

# Improving The Performance of Sparse Cholesky Factorization with Fine Grain Synchronization

by

Manish Kumar Tuteja

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degrees of

Master of Science

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

© Massachusetts Institute of Technology 1994. All rights reserved.

Author .....

Department of Electrical Engineering and Computer Science

May 12, 1994

Certified by .....

Anant Agarwal

Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by .....

Frederic R. Morgenthaler

Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUL 13 1994

# **Improving The Performance of Sparse Cholesky Factorization with Fine Grain Synchronization**

by

**Manish Kumar Tuteja**

Submitted to the Department of Electrical Engineering and Computer Science  
on May 12, 1994, in partial fulfillment of the  
requirements for the degrees of  
Master of Science  
and  
Bachelor of Science in Computer Science and Engineering

## **Abstract**

Performance of fan-out cholesky factorization on multiprocessors can be improved by introducing fine grain synchronization into the application. Fine grain synchronization exposes greater parallelism over the coarse grain synchronization used in standard fan-out implementations. With finer synchronization, it is also possible to redistribute cholesky operations across processors for even more parallelism.

On an ideal machine, the exposed parallelism can substantially reduce execution time of fan-out cholesky factorization. On a practical parallel machine, the execution time can be more than halved over coarse fan-out factorization. This conclusion was reached by executing coarse grain and fine grain versions of fan-out code using a high level parallel factorization simulator. The simulator models the Alewife Machine, a research multiprocessor with hardware support for fast word level synchronization. The machine is under development in the Laboratory for Computer Science at the Massachusetts Institute of Technology.

Thesis Supervisor: Anant Agarwal

Title: Associate Professor of Electrical Engineering and Computer Science

## **Acknowledgments**

With great personal hardship, my mother, Sharda Tuteja, and my father, Wazir C. Tuteja, have provided many opportunities to my sister, brother and me. For these, I am eternally grateful. I hope and pray that I will be able to repay them in kind. To them, I dedicate this thesis and my forthcoming degrees.

Professor Anant Agarwal has helped me through MIT as a mentor, advisor, and role model. His enthusiasm and energy for research have been inspiring. I thank Professor Agarwal for his exceptional guidance over the past two and a half years with this thesis. I also sincerely thank every other member of the Alewife research group.

During the past five years, my friends, old and new, at New House 4 have provided a comfortable retreat from the firehose of MIT. I am glad that I have met so many wonderful people who have enriched my life in all ways imaginable.

# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Studying Parallel Cholesky on a Large Multiprocessor . . . . .	10
1.2	Fine Grain Synchronization in Cholesky Factorization . . . . .	11
1.3	Organization of Thesis . . . . .	12
<b>2</b>	<b>Cholesky Factorization</b>	<b>13</b>
2.1	Math of Cholesky Factorization . . . . .	13
2.2	Sparse Cholesky Factorization . . . . .	14
2.3	Sparse Parallel Cholesky Factorization . . . . .	15
2.3.1	Where is the Parallelism? . . . . .	15
2.3.2	Fan-out Factorization . . . . .	16
2.3.3	Determining Tasks . . . . .	17
2.3.4	Multiprocessor Operation . . . . .	18
2.4	Mod Mapped Column Distribution . . . . .	19
<b>3</b>	<b>Fine Grain Cholesky Factorization</b>	<b>20</b>
3.1	Fine Grain Fan-Out . . . . .	20
3.1.1	Fine Grain w/Task Redistribution . . . . .	22
<b>4</b>	<b>CSIM: An Event Driven Parallel Factorization Simulator</b>	<b>25</b>
4.1	CSIM Internals . . . . .	26
4.1.1	Processor Simulation . . . . .	26
4.1.2	Modeling Coarse Grain Synchronization . . . . .	28
4.1.3	Modeling Fine Grain Synchronization . . . . .	29

4.1.4	Cache Simulation . . . . .	31
4.1.5	Local Memory . . . . .	32
4.1.6	Shared Memory/Network Model . . . . .	32
4.2	Knobs and Switches . . . . .	33
4.3	Equipment Required . . . . .	33
<b>5</b>	<b>Results and Analysis</b>	<b>34</b>
5.1	Ideal Machine Results . . . . .	35
5.1.1	Ideal:Coarse–Single Processor Run Times . . . . .	35
5.1.2	Ideal:Coarse–Speedup Curves . . . . .	36
5.1.3	Ideal:Fine Grain no Task Redistribution Speedup . . . . .	37
5.1.4	Ideal:Fine Grain w/Task Redistribution Speedup . . . . .	38
5.1.5	Ideal:Coarse–Utilization . . . . .	39
5.1.6	Ideal:Fine Grain w/Task Redistribution–Utilization Curves . . . . .	43
5.2	Practical Machine Results . . . . .	46
5.2.1	Coarse–Single Processor Run Times . . . . .	46
5.2.2	Coarse–Speedup Curves . . . . .	46
5.2.3	Comparison of <i>CSIM</i> 's output to Rothberg . . . . .	48
5.2.4	Fine Grain no Task Redistribution Speedup . . . . .	48
5.2.5	Fine Grain w/Task Redistribution Speedup . . . . .	49
5.2.6	Coarse–Utilization Curves . . . . .	51
5.2.7	Fine Grain w/Task Redistribution–Utilization Curves . . . . .	55
<b>6</b>	<b>Conclusions</b>	<b>59</b>

# List of Figures

2-1	<i>Sequential Algorithm for Cholesky Factorization</i> . . . . .	14
2-2	<i>A Sparse Cholesky Factorization Algorithm</i> . . . . .	15
2-3	<i>A Sample Matrix and its Factor</i> . . . . .	16
2-4	<i>Elimination tree of Sample matrix</i> . . . . .	16
2-5	<i>Fan-Out Algorithm for Cholesky Factorization</i> . . . . .	17
2-6	<i>Work Available</i> . . . . .	18
2-7	<i>4 Processor Operation</i> . . . . .	19
3-1	<i>4 Processor Operation—Arrows mark Data Dependencies</i> . . . . .	21
3-2	<i>4 Processor Operation With Fine Grain Synchronization</i> . . . . .	21
3-3	<i>4 Processor Operation With Fine Grain Synchronization and Task Redistribution</i> . . . . .	23
4-1	<i>Polling Overhead Model for Coarse Cholesky</i> . . . . .	28
5-1	<i>Ideal:Coarse Grain Speedup</i> . . . . .	36
5-2	<i>Ideal:Fine Grain—NO Task Redistribution Speedup</i> . . . . .	38
5-3	<i>Ideal: Fine Grain w/Task Redistribution Speedup</i> . . . . .	39
5-4	<i>Ideal:bcsstk14 Coarse Grain Utilization Curve</i> . . . . .	40
5-5	<i>Ideal:bcsstk15 Coarse Grain Utilization Curve</i> . . . . .	41
5-6	<i>Ideal:d750.0 Coarse Grain Utilization Curve</i> . . . . .	41
5-7	<i>Ideal:lshp Coarse Grain Utilization Curve</i> . . . . .	42
5-8	<i>Ideal:w15.1 Coarse Grain Utilization Curve</i> . . . . .	42
5-9	<i>Ideal:bcsstk14 Fine Grain w/Task Redistribution Utilization Curve</i> . . . . .	43

5-10	<i>Ideal:bcsstk15 Fine Grain w/Task Redistribution Utilization Curve . . . . .</i>	44
5-11	<i>Ideal:d750.O Fine Grain w/Task Redistribution Utilization Curve . . . . .</i>	44
5-12	<i>Ideal:lshp Fine Grain w/Task Redistribution Utilization Curve . . . . .</i>	45
5-13	<i>Ideal:w15.1 Fine Grain w/Task Redistribution Utilization Curve . . . . .</i>	45
5-14	<i>Coarse Grain Speedup . . . . .</i>	47
5-15	<i>Fine Grain–NO Task Redistribution Speedup . . . . .</i>	49
5-16	<i>Fine Grain w/Task Redistribution Speedup . . . . .</i>	50
5-17	<i>Fine Grain w/Task Redistribution Speedup . . . . .</i>	51
5-18	<i>bcsstk14 Coarse Grain Utilization Curve . . . . .</i>	52
5-19	<i>bcsstk15 Coarse Grain Utilization Curve . . . . .</i>	52
5-20	<i>d750.O Coarse Grain Utilization Curve . . . . .</i>	53
5-21	<i>lshp Coarse Grain Utilization Curve . . . . .</i>	53
5-22	<i>w15.1 Coarse Grain Utilization Curve . . . . .</i>	54
5-23	<i>bcsstk14 Fine Grain w/Task Redistribution Utilization Curve . . . . .</i>	56
5-24	<i>bcsstk15 Fine Grain w/Task Redistribution Utilization Curve . . . . .</i>	56
5-25	<i>d750.O Fine Grain w/Task Redistribution Utilization Curve . . . . .</i>	57
5-26	<i>lshp Fine Grain w/Task Redistribution Utilization Curve . . . . .</i>	57
5-27	<i>w15.1 Fine Grain w/Task Redistribution Utilization Curve . . . . .</i>	58

# List of Tables

4.1	<i>Statistics Gathered</i> . . . . .	26
4.2	<i>Simulated cdiv operation</i> . . . . .	27
4.3	<i>Simulated cmod operation</i> . . . . .	27
4.4	<i>CSIM controls</i> . . . . .	33
5.1	<i>Test Matrices</i> . . . . .	34
5.2	<i>Ideal:Coarse–Single Processor Run Times</i> . . . . .	35
5.3	<i>Coarse–Single Processor Run Times</i> . . . . .	46
5.4	<i>Comparsion of CSIM with Rothberg’s results</i> . . . . .	48



# Chapter 1

## Introduction

The cholesky factorization of a large sparse positive definite matrix is a useful value in scientific computations. Execution time can be substantially reduced by computing the factorization in parallel. There are a number of parallel algorithms for performing the computation. Only the fan-out algorithm will be studied in this thesis. As with any parallel computation, establishing the optimal choice of synchronization, work distribution and data distribution is a non-trivial issue.

Rothberg and Gupta of Stanford University have done a detailed study of parallel cholesky factorization[10, 9]. They experimented with algorithmic changes to increase memory reference locality. Other researchers include Gilbert[5, 6], Kumar[8], Venugopal[12] and Zhang[14]. Of these studies, only Gilbert[6] has attempted to use more than 64 processors to perform the factorization. He found a moderate speedup in factoring his test matrices on a data parallel Connection Machine 2.

What kind of speedup would there be on a large MIMD machine? Is the synchronization method used in fan-out cholesky factorization adequate for large machines? How does the performance change as machine parameters such as network latency, and memory system overhead change? This thesis will present an analysis of fan-out factorization on a large MIMD machine with data from a simulator for parallel cholesky factorization.

## 1.1 Studying Parallel Cholesky on a Large Multiprocessor

Ideally, all experiments would be carried out using an easily configurable 1024 node multiprocessor. Since no such machine is available, the best solution is a machine simulator. The Alewife research group in the Laboratory for Computer Science at the Massachusetts Institute of Technology has an advanced simulation environment for the Alewife multiprocessor[1, 3]. Some very large matrices with factorization times on the order of a billion cycles were used for this study. This number of cycles would take an excessive amount of time to simulate under a detailed simulation environment like Alewife's. The large simulation time effectively disallows any study with a range of system parameters. Though NWO, the Alewife simulator, would have provided highly accurate statistics, the simulation time was unacceptable.

The Proteus simulation system was also investigated as a possible platform for study[2]. Proteus adds parallel functionality and statistics gathering to an application without the overhead of detailed simulation. Even though Proteus applications can run much faster than NWO, the simulation time is still high. As with NWO, studying large matrices under a variety of system configurations would have taken an unacceptable amount of time.

To decrease simulation time, this study relies on an event-based parallel factorization simulator called *CSIM*. A similar simulator was developed by Rothberg for studying large factorizations[10]. *CSIM* outperforms both Proteus and NWO. The key to the speed of *CSIM* is that it does not perform the floating point operations of the factorization. Instead it estimates the time a processor would take to perform the operation with attention to cache effects, local and distributed memories, and synchronization. The processor's cycle time is increased by this estimate. *CSIM* can handle very large matrices and machine size up to 2048 processors. It is highly configurable and produces detailed statistics on processor utilization, cache, local memory, and distributed memory performance. The simulator also supports a fine grain synchronization mode with statistics.

Since *CSIM* is not a real cycle-by-cycle simulator of a multiprocessor, results from the simulation cannot be highly accurate. This does not mean that they are unreliable. *CSIM*

results for a coarse-grain factorization are very close to results obtained by Rothberg from a real machine and a simulator. This comparison of results along with the assumptions made in *CSIM* and their resulting effects are discussed in greater detail later.

## 1.2 Fine Grain Synchronization in Cholesky Factorization

The fan-out algorithm for cholesky factorization limits parallelism to the column level by synchronizing in a coarse way. Entire columns are locked, posing an unnecessarily strict data dependency. Due to the nature of the computation, the locking processor does not need a lock on an element after it has written to it. But the processor cannot release the element; it must wait until it is done with the entire column. Thus, any processor waiting to perform updates to the column must wait for the first processor to finish computing.

The solution is to have processors synchronize in a fine way by locking individual column elements instead of whole columns. The effect of using fine grain synchronization should be increased parallelism. Synchronizing on a memory element incurs an overhead on each memory read and write. Also, if a synchronization fails, the memory system must implement some mechanism to find out when the synchronization succeeds.

*CSIM* implements a simple fine grain synchronization model. Successful synchronization operations incur a small overhead for determining when they were successful. A backoff strategy is used for failed synchronizations. The fan-out algorithm was modified to synchronize on matrix elements. Due to the statically scheduled order of the factorization, this simple model is sufficient. The assumptions of the model are presented later.

It turns out that the distribution of work to processors with the fan-out algorithm does not leave much optimization for fine grain synchronization to perform. The resulting increase in parallelism is slight. With the finer synchronization, however, the work can be distributed in many more ways. A simple change in the distribution algorithm led to much better performance of the fine grain cholesky application. The new distribution of tasks is described in chapter 3.

## 1.3 Organization of Thesis

This thesis is organized into the following sections.

**Ch. 2: Cholesky Factorization** A detailed description of sequential and parallel cholesky factorization.

**Ch. 3: Fine Grain Cholesky Factorization** Modifications to the algorithm for exploiting fine grain synchronization.

**Ch. 4: *CSIM*: An Event Based Parallel Factorization Simulator** A Detailed description of *CSIM*.

**Ch. 5: Results and Analysis** Coarse and fine grain factorization results from *CSIM* along with analysis.

**Ch.6: Conclusions** Concluding remarks with discussion of further work.

**Bibliography** A list of references from this thesis.

# Chapter 2

## Cholesky Factorization

### 2.1 Math of Cholesky Factorization

Cholesky factorization is an intensive matrix computation that is common in structural analysis, device and process simulation, and electric power network problems. Given a positive definite matrix  $A$ , the Cholesky factor is a lower triangular matrix  $L$ , such that  $A = LL^T$ . This factorization simplifies finding solution of linear systems of  $A$ .

$$Ax = B \quad (2.1)$$

$$LL^T x = B \quad (2.2)$$

By replacing  $A$  with its factors,  $LL^T$ , the system  $Ax = B$  can be quickly solved by backward and forward substitution.

$L$  is found by writing  $A = LL^T$  and using the following relations.

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots \\ A_{21} & A_{22} & \dots \\ \dots & \dots & \dots \end{bmatrix}, L = \begin{bmatrix} L_{11} & L_{12} & \dots \\ L_{21} & L_{22} & \dots \\ \dots & \dots & \dots \end{bmatrix}$$

$$A_{11} = L_{11}^2 \quad (2.3)$$

$$A_{i1} = L_{i1}L_{11} \quad (2.4)$$

$$L_{i1} = A_{i1}/L_{11} \quad (2.5)$$

$$A_{i2} = L_{i1}L_{21} + L_{i2}L_{22} \quad (2.6)$$

$$L_{22} = \sqrt{A_{22} - L_{21}^2} \quad (2.7)$$

These relations can be expanded to compute every element in the factor matrix,  $L$ . A simple sequential algorithm for calculating  $L$  follows.

```

for  $k = 1$  to  $n$ 
   $L_{kk} = \sqrt{L_{kk}}$ 
  for  $i = k + 1$  to  $n$ 
     $L_{ik} = L_{ik}/L_{kk}$ 
  for  $j = k + 1$  to  $n$ 
    for  $i = j$  to  $n$ 
       $L_{ij} = L_{ij} - L_{ik} * L_{jk}$ 

```

Figure 2-1: *Sequential Algorithm for Cholesky Factorization*

## 2.2 Sparse Cholesky Factorization

In scientific computations, the  $A$  matrix is commonly a sparse matrix. For example, a large set of differential equations describing the mechanical structure of a system might be written as a banded matrix. A majority of elements in a banded matrix are zero (non zero elements are  $O(n)$  as opposed to  $O(n^2)$ ). Blindly following the technique outlined in figure 2-1 wastes computation time on factor elements in  $L$  that will turn out to be 0. The non-zero elements of  $L$  can be predetermined by the non-zero structure of  $A$ . With this information, sparse factorization algorithms spend resources on only these non-zero elements of  $L$ .

The dense cholesky algorithm of figure 2-1 is rewritten as in figure 2-2[9]. The first difference from figure 2-1 is that for clarity, two pieces of code have been moved into subroutines called *cdiv* and *cmod*. The main difference is that for faster sparse matrix factorization times, the *cmod* operation is only performed on non-zero entries.

```

CDIV :
    Lkk = √Lkk
    for i = k + 1 to n
        Lik = Lik/Lkk

CMOD :
    for i = j to n
        Lij = Lij - Lik * Ljk

for k = 1 to n
    cddiv(k)
    for each j such that Ljk ≠ 0 do
        cmod(j, k)

```

Figure 2-2: A Sparse Cholesky Factorization Algorithm

## 2.3 Sparse Parallel Cholesky Factorization

### 2.3.1 Where is the Parallelism?

The parallelism available in the sparse factorization algorithm is best illustrated through an example. Figure 2-3 shows the non-zero structure of a sample matrix,  $A$ , along with the non-zero structure of its factor,  $L$ . Columns are numbered along the diagonal. Element values are not important—so non zero elements are marked with bullets(•). Notice that the factorization algorithm needs the non-zero structure of  $L$  precomputed. This step is performed sequentially and is quick. (There are techniques for minimizing non-zero elements in  $L$  through column reordering in  $A$ , but these optimizations were not studied.)

With the non-zero structure of  $L$ , an elimination tree can be built.

Given a matrix  $A$ , with factor  $L$ , the parent of column  $j$  in the elimination tree of  $A$  is:  $parent(j) = \min \{i | L_{ij} \neq 0, i > j\}$ [9].

The elimination tree shows an important relationship between columns. Columns are only modified by their descendants in the elimination tree. Figure 2-4 shows that columns

$$A = \begin{bmatrix} 1 & \bullet & & \bullet & & & & \\ & 2 & & \bullet & & & & \\ \bullet & & 3 & \bullet & & & & \\ & & \bullet & 4 & & & & \\ & \bullet & & & 5 & \bullet & & \\ & & & & \bullet & 6 & \bullet & \\ \bullet & \bullet & & & & \bullet & 7 & \bullet \\ & & & \bullet & & \bullet & & 8 \end{bmatrix}, L = \begin{bmatrix} 1 & & & & & & & \\ & 2 & & & & & & \\ \bullet & & 3 & & & & & \\ & & \bullet & 4 & & & & \\ & \bullet & & & 5 & & & \\ & & & & \bullet & 6 & & \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 7 & \\ & & & \bullet & & \bullet & & 8 \end{bmatrix}$$

Figure 2-3: A Sample Matrix and its Factor

1, 3, 4 are independent of columns 2, 5, 6. Therefore, operations on columns 1, 3, 4 can occur in parallel with operations on columns 2, 5, 6. For a large matrix  $A$ , many such parallel branches will be available.

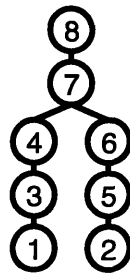


Figure 2-4: Elimination tree of Sample matrix

### 2.3.2 Fan-out Factorization

Various parallel algorithms have been developed to find the cholesky factorization of sparse matrices. One is the fan-out algorithm developed at the Oak Ridge National Laboratory[4]. In fan-out factorization, the columns are first divided among processors. Each processor able to perform a *cdiv* operation does so. It then sends the result to all processors that will use the data for a *cmul*. When a column has received all the modifications it can receive, it is *completed* with a *cdiv* operation. Now this column can be sent to processors that need it. The algorithm is shown in figure 2-5. (This pseudo-code (figure 2-5) is copied directly from [9]).



```

while I own a column which is not complete do
  if  $nmod[k] = 0$  then —  $k$  is ready to be completed
     $cdiv(k)$ 
    send  $k$  to processors which own columns  $\{j | j > k, L_{jk} \neq 0\}$ 
  else
    receive a column  $k$ 
    for each  $j$  such that  $L_{jk} \neq 0$  which i own do
       $cmod(j, k)$ 

```

Figure 2-5: *Fan-Out Algorithm for Cholesky Factorization*

The  $nmod[k]$  structure is introduced so that each column can keep track of column modifications. It is initialized to the number of  $cmod$  operations that each column will receive. When a column has a  $cmod$  operation performed, the column's value in  $nmod[k]$  is decremented. When the value reaches zero, the column is completed with a  $cdiv$  and distributed.

### 2.3.3 Determining Tasks

The algorithm in figure 2-2 uses **send** and **receive** primitives for data communication. In a shared memory implementation, the column data will reside in global memory. Explicit **sends** and **receives** are replaced by column locks and shared memory reads and writes. The column locks are used to synchronize between producer and consumer processors. Data is communicated by reading from and writing to shared memory.

To support a shared memory model, the fan-out algorithm was modified to include a task determining step. All  $cdiv$  and  $cmod$  operations are predetermined by examining the non-zero structure of the  $L$  matrix. Figure 2-6 shows all the  $cdiv$  and  $cmod$  operations required to calculate  $L$  for the sample matrix from figure 2-3.

These operations have been placed on a time line. Operations within the same time step may be performed in parallel. A box with a single digit inside represents the  $cdiv$  operation, such as  $\boxed{1}$  and  $\boxed{2}$ . All other boxes represent  $cmod$  operations as in  $\boxed{3.1}$  and  $\boxed{5.2}$ . The notation 3.1 stands for column 3 getting modified by column 1.

Since columns 1 and 2 are not receiving any modifications they can be computed

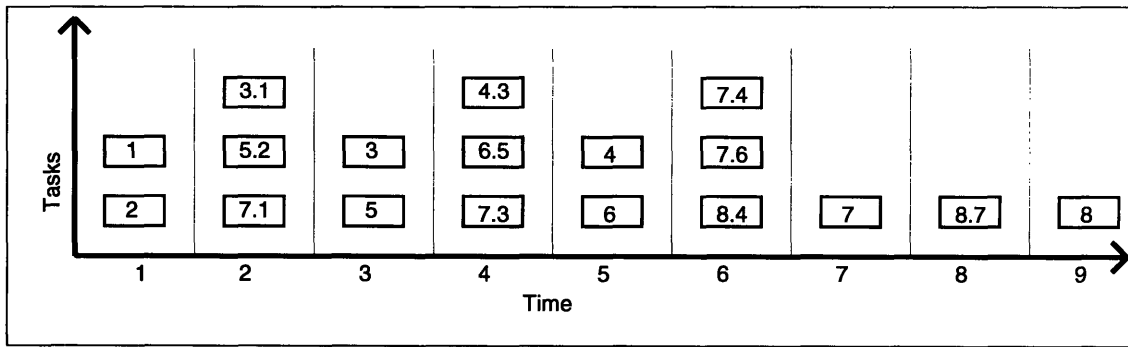


Figure 2-6: *Work Available*

in parallel in time step 1. A column may not be modified until its modifier has passed through the *cdiv* operation. So operations  $\boxed{3.1}$ ,  $\boxed{5.2}$  and  $\boxed{7.1}$  must wait till time step 2—when columns 1 and 2 are available. Once a column has received all of its modifications, it can be *cdived*, as in columns 3 and 5 in time step 3.

### 2.3.4 Multiprocessor Operation

With the tasks established, let us see how they might be distributed among 4 processors. The fan-out algorithm requires that processors modify only the columns that they own. Columns have been assigned using a *mod* function—Processor 0 owns column 1 and 5, Processor 1 owns 2 and 6, Processor 2 owns 3 and 7 and Processor 3 owns 4 and 8.

Figure 2-7 shows the timelines of four processors executing the tasks determined in figure 2-6. For simplicity, the amount of work in each block is assumed to be identical. Synchronization overhead and memory latencies are not shown.

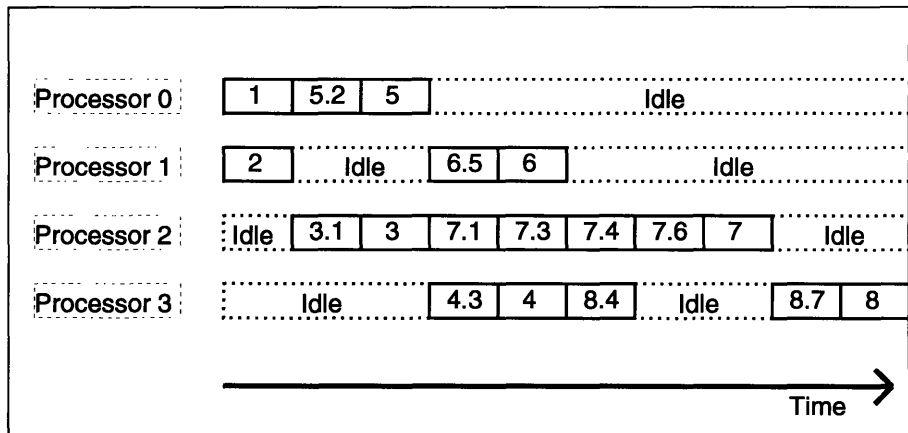


Figure 2-7: 4 Processor Operation

## 2.4 Mod Mapped Column Distribution

Fan-out factorization has one important variable that affects performance—the distribution of columns to processors. If, for some reason, all the independent columns are assigned to the same processor, the performance of the system decreases dramatically due to lost parallelism. In this study, the columns will be distributed using a *mod* function. Each processor will get the columns where:

$$Column \bmod Processor_{number} = 0 \quad (2.8)$$

# Chapter 3

## Fine Grain Cholesky Factorization

With fine-grain synchronization, a consumer processor can grab the smallest amount of data as soon as it is available from the producer. For a discussion of fine grain synchronization implementations in multiprocessors, see Kranz, Lim, and Agarwal[7].

As the Cholesky application is currently written, consumers must wait until producers release the entire column they are computing. This poses an artificial data dependency that inhibits parallelism. Two things can be done.

1. Using fine grain synchronization, allow a processor to grab data as soon as it is available from the producer.
2. Reschedule the computation to allow multiple accesses to columns, synchronize with fine grain synchronization.

### 3.1 Fine Grain Fan-Out

The effect of choice 1 is that consumers are allowed to read data as soon it is available. Figure 3-1 shows when consumers are waiting for data from producers in the example. Arrows mark the locations where fine grain synchronization would help. For example, Processor 1 waits for Processor 0 before it can compute 6.5 since it needs column 5 from Processor 0.

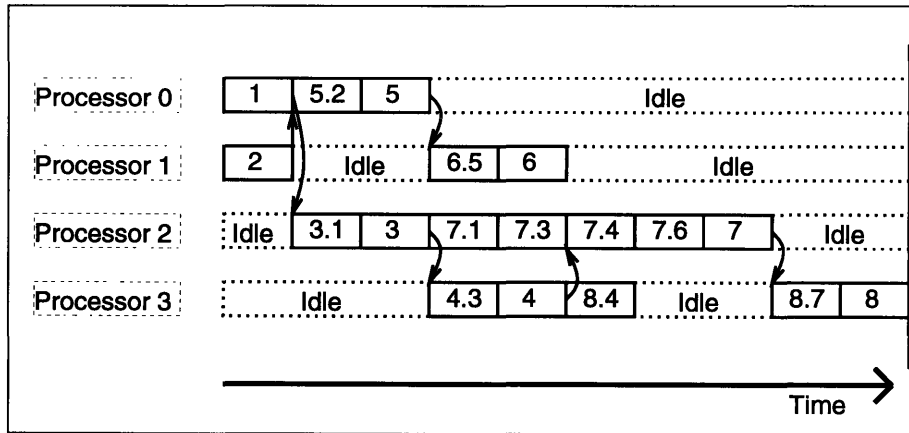


Figure 3-1: 4 Processor Operation—Arrows mark Data Dependencies

With fine grain synchronization, Processor 1 does not have to wait for Processor 0 to completely finish its *cdiv* operation on column 5. It can grab parts of column 5 as soon as they are available. To implement fine-grain synchronization, column locks were removed from the fan-out code. A locking structure was assumed for every matrix element. Before, a processor locked a column, computed, and then unlocked the column, now it locks an element, computes, and then unlocks the element. This optimization changes the finish time as shown in figure 3-2.

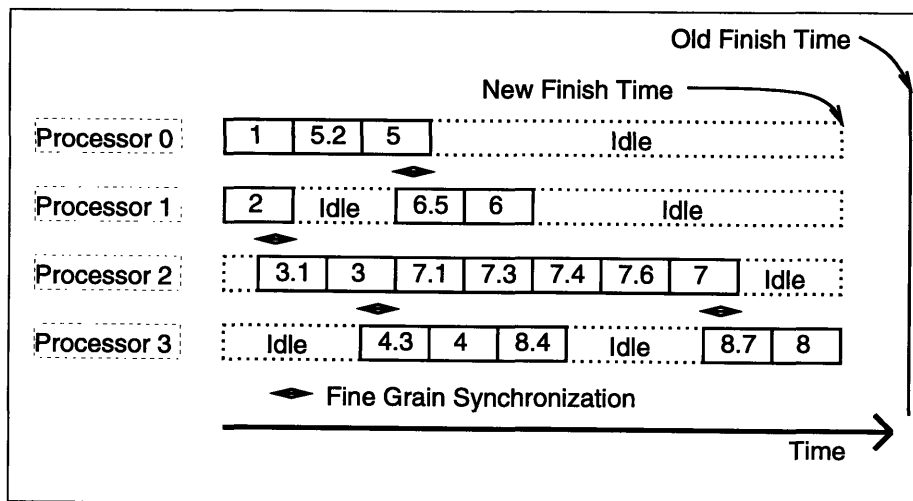


Figure 3-2: 4 Processor Operation With Fine Grain Synchronization

- Processor 1 begins computing 6.5 earlier.
- Processor 2 begins computing 3.1 earlier.
- Processor 3 begins computing 4.3 earlier.
- Processor 3 begins computing 8.7 earlier.

Figure 3-2 assumes no overhead related with fine grain synchronization.

### 3.1.1 Fine Grain w/Task Redistribution

With finer synchronization, we get greater parallelism. But how much? From the example in the previous section, fine grain synchronization helped out only part of the time. What was the problem? This optimization helps only during a *cdiv* operation with *cmo*d waiters. *Cmo*d operations waiting for the *cdiv* to complete can grab data as soon as it is available.

The trouble is that for a matrix of size  $n$ , only  $n$  *cdiv* operations will be performed. This is compared to a very large number of *cmo*d operations. Furthermore, not every *cdiv* will have a waiter. So, this fine grain optimization should prove to have very little speedup over the coarse fan-out algorithm. Can fine grain synchronization be exploited more in cholesky factorization?

The answer lies in the observation that a *cmo*d operation is associative. In the running example, column 8 needs two *cmo*ds, [8.4] and [8.7]. These operations can occur in either order. More importantly, they can be *interleaved* and still produce the correct final value. With the fan-out algorithm, interleaving of *cmo*d operations is impossible since all operations on a particular column are handled by the same processor. In figure 3-2, Processor 4 will do both [8.4] and [8.7]. If they were handled by different processors, then they could be run in parallel with fine grain synchronization. With many *cmo*ds handled in this way, the resulting speedup should be higher than with simple fan-out with fine grain.

So *cdiv* and *cmo*d operations have to be assigned to processors in another way. The current assignment method is good because it promotes data locality. Moving *cmo*d tasks

to other processors reduces the overall locality in the application. Hopefully the loss in locality will be offset by an increase in parallelism of the *cmod* operations.

How should the tasks be redistributed? Many factors are now important for optimal performance—data locality should be kept high. Parallelism should be increased and the load should be balanced. This study relies on a simple solution that proves to be effective. Columns are still assigned using a mod function. But the *cdiv* and *cmod* operations are distributed sequentially. The first operation is given to Processor 0, the next to Processor 1 and so on.

This simple remapping of tasks gives the performs as shown in figure 3-3. The tasks from figure 2-6 have been mapped sequentially across the processors.

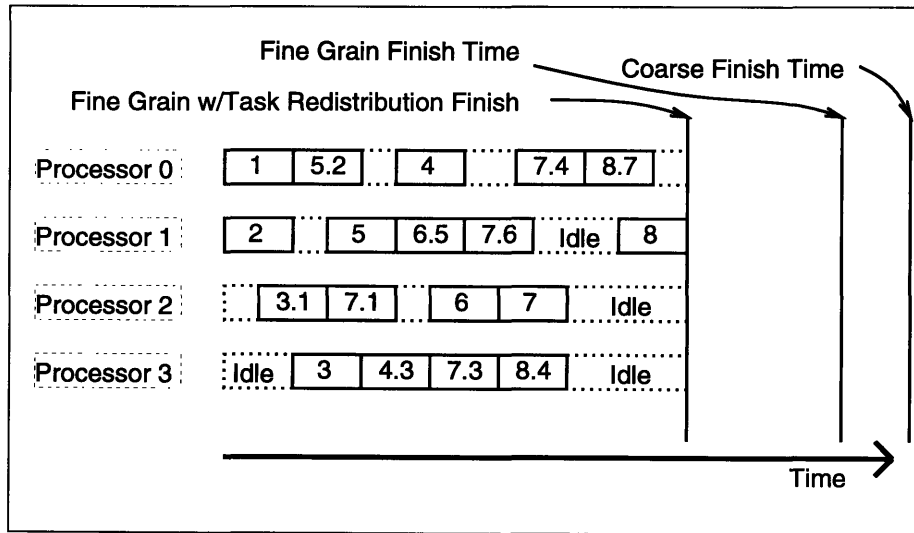


Figure 3-3: 4 Processor Operation With Fine Grain Synchronization and Task Redistribution

This implementation has reduced data locality since processors are assigned work on columns they do not own. Figure 3-3 does not take into account any locality effects. It assumes that the decrease in locality does not affect performance. This assumption is made to illustrate the possible gains in parallelism. *CSIM*, described in the next chapter, was used to carefully study the performance. For our idealized case, it is clear that remapping the tasks decreased the execution time.

Many other task mapping schemes are possible. A heuristic-based scheme that at-

tempts to preserve some locality might perform better in a real system. Other schemes have been left open for study.



# Chapter 4

## *CSIM*: An Event Driven Parallel Factorization Simulator

*CSIM* is a parallel cholesky factorization simulator that can process large matrices at very high speeds. It can simulate a large machine with caches and a simple network model. The introduction gave a brief motivation for the development of *CSIM*.

*CSIM*'s target was a model of the Alewife machine. The Alewife Machine is a scalable multiprocessor with identical processing nodes connected in a 2-D mesh. Each node consists of a 32 bit RISC processor, a floating point chip, 64 KB cache, 8 MBytes of dynamic RAM and a network routing chip[7]. The distributed memory of the machine appears as a shared memory to applications. Hardware for cache coherence maintains this abstraction. The machine has hardware support for fine grain synchronization at the lowest level of memory—processors in an Alewife system can arbitrate on a single word of memory.

*CSIM* simulates machine operations at the column level. The smallest events characterized are individual *cdiv* and *cmop* operations from figure 2-5 and figure 2-6. The simulator is initialized with a matrix to factor,  $A$ . The structure of  $A$ 's factor matrix,  $L$ , is then computed. This structure is used to determine the *cdiv* and *cmop* operations as in figure 2-6. For a coarse grain simulation, the tasks are assigned to processors as in figure 2-7. For a fine grain simulation, the tasks are assigned as in figure 3-2 or figure 3-3.

With the tasks<sup>1</sup> now assigned to processors, the parallel simulation begins. The simulator goes down the task queue of each processor and tries to see if the task at the head of the queue can execute. A task can execute if the data requested by the task is not locked. If it is locked, the simulator stalls the processor and continues to the next. This process continues until all processor queues are empty. At the end of the simulation, a breakdown of cycles is presented from the factorization, with total and average number of cycles spent on the following:

Utilization	Average Processor Utilization for the Simulation
Column Wait Idling	Average Column Wait Idling Time per Processor
Fine Grain Overhead	Average Fine Grain Overhead for the Simulation
Coarse Grain Overhead	Average Coarse Grain Overhead for the Simulation
Remote Misses	Average Time Spent servicing Remote Misses
Local Misses	Average Time Spent servicing Local Misses
Idle	Average Idle Time

Table 4.1: *Statistics Gathered*

## 4.1 CSIM Internals

### 4.1.1 Processor Simulation

For each processor in the simulation, a detailed set of statistics is kept. As each individual *cdiv* and *cmop* operation is processed, the processor statistics are updated. The most important statistic is the current time—maintained in cycles for a processor. For each *cdiv* and *cmop* operation, the processor time is incremented by the number of cycles it takes to perform the operation.

The amount of time taken by an operation is determined by the amount of work and the location of the required data.<sup>2</sup> A *cdiv* operation is handled as follows. An element in a *cdiv* operation is:  $L_{ik} = \frac{L_{ik}}{L_{kk}}$ .

<sup>1</sup>On a real machine, these tasks would not be high level processes. Each node would run a single process and would manage its own *cdiv* and *cmop* operation queue. This setup is easy to model and is a good implementation.

<sup>2</sup>Other operations, such as loop indexing, are not added.

$L_{ik}$  is a memory read. It may be a cache access, a local memory access or a remote memory access.

$L_{kk}$  is a cached value. It is read at the beginning of the of each *cdiv* operation. The time to read it into the cache for the first time is not added to the processor time.

*divide* is a floating point operation.

$L_{ik}$  is a memory write.

Summing up the *cdiv* operation:

1	Cache Accesses
2	Memory Accesses
1	Floating Point Operation

Table 4.2: *Simulated cdiv operation*

A single element *cmod* operation,  $L_{ij} = L_{ij} - L_{ik} * L_{jk}$  is simulated as follows:

$L_{ij}$  is a memory read.

$L_{ik}$  is a memory read.

$L_{jk}$  is cached.

*subtract* is a floating point operation.

*multiply* is a floating point operation.

$L_{ij}$  is a memory write.

Summing up the *cmod* operation:

1	Cache Accesses
3	Memory Accesses
2	Floating Point Operation

Table 4.3: *Simulated cmod operation*

To calculate the time for each *cmod* and *cdiv* operation, the amount of taken for a single element computation is scaled by the number of elements. Various overheads, discussed below, are also added.

## 4.1.2 Modeling Coarse Grain Synchronization

A task queue is maintained for every processor by *CSIM*. Before a task can execute, the column required by the task must be available for reading (i.e. not locked by any other processor). *CSIM* stalls the consumer processor until the data is available for reading. But in a real machine, the consumer must incur an overhead for waiting.

*CSIM* models this by adding a polling overhead. Any consumer that receives data from a producer pays this overhead on the wait time between the request and the receipt of the data. Figure 4-1 illustrates the computation.

$$\begin{aligned}
 T_{start} &= \text{Time Consumer starts polling for data from Producer} \\
 T_{ready} &= \text{Time Producer is ready with the data.} \\
 T_{wait} &= T_{ready} - T_{start} \\
 T_{overhead} &= T_{wait} * \text{POLLING\_OVERHEAD} \\
 T_{read} &= T_{start} + T_{wait} + T_{overhead}
 \end{aligned}
 \tag{4.1}$$

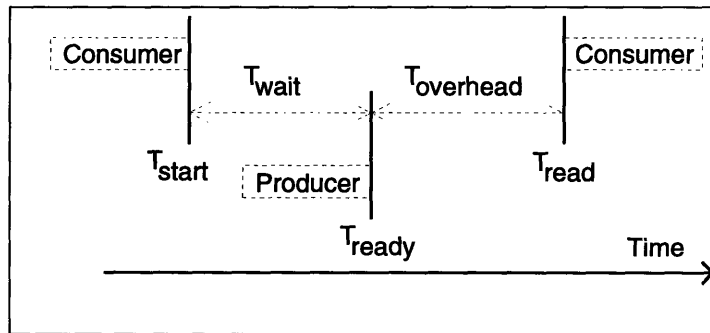


Figure 4-1: *Polling Overhead Model for Coarse Cholesky*

The polling overhead is a constant value that is a function of the machine size. This value is based on the following assumptions:

- Synchronizations are infrequent. This is the main assumption that allows the use of this polling model for coarse cholesky synchronization.
- Network traffic is low or alternatively, the network bandwidth is very high. Low

network traffic means that the only penalty a consumer pays for waiting is the polling overhead.

When multiple consumers are waiting for a producer, each incur a polling overhead. But since the producer can service only one consumer at a time, each consumer incurs an additional delay that is a function of two values: Its order in arrival at the producer, and the amount of time the producer takes to send out the requested data. The first value means that the first consumer reads first, the second reads second and so on. The second value means that the second consumer must also wait an additional time equivalent to the time it takes for the producer to service the first.

What is the extra time that the second consumer must wait? The producer has to send data to the first consumer. With an infinite network buffer, this time can be modeled as the amount of time it takes the producer to copy the data from its cache to the network.

$$\begin{aligned}
T_{start1} &= \text{Consumer 1 Requests Data from Producer} \\
T_{ready} &= \text{Producer is ready with the data.} \\
T_{wait1} &= T_{ready} - T_{start1} \\
T_{overhead1} &= T_{wait1} * POLLING\_OVERHEAD \\
T_{read1} &= T_{start1} + T_{wait1} + T_{overhead1}
\end{aligned} \tag{4.2}$$

$$\begin{aligned}
T_{start2} &= \text{Consumer 2 Requests Data from Producer} \\
T_{wait2} &= T_{ready} - T_{start2} \\
T_{overhead2} &= T_{wait2} * POLLING\_OVERHEAD \\
T_{send1} &= \#elements * CACHE\_ACCESS\_TIME \\
T_{read2} &= T_{start2} + T_{wait2} + T_{overhead2} + T_{send1}
\end{aligned} \tag{4.3}$$

### 4.1.3 Modeling Fine Grain Synchronization

A task queue is still maintained for every processor by *CSIM*. But these tasks use finer synchronization and synchronize at the matrix element level. So a task can execute as soon the first part of the data it requires is available.

The polling overhead model in figure 4-1 is inadequate for fine grain synchronization. First, every synchronization operation is slower. Even successful synchronizations pay a small penalty to check the synchronization structures. An analysis of fine grain synchronization with an application on Alewife was done by Yeung[13]. *CSIM* models this with a `FINE_GRAIN_OVERHEAD` parameter. This parameter adds a constant overhead to every successful synchronization.

Determining whether a synchronization will succeed or fail is tricky. Timing between producers and consumers becomes critical in determining the total waiting time for the consumer. There are five cases that must be dealt with individually.

- Consumer and Producer Start at Similar Times.

This is the worst case for fine grain synchronization between producers and consumers. The two might start thrashing on individual cache lines. The consumer will read a line. The producer will shortly invalidate it. Both proceed forward in this manner, producing many cache coherence protocol packets. This can severely hurt application performance.

*CSIM* handles this case by forcing the consumer to back off for a constant amount of time. The time is equivalent to the time taken by 2 *cmop* iterations. 2 iterations will fill one cache line<sup>3</sup>. To compute this value, we assume the worst case for the the *cmop* iteration—a remote read and a remote write. `BACKOFF_TIME` is calculated as follows:

$$\begin{aligned}
 T_{1 \text{ cmop}} &= (1 * \text{CACHE\_ACCESS} + 3 * \text{REMOTE\_ACCESS} \\
 &\quad + 2 * \text{FLOAT\_OP}) \\
 &\quad (1.0 + \text{FINE\_GRAIN\_OVERHEAD}) \\
 \text{BACKOFF\_TIME} &= 2 * T_{1 \text{ cmop}} \tag{4.4}
 \end{aligned}$$

With this backoff time, a consumer is *guaranteed* to be behind the producer. After waiting, The consumer can proceed with all synchronizations succeeding.

- Consumer Starts a Little After the Producer.

---

<sup>3</sup>Each operation is on a double value—two words.

The important question in this case is what is “a little after”? If the consumer starts after the `BACKOFF_TIME` window then most synchronizations will be successful. The only penalty that the consumer will pay is the overhead of a fine grain vs. a regular read. On the other hand, if the consumer starts within the `BACKOFF_TIME` window, then it will have to take the `BACKOFF_TIME` penalty. The simulator keeps careful track of this timing and correctly handles this case.

- Consumer Starts Much Later Than Producer.

This case is not very different from coarse grain synchronization. Every read by the consumer will be met with success. The only penalty that the consumer will pay is the overhead of a fine grain vs. a regular read.

- Many Consumers

All three scenarios above become more complex as the number of consumers increases beyond 1. *CSIM* imposes the backoff penalty on each consumer and insures that the producer serves sequentially serves the consumers.

- Many Writers

Many processors attempting to modify locations in the same cache line at the same time will be heavily penalized by cache coherence protocol overhead. As explained below, *CSIM* avoids the cache coherence protocol issue altogether. This case is handled just like the Many Consumers case, with the writer suffering a backoff penalty and being handled sequentially by owner.

#### **4.1.4 Cache Simulation**

- Coarse Grain Factorization

*CSIM* maintains a 64K cache for each processor. Cache coherence is not maintained. In coarse grain operation, this is not a problem. The owner of a column is the only processor that attempts any writes to the column. Readers of that column will not attempt to read it until it is completed by the owner. Since after the completion, no further writes occur, there are no coherency issues.

The caches are maintained with a random replacement strategy. Cache contents can include local and remote locations. A cache hit takes 2 cycles.

- **Fine Grain Factorization**

In the fine grain case, multiple processors might attempt to write to the same location. To avoid maintaining coherent caches in the simulator, remote locations are not allowed to be cached if they are being written. This restriction effectively turns off caching of remote locations.

Due to the complexity associated with cache coherence, *CSIM* does not support removing this restriction. With cache coherence, the overall results of fine grain factorization should be better.

#### **4.1.5 Local Memory**

Local Memory is maintained for each processor as a list of columns that it owns. The size of memory is not bounded. Though, it is unlikely that any processor will get an unevenly large number of columns assigned due to the *mod* mapping of columns.

Local memory accesses produce a cache line of 4 words in 15 cycles. Or a per double word access time of 7.5 cycles.

#### **4.1.6 Shared Memory/Network Model**

*CSIM* simulates a distributed shared memory. Each memory access is mapped to local or global memory by a vector that maintains the location of columns. This in effect simulates a shared memory since in a real machine, a memory access would use some bits in the address to determine where the location is stored.

Shared memory access times are determined by the square root of the total number of processors. Alewife takes about 2 cycles per network hop. On an  $P$  processor machine configured in a 2-D mesh, the average distance should be about  $\frac{\sqrt{P}}{2}$ . Access times are constant across a simulation. No contention on the network is modeled. The formula used for the access time is:



$$T = \frac{2 * \sqrt{P}}{2} + 15 \quad (4.5)$$

This time is the latency for a cache line(4 words) read. A remote column read uses the per word time multiplied by the number of words read to compute the latency for the read.

## 4.2 Knobs and Switches

Table 4.4 summarizes important controls in the simulator.

P	Number of processors in the simulation
CACHE_SIZE	Size of per processor cache
FLOAT_OP	Cycle Time to perform a floating point operation.
CACHE_ACCESS	Cache Line Access Time
LOCAL_ACCESS	Local Memory Access Time on a Cache Miss
REMOTE_ACCESS	Remote Memory Access Time on a Cache Miss
POLLING_OVERHEAD	Overhead added to polling for data.
FINE_GRAIN_OVERHEAD	Fine Grain Read/Write Overhead
BACKOFF_TIME	Backoff time for a failed synchronization

Table 4.4: *CSIM controls*

## 4.3 Equipment Required

*CSIM* is written in C and runs on UNIX workstations. Simulation time for a 3466 column matrix with 13681 non zeros with 512 processors is about 5 minutes on a Sparc Station 10/30.

# Chapter 5

## Results and Analysis

Five test matrices were used in this study. The first four matrices, part of the SPLASH suite of benchmarks[11] are the same matrices used by Rothberg[9]. The fifth matrix, w15.1, is bcsstk15 sampled to 1/3 the number of columns. w15.1 was created for rapid testing.

A quick summary of the matrices:

Matrix	Number of Columns	Number of Non Zeros
bcsstk14	1806	32630
bcsstk15	3948	60882
d750.O	750	281625
lshp.0	3466	13681
w15.1	1316	7341

Table 5.1: *Test Matrices*

Ideal machine and practical machine results will be presented for the following three variations of fan-out cholesky factorization.

**Coarse:** This is the fan-out algorithm modified for shared memory. A column lock mechanism is used for synchronization between processors.

**Fine No Task Redistribution:** The coarse grain version was modified to use fine grain synchronization on matrix elements. *cdiv* and *cmmod* tasks are assigned in the same way as the Coarse grain version.

Fine w/Task Redistribution: Fine grain synchronization is used on matrix elements.

*Cdiv* and *cmop* tasks are redistributed using a round robin system.

## 5.1 Ideal Machine Results

To see how the three versions of the fan-out algorithm compare, ideal machine results are presented first. In the results that follow, *CSIM* was configured to have a uniform memory access time of 1 cycle. All floating point computations took a cycle as well.

### 5.1.1 Ideal:Coarse-Single Processor Run Times

*CSIM* was configured as follows to produce single processor run times.

Parameter	Value
P	1
CACHE_SIZE	64 KB
FLOAT_OP	1 cycle
CACHE_ACCESS	1 cycle
LOCAL_ACCESS	1 cycle
REMOTE_ACCESS	n/a
POLLING_OVERHEAD	n/a
FINE_GRAIN_OVERHEAD	n/a
BACKOFF_TIME	n/a

Matrix	1 Processor Run Time
bcsstk14	5.9838e+07
bcsstk15	9.89893e+08
d750.O	4.23282e+08
lshp.0	2.84074e+07
w15.1	2.97619e+07

Table 5.2: *Ideal:Coarse-Single Processor Run Times*

### 5.1.2 Ideal:Coarse-Speedup Curves

The coarse grain simulation uses a polling overhead model that imposes an overhead on all synchronization waiting times. For this ideal run, the overhead was set at zero. Speedup curves, shown in figure 5-1 were produced relative to an ideal single processor coarse grain performance. The simulator was configured as follows:

Parameter	Value
P	Number of Processors
CACHE_SIZE	64 KB
FLOAT_OP	1 cycle
CACHE_ACCESS	1 cycle
LOCAL_ACCESS	1 cycle
REMOTE_ACCESS	1 cycle
POLLING_OVERHEAD	0.0
FINE_GRAIN_OVERHEAD	n/a
BACKOFF_TIME	n/a

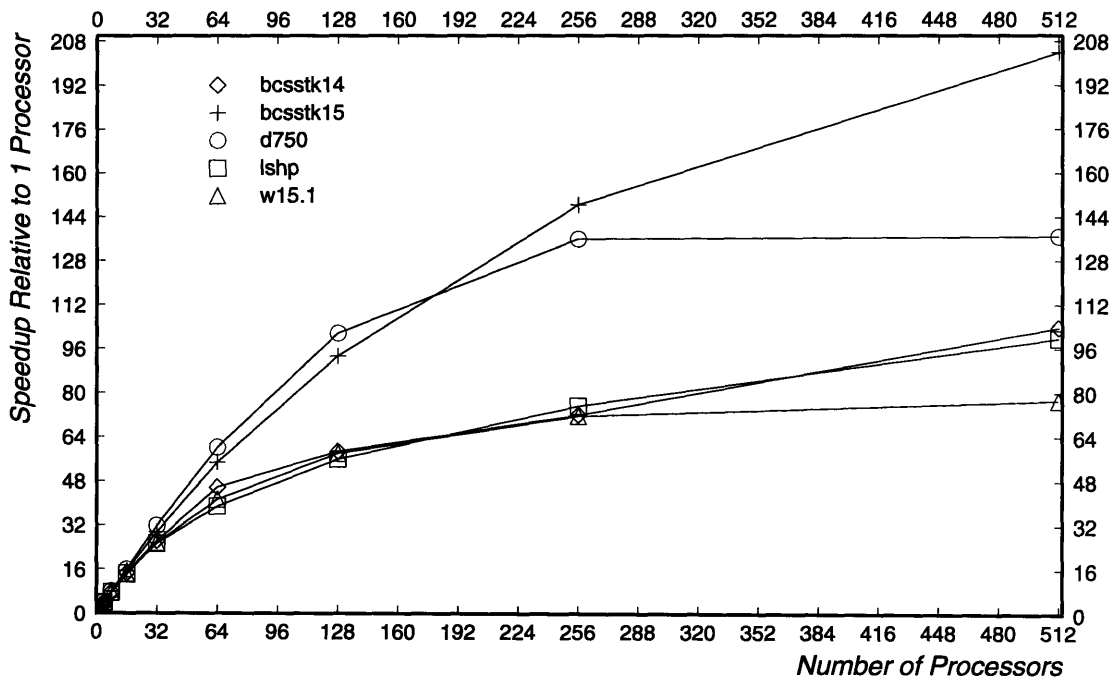


Figure 5-1: Ideal:Coarse Grain Speedup

From figure 5-1, it is clear that even on an ideal machine, the fan-out version of the application shows moderate speedup. The best performance is bcsstk15's speedup, at about 208 on 512 processors. The worst is w15.1, at about 80 for 512 processors. Since there is no overhead of any sort in these simulations, the lack of parallelism must be due to the algorithm and the data.

### 5.1.3 Ideal:Fine Grain no Task Redistribution Speedup

This is the first fine grain implementation. Column locks were replaced by element locks. All fine grain overhead was removed from the simulator for this run. Figure 5-2 shows much better speedup over the coarse grain application. The best speedup is delivered by d750 at about 224, the worst by w15.1 at about 90. These results suggest that the few *cdiv* operations are important because they provide the data for *cmop* operations to continue.

Parameter	Value
P	Number of Processors
CACHE_SIZE	64 KB
FLOAT_OP	1 cycle
CACHE_ACCESS	1 cycle
LOCAL_ACCESS	1 cycle
REMOTE_ACCESS	1 cycle
POLLING_OVERHEAD	n/a
FINE_GRAIN_OVERHEAD	0.0
BACKOFF_TIME	0.0

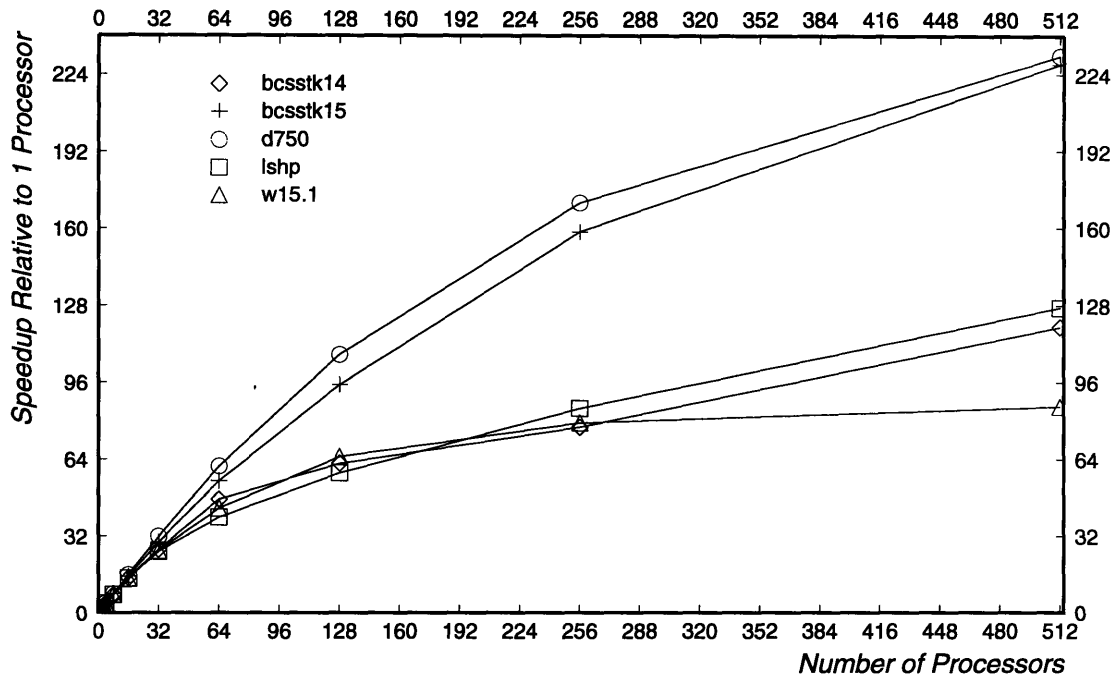


Figure 5-2: *Ideal:Fine Grain–NO Task Redistribution Speedup*

### 5.1.4 Ideal:Fine Grain w/Task Redistribution Speedup

In the version of the application, the tasks have been redistributed as well, figure 5-3 shows by far the best performance—approaching a speedup of about 448 for all matrices. This ideal run allows certain behavior that cannot be allowed in a real machine. For example, many readers and writers can be accessing the same memory location without penalty. This kind of behavior would be severely penalized in a real machine by cache coherency protocol overhead.

Parameter	Value
P	Number of Processors
CACHE_SIZE	64 KB
FLOAT_OP	1 cycle
CACHE_ACCESS	1 cycle
LOCAL_ACCESS	1 cycle
REMOTE_ACCESS	1 cycle
POLLING_OVERHEAD	n/a
FINE_GRAIN_OVERHEAD	0.0
BACKOFF_TIME	0.0

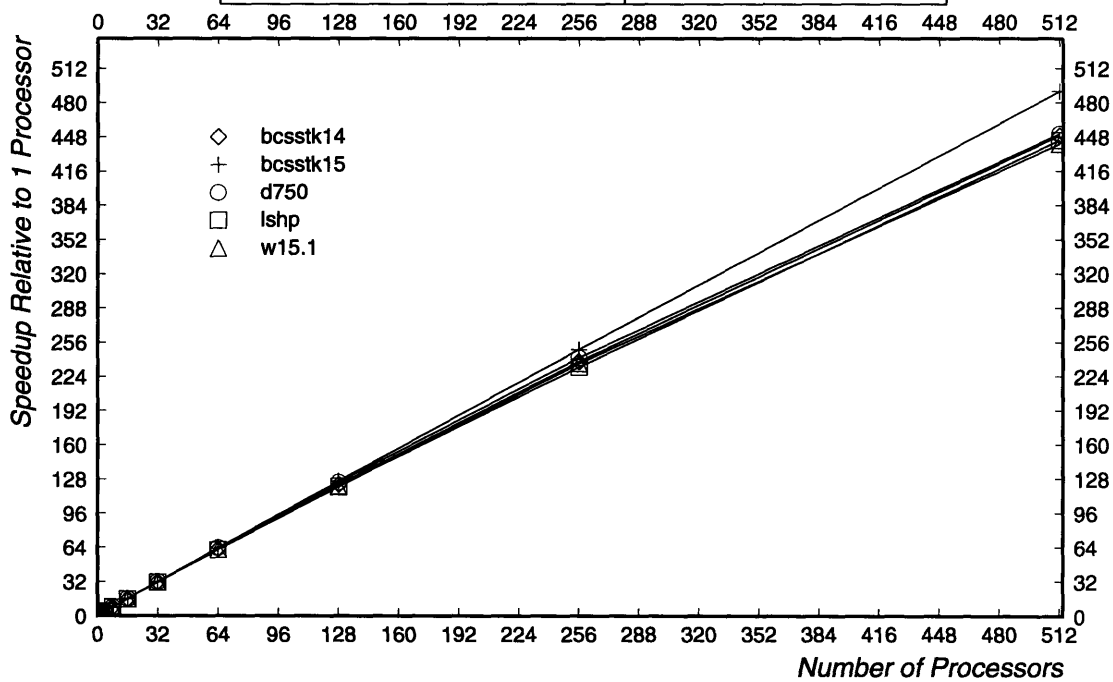


Figure 5-3: *Ideal: Fine Grain w/Task Redistribution Speedup*

### 5.1.5 Ideal:Coarse-Utilization

The following set of figures show the processor utilization, for an ideal machine with the coarse grain simulation. The X-axis is the number of processors. The Y-axis is the average processor utilization. Right above the utilization curve is the local miss curve. Above that is the remote miss curve. Above that is the *Column Sync Idle* curve. This

curve represents the average amount of time spent waiting for column synchronizations. Finally, above that, is the idle time from lack of work. From these curves, it is clear that after 256 processors, from a quarter to more than half the time is spent waiting for Column Synchronizations—or column locks.

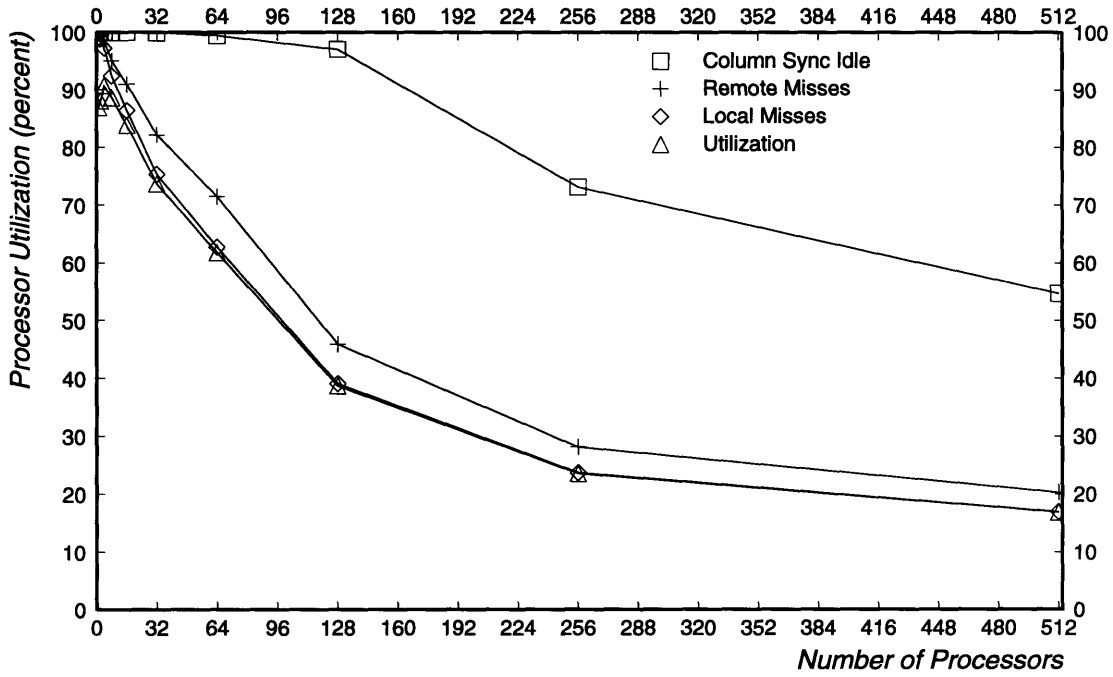


Figure 5-4: *Ideal:bcstkl4 Coarse Grain Utilization Curve*



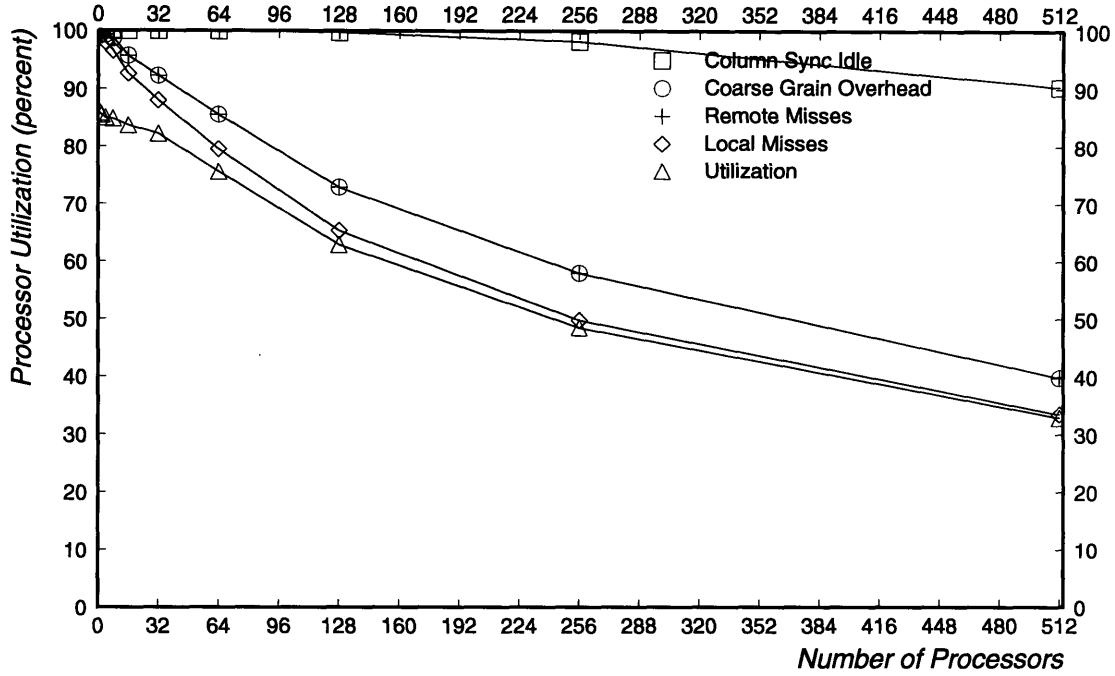


Figure 5-5: *Ideal:bcstk15* Coarse Grain Utilization Curve

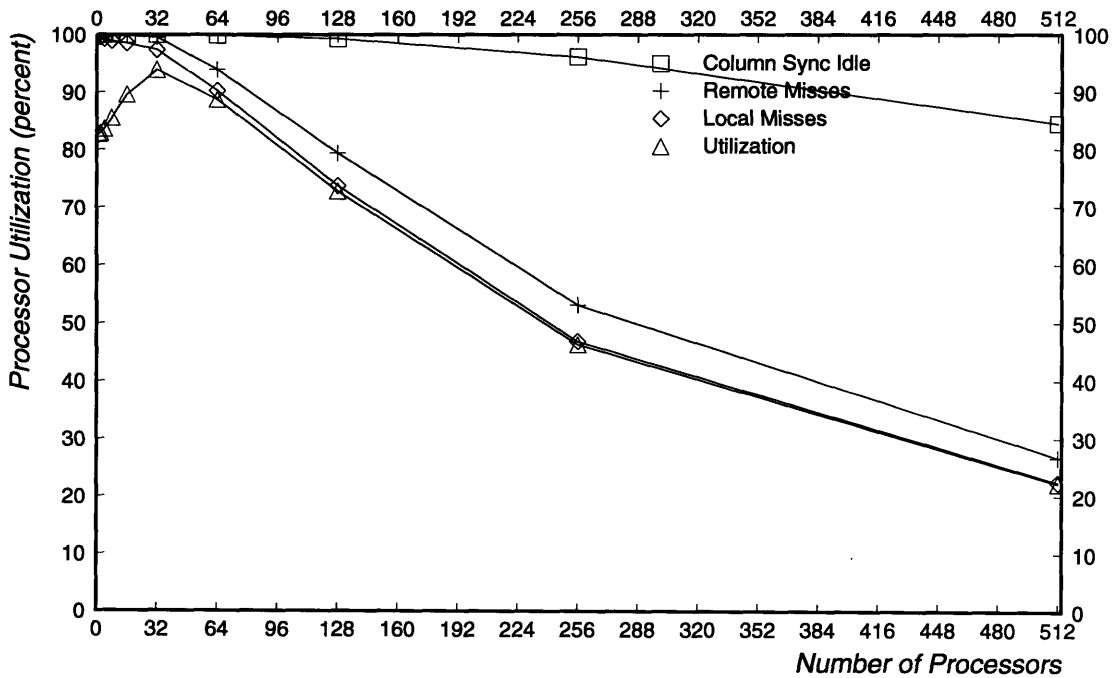


Figure 5-6: *Ideal:d750.O* Coarse Grain Utilization Curve

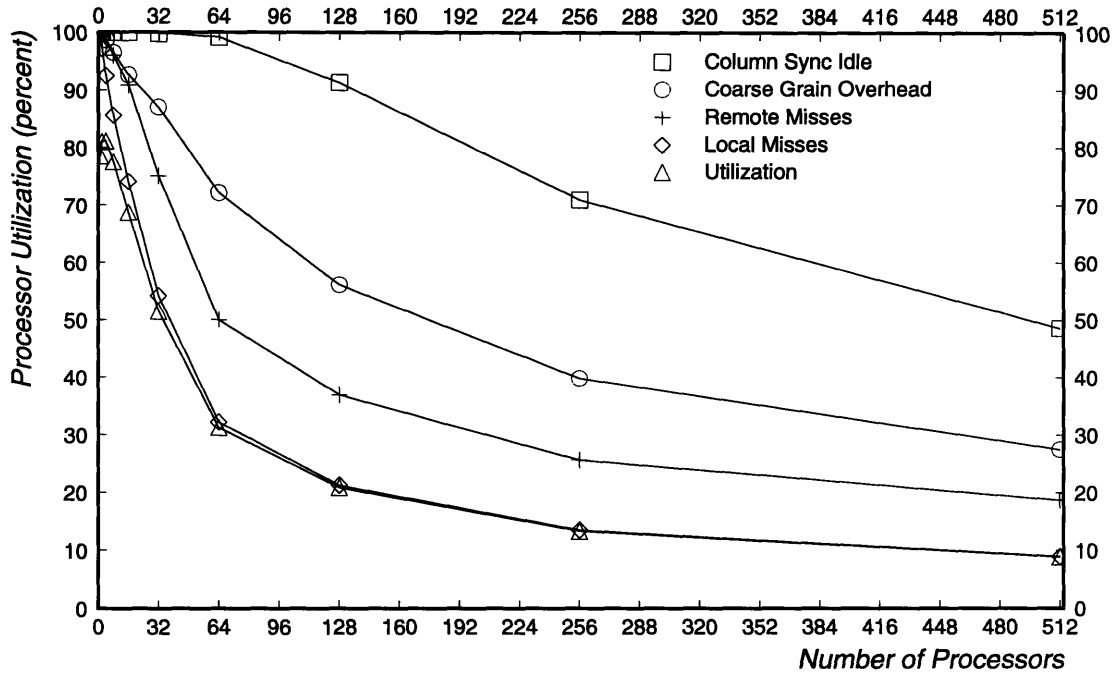


Figure 5-7: *Ideal:lshp Coarse Grain Utilization Curve*

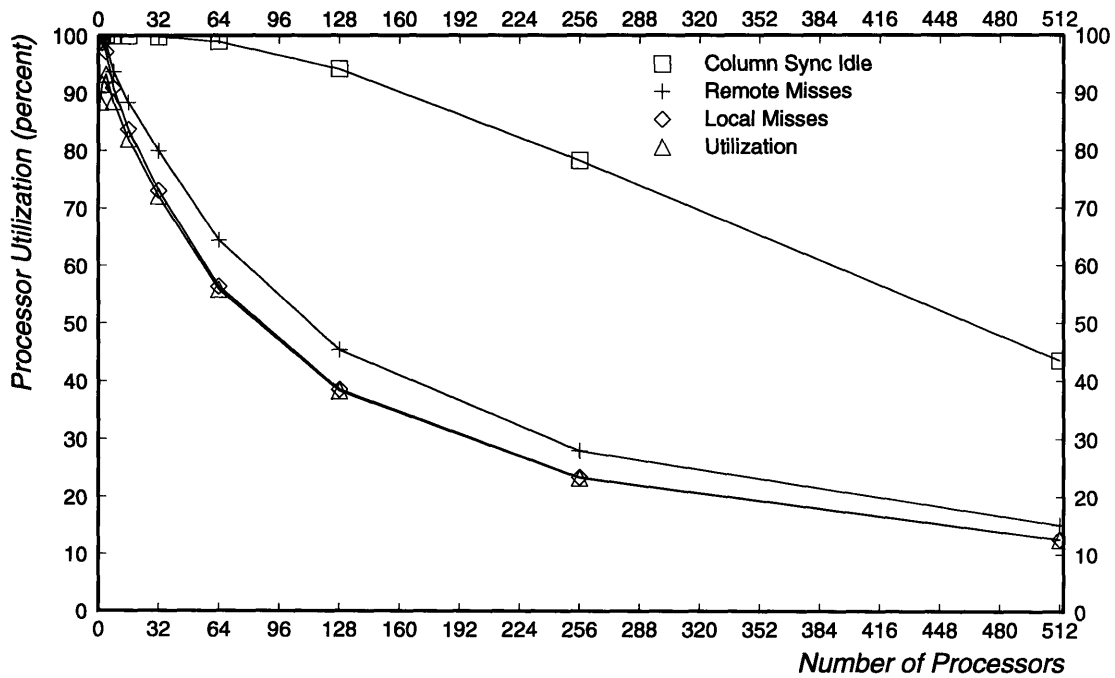


Figure 5-8: *Ideal:w15.1 Coarse Grain Utilization Curve*

### 5.1.6 Ideal:Fine Grain w/Task Redistribution–Utilization Curves

The following figures show the utilization curves for a fine grain simulation with task redistribution. In comparing the curves with their respective coarse grain versions, we see that overall idle time has been nearly eliminated. The *Column Sync Idle* time has been substantially reduced. Remote misses, on the other hand, have increased dramatically. This suggests that the performance on a real machine will depend on the remote latency.

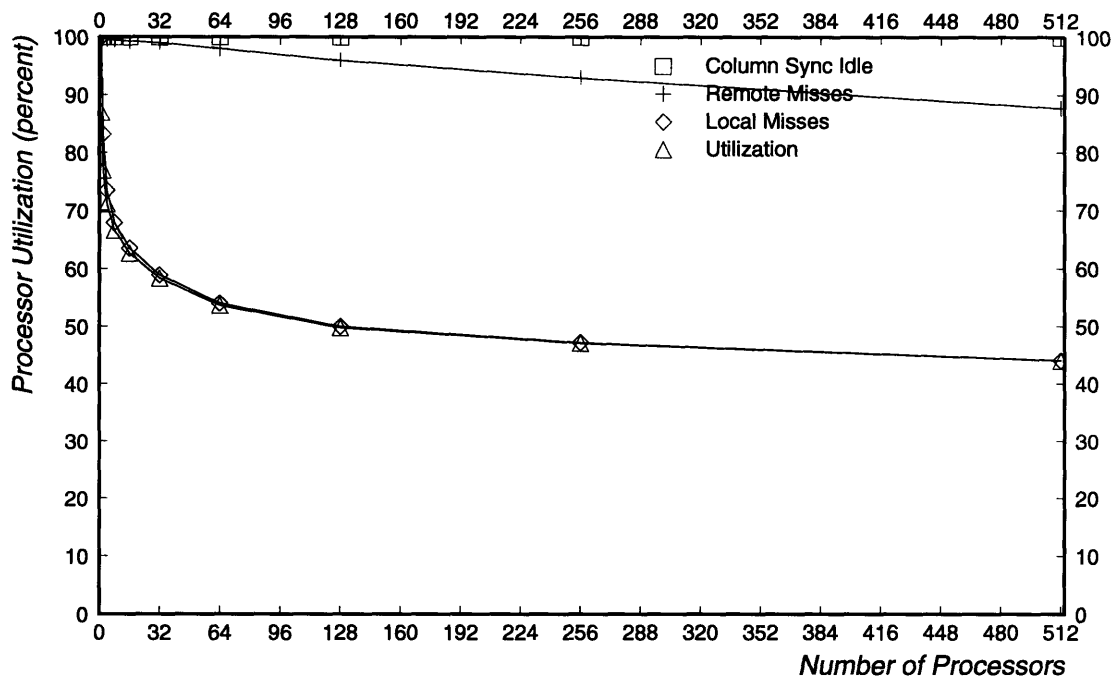


Figure 5-9: *Ideal:bcstk14 Fine Grain w/Task Redistribution Utilization Curve*

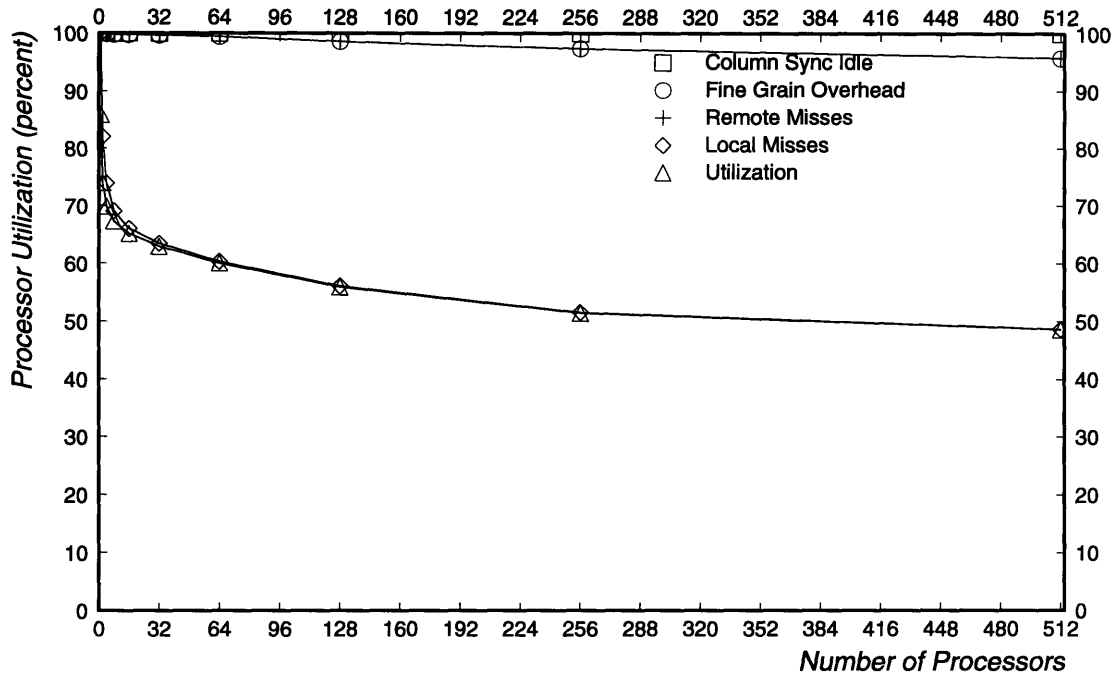


Figure 5-10: *Ideal:bcstkl5 Fine Grain w/Task Redistribution Utilization Curve*

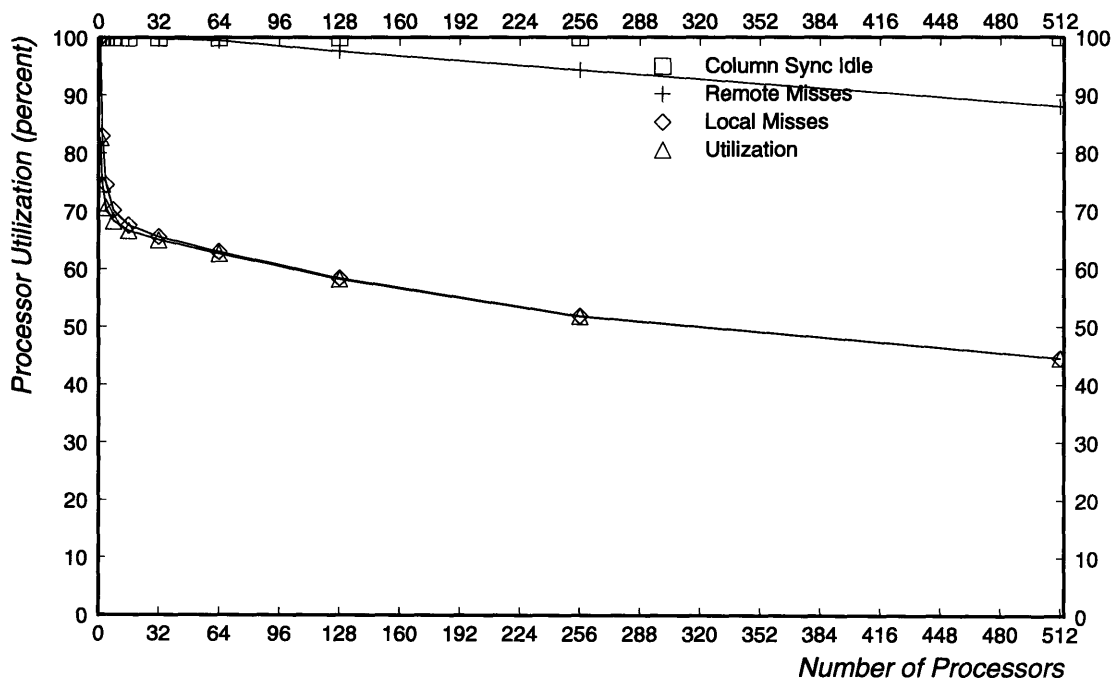


Figure 5-11: *Ideal:d750.O Fine Grain w/Task Redistribution Utilization Curve*

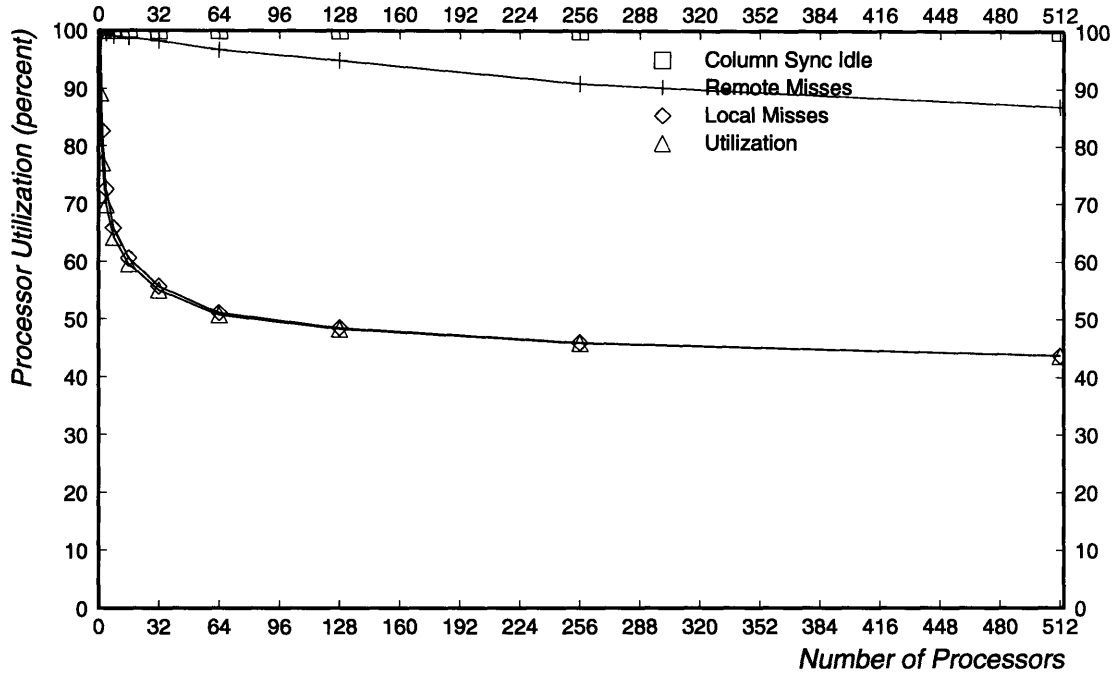


Figure 5-12: *Ideal:lshp Fine Grain w/Task Redistribution Utilization Curve*

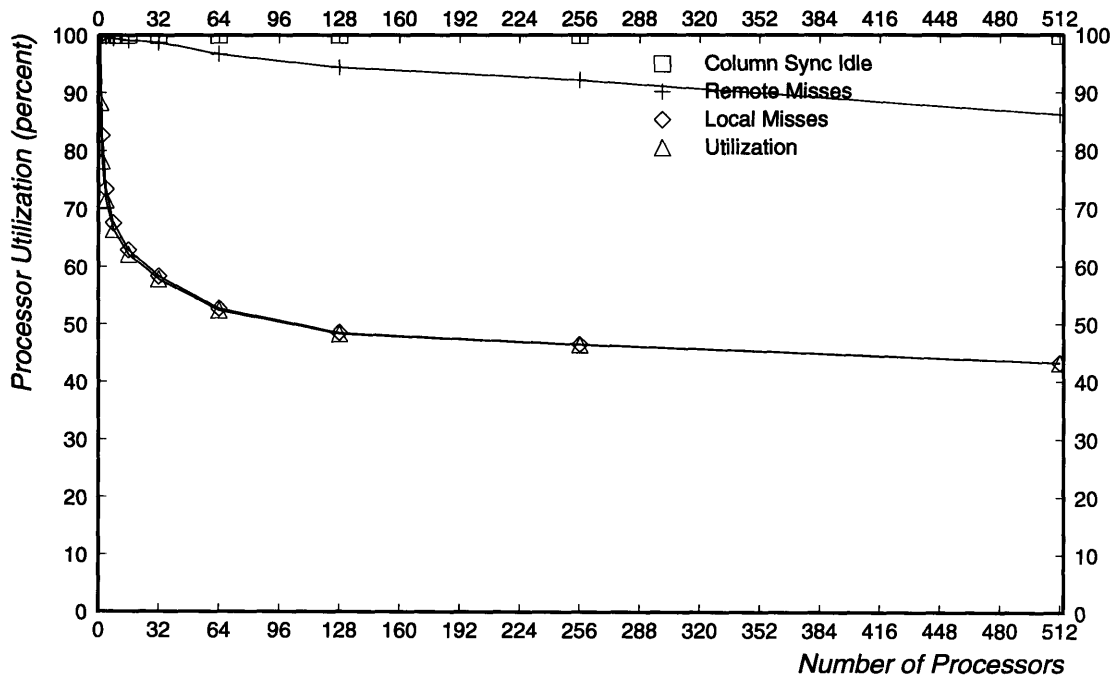


Figure 5-13: *Ideal:w15.1 Fine Grain w/Task Redistribution Utilization Curve*

## 5.2 Practical Machine Results

### 5.2.1 Coarse–Single Processor Run Times

*CSIM* configured as follows produced the results in table 5.3 for single processor run times of the coarse grain model. All appropriate overheads have been reintroduced.

Parameter	Value
P	1
CACHE_SIZE	64 KB
FLOAT_OP	5 cycles
CACHE_ACCESS	2 cycles
LOCAL_ACCESS	15 cycles
REMOTE_ACCESS	n/a
POLLING_OVERHEAD	n/a
FINE_GRAIN_OVERHEAD	n/a
BACKOFF_TIME	n/a

Matrix	1 Processor Run Time
bcstk14	2.18823e+08
bcstk15	3.67071e+09
d750.O	1.63593e+09
lshp.0	1.00731e+08
w15.1	1.06694e+08

Table 5.3: *Coarse–Single Processor Run Times*

### 5.2.2 Coarse–Speedup Curves

Speedup curves, shown in figure 5-14 were produced relative to single processor coarse grain performance. The simulator was configured as follows:

Parameter	Value
P	Number of Processors
CACHE_SIZE	64 KB
FLOAT_OP	5 cycles
CACHE_ACCESS	2 cycle
LOCAL_ACCESS	15 cycles
REMOTE_ACCESS	$T = \sqrt{P} + 15$
POLLING_OVERHEAD	0.10
FINE_GRAIN_OVERHEAD	n/a
BACKOFF_TIME	n/a

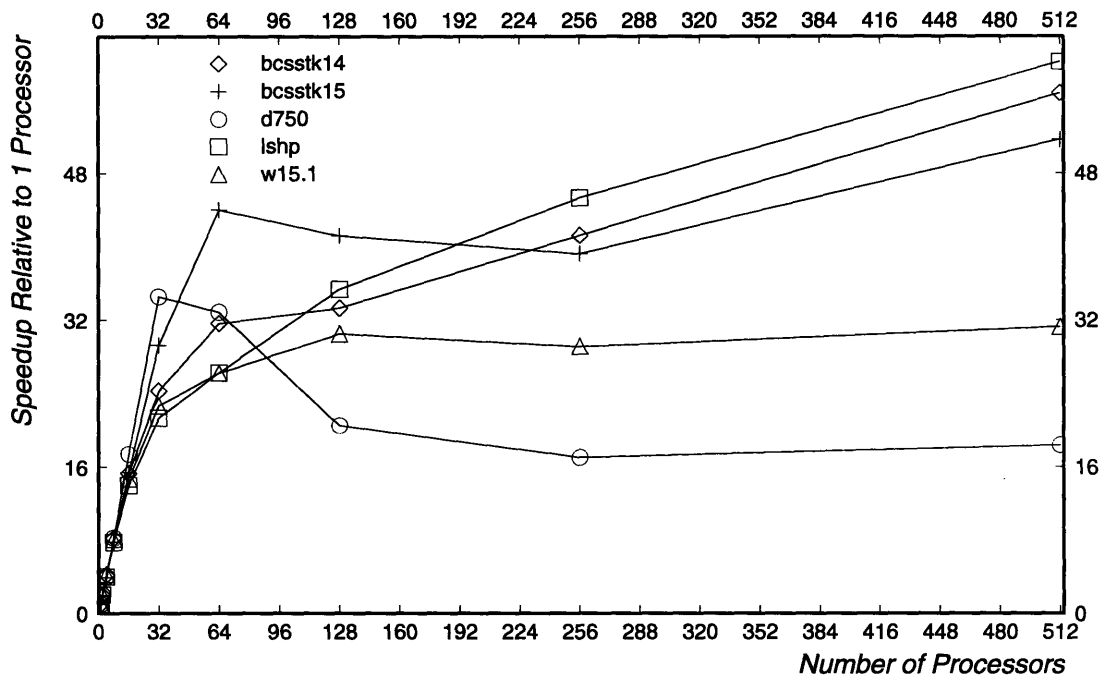


Figure 5-14: Coarse Grain Speedup

Figure 5-14 shows little speedup at large machine sizes for the coarse grain version. The best performance is shown by the lshp matrix, which gets a speedup of about 60 on 512 processors. The worst is shown by the d750 matrix. This matrix is a dense matrix and its performance is dependent on column distribution. As the utilization curve in figure 5-20 shows, this matrix requires few remote accesses for small machine sizes. But as the machine size is increased, the locality gets completely destroyed—resulting in

extremely poor performance.

### 5.2.3 Comparison of *CSIM*'s output to Rothberg

Rothberg presents speedup numbers from a simulated multiprocessor for *lshp*, *bcsstk15* and *dense750*. His speedup numbers are relative to single processor run times of the parallel code. Rothberg's numbers in the table below were read from a graph—so they are approximate.

Matrix	Processors	Rothberg Speedup	<i>CSIM</i> Speedup
<i>bcsstk15</i>	16	14	15.3
<i>bcsstk15</i>	32	24	29.3
<i>d750.O</i>	16	16	17.4
<i>d750.O</i>	32	33	34.5
<i>lshp.0</i>	16	7.5	14.0
<i>lshp.0</i>	32	10.5	21.3

Table 5.4: *Comparison of CSIM with Rothberg's results*

*CSIM* performs comparably with Rothberg's simulator on *d750* since it has very good locality for small machine sizes. *Lshp* doubles in performance. Why is *d750* comparable and *lshp* different? The difference is that *lshp* requires many more remote accesses than *d750*. The remote latency model under *CSIM* gives lower latency numbers than Rothberg's model. As a result, the matrix runs about twice as fast. *Bcsstk15* shows similar speedup.

### 5.2.4 Fine Grain no Task Redistribution Speedup

Figure 5-15 shows the performance of the fine grain application without any task redistribution. Speedup is relative to the 1 processor coarse grain version. All matrices perform better. *d750* especially improves. Even though the fine grain memory accesses take 10% longer to complete, the extra parallelism exposed more than overcomes this penalty. Since this version is not better than fine grain w/task redistribution, utilization curves for this implementation will not be shown. *CSIM* was configured as follows:



Parameter	Value
P	Number of Processors
CACHE_SIZE	64 KB
FLOAT_OP	5 cycles
CACHE_ACCESS	2 cycle
LOCAL_ACCESS	15 cycles
REMOTE_ACCESS	$T = \sqrt{P} + 15$
POLLING_OVERHEAD	n/a
FINE_GRAIN_OVERHEAD	0.10
BACKOFF_TIME	see equation

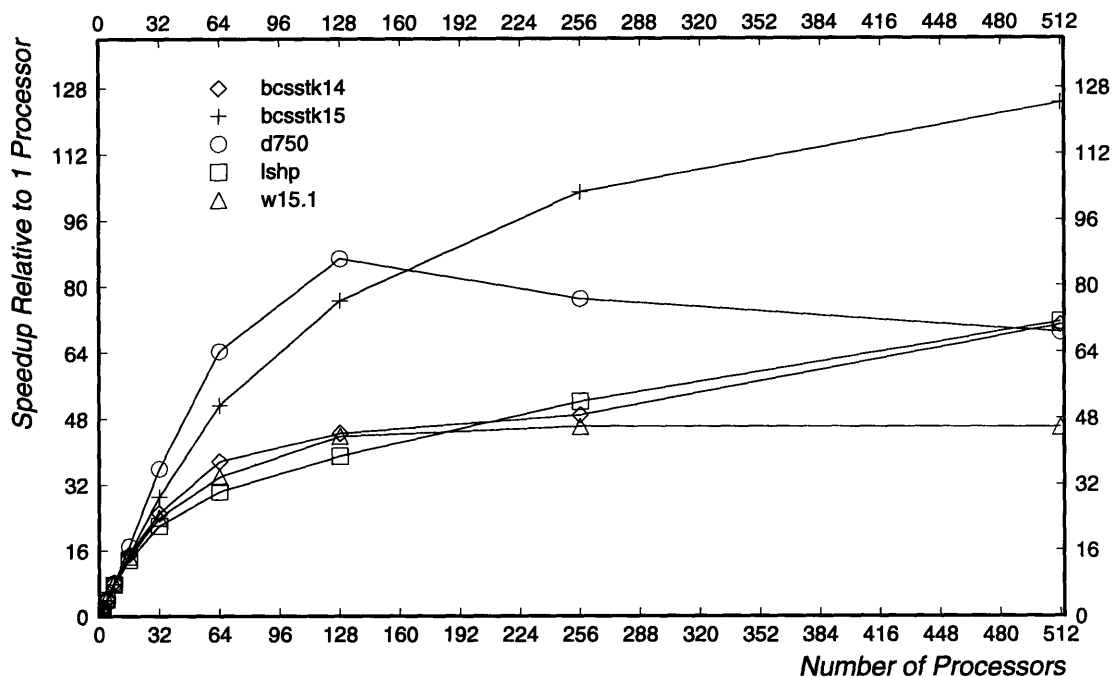


Figure 5-15: *Fine Grain–NO Task Redistribution Speedup*

### 5.2.5 Fine Grain w/Task Redistribution Speedup

Figure 5-16 shows the performance of the fine grain code with the tasks redistributed. Speedup is relative to the 1 processor coarse grain version. These results are much better than the coarse grain and the fine grain without task redistribution version. The

simulator was configured as follows. Figure 5-17 shows the ratio of the fine grain w/task redistribution runtime to the coarse grain runtime.

Parameter	Value
P	Number of Processors
CACHE_SIZE	64 KB
FLOAT_OP	5 cycles
CACHE_ACCESS	2 cycle
LOCAL_ACCESS	15 cycles
REMOTE_ACCESS	$T = \sqrt{P} + 15$
POLLING_OVERHEAD	n/a
FINE_GRAIN_OVERHEAD	0.10
BACKOFF_TIME	see equation

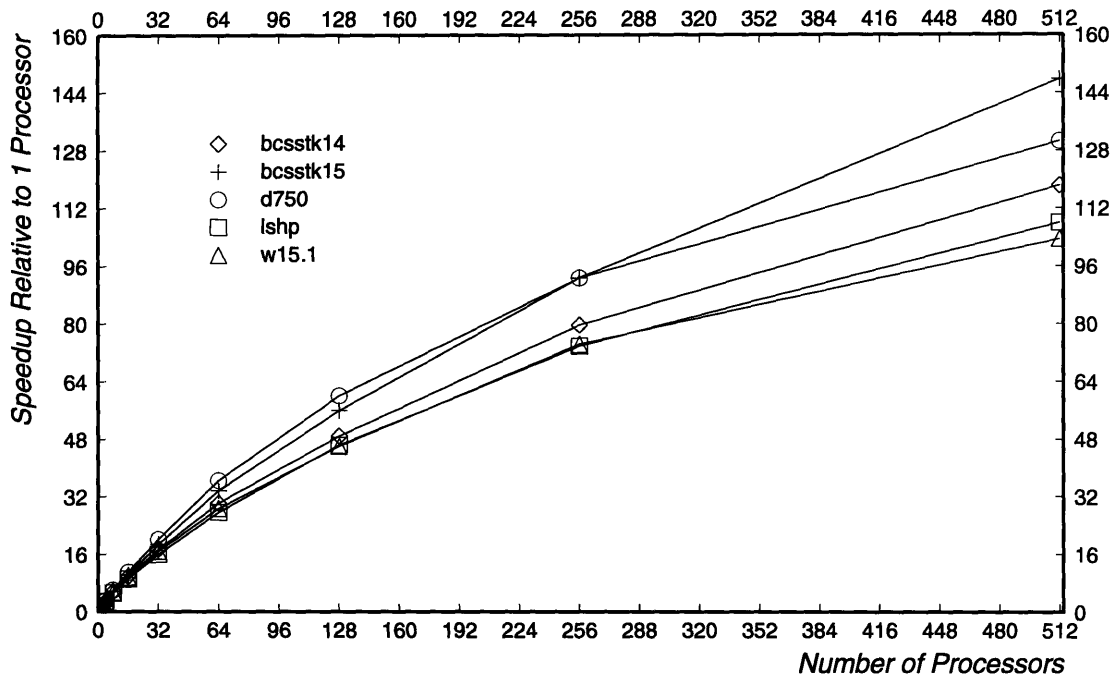


Figure 5-16: *Fine Grain w/Task Redistribution Speedup*

It is interesting that all speedup curves in figure 5-16 follow a similar path. Why is that? It is worthwhile to look at what the processors are spending their time on to answer this question.

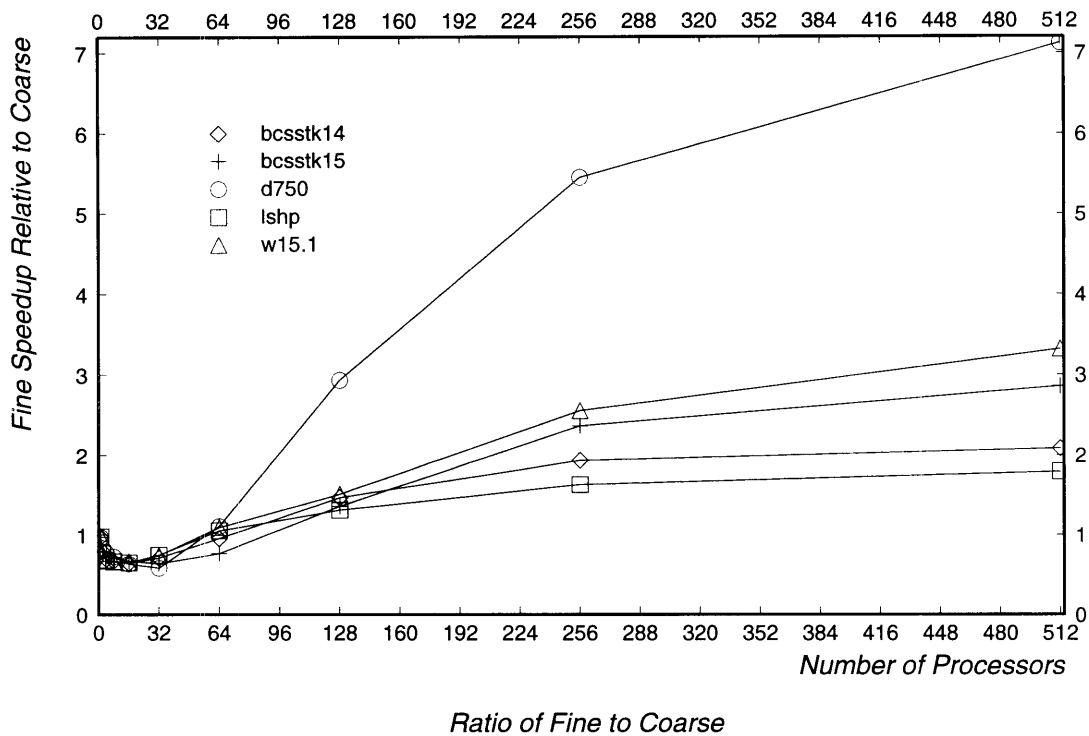


Figure 5-17: *Fine Grain w/Task Redistribution Speedup*

### 5.2.6 Coarse-Utilization Curves

The following set of figures show the processor utilization for the coarse grain version of the application. On the X-axis is the number of processors. The Y-axis is a cumulative plot of the average time spent on utilization, local misses, remote misses, and overhead. The time above the highest line is idle time.

All of these curves show a similar distribution of time as the machine size is increased. First, the overall idle time from lack of parallelism increases. Second the idle time from consumers waiting for producers, *Column Sync Idle*, increases. Local misses become negligible as the machine size increases.

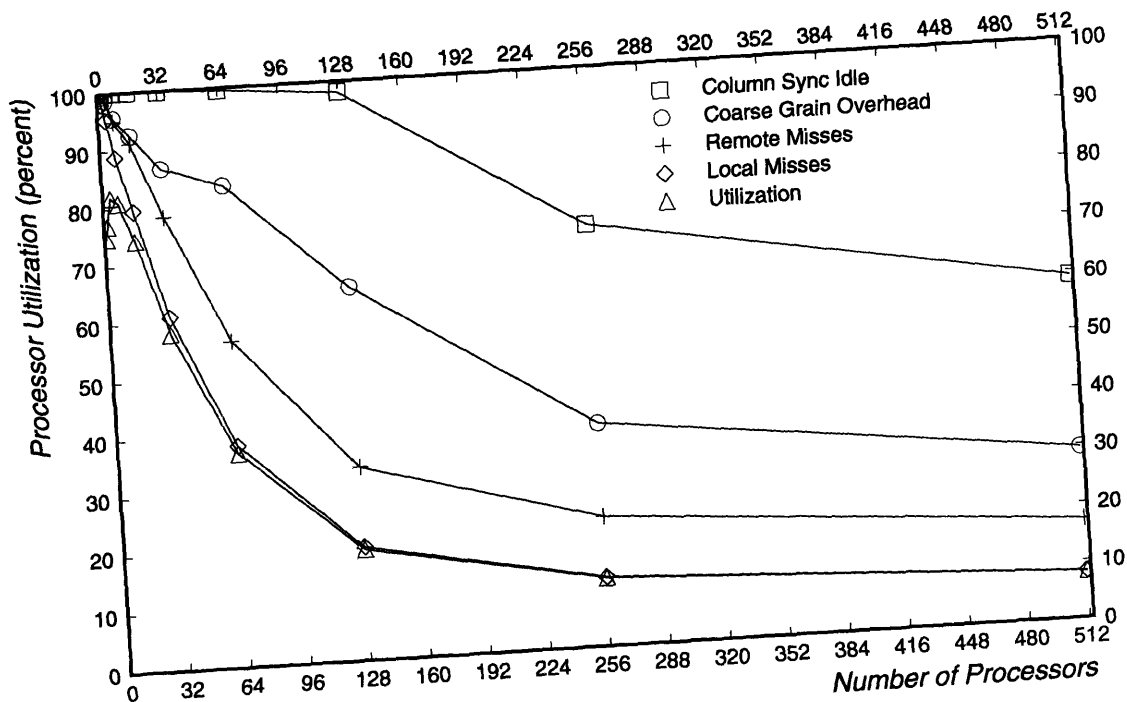


Figure 5-18: *bcsstk14* Coarse Grain Utilization Curve

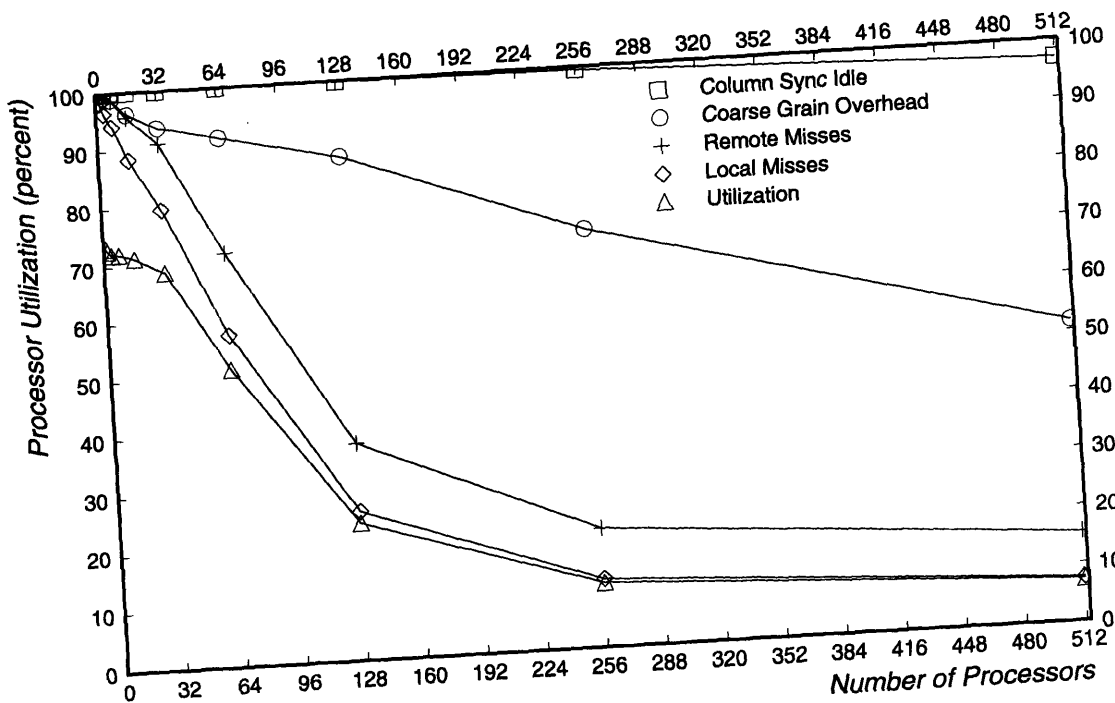


Figure 5-19: *bcsstk15* Coarse Grain Utilization Curve

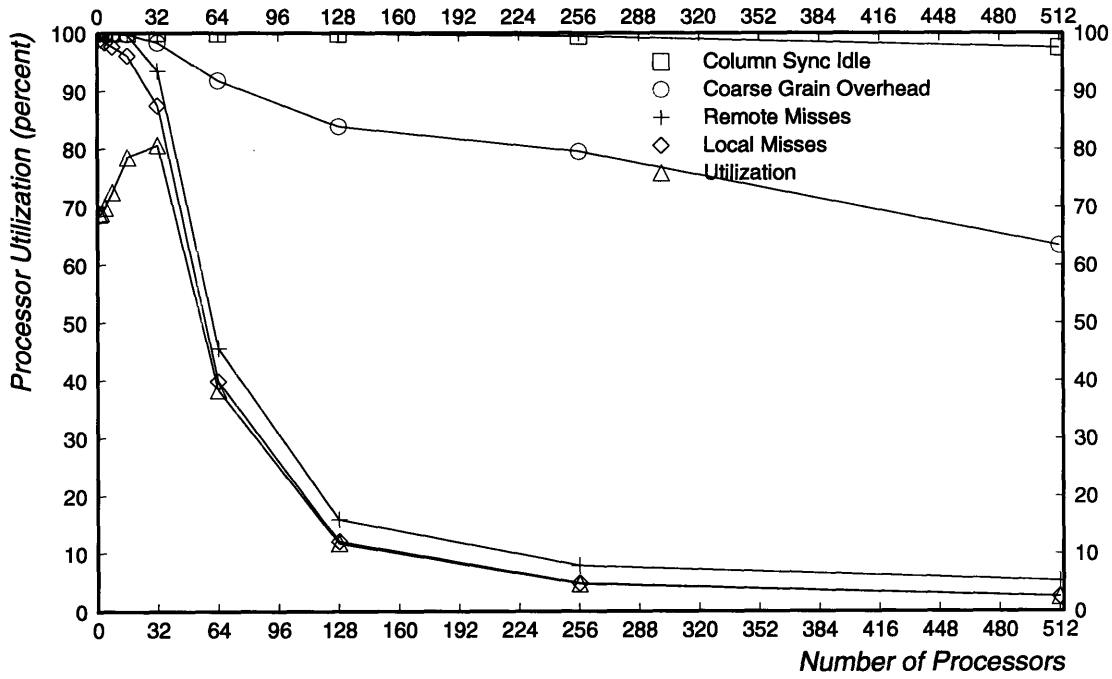


Figure 5-20: *d750.O* Coarse Grain Utilization Curve

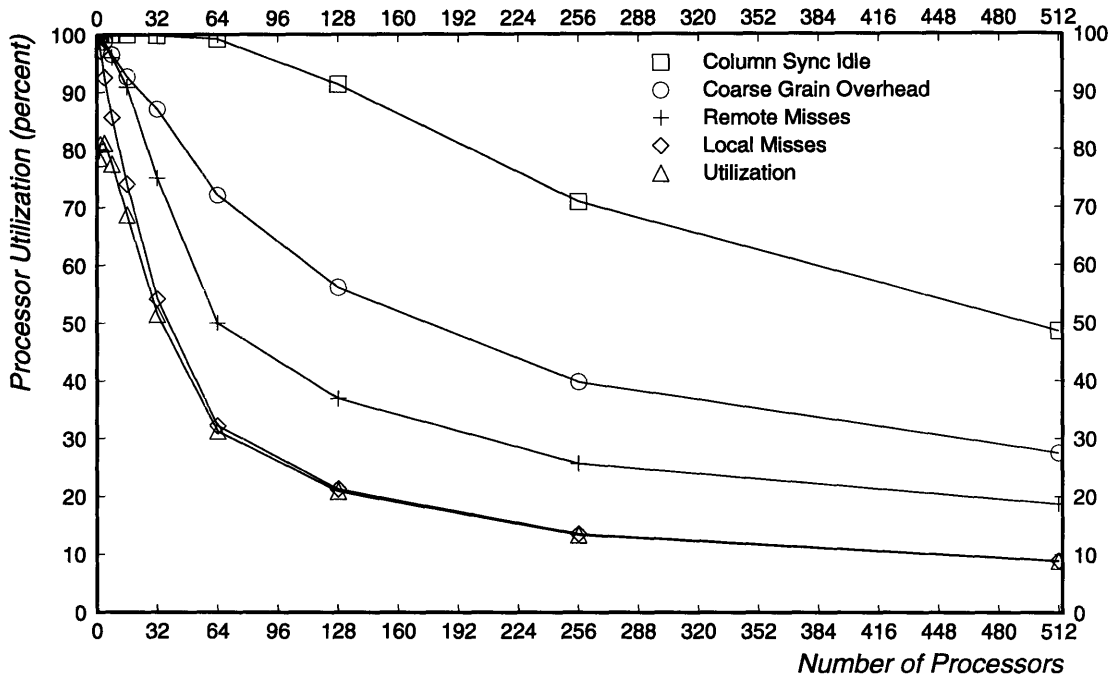


Figure 5-21: *lshp* Coarse Grain Utilization Curve

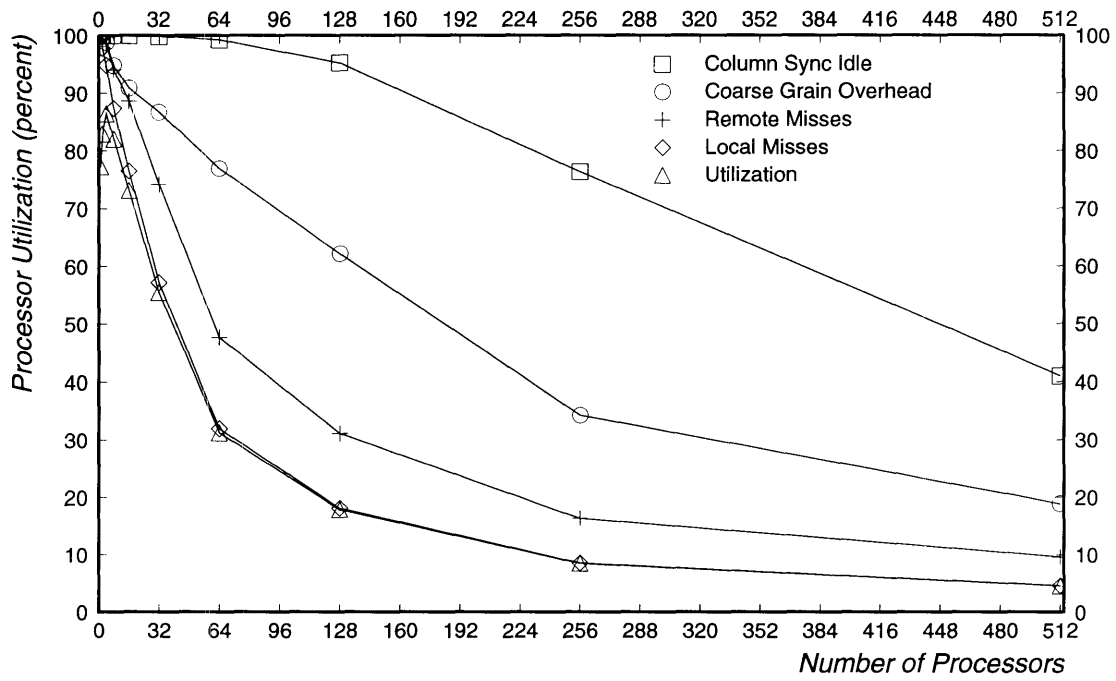


Figure 5-22: w15.1 Coarse Grain Utilization Curve

### 5.2.7 Fine Grain w/Task Redistribution–Utilization Curves

The following set of figures show the processor utilization for the fine grain w/task redistribution version of the application. These figures look very different from the coarse grain utilization curves. The overall idle time from lack of parallelism is now negligible. So, the fine grain version has much much better load balancing. A substantial amount of the time is spent on fine-grain overhead. This is the amount of time the machine takes to service a successful synchronization along with the backoff delay from polling. By far, the biggest difference is the increased percentage of remote references. This was expected as the task redistribution destroyed data locality.

It is now possible to answer why the speedup curves shown in figure 5-16 all seem to follow a similar path. Almost all the operations performed in the application are *cmop* operations. With fine grain synchronization and task redistribution, there is almost perfect load balancing. So the speedup curves follow the speedup of one *cmop* operation. The amount of time a *cmop* operation takes depends on the remote miss latency. This latency scales with the square root of the number of processors. So ideally, the order of the speedup curve for the fine grain application is  $O(\sqrt{P})$ .

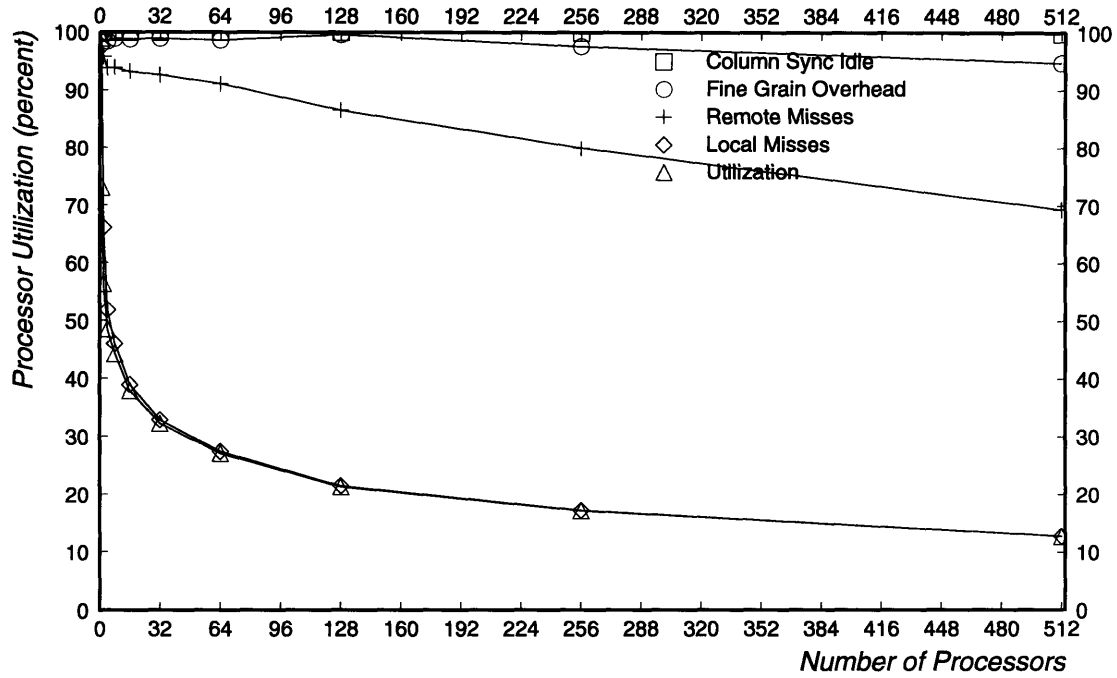


Figure 5-23: *bcsstk14 Fine Grain w/Task Redistribution Utilization Curve*

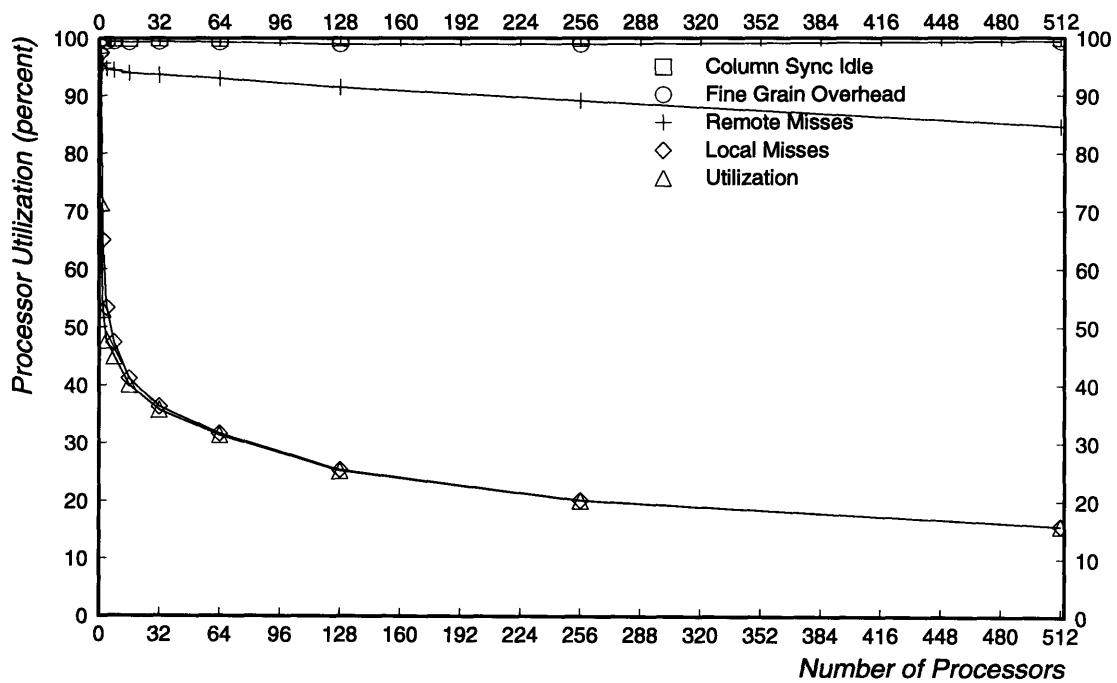


Figure 5-24: *bcsstk15 Fine Grain w/Task Redistribution Utilization Curve*



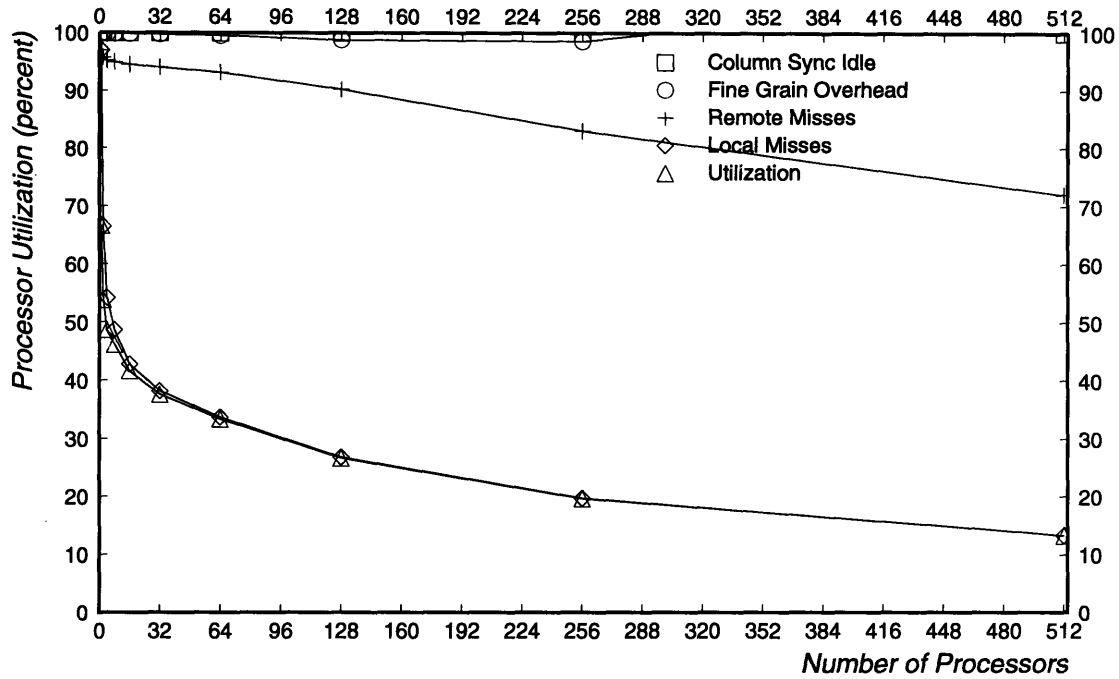


Figure 5-25: *d750.O Fine Grain w/Task Redistribution Utilization Curve*

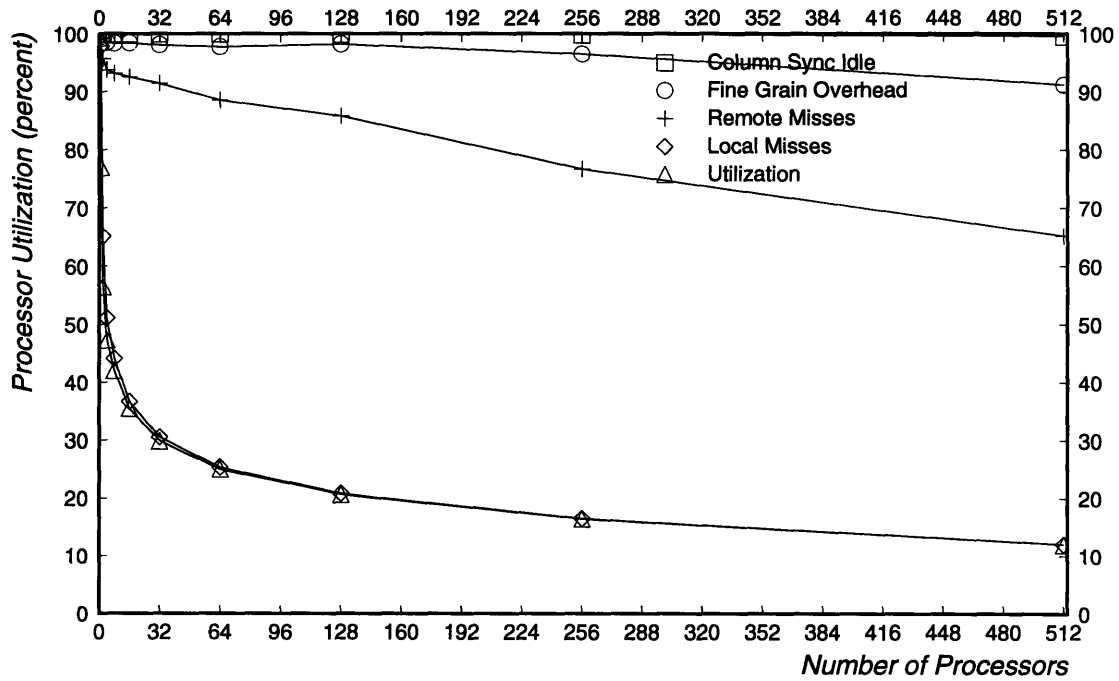


Figure 5-26: *lshp Fine Grain w/Task Redistribution Utilization Curve*

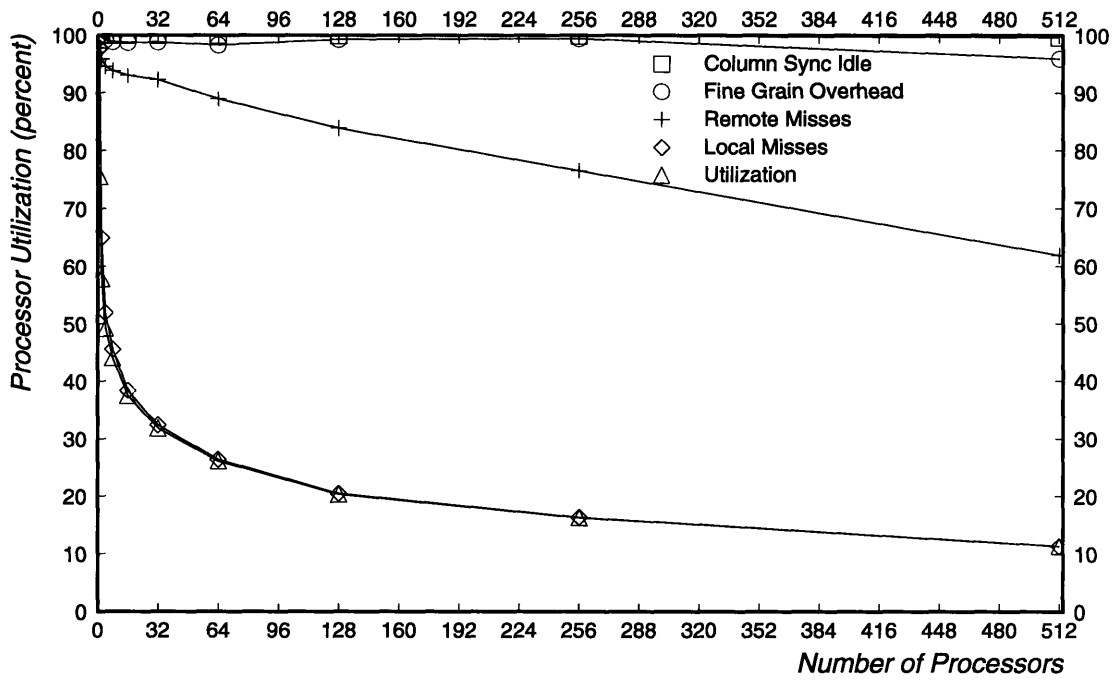


Figure 5-27: *w15.1 Fine Grain w/Task Redistribution Utilization Curve*

# Chapter 6

## Conclusions

Performance of the fan-out algorithm for Cholesky factorization can be improved using fine grain synchronization. By replacing locks on columns with locks on element, *cmod* operations can start without waiting for their *cdiv* operations to finish. This optimization gives moderate speedup over coarse grain fan-out (column locks) factorization. The performance increased from 16% percent to 45% over coarse grain for the test matrices.

Performance can be improved even more by redistributing the *cmod* and *cdiv* operations. A round robin task redistribution scheme is sufficient. With this new distribution, *cmod* operations on the same column run in parallel. The resulting speedup is much higher than fine grain without task redistribution. Speedup increased from 100% to 400% over the coarse grain version for the test matrices. It increased from 71% to 123% over the fine grain without task redistribution.

Since the processor load is close to balanced for the fine grain w/task redistribution, the overall speedup is defined by the time it takes to perform a *cmod* operation. This time is bounded by communication and can be computed as having order of  $O(\sqrt{P})$  for a 2-D network. This value is identical to the scaling of communication cost in the model. Output from *CSIM* shows speedup curves to be of order  $O(\sqrt{P})$ .

Due to the nature of the coarse grain factorization, *CSIM*'s results should be comparable to a real machine. Since there are no cache coherency issues in the computation, the simulator allows remote values to be cached locally. *CSIM* supports a polling model for column locks with a polling overhead. Multiple readers of a column are serviced

sequentially by the owner. *CSIM*'s remote read latency model does not take into account network contention.

The fine grain simulation has an important limitation—it does not model cache coherency. Remote accesses are not cached. A backoff delay model was used with the delay time computed at exactly one cache line. With this delay time, there should be no thrashing on a cache line by two writers. Each additional writer to the cache line is also delayed by the delay time—again insuring no thrashing. This model does not take into account any overhead of the cache coherency protocol. Using caches, should give better performance. Though *CSIM*'s model attempts to minimize protocol overhead by avoiding thrashing, it remains to be verified on a real machine.

In an ideal machine, fine grain factorization gives far better performance than coarse grain factorization. Though *CSIM* is a good tool for getting rapid results for a multiprocessor cholesky factorization run, it does not effectively model all aspects of a real machine. Most important are network contention and cache coherency. Without writing a complete multiprocessor simulator, it is difficult to model both of these issues. The Alewife machine will soon be available. The coarse grain and the fine grain cholesky factorization algorithms will be run on the machine to test the effectiveness of each and verify the *CSIM* models.

# Bibliography

- [1] A. Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.
- [2] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.
- [3] David Chaiken. NWO User's Manual. ALEWIFE Memo No. 36, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1993.
- [4] A. George, M. Heath, and J.W.H. Liu. Sparse cholesky factorization on local-memory multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 9(2):327–340, March 1988.
- [5] John R. Gilbert and Hjalmtyr Hafsteinsson. Parallel solution of sparse linear systems. Technical Report 87-893, Cornell University, December 1987.
- [6] John R. Gilbert and Robert Schreiber. Highly parallel sparse cholesky factorization. *SIAM Journal of Scientific and Statistical Computing*, 13(5):1151–1172, September 1992.
- [7] David Kranz, Beng-Hong Lim, and Anant Agarwal. Low-cost Support for Fine-Grain Synchronization in Multiprocessors. To appear in *Multithreading: A Summary of the State of the Art*, Kluwer Academic Publishers, 1992. Also available as MIT Laboratory for Computer Science TM-470, June 1992., January 1992.

- [8] P. Sreenivasa Kumar, M. Kishore Kumar, and A. Basu. Parallel algorithms for sparse triangular system solution. *Parallel Computing*, 19(2):187–96, February 1993.
- [9] Edward Rothberg and Anoop Gupta. A comparative evaluation of nodal and supernodal parallel sparse matrix factorization: Detailed simulation results. Technical Report STAN-CS-90-1305, Stanford University, Department of Computer Science, Stanford CA 94305, February 1990.
- [10] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse cholesky factorization. Technical Report STAN-CS-92-1438, Stanford University, Department of Computer Science, Stanford CA 94305, July 1992.
- [11] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.
- [12] Sesh Venugopal, Vijay K. Naik, and Joel Saltz. Performance of distributed sparse cholesky factorization with pre-scheduling. In *Proceedings of Supercomputing '92*, November 1992.
- [13] Donald Yeung and Anant Agarwal. Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient. In *Principles and Practices of Parallel Programming, 1993*, pages 187–197, San Diego, CA, May 1993. IEEE. Also as MIT/LCS-TM 479, October 1992.
- [14] G. Zhang and H.C. Elman. Parallel sparse cholesky factorization on a shared memory multiprocessor. *Parallel Computing*, 18(9):1009–22, September 1992.