IN-62
DATE OVERRIDE
43029
P,33

Efficient ICCG on a Shared Memory Multiprocessor

Steven W. Hammond
Robert Schreiber

May, 1989

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 89.24

NASA Cooperative Agreement Number NCC 2-387

RIACS
Research Institute for Advanced Computer Science

Efficient ICCG on a Shared Memory Multiprocessor

Steven W. Hammond
Robert Schreiber

May, 1989

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 89.24

NASA Cooperative Agreement Number NCC 2-387

Efficient ICCG on a Shared Memory Multiprocessor [*]

Steven W. Hammond [†] Robert Schreiber [‡]

May 1989

Abstract

In this paper we discuss different approaches for exploiting parallelism in the ICCG method for solving large sparse symmetric positive definite systems of equations on a shared memory parallel computer. Techniques for efficiently solving triangular systems and computing sparse matrix-vector products are explored. Three methods for scheduling the tasks in solving triangular systems are implemented on the Sequent Balance 21000. Sample problems that are representative of a large class of problems solved using iterative methods are used. We show that a static analysis to determine data dependences in the triangular solve can greatly improve its parallel efficiency. We also show that ignoring symmetry and storing the whole matrix can reduce solution time substantially.

[*]submitted to: International Journal of High Speed Computing.

[†]Ph.D. Student at Rensselaer Polytechnic Institute, Troy, NY 12180 and Visiting Research Associate at Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA 94035.

[‡]Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA 94035.

[0]Work reported herein was supported by Cooperative Agreement NCC2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA)..

1 Introduction

We explore different schemes for exploiting the parallelism available in the ICCG method for solving large sparse systems of linear equations on a shared memory computer. All of this work has been conducted on a 12 processor Sequent Balance 21000. We have looked at the efficient implementation of methods for solving triangular systems and at sparse matrix vector multiplication.

An important difficulty in solving general sparse triangular systems is that the available parallelism depends on the zero structure of the matrix, and is therefore not known at compile time. The concurrency is data dependent and can be determined only at run time. We show that by performing a small amount of analysis to determine the data dependences one can drastically improve the parallel efficiency. We permute (reorder) the index set of the recurrence equation for the triangular solve and put the indices in a queue. The processors repeatedly take indices from the queue, perform the associated calculations, and then take another index until all unknowns have been computed. Data dependences are resolved by *semaphores*. A *semaphore* is a variable that can be operated upon only by synchronizing primitives. We check indices in a shared array that indicate whether each of the unknowns has been computed. If a calculation depends on a piece of data and an entry in the shared array indicates that it has not been computed then the processor performing the calculation must *busy wait*. Busy waiting is when a processor loops waiting for a flag to change value.

Also, we show that there is a tradeoff between storing the lower triangular part of a symmetric matrix and storing the entire matrix. Storing the lower part to save storage complicates the multiplication since both outer products (which require synchronization) and inner products must be performed. The synchronization overhead slows down this operation.

For our experiments we work with systems of equations in the form they are presented. We do not consider the problem of reordering the rows and columns to enhance parallelism.

The rest of this paper is organized as follows. Section 2 reviews related research. Section 3 contains a brief discussion of the ICCG method. Section 4 discusses how the dependence graph is used to exploit the parallelism in solving sparse triangular systems. Section 5 contains numerical experiments that show it is more efficient to store the whole symmetric matrix than only the upper or lower triangular part. In Section 6 we compare solving a lower triangular system by inner products versus solving by outer products. Section 7 presents the efficiency of the ICCG method using the techniques described in the previous sections. Section 8 dicusses other scheduling methods not used in this paper. Section 9 contains remarks and conclusions. Appendix A describes the 7 test cases used in the experiments. Appendix B discusses the architecture of the Sequent Balance 21000 and provides times for arithmetic operations and synchronization primitives. In Appendix C we show how the time to access array elements increases as a function of the array size on the Sequent Balance 21000.

2 Related Work

Level scheduling methods are considered in [2,12,27]. Anderson [2] compares two different scheduling methods for solving sparse triangular systems on the Alliant FX/8, a shared memory machine. They are *forward level scheduling*, in which each unknown in the triangular solve is computed as early as possible, and *backward level scheduling*, in which each unknown is computed at the latest possible time. A level scheduling approach partitions the loop of the recurrence equation into a sequence of fully parallelized do loops (levels) separated by global synchronizations. He shows that the overhead in scheduling tasks to be performed as late as possible is not worth the time savings.

Baxter *et. al.* [3] compare *level scheduling* with a *self scheduling* method using a shared memory computer, an Encore Multimax/320. The *self scheduling* method is a two step procedure to parallelize the recurrence equation of the triangular solve. First, one performs a topological sort of the dependence graph to permute the index loop. Next, statically assign elements of the index set to the processors of the system. Global synchronizations are avoided by requiring processors to write into specified locations of shared arrays when work on a particular index is completed. Before a variable can be used, a processor makes sure that the appropriate values have been calculated by *busy waiting* on a designated value in the shared memory. They show that *self scheduling* performs better than *level scheduling* for all but one of their test cases.

The work of Saltz *et. al.* [25] is similar to the work of Baxter. They also compare *level scheduling* and *self scheduling* on an Encore and reach similar conclusions. Saltz also proposes a new programming construct, *doconsider* which allows compilers to parallelize many problems in which substantial loop-level parallelism is available but cannot be detected by standard compile-time analysis.

The difference between the work presented here and previous work on triangular systems is that we use dynamic scheduling to assign tasks to processors and the others use static scheduling.

In this paper we focus on general parallel processors but others have studied implementations on parallel vector machines [15,16,20,26]. Additionally, Saad [21] presents a survey of recent research in Krylov subspace methods with an emphasis on parallel and vector implementations.

3 ICCG Background

Here we give a brief introduction to the Incomplete Cholesky Conjugate Gradient (ICCG) method. For detailed information of its derivation and properties see references [6,11,13,17,18,23]. The Conjugate Gradient (CG) method was proposed by Hestenes and Stiefel [13] for the solution of

$$Ax = b, \tag{1}$$

where A is a given symmetric positive definite N by N matrix, b is a given N-vector and x is an N-vector to be computed.

Starting from an initial guess $x^{(0)}$, the CG method generates a series of approximate solutions $x^{(k)}$. The convergence rate is very poor for ill-conditioned problems [11]. One way to improve the convergence is to *pre-condition* (1) – premultiply it by a conditioning matrix and thereby condense the eigenvalue spectrum [4].

A popular preconditioner is the Incomplete Cholesky preconditioner proposed by Meijerink and Van der Vorst [18]: they perform an approximate Cholesky-factorization LL^T of A with zero fill. Equation (1) now becomes:

$$\left[(L^{-T})L^{-1}A\right]x = (L^{-T})L^{-1}b. \tag{2}$$

L^{-1} is not explicitly computed, instead triangular systems are solved. Each iteration of the ICCG method requires the solution of two sparse triangular systems, a sparse matrix vector product, 3 saxpy's and 2 inner products.

We warn the reader that we use an inconsistent notation here from the rest of the paper. Here we subscript a vector to indicate that it is a member of a sequence rather than referring to an individual element. The greek letters represent scalars. The ICCG method is below:

```
$x_0 := 0$
$r_0 := b$
$\delta := \epsilon\|r_0\|_2$
repeat   For k=1, 2, ...
        Solve $LL^T z_{k-1} = r_{k-1}$ for $z_{k-1}$
        $\beta_k := z_{k-1}^T z_{k-1} / z_{k-2}^T z_{k-2}$          $(\beta_1 \equiv 0)$
        $p_k := z_{k-1} + \beta_k p_{k-1}$                $(p_1 \equiv z_0)$
        $\alpha_k := z_{k-1}^T r_{k-1} / p_k^T A p_k$
        $x_k := x_{k-1} + \alpha_k p_k$
        $r_k := r_{k-1} - \alpha_k A p_k$
        until $\|r_k\|_\infty \le \delta$
    $x := x_k$
```

For our codes we choose $\varepsilon = 10^{-6}$ so our iteration stops when the infinity norm of the residual is reduced by 6 orders of magnitude.

4 Triangular Systems

At each ICCG iteration we solve the triangular systems

$$Lq = r \tag{3}$$

and

$$L^T z = q. \tag{4}$$

Together, these two operations consume between 30% and 41% of the total cpu time required to solve the system on a single processor for our test cases. The percentage depends on the sparsity of L - the more nonzero elements in L the higher the percentage. The remaining time is consumed by sparse matrix-vector products, inner products and saxpy's. These are relatively easy to compute in parallel. Efficient parallel computation of the triangular solves is necessary to accelerate the entire computation.

The system (3) is solved by

$$q_i = \frac{r_i - \sum_{j=1}^{i-1} L_{i,j} q_j}{L_{i,i}} \quad i = 1, \ldots, N \tag{5}$$

In the dense case, each q_i depends on all q_j, $j = 1, \ldots, i-1$. When L is sparse, each q_i depends on a few other q_j. Another way to look at it is that once some q_j has been computed, several other q's may be computed in parallel. It is possible to perform some simple analysis of the data dependences to determine which elements of q can be computed in parallel and determine which q_j's each q_i depends on. This information can be utilized to schedule tasks. For example, if q_i depends on q_j then q_j should be scheduled before q_i. Also, if q_j and q_k are independent tasks then we may schedule them to be computed in parallel.

$$\begin{pmatrix}
\times & & & & & & & \\
\times & \times & & & & & & \\
& & \times & & & & & \\
& & \times & \times & & & & \\
& \times & & \times & \times & & & \\
\times & & & & & \times & & \\
& & & \times & & \times & \times & \\
& & \times & & \times & & \times & \times
\end{pmatrix}$$

Figure 1: Sparsity Structure of L

4.1 Computing the Dependence Graph

The problem one faces when exploiting this type of parallelism is that it is data dependent and can only be recognized at run time, not at compile time. It depends entirely on the sparsity structure of L. L is usually read in as input or computed at an earlier stage of the program. The focus in this section and the next one is on lower triangular systems. A similar analysis can be done on upper triangular matrices.

Consider solving (3) where L has the sparsity structure shown in Figure 1. Analysis of the structure of L enables us to construct a corresponding directed graph (digraph), the *dependence graph* $G(L) = (V, E)$. There are N vertices, $V = \{1, \ldots, N\}$, corresponding to the N rows of L (and N elements of q). A nonzero element at $l_{i,j}$ means that q_i depends on q_j; i.e., q_j must be calculated before q_i. Therefore, we define the edges of $G(L)$ as follows: $E = \{(j, i) \mid l_{i,j} \neq 0\}$. We ignore the loops corresponding to the diagonal elements of L ($G(L)$ is acyclic). The *depth* of a vertex v_i is 0 if it has no predecessors otherwise the *depth* of v_i is the length of the longest directed path in $G(L)$ whose origin is a vertex of depth 0 and terminus is v_i.

The dependence graph of L is shown in Figure 2. All nodes at depth 0 can be computed immediately. They have no dependences. q_1 and q_3 can be solved directly. Once q_3 is computed we can solve for q_4. After q_1 is computed we can solve for q_2 and q_6 in parallel. The unknowns q_2 and q_6 depend only on q_1. Once q_2, q_6 and q_4 have been computed, we can solve for q_5, q_7 in parallel. Vertices that have equal depth represent independent tasks. The fact that q_2 and q_6 can be computed as soon as q_1 has been computed, even if q_3 has not been completed, illustrates the difference between *level scheduling* methods and *self scheduling* methods. *level*

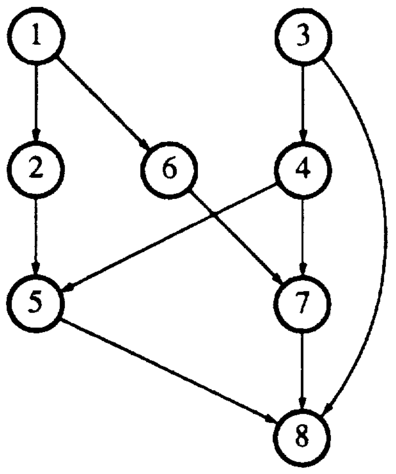

Figure 2: Dependence Graph of L

scheduling computes tasks corresponding to vertices of equal depth in G in parallel. All tasks at a certain depth must be completed before tasks at the next level can be started. Global synchronizations are used to separate tasks at different depths. *self scheduling* allows tasks to start as soon as their associated dependences have been computed.

4.2 Permuting the Index Set

The index set for the sequential solution of equation (5) is $i = 1, \ldots, N$. To exploit the parallelism in the forward solve we reorder the index set according to the depth of each index in the dependence graph. A vertex of a certain depth is put in the permuted set before all vertices of greater depth. We define *postion(k)* to be the number of elements in the premuted index set that precede k. If two vertices v_i and v_j have equal depth then we put v_i in the permuted index set before v_j if

$$\max_n(position(n)) < \max_m(position(m)) \quad \text{such that } (n,i),(m,j) \in E.$$

If

$$\max_n(position(n)) = \max_m(position(m)) \quad \text{such that } (n,i),(m,j) \in E$$

then $n = m$ and v_i is placed in the list before v_j if $i < j$. This is a side effect of the sequential traversal of the data structure for L.

We call this permuted index set **fwd_schedule**. For example, the permuted index set from the dependence graph in Figure 2 is

$$\texttt{fwd\_schedule} = \{1, 3, 2, 6, 4, 5, 7, 8\}$$

Note that 6 appears before 4. 6 is a descendent of 1 and 4 is a descendent of 3 and $position(1) < position(3)$.

One way to compute the **fwd_schedule** list is outlined here. First, as the matrix is assembled or read in, construct an array of length N, called the *ready* array such that $ready[i]$ is the number of nonzero elements in row i of L. We then scan the entries of the *ready* array looking for entries with a value of 1. If $ready(i) = 1$ then q_i can be solved for directly. This entry is put in a queue, Q. When we have inserted all entries with value 1 in Q we start the following loop. We follow the notation used in [1] for operations on a queue. A *queue* is a special kind of a list, where items are inserted at one end (the *rear*) and deleted from the other end (the *front*).

```
fwd_schedule = nil

While ($empty(Q) \neq$ true)

        1. $i := front(Q)$
        2. $dequeue(Q)$
        3. append i to fwd_schedule list
        4. for each nonzero element $L_{ki}$
            (a) $ready(k) := ready(k) - 1$
            (b) if $(ready(k) = 1)$ then $enqueue(Q, k)$
```

The dependence graph is not explicitly computed but the information it represents is implicit in *ready* and the ordering of **fwd_schedule**. We require two integer arrays of length N to hold **fwd_schedule** and *ready*. This additional storage is small relative to the storage for A, L and the other N-vectors needed for ICCG.

Equation (4) is also solved with a permuted index set, which we store in the array **back_schedule[]**. It is computed by analyzing the dependence graph of L^T, $G(L^T)$, in a manner similar to that used to compute **fwd_schedule**. Let $G(L^T) = (V_T, E_T)$, $V_T = V$ and $E_T = \{(j,i)|(i,j) \in E\}$. $G(L^T)$ is the same as $G(L)$ with the direction of the edges reversed. For the example shown in Figure 1, the schedules for solving the upper and lower triangular systems are the reverse of each other. This is not true in general. Suppose that we have the same lower triangular matrix as in Figure 1 except $L_{8,1}$ and $L_{8,8}$ are the only nonzero elements in row 8 of L. Then, v_8 will

case	fwd_schedule	fwd solve	back_schedule	bck solve	Ax=b
1	.12	.18	.11	.16	1.89
2	.06	.06	.05	.06	2.45
3	.05	.05	.04	.04	3.07
4	.07	.08	.06	.08	3.21
5	.10	.11	.09	.10	8.85
6	.25	.30	.23	.28	8.52
7	3.12	3.25	2.09	2.48	655.39

Table 1: Time in seconds to compute task schedules vs. single sequential triangular solve and solving Ax=b in parallel

be depth 1 in $G(L)$ and 8 will be the 6^{th} element in **fwd_schedule[]**. But, v_8 will be at depth 0 in $G(L^T)$ and the first element in **back_schedule[]**.

The time to compute the permuted index sets is a little less than the time to compute a single sequential triangular solve and a small fraction of the time to solve (2) in parallel. The time in seconds to compute the forward and backward schedules for the test cases is shown in Table 1. We compare the time to compute the **fwd_schedule** and the **back_schedule** lists with the time to sequentially compute one forward and backward solve and with the time to solve $Ax = b$ in parallel.

4.3 Forward Solve

We solve (3) as follows. L is stored by columns and the forward solve is computed as a set of outer products. **fwd_schedule** is the list of indices which correspond to elements of q to be computed. It is treated as a queue of tasks to be executed by the pool of processors. Let there by $\mathcal{P}$ processors. Initially, the first $\mathcal{P}$ indices in the queue are assigned one to each processor. Let i be the index a processor gets from the queue. Before we compute each forward solve we set **fwd_ready[]** to be the number of nonzero elements in each row of L. If **fwd_ready[i]** $\neq 1$, then the processor must *busy wait*, else, compute $r_i = b_i/L_{i,i}$. Next, compute $r_j -= L_{j,i}q_i$ and decrement **fwd_ready[j]** for all nonzero elements j of column i of L. Finally, if the queue is not empty get the next task.

For the triangular solve, we experimentally compared three different techniques for parallelizing the code. We call the first method *dynamic scheduling* (DS). The elements of q are assigned to processors in order, from 1 to N. They are computed as soon as the data they depend on is ready to be used. The data illustrate that poor performance may be expected if the index set is left in its original order.

The second technique is due to Baxter *et. al.* [3]. We call this technique *reordered static scheduling* (RSS). They use the reordering strategy above, but employ a static assignment of tasks from the permuted index set to processors. Let $\mathcal{P}$ be the number of processors. Processor i, $1 \leq i \leq \mathcal{P}$, gets tasks $i + \mathcal{P} \times j$ for $j = 0, \ldots, \lfloor \frac{N-1}{\mathcal{P}} \rfloor$. It has the advantage that for every iteration each processor will solve for the same values of q. This characteristic is especially noticeable for small problems when the entire problem fits in the local memory (or cache) of the processors. But this may not be very good at load balancing. If there are wide variations in the number of nonzero elements in the rows/columns of the matrix then the static mapping may cause unnecessary busy waiting. This variation arises in many different situations: non-uniform discretizations, adaptively refined meshes, or mixed element types (triangular and quadrilateral elements in the same grid) for instance.

The problem with *reordered static scheduling* is that the position of the task in the schedule is determined solely by its depth in the dependence graph. The strategy does not consider the amount of time needed to perform the task. It is possible that a static assignment of tasks to processors could result in uneven distribution of work and lower or less throughput.

The third technique is called *reordered dynamic scheduling* (RDS). We reorder the index set as above, but we put the indices (tasks) in a queue rather than statically mapping them to processors. The first processor done with the work initially assigned to it takes the next job from the front of the queue. This is done to reduce the time spent *busy waiting* due to potential load imbalance. There is an additional expense of maintaining a global pointer (**m_next()** on the Sequent) to the first element in the queue.

The C code for reodered dynamic scheduling is shown in Figure 3. **m_next()** is the system function which increments a global counter and returns its current value. **fwd_schedule[]** is our permuted index set for the forward solve. As suggested by Duff, *et. al.* [7], we store the columns as packed sparse vectors held contiguously in the array **l[]**. The row numbers of the corresponding nonzero entries held in **l[]** are held in the integer array **row_num[]**. The integer array **start[i]** points to the start of column **i** in array **l[]** containing the nonzero elements of matrix L. In fact **l[start[i]]** is the diagonal element $L_{i,i}$. The global variables **unknowns** and **tot_nonzero** hold the number of rows in L and the total nonzero elements in L respectively **start[unknowns+1]** $\equiv$ **tot_nonzero + 1**.

Once we have gotten a task from the queue, we check whether all of the data it needs are ready. This is done by looking at the value of the **fwd_ready[i]** array containing the number of direct dependences for row i. If the value is greater than

```
parallel_fwdslv(l, q, row_num, my_id, num_proc)

double
   l[], q[];                               /* l[] - nonzero elements of L  */
short int                                  /* q[] vector to be computed    */
   row_num[];                             /* row_num[i] - row of element i */
int
   my_id, num_proc;                          /* proc's id and # of proc's */
{
register double
   tmp;
register int
   column, row, pointer;
int
   task, m_next();

   task = m_next();                             /* get pointer into queue    */

   while (task <= unknowns) {
      column = fwd_schedule[task];              /* get column for this task  */
      bck_ready[column] = WAIT;                 /* reset for back solve      */
      pointer = start[column];                  /* get pointer into DS       */
      while (fwd_ready[column] > 1) continue;  /* busy wait until ready     */
      q[column] /= l[pointer++];                /* solve for our q[i]        */
      tmp = q[column];                          /* store it in a local var   */

      while ( pointer < start[column+1]) {
         row = row_num[pointer];
         S_LOCK(lp[row]);                       /* set lock                  */
         q[row] -= l[pointer] * tmp;            /* mult q[i] by column j     */
         fwd_ready[row]--;                      /* decrement depend. vector  */
         S_UNLOCK(lp[row]);                     /* unlock lock               */
         pointer++;                          /* move to next nonzero element */
         }
      task = m_next();                          /* get next task             */
      }
   m_sync();                                 /* synchronize before returning */
   }
```

Figure 3: Procedure for Parallel Forward Solve

Case	Type						
	Sequential	DS		RSS		RDS	
	t_s	t_p	effic.	t_p	effic.	t_p	effic.
1	2.97	1.61	.15	.73	.34	.78	.32
2	2.80	.78	.30	.55	.42	.77	.30
3	3.29	1.07	.26	.72	.38	.99	.28
4	3.84	1.54	.21	.90	.36	1.03	.31
5	11.32	3.48	.27	2.13	.44	2.92	.32
6	13.29	11.04	.10	2.73	.41	3.29	.34
7	665.21	418.21	.13	219.46	.25	206.81	.27

Table 2: Time in seconds and efficiency of parallel forward solve on 12 processors relative to sequential code

1 then we *busy wait*. When `fwd_ready[i]` is equal to 1 the dependences for q_i have been satisfied and we can compute q_i. We set $q_i = q_i/L_{i,i}$ and then loop over the nonzero elements in column i below the diagonal, computing $q_j = q_j - L_{j,i} \times q_i$. Then we decrement the value of the `fwd_ready[j]` array to indicate that one dependence for q_j has been satisfied. The array `q` and the `fwd_ready` array are shared and access to individual elements must be synchronized using the system calls `S_LOCK()` and `S_UNLOCK()`. These synchronization procedures are called once for each nonzero off-diagonal element in the lower triangular matrix each iteration. This locking and unlocking operation takes about half of the time in the forward solve routine when the matrix is stored by columns. We also reset `bck_ready[]` for the next back solve operation.

In Table 2 we show the results for the three methods explained above on seven test problems. This is the time spent during the iterative solution of $Ax = b$ doing forward solves. All times are measured in seconds. We also include the sequential time for for each problem, t_s. The sequential time given is the best sequential code we could write running on one processor; there are no parallel constructs or synchronizations used. Parallel code running on 12 processors of the Sequent took time t_p. We measure efficiency as

$$\text{effic.} = \frac{t_s}{t_p \times \#\text{proc}}, \quad \#\text{proc} = 12.$$

The DS timings are included for comparison to illustrate the benefit of computing the dependence graph and permuting the index set. We see that both RSS and

Case	$t_{RDS} - t_{RSS}$	Predicted	# iterations
1	0.05	0.06	16
2	0.22	0.11	41
3	0.29	0.14	68
4	0.13	0.13	15
5	0.79	0.47	101
6	0.44	0.44	43

Table 3: Time Difference between RDS and RSS vs. Estimated time in seconds

RDS are significantly better than DS, sometimes more than twice as efficient. The RSS method performs better than RDS in all but the last problem. In the first Test Case, RSS and RDS the two took almost the same time, and half as long as dynamic scheduling. In the last case, RDS was more efficient than RSS despite the calls to the global counter. This has several possible explanations. First, the number of nonzero elements in each column was more in the first and last cases than in the second through sixth cases. Thus, the relative overhead associated with the global counter versus the amount of work to do per call is less. In the last case, the number of nonzero elements per column varied between 8 and 40. Good load balancing is especially important in this case for increased throughput. Statically assigning tasks to processors by their depth in the dependence graph alone (as in RSS) cannot achieve this. There must be some way to account for the amount of work to be done in each task, not just the dependences of the task. The RDS method performs better at this than the RSS method as shown in Test Case 7.

When the problems have a regular sparsity structure (most of the columns/rows have the same number of nonzero elements) the time to compute each q_i is roughly the same and the load is balanced as long as each processor gets roughly the same number of q_i's to solve for. Test Cases 1-6 have a regular sparsity structure and thus the RSS method performed slightly better. The main contribution to this difference is the fact that in the RDS technique a global counter is used to maintain the queue of tasks. It takes about 50μ-seconds for each call and this is done before each q_i is computed. A prediction for the time difference when there is a regular sparsity structure is

$$t_{RDS} - t_{RSS} \approx (50\mu\text{-seconds})(\#\text{iterations})\left(\frac{\#\text{unknowns}}{\#\text{proc's}}\right). \qquad (6)$$

In Table 3 we compare the actual difference with the prediction for the first 6 cases. The right most column shows the number of iterations for convergence for each test case. This model gives an estimate of the size of the difference that is correct to within a factor of two.

4.4 Backward Solve

The backward solve is similar to the forward solve, but there are subtle differences in implementation. To solve (4) we carry out the computation as a series of inner products rather than outer products. L^T is accessed by rows since we store L by columns.

An outline of our back solve procedure follows. Just as in the forward solve, we have a list of permuted indices `back_schedule[]`. It is computed in a manner analogous to `fwd_schedule[]`. `back_schedule[]` is treated as a queue of tasks to be computed by the processors. `bck_ready[]` is initialized to the value `''WAIT''`. For some j, if `bck_ready[j] = WAIT` then this indicates that `z[j]` has not been computed yet. Each processor gets an index from the queue as it begins the back solve. Let i be the index that some processor gets. For each nonzero element j in row i of L^T, check `bck_ready[j]`. If `bck_ready[j] = WAIT`, then *busy wait*. Else, compute $\texttt{z[i]} -= \texttt{L}^{\texttt{T}}_{\texttt{i,j}}\texttt{z[j]}$. When all nonzero off-diagonal elements in row i have been used we calculate $\texttt{z[i]} = \texttt{z[i]}/\texttt{L}^{\texttt{T}}_{\texttt{i,i}}$ and set `bck_ready[i] = ''DONE''`. The value `DONE` indicates that the element of `z[]` is computed. Finally, if the queue is not empty get the next index.

The C code for this technique is shown in Figure 4. As in the forward solve routine we compute the new vector in place, overwriting the previous entries of `z[]`. `l[]` is the array containing the nonzero elements of the rows of the upper triangular matrix. The beginning of row i is pointed to by the array `start[i]`. To move across the nonzero elements of row i, from right to left, we start at `pointer = start[i+1]-1`. `start[i+1]` points to $L_{i+1,i+1}$ in `l[]` and `start[i+1]-1` points to the right-most nonzero element in row i. The `bck_ready[]` array is set to "BUSY" during the previous forward solve. Therefore, if `bck_ready[j] = BUSY`, then `z[j]` has not been computed yet in the back solve. To reset `fwd_ready[j]` for the next forward solve we set `fwd_ready[j] = fwd_depend[j]`. `fwd_depend[j]` is the number of nonzero elements in row j of L. To indicate that `z[j]` has been computed in the back solve we set `bck_ready[j]` to "DONE". The `back_schedule[]` array contains the index set that has been permuted appropriately for the back solve operation. Finally, we set a barrier `m_synch()` to synchronize all processors at the end of the procedure before returning.

There is no need to do the locking and unlocking as in the forward solve routine. This procedure only writes to three shared arrays `fwd_ready[]`, `bck_ready[]`, and `z[]`. Each location is read by many processors but only written to by one processor.

```
parallel_bckslv(l, z, row_num, my_id, num_proc)

double
   l[], z[];                                         /* arrays for L and z */
short int
   row_num[];                         /* row_num[i] is row of element l[i]  */
int
   my_id, num_proc;      /* variables for processor # and # of processors */
{
register double
   tmp;
register int
   row, column, pointer;
int
   task, m_next();

   task = m_next();                               /* get first task to do */

   while (task <= unknowns) {
      row = back_schedule[task];
      pointer = start[row+1] - 1;

      column = row_num[pointer];
      tmp = z[row];                      /* copy z to local register variable */

      while ( column > row) {
         while (bck_ready[column] == SPIN) continue;/* busy wait until ready */
         tmp -= l[pointer] * z[column];
         column = row_num[--pointer];             /* get next column number */
         }
      z[row] = tmp / l[pointer];
      bck_ready[row] = DONE;                  /* set flag that it is done */
      fwd_ready[row] = fwd_depend[row];     /* reset for next forward solve */
      task = m_next();                           /* get next one to do  */
      }
   m_sync();    /* get all proc's synched before returning */
   }
```

Figure 4: Procedure for Parallel Backward Solve

Case	Method						
	Sequential	DS		RSS		RDS	
	t_s	t_p	effic.	t_p	effic.	t_p	effic.
1	2.70	.95	.24	.38	.58	.45	.50
2	2.65	.67	.33	.42	.53	.62	.36
3	3.07	.83	.31	.49	.52	.78	.33
4	3.56	1.01	.29	.55	.54	.81	.37
5	10.75	2.63	.34	1.54	.58	2.56	.35
6	12.38	7.18	.14	1.72	.60	2.35	.44
7	507.90	103.92	.41	85.44	.50	85.80	.49

Table 4: Time in seconds and efficiency of parallel backward solve on 12 processors relative to sequential code

No locking is required in this situation. The inner product form of the triangular solve therefore has much less overhead.

In Table 4 we compare the three methods for the backward solve. DS is clearly slower than the other two. It is only included for comparison. We see that the RSS method performs better than the other methods. Just as for the forward solve, the time difference is due to the fact that in RDS a global counter is required to maintain the queue of tasks. The difference is very pronounced for problems 1-6: since there is very little work to do to compute each z_i; i.e., there are only a few nonzero off-diagonal elements in each row/column. The efficiency of RSS and RDS are almost identical for Test Case 7. The load balancing that is provided in RDS makes up for the overhead of using the global counter. The amount of work to be done to compute some z_i is directly related to the number of nonzeros in row i of L. The amount of work per task affects the load balancing. Test Case 7 has the most variation in the number of nonzero elements in its rows (and columns). As the variation increases so does the need to account for this in the scheduling of tasks.

5 Matrix-Vector Product

In this section we discuss the implementation of sparse matrix-vector products on the Sequent. We show that it is more efficient to store the whole symmetric matrix by rows rather than trying to save storage and storing only the lower or upper triangular half. This is true for both the sparse matrix-vector product and the triangular solves. To compute a general symmetric sparse matrix-vector product $Ax = b$ on the Sequent it is more efficient to store all of A by rows than to store only the lower triangular part by rows (or columns).

When a sparse symmetric matrix is stored as a lower triangular matrix by columns or rows (or if the upper triangular matrix is stored by columns or rows) the multiplication must be carried by a combination of inner and outer products. The implementation becomes complicated and requires synchronization to protect elements of shared arrays from being modified by more than one processor at a time.

An implementation of a symmetric sparse matrix-vector product written in C is shown in Figure 5. For this example, only the nonzero elements of the lower triangular part of A are stored (by columns). First, each processor initializes a portion of the array `b[]` to be zero. Next, each processor gets a column of the data structure. This is a column of the lower triangular part of the matrix and a row of the upper triangular part. A column in the lower part, say column i, is multiplied by element `x[i]`. The product is accumulated into the *shared* array `b[]`: `b[j]+=a[pointer]×x[i]`. To be sure that only one processor is writing to `b[j]` at a time we must use the system synchronization routines `S_LOCK()` and `S_UNLOCK()`. Next we multiply the element of column `i` by x[j] and add the product to the local variable `inner_prod`. When we have exhausted all elements of the upper triangular row, we add the local inner product into the shared array `b[]` using the appropriate locks. In essence, we accumulate inner products locally and add outer products globally. This approach requires two system synchronization calls per nonzero element in the lower triangular part of A. Even though the probability of collision is small since we are dealing with a sparse matrix, this has to be done to insure that only one processor updates an element of `b[]`.

A procedure for computing a general sparse matrix-vector product where the full A is stored by rows is much simpler and is shown in Figure 6. Each processor computes a set of inner products. The processors dynamically get an element of `b[]` to compute using the system global counter `m_next()` The array `row_start[]` is an array holding the starting point for each row as it is stored in the data structure. The inner product of each row with x is computed and stored in `b[]`. This algorithm requires no synchronization since the work is divided into non-overlapping groups of rows.

In Table 5 we compare two methods for parallel computation of the sparse matrix vector product with the time it takes to compute it sequentially. The first method, "Symmetric", is the symmetric code from Figure 5. It takes advantage of symmetry and stores only the lower triangular part of the matrix. The second method, "Nosynch", is the same algorithm but we have commented out all of the synchronization calls to `S_LOCK()` and `S_UNLOCK()`. The answer we get is incorrect

```
mult(a, x, b, row_num, first, last, my_id, num_proc)
double
   a[], x[], b[];
short int
   row_num[];
int
   first, last, my_id, num_proc;
{
double
   x_elem, inner_prod;
register int
   pointer, k;
int
   row, column, m_next();

   for (k=first; k<last; k++) b[k] = 0.0;     /* zero array */

   m_sync();

   column = m_next();                    /* get our 1st column to start on */
   pointer = start[column];                /* get position of 1st element */

   while (column <= unknowns) {
      x_elem = x[column];

      inner_prod = a[pointer++] * x_elem;    /*  compute a[i,i]*x[i]   */

      while (pointer < start[column+1]) {
         row = row_num[pointer];
         S_LOCK(lp[row]);
         b[row] += a[pointer] * x_elem;  /* this is part of the outer prod. */
         S_UNLOCK(lp[row]);
         inner_prod += a[pointer++]*x[row];/* this is part of the inner prod. */
         }
      S_LOCK(lp[column]);
      b[column] += inner_prod; /* store the inner product now */
      S_UNLOCK(lp[column]);

      column = m_next(); /* get next column to work on */
      pointer = start[column]; /* get pointer into array       */
      }
   m_sync(); /* wait until everyone else is done */
   }
```

Figure 5: Code for Symmetric Sparse Matrix-Vector Product

```
full_mult(a, x, b, col_num, first, last)

double
   a[],                            /* the nonzero entries of A */
   x[],                            /* the vector to mult by */
   b[];                            /* the result gets put here */
short int
   col_num[];                      /* array of column numbers */
int
   first,                          /* first row we work on */
   last;                       /* we do up to by not including this row */
{
double
   inner_prod;
register int
   pointer,                      /* pointer into global DS */
   row,                        /* row that we are working on */
   column;                     /* column number in row that we are using */

   row = 1;
   while(row <= unknowns) {
      row = m_next();              /* get row to work on */
      inner_prod = 0.0;
/*
 *  compute a[row,*] * x[*]     { inner product}
 */
      for (pointer = row_start[row]; pointer<row_start[row+1]; pointer++){
         inner_prod += a[pointer] * x[col_num[pointer]];
         }
      b[row] = inner_prod;
      }
   m_sync();
   }
```

Figure 6: Code for Full sparse matrix-vector product

Case	Method						
	Sequential	Symmetric		Nosynch		Full	
	t_s	t_p	effic.	t_p	effic.	t_p	effic.
1	5.86	.91	.54	.58	.84	.56	.87
2	5.29	.99	.45	.59	.75	.63	.70
3	6.03	1.06	.47	.71	.71	.66	.76
4	7.23	1.35	.45	.77	.78	.76	.79
5	21.20	3.64	.49	2.41	.73	2.27	.78
6	13.29	3.82	.56	2.35	.91	2.55	.84
7	1270.80	321.85	.33	226.13	.47	207.16	.51

Table 5: Time in seconds and efficiency of parallel Sparse Matrix-Vector product on 12 processors relative to sequential code

but we stop after the same number of iterations. This is to show the impact of the synchronization. It also gives us a lower bound on the time for this method of matrix-vector product. The last method, "Full", is the times from the code in Figure 6.

We see that storing the full matrix is best. Timing and efficiency is better than 70% except for large problems. We expect this, since matrix-vector products are very parallel computations. If we look at the difference between the parallel times of the Symmetric and Nosynch columns it is clear that to use the system synchronization calls adds almost 50% to the cost of the computation. But, even without the synchronization, we see that the Full method is better than the Nosynch method. From this we conclude that there is no advantage in storing only half of a symmetric matrix for parallel computation of the sparse matrix-vector product on this machine.

An alternative to sparse matrix vector multiplication computed as inner or outer products is proposed my Melhem [19] where he suggests a general technique of using striped matrix storage.

6 Triangular Solve Revisited

The decision to store the full matrix A affects other parts of the code. We also stored the full preconditioner as two triangular matrices, L and L^T, both by rows. The new values for the timings of the triangular solves are compared with the old values in

Problem	Method			
	Symmetric		Full	
	fwd	bck	fwd	bck
1	.73	.38	.44	.42
2	.55	.42	.41	.42
3	.72	.49	.53	.48
4	.90	.55	.60	.58
5	2.13	1.54	1.58	1.54
6	2.73	1.72	1.79	1.70
7	206.81	85.80	200.52	216.63

Table 6: Time in seconds for Triangular Solve on 12 processors

Table 6. The columns labeled "Symmetric" are for storing only the lower triangular half of the symmetric matrix. The columns under "Full" are the timings for storing both the upper and lower triangular matrices of the preconditioner by rows. The forward solve is faster because it uses inner products. There is no synchronization for every element of L, only one for each row. We cannot, however, explain the data from Case 7.

7 Parallel Efficiency of ICCG

In Table 7 we show the time required to solve (1) for each implementation, assuming the preconditioner has be previously computed. In the first six cases it is clear that storing the full matrix is better than storing only its lower triangle. The efficiency is near or above 60% for the entire code. This is a very reasonable level and what we expected. But, for the seventh case, the code was not efficient. In Appendix C we show how the time to access array elements increases as a function of the array size and discuss the time for Test Case 7.

8 Scheduling

Other scheduling methods not considered here are discussed in [9,10,14,22]. The general problem is to schedule a set of partially ordered tasks onto a multiprocessor system so that the time required to complete the tasks is miminized. This problem is known to belong to the class of "strong" NP-hard problems. The work by [9,10,22]

Case	Method						
	Sequential	Symmetric		Nosynch		Full	
	t_s	t_p	effic.	t_p	effic.	t_p	effic.
1	15.26	2.54	.50	2.28	.55	1.89	.67
2	17.80	2.97	.50	2.67	.55	2.45	.60
3	21.00	3.89	.45	3.48	.50	3.07	.57
4	22.52	4.16	.45	3.43	.54	3.21	.58
5	72.39	10.83	.56	9.58	.62	8.85	.68
6	78.09	11.01	.59	9.64	.67	8.52	.76
7	2811.14	655.39	.36	609.36	.38	660.19	.35

Table 7: Time in seconds and efficiency of Total ICCG code on 12 processors relative to sequential code

presents bounds on the number of processors required to compute the tasks in a minimum amount of time and bounds the time to compute the tasks with a fixed number of processors. Also, in [10], bounds on the ratio of times for two different feasible schedules are given.

Kasahara and Narita [14] present two different scheduling methods, CP/MISF and DF/IHS. CP/MISF stands for critical path/most immediate successors first and DF/IHS stands for depth first/implicit heuristic search. The primary difference between the two is that the former schedules tasks as soon as possible and the latter schedules tasks as late as possible. Both require sorting of tasks at the same level according to the number of predessors they have and both are $O(N^2)$ algorithms, where N is the number of vertices in the dependence graph.

We do not use either of these scheduling techniques. Sorting the tasks at each level is expensive. We choose a scheduling method that is not optimal but requires very low overhead.

9 Summary

We have discussed different approaches for exploiting parallelism in the ICCG method for solving large sparse symmetric positive definite systems of equations on a shared memory parallel computer. We showed that performing a small amount of analysis to determine the data dependences can drastically improve the parallel efficiency. Additionally, when the sparsity structure of a triangular matrix was regular then a reordered static scheduling method performed more efficiently than a reordered dynamic scheduling method. Finally, we showed that for the Sequent it was more efficient to store the whole symmetric matrix by rows rather than only the upper or lower triangular part. The code for a full matrix was simpler and required less synchronization overhead.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] E. Anderson. *Parallel Implementation of Preconditioned Conjugate Gradient Methods for Solving Sparse Systems of Linear Systems*. Technical Report 805, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, Illinois, August 1988.

[3] D. Baxter, J. Saltz, M. Shultz, and S. Eisenstat. *Preconditioned Krylov Solvers and Methods for Runtime Loop Parallelization*. Technical Report TR-655, Department of Computer Science, Yale University, New Haven CT, October 1988.

[4] G. Bedrosian. *FORTRAN Subroutine Package for Solving Large, Sparse, Symmetric Linear Systems*. Technical Report 84CRD284, General Electric Co., Corporate Research and Development Center, 1984.

[5] G. Bedrosian. Private communication. 1986.

[6] P. Concus, G. H. Golub, and D. P. O'Leary. A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations. In Gene H. Golub, editor, *Studies in Numerical Analysis*, pages 179–198. The Mathematical Association of America, 1984.

[7] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[8] I. S. Duff, R. G. Grimes, and J. G. Lewis. *Sparse Matrix Problems*. Technical Report CSS 191, Harwell Laboratory, October 1987.

[9] E. B. Fernandez and B. Bussell. Bounds on the number of processors and time for multiprocessor optimal schedules. *IEEE Trans. Comput.*, c-22(8):745–751, Aug 1973.

[10] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM J. Comput.*, 4(2):187–200, 1975.

[11] G. H. Golub and C. F. VanLoan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, Maryland, 1983. Second Printing.

[12] A. Greenbaum. *Solving Triangular Linear Systems Using FORTRAN with Parallel Extensions on the NYU Ultracomputer Prototype.* Technical Report 99, Courant Institute, New York University, New York, NY, 1987.

[13] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. National Bureau of Standards*, (49):409–436, 1952.

[14] H. Kasahara and Narita S. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. Comput.*, c-33(11):1023–1029, Nov 1984.

[15] A. Koniges. *Parallel Processing of a Preconditioned Biconjugate Gradient Algorithm in CRAY supercomputers.* Technical Report, Lawrence Livermore National Lab, Livermore, CA, 1987.

[16] A. Lichnewski. Some vector and parallel implementations for preconditioned gradient algorithms. In J. Kowalik, editor, *Proceedings of the NATO workshop on high speed computing*, pages 343–359, 1984.

[17] T. A. Manteuffel. *The Shifted Incomplete Cholesky Factorization.* Technical Report SAND78-8226, Sandia Laboratories, May 1978.

[18] J. A. Meijerink and H. A. Van der Vorst. An iterative solution method for linear equation systems of which the coefficient matrix is a symmetric M-Matrix. *Math. Comp.*, 31:148–162, 1977.

[19] R. G. Melhem. Solution of linear systems with striped sparse matrices. *Parallel Computing*, 6:165–184, 1988.

[20] G Meurant. Multitasking the conjugate gradient method on the CRAY X-MP/48. *Parallel Computing*, 5:267–280, 1987.

[21] A. Osterhaug. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Computer Systems, Inc., 1986. Sequent Technical Publications.

[22] C. V. Ramamoorthy, K. M. Chandry, and M. J. Gonzales Jr. Optimal scheduling strategies in a multiprocessor system. *IEEE Trans. Comput.*, c-21(2):137–146, Feb 1972.

[23] J. K. Reid. On the method of conjugate gradients for the solution of large sparse systems of linear equations. In J. K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 231–254, Academic Press, New York, 1971.

[24] Y. Saad. *Krylov Subspace Methods on Supercomputers.* Technical Report TR88.40, RIACS, NASA Ames Research Center, Moffett Field, CA 94035, September 1988. To appear SIAM J. SCI. STAT COMPUT.

[25] J. Saltz, R. Mirchandaney, and D. Baxter. *Run-Time Parallelization and Scheduling of Loops.* Technical Report ICASE Report No. 88-70, ICASE, NASA Langley Research Center, Hampton, VA 23665, December 1988.

[26] M. K. Seager. *Parallelizing Conjugate Gradient Method for the CRAY X-MP.* Technical Report, Lawrence Livermore National Lab, Livermore, CA, 1984.

[27] Omar Wing and John W. Huang. A computational model of parallel solution of linear equations. *IEEE Trans. on Computers*, 29(7):632–638, 1980.

A Test Problems

For this work we have chosen 7 test cases which are representative of the class of problems solved by iterative methods. All are from two-dimensional domains. The first five are from the Harwell-Boeing collection [8] and the last two are from electro-magnetic analysis [5]. The test cases are described in Table 8.

Case	Ref.	Description	Order	Nonzeros
1	[8]	A nine point discretization of the Laplacian on a unit square with Dirichlet boundary conditions. LAP30	900	4322
2	[8]	Matrix used in modeling power system networks. PSADMIT1	662	1568
3	[8]	Matrix used in modeling power system networks. PSADMIT2	494	1080
4	[8]	Matrix used in modeling power system networks. PSADMIT3	685	1967
5	[8]	Matrix used in modeling power system networks. PSADMIT4	1138	2596
6	[5]	A first-order triangular finite element discretization of the Laplacian operator on a unit square.	2500	7251
7	[5]	Matrix from a nonlinear magnetostatic model of a permanent magnet motor, using an unstructured finite element mesh with mixed triangular and quadrilateral third-order elements.	6517	69,670

Table 8: Test Case Descriptions

B Sequent Overview and Performance Figures

This section provides an overview of the architecture of the Sequent Balance 21000 and the execution times for the operations that were used in the timings given in this paper. The architectural description is due to Osterhaug [21].

B.1 The Sequent Architecture

The Sequent Balance 21000 is a shared memory multiprocessor. The processors are identical 10-MHz National Semiconductor 32032's. These are 32-bit processors. They operate on a peer basis, executing a single copy of the operating systems executive, or "kernel".

There is no designated "master" cpu. All processors. memory modules, and i/o controllers plug into a single high-speed bus. There is hardware support for mutual exclusion - to support exclusive access to shared data structures. the system includes up to 64K user-accessible hardware spin-locks.

The system we used has 12 processors and 28 Mbytes of memory. In addition. each cpu has 8 Kbytes of local RAM and 8 Kbytes of cache RAM. The local RAM holds a copy of certain frequently used kernel code and read-only kernel data structures. The cache RAM holds blocks of system memory most recently used by the cpu.

	Operand		
Operation	4-Byte Integer	4-Byte Real	8-Byte Real
Addition	4.4	32.4	18.9
Multiplication	12.7	28.1	20.8
Division	17.0	33.0	25.5

Table 9: Time in μseconds for Arithmetic Operations

B.2 System Timing

This section provides execution times in microseconds for a variety of operations that were used by the programs discussed in this paper. Times for arithmetic operations are shown in Table 9. These timings are computed by looping through a program segment 50.000 times. The time before the loop was executed was then subtracted from the time at the end of the loop. Some time was subtracted for loop overhead and then that time was divided by the number of iterations through the loop.

Locking and unlocking of locations in the hardware atomic lock memory was done by the in-line C macros `S_LOCK()` and `S_UNLOCK()`. If we assume there is no contention for the lock, locking and unlocking a lock takes a total of 35 microseconds. The system provided routines in the Parallel Programming Library were slower, taking 53 microseconds. A function, `m_next()`, is provided to increment a global counter and return the current value. This function takes an average of 49 microseconds per call.

C Memory Access Times on the Sequent

In the ICCG method every element of L, L^T, and A are read once each iteration. To explain the inefficiency of Test Case 7 we ran a simple test that iterates over different array sizes accessing each element once per iteration. We measured the average time to access an array element as a function of the size of the array. We created a program with a double-precision array, **big_array[]**, with 200,000 elements. Then, we timed the following two loops:

```
for (test_size =1000; test_size<10000; test_size += 1000) {
   timer(&start_time);
   for (i=0; i<200; i++) {
      for (j=0; j<test_size; j++) {      /* first loop  */
         local = big_array[j];
         }
      }
   sep_timer(&end_time);
   }

for (test_size =10000; test_size<200001; test_size += 10000) {
   timer(&start_time);
   for (i=0; i<200; i++) {
      for (j=0; j<test_size; j++) {      /* second loop */
         local = big_array[j];
         }
      }
   sep_timer(&end_time);
   }
```

We copy the elements of the array one at a time to a scalar variable **local**. In the first loop, the number of array elements accessed, **test_size**, varies from 1,000 to 9,000 in increments of 1,000. In the second loop, the number of array elements accessed, **test_size**, varies from 10,000 to 100,000 in increments of 10,000. We loop 200 times for each test size and divide the time by the total number of array accesses. This was done 5 times for each test and the results were averaged.

The timing routine returns both user time (**utime**) and system time (**stime**) separately rather than the sum of the two We used the sum in all previous timings. These quantities are defined as follows:

user time the total amount of time spent executing in user mode

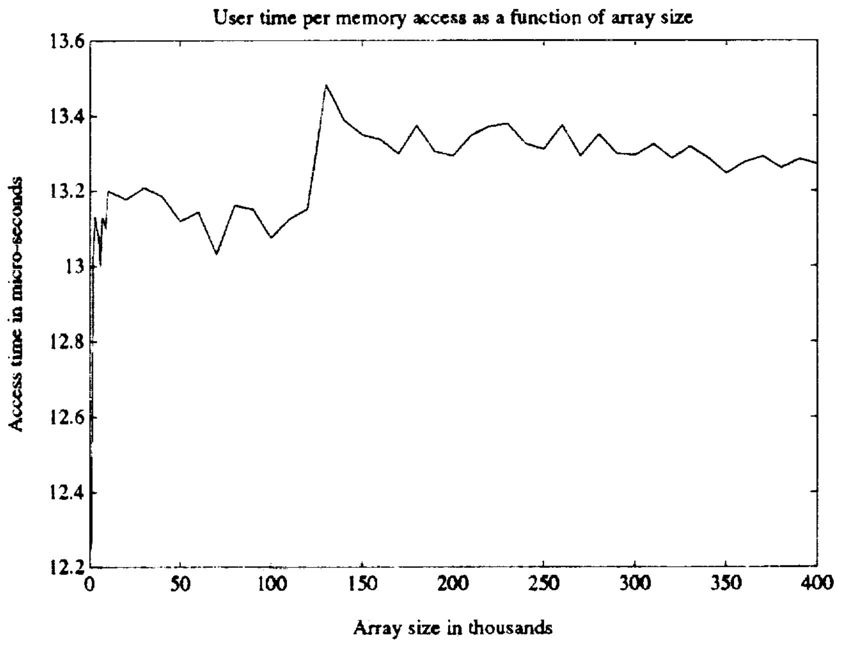

Figure 7: User Time in μsecs for Memory Access as a function of Array Size

system time the total amount of time spent in the system on behalf of the process.

The results of this test are shown in Figures 7 and 8. The times are measured in microseconds. There is approximately a 2% increase in user time per access as the array size is increased from 120K to 130K. But, there is a factor of 5 increase in system time per access as the number of array elements in the test case increases from 120K to 130K. Recall, the total storage for full A in Test Case 7 is 132,823 double-precision numbers. The Sequent takes longer to access each element for this large problem than for all the other test cases.

In Table 10 we show separate entries for the user time and the system time for the sequential, symmetric and full matrix implementations of Test Case 7. The feature to notice is the drastic increase in system time for the forward and backward solve in the "Full" case as compared with the corresponding times of the "Symmetric" and the "Sequential" implementations. This is partially explained by the test loops above. We see that large problems such as Test Case 7 are not efficient on the Sequent.

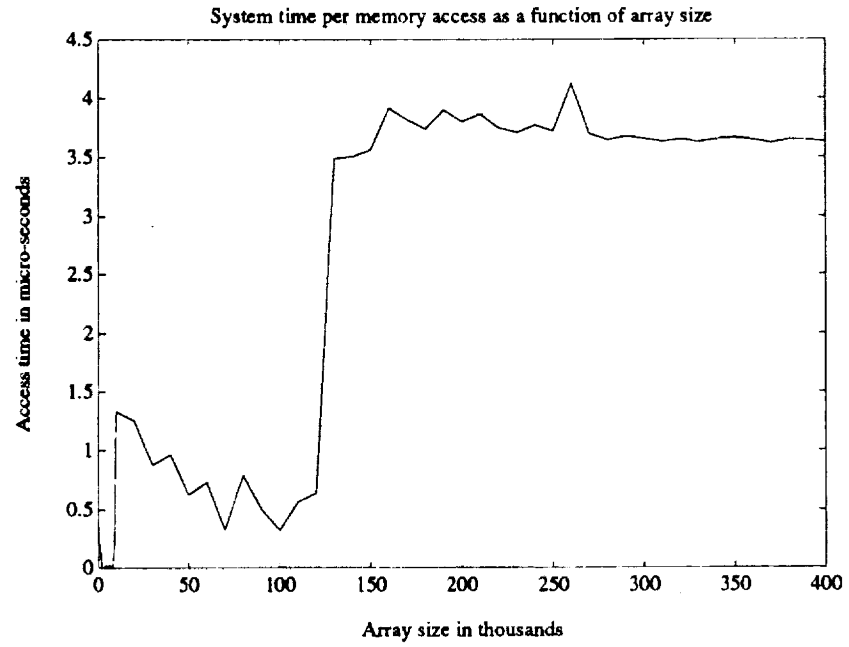

Figure 8: System Time in μsecs for Memory Access as a function of Array Size

Problem		Operation			
		fwd	bck	mult	total
sequential	utime	576.05	506.50	1157.40	2575.25
	stime	94.53	4.67	119.74	254.36
	totals	670.58	511.17	1277.14	2829.61
symmetric (parallel)	utime	117.97	65.85	182.06	397.57
	stime	88.84	19.95	143.97	257.82
	totals	206.81	85.80	326.03	655.39
full (parallel)	utime	73.33	72.17	173.28	349.94
	stime	*127.19*	*144.46*	34.18	310.25
	totals	200.52	216.63	207.46	660.19

Table 10: Detailed timing of Case 7 in seconds