PARALLEL CHOLESKY FACTORIZATION ON A MULTIPROCESSOR*

Alan George
University of Waterloo
Waterloo, Ontario, Canada

Michael T. Heath
Oak Ridge National Laboratory
Oak Ridge, Tennessee, U.S.A.

Joseph Liu
York University
Downsview, Ontario, Canada

Research Report CS-84-49
November 1984

Parallel Cholesky Factorization on a Multiprocessor*

Alan George

University of Waterloo
Waterloo, Ontario, Canada

Michael T. Heath

Oak Ridge National Laboratory
Oak Ridge, Tennessee, U. S. A.

Joseph Liu

York University
Downsview, Ontario, Canada

ABSTRACT

A parallel algorithm is developed for Cholesky factorization on a multiprocessor. The algorithm is based on self-scheduling of a pool of tasks. The subtasks in several variants of the basic elimination algorithm are analyzed for potential concurrency in terms of precedence relations, work profiles, and processor utilization. This analysis is supported by simulation results. The most promising variant, which we call column-Cholesky, is identified and implemented for the Denelcor HEP multiprocessor. Experimental results are given for this machine.

*Research supported in part by the Canadian Natural Sciences and Engineering Research Council under grants A8111 and A5509, by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems Inc., and by the U.S. Air Force Office of Scientific Research under contract AFOSR-ISSA-84-00056.

1. Introduction

Designing parallel algorithms amounts to deciding how to break up a given computational problem into subtasks that can be carried out concurrently on some given number of processors. The optimal size or number of these subtasks, usually referred to as the granularity of the parallel algorithm, and the appropriate type of communication of data or control information among them are strongly dependent on the target machine architecture. Our specific purpose in this paper is to present a parallel algorithm for Cholesky factorization on a multiprocessor. A larger aim is to exhibit a mode of analysis and a computational methodology that are applicable to finding and exploiting potential parallelism in a broad range of problems, algorithms, and computer architectures.

The computational paradigm we employ is that of a *pool of tasks* whose parallel execution is governed by a *self-scheduling* discipline. A pool of tasks may range from a relatively random, heterogeneous collection having no strong sense of order or precedence among them to a systematic sequence having a well defined order, such as the successive rows or columns of a matrix. In any case we will assume that the tasks are assigned some well ordered sequence of task numbers or task id's, whether naturally arising or more arbitrarily chosen. Some guidance as to what constitutes a good ordering of tasks will emerge from our analysis of precedence relations and work profiles.

In some parallel algorithms, specific tasks are mapped onto specific processors in advance of initiating the computation, and therefore effective load balancing among the processors requires that the tasks be reasonably uniform in size. Self-scheduling can be viewed as a technique for automatic and dynamic load balancing that does not necessarily require uniformly sized tasks. In self-scheduling, p processes are invoked to perform a job consisting of T tasks ($p \leq T$). When a given process completes an assigned task, it checks whether any unassigned tasks remain in the pool, and if so it is assigned the next one. Thus, if a processor happens to have drawn a relatively small task, this simply means that it will become free to take on yet another task from the pool sooner than a processor occupied by a larger one. In this way all of the processors tend to be kept busy even if the tasks vary in their computational difficulty. Similar advantages are gained from this approach when processors having different computational speeds are employed; i.e., faster processors tend to share a greater portion of the total work load.

As noted above, tasks are claimed by free processors in an order specified by the task numbers or id's. The primary reason tasks are not selected from the pool at random is that we must satisfy any precedence relations that may hold among the tasks. Between two tasks, say task a and task b, there are several possible types of precedence relations affecting their potential parallel execution. Three that are of interest to us in this context are:

Type 1: task a must finish before task b can begin;

Type 2: task a must finish before task b can finish;

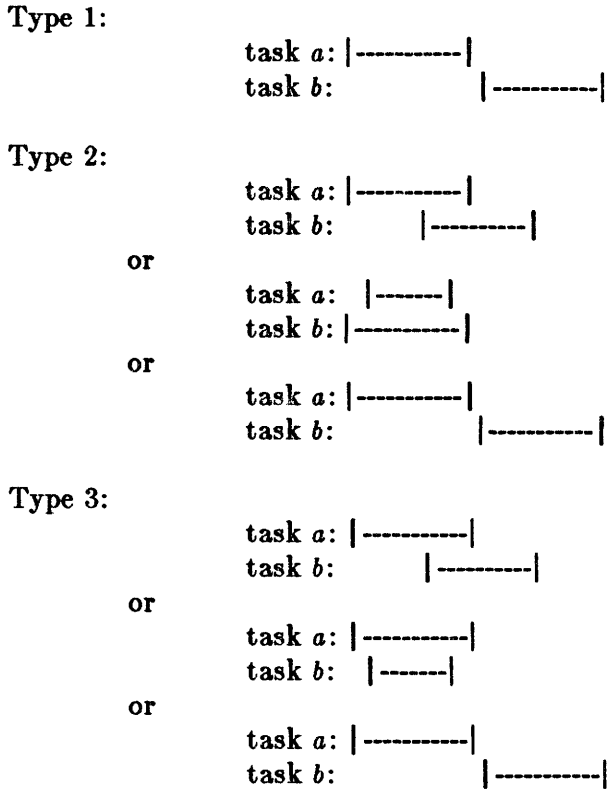
Type 3: task a must begin before task b can begin.

The first type permits only serial execution of the two tasks, but the other two types permit at least some degree of concurrent execution. Obviously the order chosen for the pool of tasks should be consistent with any such precedence relations in order to

Table of Contents

take maximum advantage of parallelism. We shall see examples of all three types of precedence relations in various forms of Cholesky factorization.

The implication of these precedence relations can be depicted in terms of the following horizontal time-line diagrams:



This self-scheduled pool-of-tasks approach is flexible in that it is not strongly dependent on the number of processors available, but it is best suited to large-to-medium grained parallelism, which, in the present context, means problems for which the total number of tasks T exceeds the number of processors p by a substantial margin. Since the pool of tasks must be made available to each processor, this paradigm is not appropriate for all parallel architectures, but it is appropriate for several important ones, including systems having a significant amount of shared global memory and systems having a master processor that can dispatch tasks to a number of other processors.

The concept of self-scheduling, at least in the sense we are using it, seems to be due to the designers of the Denelcor HEP multiprocessor, principally Burton Smith. It is discussed in several of Harry Jordan's early unpublished manuscripts on programming methodology for the HEP. It is mentioned briefly in his published paper [7] and is given as an example in the HEP Fortran manual [2].

2. Cholesky Factorization and Self-Scheduling Loops

2.1. Overview

We turn our attention now to the specific problem we wish to address, computing the Cholesky factorization of a symmetric positive definite matrix of order n. With a single processor, the amount of work required is $O(n^3)$ arithmetic operations. With more processors, the total number of arithmetic operations performed remains the same, but the total execution time will be reduced as a result of sharing the work among the processors, even though some additional overhead may be introduced by necessary communication or synchronization among the tasks and processors. If the number of processors available is very large, say $O(n)$ or $O(n^2)$, then the work performed by each processor will be correspondingly small, perhaps just a few arithmetic operations. Some appropriate parallel algorithms for this case include systolic arrays [1] and data-flow [10], and the corresponding architectures involve very simple processors with limited communication among them and only local memory. We are interested in the case $p \ll n$. Most existing machines with architectures suitable for supporting our approach have a number of general purpose processors in the range $4 \leq p \leq 64$.

It has long been known that there are numerous ways of organizing Gaussian elimination (Crout, Doolittle, etc.), each of which has advantages in specific circumstances (memory access patterns, vectorization, etc.). A systematic study of these variations on Gaussian elimination and their implications for particular computer architectures is given by Dongarra, Gustavson and Karp [3]. Their formulation, which serves as the point of departure for our analysis, is given by the following generic algorithm describing Gaussian elimination for a given $n \times n$ matrix A:

```
for _____________
    for _____________
        for _____________
            a_ij := a_ij - (a_ik * a_kj)/a_kk
        end
    end
end
```

In this formulation, a particular algorithm results from filling the blanks with the limits on the loop parameters i, j, and k in some order. A minor variant of this generic algorithm can be used to describe the Cholesky factorization LL^T for symmetric positive definite matrices. In the latter case we can take advantage of symmetry (i.e., $a_{kj} = a_{jk}$) in order to access only one triangle of the matrix (say the lower triangle) and to access the matrix by rows or columns, as desired. Our discussion of Cholesky factorization will be based on this formulation. Six different algorithmic forms of Cholesky factorization can be obtained, depending on the arrangement of the three loop indices i, j, and k. Dongarra et al [3] have appropriately called these the

ijk, *ikj*, *jik*, *jki*, *kij*, and *kji*

forms of the algorithm. In this paper, we consider the self-scheduling of tasks for parallel execution in these different forms of Cholesky decomposition. To aid our discussion, we define the following three classes of factorization forms (see [5, pp. 17-20] for a detailed discussion):

Row-Cholesky: The rows of the Cholesky factor L are computed one by one. This formulation is sometimes referred to as the *bordering method.*

Column-Cholesky: The columns of the Cholesky factor L are computed one by one using the previously computed columns of L. This is sometimes called the *inner-product* formulation of symmetric decomposition.

Submatrix-Cholesky: The submatrix modifications from columns of L are applied one by one to the remaining submatrix to be factored. This is sometimes called the *outer-product* formulation of symmetric decomposition.

Fig. 2.1 illustrates these three forms of Cholesky decomposition.

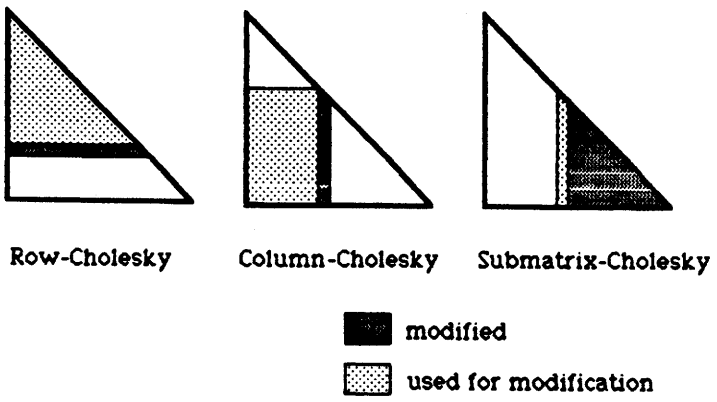


Fig. 2.1. Three Forms of Cholesky Decomposition

We shall study the following scheduling loops:

```
for i := 1 to n          for j := 1 to n          for k := 1 to n
  schedule Trow(i)         schedule Tcol(j)         schedule Tsub(k)
end                      end                      end
```

for the three different forms, where $Trow(i)$ is the task to compute the i^{th} row L_{i*} of the Cholesky factor L, $Tcol(j)$ is the task to determine the j^{th} column L_{*j}, and $Tsub(k)$ is the task to perform the submatrix modification from the k^{th} column of L. We shall consider the self-scheduling of these tasks for the three algorithms and compare their respective performance in a multiprocessor environment.

It should be emphasized that there are many other ways of setting up self-scheduling loops to perform Cholesky factorization, depending on how we split up the entire factorization into tasks. The three scheduling loops under consideration are logical ways of defining tasks, are appropriate for the level of granularity we wish to

exploit, and they serve to illustrate the various techniques used in §3 to compare different parallel algorithms.

2.2. Row-Cholesky

In order to compute the i^{th} row of the Cholesky factor, we require access to the previous $i-1$ rows of L. Computationally, these $i-1$ rows are used to do a lower triangular solve to determine L_{ij}, for $j=1,\ldots,i-1$. Then the diagonal element L_{ii} can be obtained from these computed entries of the i^{th} row. Depending on how the previous $i-1$ rows are accessed, whether by row or by column, we have the ijk or the ikj forms of factorization, respectively, as used by Dongarra et al [3]. Fig. 2.2 illustrates these two forms pictorially.

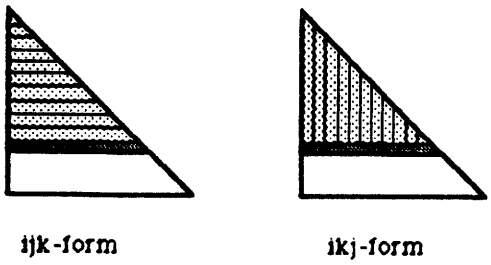


Fig. 2.2. Two Forms of Row-Cholesky

The i^{th} task $Trow(i)$ depends on results from all the previous $i-1$ tasks. It should be noted, however, that in the ijk form, since the first $i-1$ rows of L are being accessed row by row in the execution of task $Trow(i)$, the part involving the first r rows can be performed once the first r tasks $Trow(1),\ldots,Trow(r)$ to compute $L_{1*},\ldots,L_{r*}$ have been completed. In other words, although $Trow(i)$ uses results from $Trow(i-1)$, a major portion of these two tasks can be executed concurrently, except when $i \leq 2$.

Thus there is a Type 2 precedence relation among the n tasks

$$Trow(1) \rightarrow Trow(2) \rightarrow \cdots \rightarrow Trow(n)$$

since task $Trow(i)$ cannot be completed until $Trow(i-1)$ has finished, but $Trow(i)$ can begin before $Trow(i-1)$ is finished. Thus, the scheduling of these tasks on a number of parallel processors becomes potentially advantageous. The ikj form of row-Cholesky, on the other hand, leads to a Type 1 (i.e., serial) precedence relation among the tasks $Trow(*)$. Since we want to maximize parallelism, we shall henceforth use "row-Cholesky" to refer to the ijk form of Cholesky.

2.3. Column-Cholesky

To compute column j of the Cholesky factor, we require access to the rectangular submatrix enclosed (inclusively) by the j^{th} row and j^{th} column of L. This rectangular submatrix can be accessed either by row or by column, so that we have the *jik* and *jki* forms of Cholesky as used by Dongarra et al. These two forms are illustrated in Fig. 2.3.

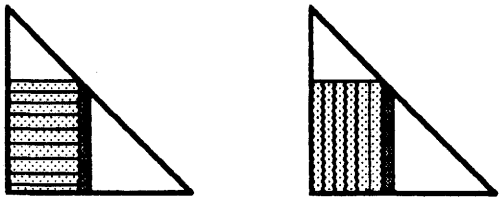


Fig. 2.3. Two Forms of Column-Cholesky

The *jki* form of column-Cholesky shares the same advantage as the *ijk* form in row-Cholesky. This may be attributed to the fact that the submatrix that modifies column j is being accessed column by column. Indeed, the same Type 2 precedence relation exists among the n tasks

$$Tcol(1) \rightarrow Tcol(2) \rightarrow \cdots \rightarrow Tcol(n)$$

so that there is a high degree of potential concurrency among these tasks. It makes sense, therefore, to schedule these tasks for a number of parallel processors. We shall use "column-Cholesky" to refer to this *jki* form. Note that in the *BLAS* (Basic Linear Algebra Subroutines [8]) terminology, the basic operation here is a *SAXPY*; that is, a computation of the form $Ax+y$, where A is a matrix, and x and y are vectors.

It should be pointed out that the parallel implementation of Cholesky factorization by Dongarra and Hiromoto [4] is the *jik* version. Since at the outer loop level of *jik* the precedence relation among the tasks $Tcol(*)$ is of Type 1 (i.e., serial), any parallelism must come within the inner loops, where the basic operation is a matrix-vector product. The latter operation, called a *GAXPY* in [3] is implemented in [4] as a set of inner products computed in parallel. More will be said about this implementation in §4.

2.4. Submatrix-Cholesky

To apply the modification from column k of the Cholesky factor, we need to modify entries in the submatrix as given by the remaining $n-k$ columns of the matrix. Again, the modification can be performed either by row or by column to give the *kij* and *kji* forms of Cholesky, respectively. Fig. 2.4 illustrates these two forms.

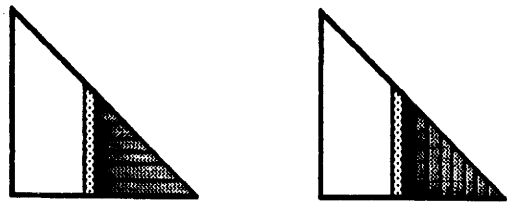


Fig. 2.4. Two Forms of Submatrix-Cholesky

The *kji* form is more appropriate for self-scheduling loops, for the same reason as in the case of *ijk* and *jki* forms. For our purpose, we shall use "submatrix-Cholesky" to refer to the *kji* version, in which modification is applied column by column. Again, the basic operation for this scheme is a *SAXPY*.

It is interesting to note that the n tasks $Tsub(*)$ are related in a rather different way. The task $Tsub(k)$ can start whenever the modifications to column k from the previous $k-1$ tasks are done. It is therefore possible that the task $Tsub(k)$ is completed before a task $Tsub(c)$, where $c < k$. Thus the precedence relation

$$Tsub(1) \rightarrow Tsub(2) \rightarrow \cdots \rightarrow Tsub(n)$$

is of Type 3.

Another notable difference is the possibility of modifying the same column by different tasks at the same time. Some mechanism for mutual exclusion must therefore be incorporated to avoid simultaneous updates. This problem does not arise in row-Cholesky and column-Cholesky, since the modifications to a particular row or column are performed by only one task, so that the updates are done serially by one processor. Finally, we note that the algorithm of O'Leary and Stewart [10] is associated with the *kji* form of submatrix-Cholesky, but with a finer grain of parallelism in that the inner loops are parallelized as well.

3. Analysis of Cholesky Factorization with Self-scheduling Loops

3.1. Work Profiles and Processor Utilization

In §2, we have discussed three ways (row-Cholesky, column-Cholesky, and submatrix-Cholesky) of scheduling tasks in a parallel environment for the solution of symmetric positive definite linear systems. For each of the three forms, we saw that a comparison of the precedence relations among the resulting tasks enabled us to identify the more inherently parallel of the two possible variations. In this section we turn to the problem of choosing from among the three basic forms the best for parallel execution. In order to do this we introduce the notion of *work profiles* of the different self-scheduling algorithms and study the corresponding processor utilization curves.

In general, consider the self-scheduling of the following loop:

```
for t := 1 to n
    schedule Task(t)
end
```

where t is the task number or task id used for the purpose of task scheduling. In other words, if there are p processors available (assuming that $p < n$), the first p tasks will be claimed by these processors. Whenever a processor becomes free, it will be responsible for the next task in the sequence, namely $Task(p+1)$.

For each task, we define $TaskWork(t)$ to be the amount of work required to complete it. The ***work profile*** is then the graph of $TaskWork(t)$ plotted against t. In Cholesky factorization, for simplicity, we assume that the amount of work for a task is the number of multiplicative operations required. Here, for uniformity, we shall regard a square root operation as another multiplicative operation. For the three basic forms of Cholesky it is easy to verify that:

$$TrowWork(i) = i(i+1)/2$$

$$TcolWork(j) = j(n-j+1)$$

$$TsubWork(k) = (n-k+1)(n-k+2)/2$$

Figs. 3.1 to 3.3 show the work profiles of the row-, column-, and submatrix-Cholesky schemes for $n = 50$. It should be noted that the area under each curve represents the total number of multiplicative operations required to perform Cholesky factorization on a 50×50 linear system. The areas under the three graphs are therefore the same, but their different shapes lead to quite different processor utilization characteristics.

With row-Cholesky, the relatively small tasks at the beginning should enable all processors to become fully utilized quickly, but saving the larger tasks to the end is likely to cause a significant number of processors to become idle while other processors finish with tasks involving the last few rows. Since these tasks require comparatively more time (proportional to $n^2/2$), the degradation in the overall efficiency of the scheme is non-trivial. Submatrix-Cholesky is rather the opposite: the relatively large tasks at the beginning may tend to inhibit full processor utilization early on, but its terminal behavior should be good because the tasks are getting smaller toward the end. Column-Cholesky has the best properties of both: task sizes taper up and then down in a smooth manner, leading to good processor utilization throughout the computation.

In Figs. 3.4 to 3.6 these conclusions are graphically illustrated in the processor utilization curves, resulting from simulations described in §3.2, for the three forms of Cholesky. As expected, row-Cholesky (Fig. 3.4) "ramps up" quickly to full processor utilization, but its utilization degrades near the end due to processors becoming idle while the last few large tasks are completed. The processor utilization graph for submatrix-Cholesky (Fig. 3.6) is somewhat deceptive in that the processors may all *begin* tasks rather quickly, but that does not mean that they are actually engaged in productive work, since the processor for a given task may spend a good deal of its time waiting for modifications from earlier columns to be completed (this effect is discussed in detail below in §3.2). Thus, in effect, submatrix-Cholesky ramps up to full utilization more slowly because of the large early tasks, but then maintains reasonably good utilization until near the end. Column-Cholesky (Fig. 3.5) shows more uniformly good behavior and is clearly the superior method. Note that in the example the efficiencies,

which will be defined below, of the row-Cholesky and submatrix-Cholesky methods happen to be the same. This is entirely accidental; in general, their efficiencies would be different.

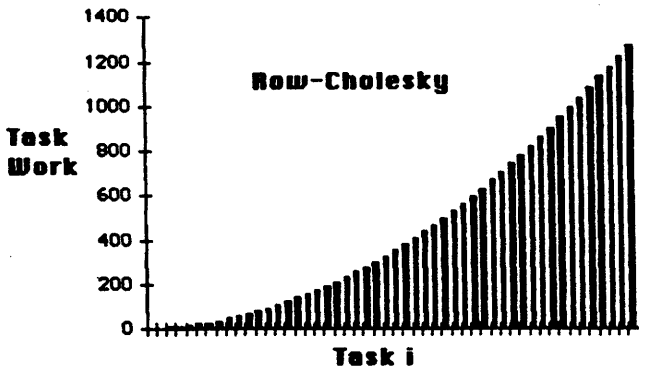


Fig. 3.1. Work Profile of Row-Cholesky.

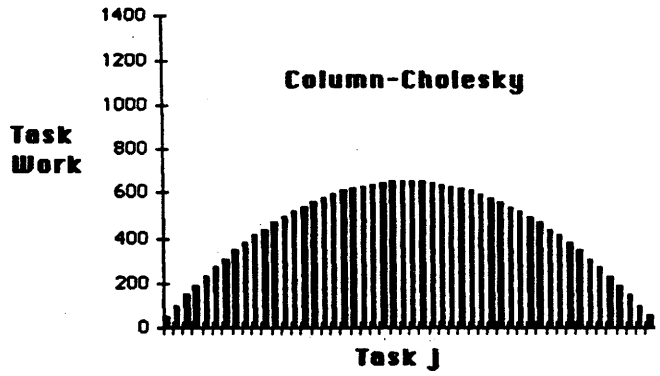


Fig. 3.2. Work Profile of Column-Cholesky.

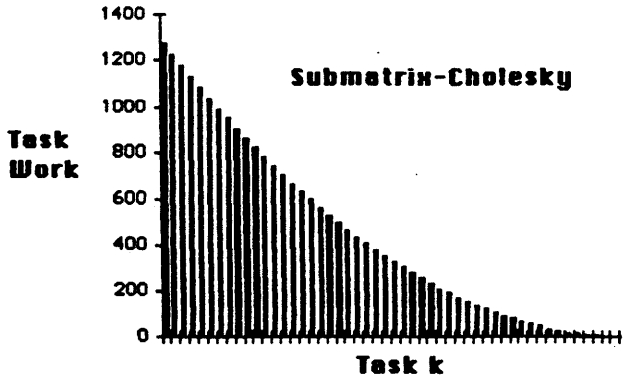


Fig. 3.3. Work Profile of Submatrix-Cholesky.

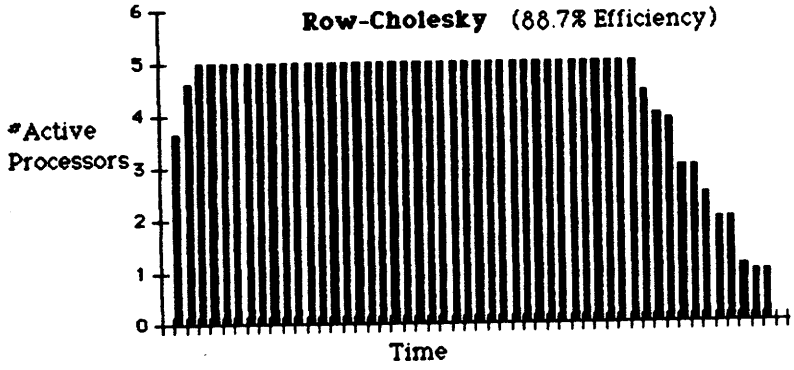


Fig. 3.4. Processor Utilization Graph for Row-Cholesky ($n = 50$, $p = 5$).

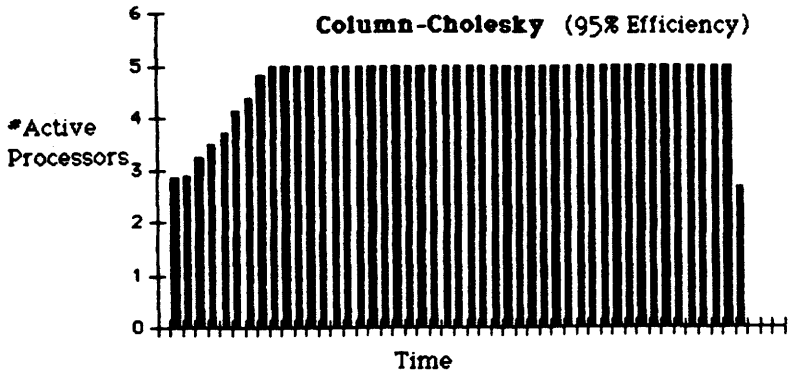


Fig. 3.5. Processor Utilization Graph for Column-Cholesky ($n = 50$, $p = 5$).

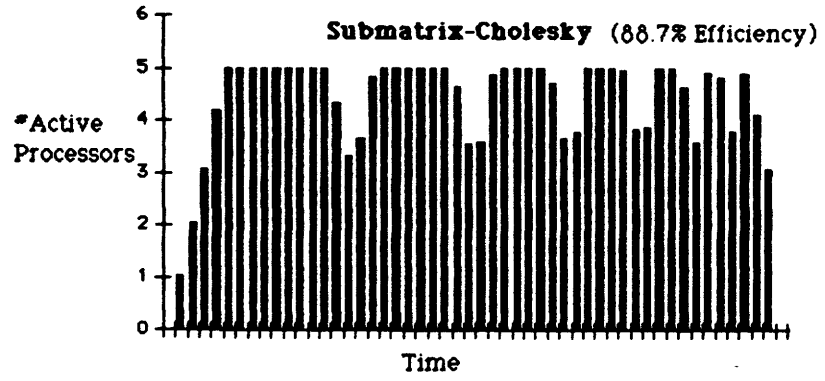


Fig. 3.6. Processor Utilization Graph for Submatrix-Cholesky ($n = 50$, $p = 5$).

3.2. Simulation Results

In this section we turn from a theoretical analysis of the three forms of Cholesky factorization to a simulation of their actual behavior for $n \times n$ systems on a multiprocessor with p processors. Numerical experiments on an existing multiprocessor, the Denelcor HEP, will be provided in §4.

In the simulation, we assume that the p processors have the same performance. Each takes one time unit to perform an additive operation or a multiplicative operation. Two unit time steps are required to compute the square root of a real number. In the simulation, time that might be lost to memory contention caused by simultaneous *access* (as opposed to simultaneous update) has been ignored.

In the results reported, we give the total number of time steps required to complete the entire factorization using p processors. The *speed-up* for p processors is defined to be:

$$\text{speed-up} = (\text{time used by 1 processor}) \,/\, (\text{time used by } p \text{ processors}).$$

Efficiency for p processors is computed as:

$$\text{efficiency} = \text{speed-up} \,/\, p.$$

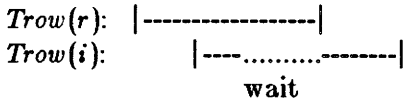
In general, efficiency is less than 100% except for the case when $p = 1$. This is due to the fact that during the course of the computation, some processors spend some time steps waiting, so that time is spent not directly for the actual numerical computation. Indeed, efficiency can also be computed as:

$$\text{efficiency} = 1 - (\text{total wait time}) \,/\, (\text{total time steps} * p) .$$

We classify the wait time into two types. *Busy-wait time* is time spent by a processor waiting in order to do further work on a task it has already been assigned. *Idle-wait time* is time in which a processor is in a wait state with no task on hand. Idle-wait time should occur only towards the end of the computation when all tasks have been assigned and yet there are some free processors.

Implicit in the notion of idle-wait time is the assumption that the p processors are released only after all tasks in the computation have been completed. In some contexts, of course, where the Cholesky factorization is only one job in a long chain of computations, there might be the possibility of "idle" processors being assigned to tasks in a subsequent member of the chain. However, our attitude in this paper is that we are designing a "library subroutine", and are concerned with maximum utilization of resources for the specific computational problem of Cholesky factorization.

In what follows, we discuss the way busy-wait time is computed in the three schemes. In the row-Cholesky method, consider the task $Trow(i)$. As described in §2, this task consists of a lower triangular solve involving the previous $i-1$ rows of the factor L. If the first $r-1$ rows of L have been used in this lower triangular solve and yet L_{r*} is not ready (that is, $Trow(r)$ has not been completed), the processor working on $Trow(i)$ will enter into a busy-wait state. It will be in this wait state until $Trow(r)$ is completed. This situation can be depicted by a "horizontal time-line" as shown below.



In the case of column-Cholesky simulation, the busy-wait time is determined in the same way as row-Cholesky except that columns are considered instead of rows. The task $Tcol(j)$ requires modifications of column j from the previous $j-1$ columns. Thus, if the column modifications from the first $c-1$ columns have been performed on column j and column c is not yet ready, the processor with task $Tcol(j)$ will have to wait for the completion of $Tcol(c)$.

Finally, in the submatrix-Cholesky scheme, the situation is quite different. Consider the task $Tsub(k)$. This task cannot start until all the modifications from the previous $k-1$ columns have been performed on column k. It should be noted that these column modifications are applied by the tasks $Tsub(1), \ldots, Tsub(k-1)$. In other words, if any one of these $k-1$ tasks has not yet modified column k, the processor working on $Tsub(k)$ will be in a busy-wait state. This can be implemented quite easily by maintaining a modification count for each column: the task $Tsub(k)$ can start only if its count is $k-1$.

As mentioned in §2, simultaneous updating of a column is a problem that must be resolved for submatrix-Cholesky. However, for simplicity in the simulation, no measure is used to guard against this. Therefore, the time reported can only be regarded as a lower bound for the actual time that would be required.

In Tables 3.1 to 3.3, results from the simulation are tabulated.

p	Total Time	Speed-up	Efficiency	Busy-Wait	Idle-Wait	Total Wait
1	2686900	1.00	100.00	0	0	0
2	1353508	1.98	99.26	16	20100	20116
3	909081	2.95	98.52	73	40270	40343
4	686900	3.91	97.79	194	60506	60700
5	553623	4.85	97.07	403	80812	81215
6	464791	5.78	96.35	721	101125	101846
8	353855	7.59	94.92	1785	142155	143940
10	287388	9.34	93.49	3573	183407	186980
12	243167	11.05	92.08	6270	224834	231104
14	211658	12.69	90.68	10062	266250	276312
16	188131	14.28	89.26	15135	308061	323196
18	169985	15.80	87.81	21675	351155	372830
20	155548	17.27	86.37	29873	394187	424060
50	84852	31.66	63.33	478310	1077390	1555700
100	79801	33.67	33.67	1994850	3298350	5293200

Table 3.1: Simulation of Row-Cholesky on a 200×200 System.

The column-Cholesky scheme emerges as the clear winner when the simulation results in Tables 3.1-3.3 are compared. For $p = 20$ (10% of the size of the matrix), the efficiency of column-Cholesky remains at a level of about 97%. On the other hand, the efficiency for the other two schemes for 20 processors drops to about 87%. The difference is more dramatic when $p = 50$. These results are consistent with our earlier analysis of processor utilization based on work profiles.

The amount of idle-wait time for the row-Cholesky method is strikingly high. Indeed, we see from Table 3.1 that for $p = 20$, over 90% of the total wait time is spent for idle-wait. This is not surprising in view of the work profile of the row-Cholesky scheme as shown in Fig. 3.1 and the idealized processor utilization curve shown in Fig. 3.4.

On the other hand, from Table 3.3, we note that most of the wait time for the submatrix-Cholesky method is attributed to busy-wait, again consistent with our analysis of its work profile and precedence relation. As explained in §3.1, the processor working on task $Tsub(k)$ has to wait until all the $k-1$ modifications on column k have been applied. It should be emphasized again that the busy-wait time reported does not

p	Total Time	Speed-up	Efficiency	Busy-Wait	Idle-Wait	Total Wait
1	2686900	1.00	100.00	0	0	0
2	1343701	2.00	99.98	403	99	502
3	896065	2.99	99.95	1150	145	1295
4	672306	3.99	99.91	2035	289	2324
5	538110	4.99	99.86	3366	284	3650
6	448836	5.98	99.77	5150	966	6116
8	337100	7.97	99.63	9393	507	9900
10	270298	9.94	99.41	15306	774	16080
12	226120	11.88	99.02	24191	2349	26540
14	194468	13.81	98.69	31106	4546	35652
16	170991	15.71	98.21	42903	6053	48956
18	152395	17.63	97.95	52844	3366	56210
20	138068	19.46	97.30	69520	4940	74460
50	65902	40.77	81.54	541744	66456	608200
100	60100	44.70	44.71	2818200	504900	3323100

Table 3.2: Simulation of Column-Cholesky on 200×200 System.

p	Total Time	Speed-up	Efficiency	Busy-Wait	Idle-Wait	Total Wait
1	2686900	1.00	100.00	0	0	0
2	1353504	1.98	99.26	20106	2	20108
3	909065	2.95	98.52	40288	7	40295
4	686868	3.91	97.80	60556	16	60572
5	553568	4.85	97.08	80910	30	80940
6	464712	5.78	96.36	101322	50	101372
8	353695	7.59	94.96	142548	112	142660
10	287128	9.35	93.58	184170	210	184380
12	242785	11.06	92.22	226168	352	226520
14	211140	12.72	90.90	268514	546	269060
16	187447	14.33	89.59	311452	800	312252
18	169090	15.89	88.28	355598	1122	356720
20	154428	17.39	87.00	400140	1520	401660
50	77552	34.64	69.29	1168650	22050	1190700
100	60100	44.70	44.71	3151500	171600	3323100

Table 3.3: Simulation of Submatrix-Cholesky on 200×200 System.

include any wait time that would be incurred due to simultaneous column modifications.

The column-Cholesky scheme exhibits a relatively good balance between busy-wait and idle-wait time. The work profile of this method as depicted in Fig. 3.2 and processor utilization curve shown in Fig. 3.5 help to explain its desirable behavior.

The relatively poor performance of all three schemes for very large numbers of processors (say 100 in the tables) is disappointing, but not entirely unexpected. It means that the granularity of our approach is not appropriate for such large numbers of processors. In other words, treating each row/column/submatrix as a task is too coarse when p is a significant fraction of n. Thus, each task should be broken up into finer subtasks for parallel computation when the number of processors is relatively large.

4. Implementation on the HEP

4.1. The Denelcor HEP Computer

The Denelcor HEP is a commercially available multiprocessor whose architecture and corresponding programming style are described, for example, in [6, pp. 669-684] and [7]. A HEP can have one or more process execution modules (PEM's), but even within a single PEM there is an eight-fold parallelism due to an eight-stage instruction pipeline that can in effect process eight independent instruction streams simultaneously (that is to say, an instruction from each of eight streams is executed in each major machine cycle). Due to latencies in certain instructions, memory fetches, etc., more than eight instruction streams (processes) are usually necessary to keep the eight-stage instruction pipeline fully occupied. Experience has shown that about twelve processes are usually needed to utilize fully the machine's throughput capability.

Other salient features of the HEP for our purposes include a large shared memory, facilities for creating processes, low-overhead synchronization primitives for coordinating processes, and hardware locks on memory that facilitate efficient implementation of mutual exclusion. In particular, the self-scheduled pool-of-tasks paradigm can be programmed on the HEP in a natural and efficient manner. See [9] for a detailed discussion of implementing self-scheduling on the HEP.

A single PEM HEP at Argonne National Laboratory was kindly made available to us for the experiments reported in this paper.

4.2. Review of the Dongarra-Hiromoto Implementation

In [4], Dongarra and Hiromoto describe an implementation of the Cholesky factorization on the Denelcor HEP computer. It is based on the *jik* formulation of the factorization. Recall from §2.3 that in this *jik* version, columns are computed one by one, and the rectangular submatrix required for column modification is accessed row by row. We shall refer to this as the Dongarra-Hiromoto implementation.

It must be emphasized that their objective was to obtain near optimum parallel performance by implementing only the underlying modules (i.e., inner loops) by parallel constructs. In this way, they are able to produce "portable algorithms with a high level of granularity in their structure." They have certainly achieved their goal in [4].

Since the outer-loop precedence relation for the *jik* form is of Type 1, in the implementation [4] the n tasks of computing columns are performed ***serially***. The self-scheduling technique is used in the second loop for performing the basic *GAXPY* operation to do a matrix-vector product. In other words, the inner-products within the *GAXPY* module (which is in fact named *SMXPY* in [4]) are executed in parallel.

For completeness, we have tabulated in Table 4.1 results from simulating the Dongarra-Hiromoto parallel implementation. Note that in this implementation, there is ***no busy-wait***, since at this level all parallel processes active at a given time are completely independent (recall that in the simulations we are ignoring any wait time due to possible memory contention).

As might be expected from the serial (Type 1) nature of the outer loop, the Dongarra-Hiromoto implementation is uniformly less effective than any of the three Cholesky variations studied in §3, except that it does beat the worst of the three, row-Cholesky, for very large p.

p	Total Time	Speed-up	Efficiency	Busy-Wait	Idle-Wait	Total Wait
1	2686900	1.00	100.00	0	0	0
2	1363600	1.97	98.52	0	40300	40300
3	922522	2.91	97.09	0	80666	80666
4	702000	3.82	95.69	0	121100	121100
5	569700	4.71	94.33	0	161600	161600
6	481510	5.58	93.00	0	202160	202160
8	371300	7.23	90.46	0	283500	283500
10	305200	8.80	88.04	0	356100	365100
12	261156	10.28	85.74	0	446972	446972
14	229710	11.69	83.55	0	529040	529040
16	206148	13.03	81.46	0	611468	611468
18	187832	14.30	79.47	0	694076	694076
20	173200	15.51	77.57	0	777100	777100
50	94800	28.34	56.69	0	2053100	2053100
100	70000	38.38	38.38	0	4313100	4313100

Table 4.1: Simulation of Dongarra-Hiromoto Implementation on 200×200 System.

4.3. Implementation of Column-Cholesky on the HEP

The simulation results in section 3.2 suggest the use of the column-Cholesky scheme on the HEP machine for parallel implementation. A closer examination of the task $Tcol(j)$ in the column-Cholesky method suggests that there are two types of subtasks:

1. $cmod(j,k)$: modification of column j by column k $(k < j)$;
2. $cdiv(j)$: division of column j by a scalar.

Fig. 4.1 contains a precedence graph for these subtasks. It should be clear that modifications from the preceding columns must be performed before scalar division on column j can start. Furthermore, column j can be used to modify subsequent columns only after its values are completely formed through the subtask $cdiv(j)$.

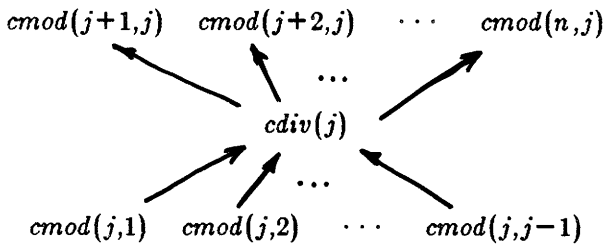


Fig. 4.1. Subtask Precedence Graph for Column-Cholesky.

The self-scheduling of the tasks $Tcol(j)$ can be implemented quite easily by maintaining a vector of flags "$ready[*]$", where $ready[j]$ indicates whether column j is ready to be used for modification of subsequent columns. The following gives an algorithmic description of the implementation:

```
ready[1] := 1;
for j := 2 to n
    ready[j] := 0;
for j := 1 to n
    schedule Tcol(j);
```

The task $Tcol(j)$ can then be implemented as:

```
for k := 1 to j-1
    begin
        wait until ready[k] = 1;
        do cmod(j,k)
    end;
do cdiv(j);
ready[j] := 1;
```

We note that this implementation has the advantage of requiring no mutual exclusion or critical section other than that directly related to self-scheduling (i.e., processes picking up a unique task id).

4.4. Experimental Results on the HEP

The column-Cholesky scheme was implemented on the HEP. Results on its performance for systems of order 200 are tabulated in Table 4.2. For the sake of comparison, we have also included the experimental results on the Dongarra-Hiromoto implementation. The times shown are actual values obtained from the timing clock on the HEP with no other user jobs running, and are in units of 10^{-7} seconds.

Our results on the HEP are reasonably consistent with our theoretical analysis and simulation results, especially when the simplifying assumptions of the latter are considered. Speed-up and efficiency are, of course, uniformly inferior when the realities of synchronization overhead, memory contention, etc., are taken into account. Still, the shape and trend of our results seem to bear out our expectations. For our implementation of column-Cholesky, in particular, a speed-up of about 8.9 on a machine having only an eight-fold hardware parallelism seems most satisfactory. Veteran HEP users have conjectured that a speed-up of about 10 is the most that could be expected of any algorithm for matrix problems of this type on a single PEM HEP [4, p. 3]. We note that due to the nature of the HEP architecture, the speed-up flattens out at about $p = 12$, since the instruction pipeline is fully saturated by that point, then actually gets slightly worse for larger p, since there is overhead associated with maintaining additional processes.

We note that the column-Cholesky algorithm outperforms the Dongarra-Hiromoto implementation, both in absolute terms and in terms of speed-up and efficiency; moreover, the differences become greater as p increases. The superiority of column-Cholesky in speed-up and efficiency for $p > 1$ is easily explained in terms of the analysis we have presented. Less readily understood is the superior performance of column-

p	Column-Cholesky			Dongarra-Hiromoto		
	HEP Time	Speed-up	Efficiency	HEP Time	Speed-up	Efficiency
1	229817620	1.00	100.00	402804992	1.00	100.00
2	116241803	1.98	98.85	208148142	1.94	96.76
3	77378577	2.97	99.00	143416418	2.81	93.62
4	59260228	3.88	96.95	111825036	3.60	90.05
5	47445196	4.84	96.88	93630438	4.30	86.04
6	40356802	5.69	94.91	81451456	4.95	82.42
8	31406321	7.32	91.47	66082956	6.10	76.19
10	27091879	8.48	84.83	61120506	6.59	65.90
12	25888952	8.88	73.98	57677942	6.98	58.20
14	25785533	8.91	63.66	57417148	7.02	50.11
16	26353589	8.72	54.50	57249314	7.04	43.97
18	26645924	8.62	47.92	57124855	7.05	39.17
20	27345522	8.40	42.02	59152152	6.81	34.05

Table 4.2: Experimental Results on the HEP for 200×200 System

Cholesky for $p = 1$, since the two algorithms are performing the same number of floating point operations, and parallelism is not a factor. We conjecture that this behavior is an artifact of the nature of the HEP architecture. A probable contributing factor is the relatively high cost of integer arithmetic (and hence index and subscript computations) on the HEP. Thus, the use of a two-dimensional array (*GAXPY*) in the Dongarra-Hiromoto code leads to relatively poor performance compared to the one-dimensional array (*SAXPY*) required for the inner loop of our implementation of column-Cholesky. It has also been observed that because of latency in memory access, some computations on the HEP can be dominated by memory references rather than arithmetic operations. Another anomaly is the discrepancy between the maximum speed-up of about 7.1 that we observed for the Dongarra-Hiromoto code and their published value of 7.6 [4, p. 3] for the same code on the HEP at Los Alamos National Laboratory.

5. Conclusion

In this paper we have developed a parallel algorithm for Cholesky factorization on a multiprocessor. The algorithm is based on the concept of self-scheduling a pool of tasks. We considered six variations on the basic elimination algorithm corresponding to all possible arrangements of the three loops. An analysis of the precedence relationships among tasks, task work profiles, and the resultant processor utilization characteristics enabled us to identify the most promising variant for parallel implementation. The algorithm chosen, which we call column-Cholesky, was implemented on the Denelcor HEP multiprocessor. Its performance surpassed that of a previous algorithm for the same problem, and seems to approach the maximum of which the machine is capable.

The self-scheduling pool of tasks seems to be a powerful paradigm for parallel computation because it tends to yield good load balancing and makes effective use of all processors even if the tasks are heterogeneous. It is not strongly dependent on the number of processors available, and therefore the resulting algorithms are potentially portable across parallel machines having different numbers of processors. In order that self-scheduling effectively exploit potential concurrency, however, the order in which tasks are scheduled from the pool must take into account the precedence relations and

work profiles of those tasks. The type of analysis that we have used should be helpful in designing parallel algorithms for many other computational problems. We have already applied the same technique, although necessarily with somewhat more complicated data structures and control logic, to the factorization of sparse matrices. These results will be reported in a future paper.

6. Acknowledgment

We are grateful to J. Dongarra and R. Hiromoto for kindly providing us with a copy of their code. We also greatly appreciate the advice and assistance provided by J. Dongarra and D. Sorensen in connection with the use of the HEP at Argonne National Laboratory.

7. References

[1] R. P. BRENT AND F. T. LUK, "Computing the Cholesky factorization using a systolic architecture", TR 82-521, Department of Computer Science, Cornell University , Ithaca, NY (September 1982).

[2] DENELCOR, INC., "HEP Fortran 77 User's Guide", Pub. No. 9000006, Denelcor, Inc., Aurora, CO (February 1982).

[3] J. J. DONGARRA, F. G. GUSTAVSON, AND A. KARP, "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine", *SIAM Review*, **26** (1984), pp. 91-112.

[4] J. J. DONGARRA AND R. E. HIROMOTO, "A collection of parallel linear equations routines for the Denelcor HEP", ANL/MCS-TM-15, Argonne National Laboratory, Argonne, IL (September 1983).

[5] J.A. GEORGE AND J. W-H. LIU, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1981).

[6] K. HWANG AND F. A. BRIGGS, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Co., New York (1984).

[7] H. F. JORDAN, "Experience with pipelined multiple instruction streams", *Proc. IEEE*, **72** (1984), pp. 113-123.

[8] C. LAWSON, R. HANSON, D. KINCAID, AND F. KROGH, "Basic linear algebra subprograms for Fortran usage", *ACM Trans. Math Software*, **5** (1979), pp. 308-371.

[9] E. L. LUSK AND R. A. OVERBEEK, "Implementation of monitors with macros: a programming aid for the HEP and other parallel processors", ANL-83-97, Argonne National Laboratory, Argonne, IL (December 1983).

[10] D. P. O'LEARY AND G. W. STEWART, "Data-flow algorithms for parallel matrix computations", Tech. Rept. 1366, Computer Science Dept., University of Maryland , College Park, MD (January 1984).