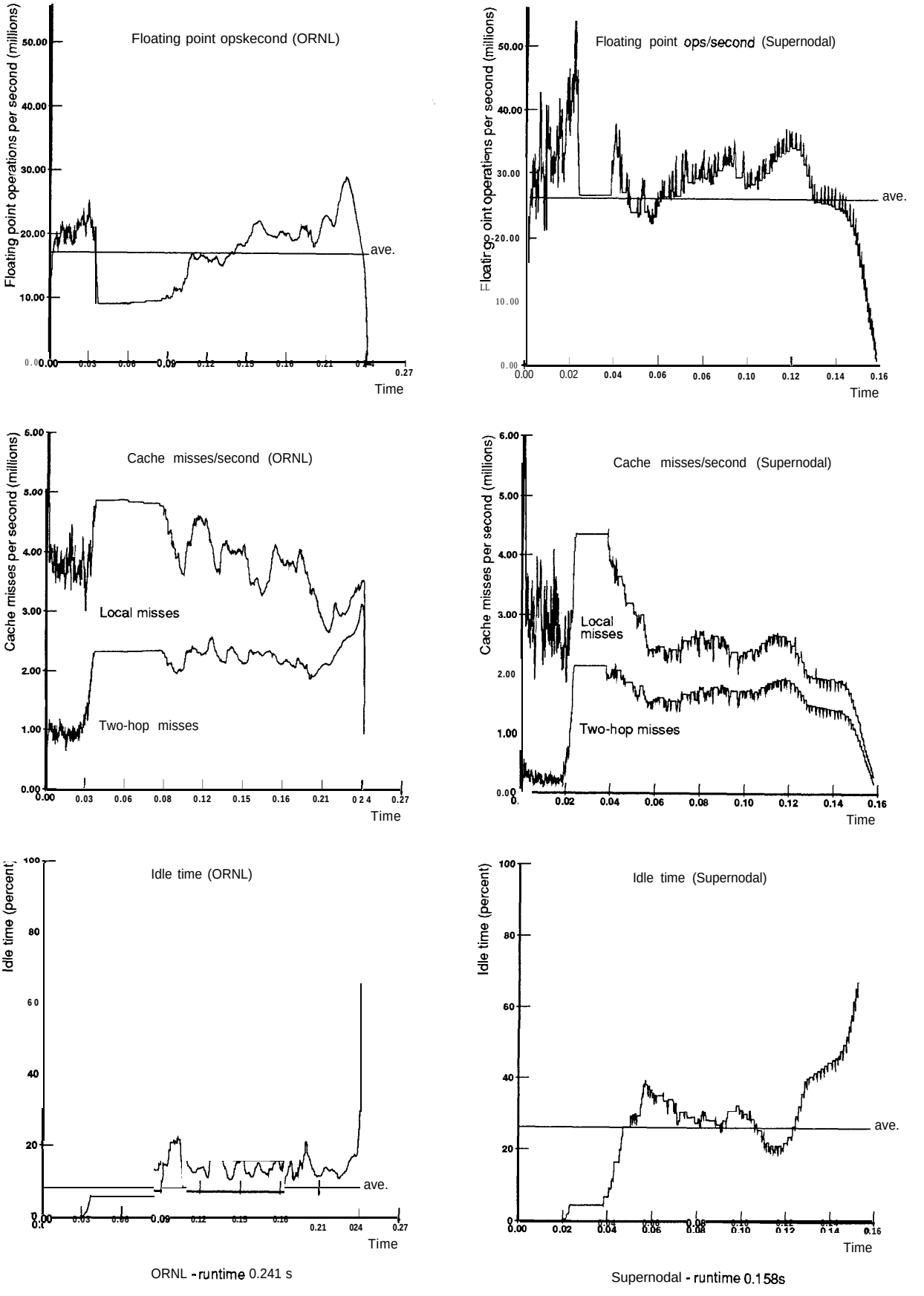


Figure 15: Dynamic behavior for matrix LSHP3466, 16 processors.

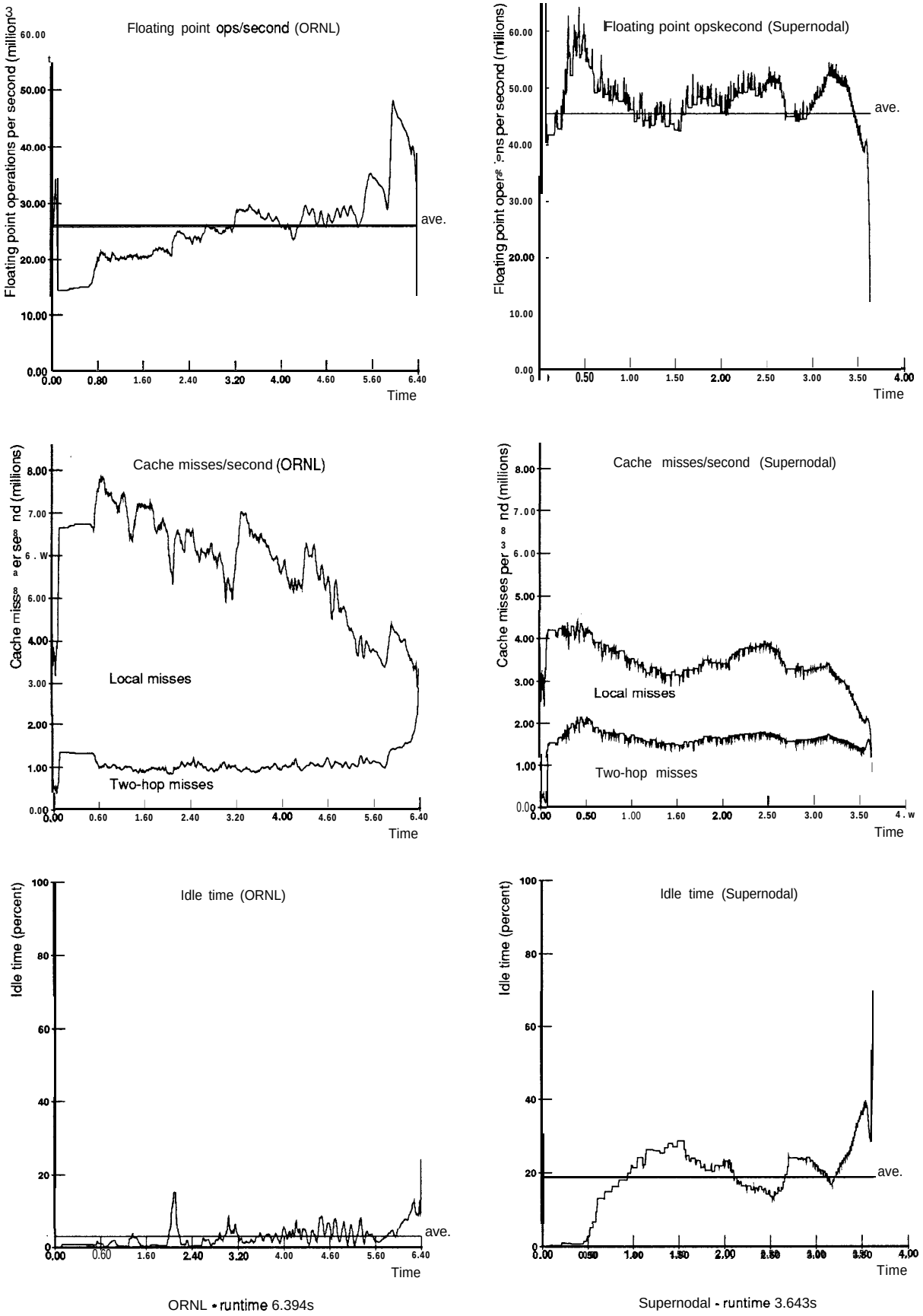


Figure 16: Dynamic behavior for matrix BCSSTK15, 16 processors.

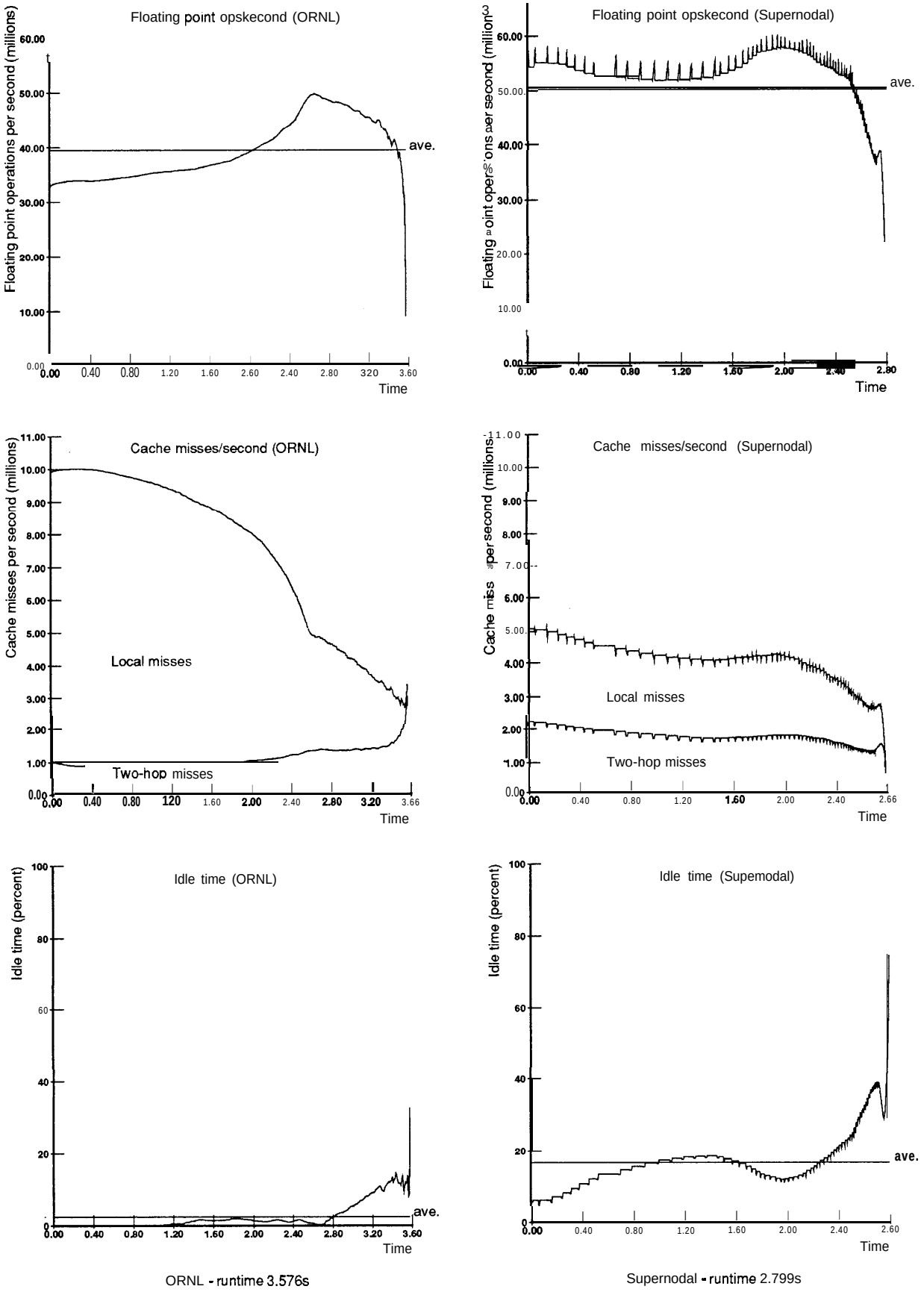


Figure 17: Dynamic behavior for matrix DENSE750, 16 processors.

the graphs indicate average values over the entire computation. The cache miss graphs contain two curves. The lower of the two curves shows the number of two-hop cache misses, while the upper shows all cache misses. Thus, the region between the two curves gives the number of local misses, The graphs on the left of a page give data about the the ORNL factorization scheme, while the graphs on the right give data about the static supemodal scheme.

In general, we observe that the factorization proceeds in two stages. The first stage consists of the processing of independent subtasks, as was discussed previously in the section on the assignment of columns to processors. As can be seen from the figures, this phase accounts for approximately 15% of the overall runtime for matrix LSHP3466, a very small percentage for matrix BCSSTK15, and this phase is naturally absent for matrix DENSE750. The computation during this first phase is characterized by very little interprocessor communication, no processor idle time, and consequently high computation rates.

Once the independent subtasks are completed, then the computation moves into the second phase, where the separator columns are computed. Recall that the work of the separators is distributed among the processors, as opposed to the independent subtasks which were processed entirely by a single processor. The beginning of this phase can be recognized by a large jump in the amount of interprocessor communication, an increase in idle time, and a drop in the computation rate.

As the second phase proceeds, the computation rate for the ORNL scheme steadily increases, due to a steady reduction in the number of local cache misses incurred. The reduction in local cache misses is to be expected, since as the computation proceeds, more and more columns are completed. Once complete, these columns no longer need to be accessed, thus the size of the active data set shrinks. Notice that the idle percentage remains low throughout the computation. That is, of course, with the exception of the very end of the computation, where idle time increase dramatically, and the computation rate falls commensurately. This increase in idle time occurs in the final columns of the matrix, where there are no longer sufficiently many columns which have not yet been completed to keep 16 processors busy. Note that the computation rates increase substantially near the end of the computation for the two sparse problems. This increase coincides with the completion of the final supemode of the matrix. Since this final supemode forms a dense lower-triangular matrix, the overheads of performing sparse cancellations can be avoided when this supemode is processed, thus allowing higher computational rates.

The second phase exhibits substantially different characteristics for the supemodal factorization scheme. As the computation proceeds, and the computation moves towards the root of the elimination tree, the idle percentage increases. This can be understood by noting that, as we move towards the top of the elimination tree, there is less and less concurrency available *between* supemodes, and thus we must make use of concurrency *within the* supemodes. As was discussed earlier, this concurrency is more difficult to exploit. As the factorization proceeds, communication rates, local cache miss rates, and computation rates depend mainly on the idle percentage. In periods where idle time is high, computation and communication rates are low, and vice-versa. Clearly, if processors are sitting idle, they will not be generating cache misses or performing floating point operations.

In the discussion of our parallel machine model, we claimed that contention for the interconnection network would not be a problem. We now have the data necessary to support this claim. Notice that in all of the cache miss graphs presented, the largest sustained rate of interprocessor cache misses is approximately 2 million per second. The ORNL scheme experiences short periods of up to 3 million misses per second, but these bursts are brief enough so that even if they were not sustainable, the impact on overall runtime would be small. In computing the total network traffic which would result from these cache misses, we must consider a number of factors. First, clearly each cache line contains 16 bytes of information. We must also take into account the messages necessary to fetch and subsequently invalidate this cache line. A conservative estimate for a single miss would be one 8-byte message in order to request the contents of the cache line from its owner memory module, and potentially another 8-byte message to invalidate it. We therefore assume that each cache miss results in 32 bytes of information being sent on the network. Thus, the entire interconnect network of the machine must sustain approximately 64 MBytes per second of communication traffic. To put this figure in perspective, we note that each point-to-point interconnect in the DASH prototype will have a peak throughput of 60 MBytes per second in each direction. Since each individual point-to-point connection can almost sustain such volumes, it is unlikely that this communication load would be a problem for the network as a whole.

We must add one important caveat here regarding communication traffic. While it is true that the interconnection network most likely has sufficient bandwidth to sustain the communication load presented by Cholesky factorization, an important related issue, hot-spotting, has not been investigated. Our simulations do not provide

us with the data necessary to determine whether the remote memory references occurring at a particular time are references to locations in a number of different memory modules or to locations in just a few modules. If references are to few modules, then contention at those modules could be a problem. Requests for memory locations could be delayed, waiting for other requests to complete. Furthermore, the data bandwidth of individual nodes in the multiprocessor could become a constraint. For example, in the previous paragraph we noted that when using 16 processors, approximately 32 MBytes per second of data is typically being transported on the interconnection network (not including the overhead messages). A single node in the DASH prototype can deliver approximately 44 MBytes per second of data. Thus, when performing a factorization with 16 processors, a single node could sustain the load if all requested data were to reside in that node. However. contention could again delay cache miss servicing. Hot-spotting in the memory system is an important issue which will require further study.

10 Future Work

A number of important issues relating to parallel sparse factorization remain to be explored. First and foremost is how idle time can be reduced in supemodal parallel factorization. One area which requires further study is the distribution of work within a supemode. Our vertical splitting scheme provided some concurrency, but it should be possible to improve on it. A natural alternative, for example, is a *horizontal splitting* of supernodes. In a vertical splitting, the work of a supemode was divided into subsets, where each subset corresponded to the work associated with some set of columns from the supemode. In a horizontal splitting, we could divide the work of a supemode into subsets, where the subsets would correspond to the work associated with some set of *rows*. This modification would require a much more complicated distribution of non-zeroes to processors, but the resulting decrease in idle time might make up for the increased overhead.

Another area for further study is that of deciding on a static distribution of separator columns to processors. As was shown in our examination of a dynamic supemodal scheme, the method which we used for distributing separator columns to processors, where we simply did an arbitrary wrap mapping, can result in substantial load imbalances. In order to improve on this distribution, we could compute the amount of work associated with each column in the matrix, and then do a precomputation in which the factorization is simulated in order to determine which processors will be idle at any given point in time. The columns could then be assigned based on this information. While finding the *best* assignment of columns to processors will certainly not be possible, it might be possible to find a better assignment than the simple one which was employed here.

We would also like to examine the issue of read-only data replication. In each of the factorization schemes studied in this paper, we have kept one copy of all read-only data for each processor, so that this data can be retrieved by any processor without requiring communication. While this replication of data reduces interprocessor communication, it also greatly increases the storage requirements of an already extremely memory-intensive computation. Some or all of this data could be distributed among the processors. This distribution is made especially easy by the shared-memory programming model. We would like to study the tradeoffs between storage overhead and interprocessor communication volume.

Another important issue in parallel factorization is the impact of various multiprocessor hardware features on factorization performance. For example, the DASH architecture includes a mechanism for program-directed prefetch. With such a feature available, a factorization code could issue a non-blocking request for a set of non-zeroes before they are needed. The program could proceed with what it is working on while the prefetch is executed, and hopefully by the time the program needs the requested data, it has been fetched and is available. A prefetch feature could substantially reduce the cost of interprocessor communication, and would present a different set of tradeoffs for parallel factorization.

Another interesting aspect of how the multiprocessor hardware affects factorization performance is the issue of -memory system and processor speed. If we were to use processors which are twice as fast, with the same memory system, would the performance double? What would happen if we were to double the speed of the memory system? Or, if we were looking to reduce the cost of the machine, which components of the hardware could be replaced with cheaper alternatives without substantially degrading factorization performance? These are important issues in the design of a cost effective multiprocessor.

11 Conclusions

In this paper, we have studied the behavior of parallel sparse factorization schemes on a simulated multiprocessor. We have presented detailed simulation results for the parallel sparse factorization scheme developed at the Oak Ridge National Laboratory. These results included not only parallel runtimes, but also a detailed analysis of the magnitude of each of the factors which contribute to the overall runtime. We found that the Oak Ridge scheme gave good speedup, relative to a sequential nodal factorization code. The cost of communicating data between processors was offset by a decrease in the number of local cache misses which resulted from having more processors and thus more cache memory available. The result was a near-linear speedup for two of the three benchmark matrices.

We then extended the Oak Ridge factorization scheme to incorporate the notion of supernodal elimination. Supemodal elimination resulted in substantially fewer instructions being executed, and substantially fewer cache misses being generated In contrast to the nodal case, the supemodal parallel factorization code had less cache miss costs to trade against communication costs. Even though supemodal parallel speedups were modest, the resulting parallel runtimes were substantially faster than those for the Oak Ridge scheme. For single processor execution, the supemodal code was two to three times as fast as the nodal code. Though the advantages of supemodal factorization decreased as the number of processors was increased, the supemodal code was still 50% faster for two of the three benchmark matrices when using 32 processors. These runtime advantages were obtained despite the fact that the incorporation of supemodes created substantial load balancing problems, and that in the 32 processor executions the processors spent from 25 to 50% of their time sitting idle.

We then attempted to further improve parallel factorization performance by decreasing processor idle time. We tried moving to a more dynamic task allocation scheme. This scheme resulted in lower idle time, but also resulted in an increase in communication. The net result was little change in overall runtime. We also tried varying the task grain size. Again it was possible to decrease idle time, but the decrease always came with an increase overhead and communication costs, leaving the runtimes unchanged. The challenge of improving sparse parallel factorization performance thus appears to be in finding a more effective means of dividing the work of a supemode among a number of processors, while keeping the overheads involved in the distribution and the costs of communication low.

Acknowledgements

We would like to thank Helen Davis and Steve Goldschmidt for creating TANGO, and we would like to thank Steve Goldschmidt for all of the help he gave us with it. This research is supported by DARPA contract N00014-87-K-0828. Edward Rothberg is also supported by a grant from Digital Equipment Corporation.

References

[1] Ashcraft, C., Eisenstat, S., and Liu, J., "A fan-in algorithm for distributed sparse numerical factorization", Technical Report CS-89-03, 1989.

[2] Ashcraft, C., Grimes, R., Lewis, J., Peyton, B. and Simon, H., "Recent progress in sparse matrix methods for large linear systems", *International Journal of Supercomputer Applications*, 1(4):10 - 30, 1987.

[3] Davis, H., Goldschmidt, S., and Hennessy, J., "Tango: A multiprocessor simulation and tracing system", Technical Report, Stanford University, 1989, to appear.

[4] Duff, I., "Multiprocessing a sparse matrix code on the Alliant FX/8", *Journal of Computational and Applied Mathematics*, 27:229-239, *1989.*

[5] Duff, I., Grimes, R., and Lewis, J., "Sparse Matrix Test Problems", *ACM Transactions on Mathematical Software*, 15(1):1 - 14, 1989.

[6] Geist, G.A., and Ng, E., "A partitioning strategy for parallel sparse Cholesky factorization", Technical Report TM-10937, Oak Ridge National Laboratory, 1988.

[7] George, A., Heath. M., Liu, J., and Ng, E., "Solution of sparse positive definite systems on a hypercube", Technical Report TM-10865, Oak Ridge National Laboratory, 1988.

[8] George, A., and Liu, J., *Computer Solution of Large Sparse Positive Definite Systems,* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[9] George, A., Liu, J., and Ng, E., "Communication results for parallel sparse Cholesky factorization on a hypercube", *Parallel Computing,* 10: 287-298, 1989.

[10] Liu, J., "The role of elimination trees in sparse factorization", *SIAM Journal on Matrix Analysis and Applications,* 11(1):134-172, 1990.

[11] Liu, J., "Reordering sparse matrices for parallel elimination", Technical Report CS-87-01, Department of Computer Science, York University, 1987.

[12] Lucas, R., *Solving Planar Systems of Equations on Distributed-Memory Multiprocessors,* PhD thesis, Department of Electrical Engineering, Stanford University, 1987.

[13] Lusk, E., Overbeek, R., et *al, Portable Programs for Parallel Processors,* Holt, Rinehart, and Winston, Inc., 1987.

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

[15] Rothberg, E., and Gupta, A., "Fast sparse matrix factorization on modem workstations", Technical Report STAN-CS-89-1286, Stanford University, 1989.