NASA Contractor Report 182056

ICASE INTERIM REPORT 11

The Preprocessed Doacross Loop

Joel H. Saltz
Ravi Mirchandaney

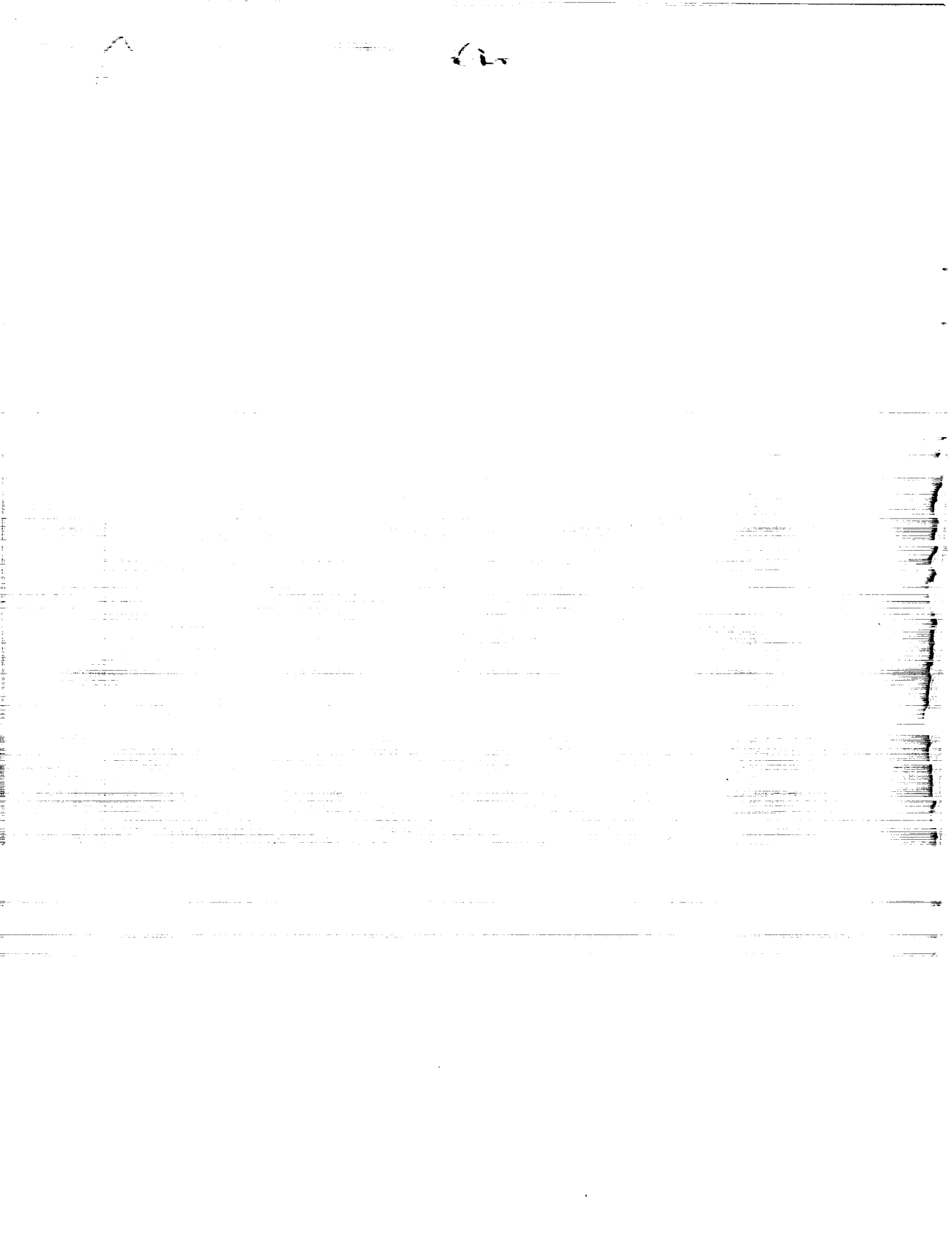
NASA Contract No. NAS1-18605
May 1990

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225



ICASE INTERIM REPORTS

ICASE has introduced a new report series to be called ICASE Interim Reports. The series will complement the more familiar blue ICASE reports that have been distributed for many years. The blue reports are intended as preprints of research that has been submitted for publication in either refereed journals or conference proceedings. In general, the green Interim Report will not be submitted for publication, at least not in its printed form. It will be used for research that has reached a certain level of maturity but needs additional refinement, for technical reviews or position statements, for bibliographies, and for computer software. The Interim Reports will receive the same distribution as the ICASE Reports. They will be available upon request in the future, and they may be referenced in other publications.

Robert G. Voigt
Director

The Preprocessed Doacross Loop *

Joel H. Saltz
ICASE NASA Langley Research Center
Hampton, VA 23665

Ravi Mirchandaney
Department of Computer Science
Yale University
New Haven, CT 06520

May 29, 1990

Abstract

Dependencies between loop iterations cannot always be characterized during program compilation. *Doacross* loops typically make use of a-priori knowledge of inter-iteration dependencies to carry out required synchronizations. We propose a type of *doacross* loop that allows us to schedule iterations of a loop among processors without advance knowledge of inter-iteration dependencies. The method proposed for loop iterations requires us to carry out parallelizable preprocessing and postprocessing steps during program execution.

*This work was supported by NASA grant NAS1-18605 while the authors were in residence at ICASE, NASA Langley Research Center

1 Introduction

Dependencies between loop iterations cannot always be characterized during program compilation. This inability to characterize dependencies can inhibit exploitation of potential parallelism if one is restricted to usual types of parallel loop constructs, i.e. *doall* or *doacross* loops [3] [2]. *Doall* loops do not impose any ordering on loop iterations while *doacross* loops impose a partial execution order in the sense that some of the iterations are forced to wait for the partial or complete execution of some previous iterations. Typically, *doacross* loops make use of a-priori knowledge of inter-iteration dependencies to carry out required synchronizations.

The method we outline here is a variant of a *doacross* loop that allows us to schedule iterations of a loop onto processors in the absence of prior knowledge about inter-iteration dependencies. We call this type of *doacross* loop the *preprocessed doacross.*

We use symbolic transformations to produce from a given loop: (1) *inspector* procedures that perform execution time preprocessing, and (2) *executors* or transformed versions of source code loop structures. These transformed loop structures carry out the calculations planned in the *inspector* procedures. Characterizing the cost of execution time preprocessing is a critical aspect of this research. One requirement is that the execution time preprocessing itself be parallelizable. The preprocessing required for the *preprocessed doacross* loop is fully parallelizable.

In Section 2, we describe the *preprocessed doacross* parallel construct, and in Section 3 we present results from two sets of experiments designed to characterize the performance tradeoffs manifest by using this construct.

2 The Preprocessed Doacross Loop

2.1 Overview

A doacross loop is frequently used when one needs to parallelize loops with non-independent loop iterations. Typically it is necessary, before executing the loop, to know the distances of dependencies between statements in different loop iterations. It is possible to carry out a simple form of execution time preprocessing that eliminates the need to know dependency distances.

```
do i=1,N
 y(a(i)) = .....    y(b(i))  ....
end do
```

Figure 1: Loop with Execution Time Determined Dependencies

```
parallel do i=1,N
S1:  while(ready(b(i)).eq.NOTDONE)
     endwhile

S2:  y(i) = .... y(b(i)) .....
S3:  ready(i) = DONE
end parallel do
```

Figure 2: Parallelized Loop with True Dependencies

In Figure 1, we present a code fragment that will be used to demonstrate the structure of the inspector and executor loops in a simplified preprocessed doacross loop. We assume that there are no output dependencies between left hand side array references; in Figure 1 this means that no two elements of array a have the same value.

We first assume that all dependencies are true dependencies, i.e., `a(i) = i` and `b(i) < i`. As we show in Figure 2, we can use a shared array `ready` to make certain that the data dependencies are satisfied. Before the loop executes, `ready` is initialized to `NOTDONE`; when a new array element `y(i)` is calculated, we set `ready(i) = DONE` (statement S3). When `y(b(i))` is required to satisfy a dependence in Figure 2, a busy wait is carried out (Statement S1) until `y(b(i))` has been calculated.

In the case when some of the `b(i) > i`, the dependence relations between loop iterations are in fact antidependencies. To accommodate these antidependencies, we transform the loop in Figure 1 so that during the course of the computation, all writes to y in Figure 1 are transformed into writes to

a new array ynew. A reference to y(b(i)) in Figure 1 may or may not have already been written to during an earlier loop iteration. When b(i) < i, we use ynew(b(i)) in the right hand side of the transformed loop and when b(i) $\geq$ i we use y(b(i)). In many cases it will be necessary to copy the newly computed elements of ynew back into y after the computation in the loop is done.

If we do not assume that a(i) is equal to i, the order in which elements of y are written in the sequential loop (Figure 1) is determined by integer array a. When a right hand side array element y(b(i)) needs to be accessed, we will need to determine whether we should use an old or an updated value of y. If y(b(i)) in Figure 1 is written to during an earlier loop iteration j < i we use y(b(i)) in the transformed code, otherwise we use ynew(b(i)). An array iter can be initialized during a preprocessing phase, so that

- the value i is stored in iter(a(i))
- all other elements of iter(a(i)) are set equal to a large integer (MAXINT).

If iter(a(i)) < i for some iteration i of the transformed loop, a true dependency involving y exists and we use y(b(i)). Alternately, if iter(a(i)) > i we use ynew(b(i)).

In order to limit the cost of initialization and the use of memory associated with this implementation of the doacross construct, we reuse the same arrays iter and ready for multiple preprocessed doacross loops. A (parallelized) postprocessing phase can be carried out after the loop is finished during which iter(a(i)) is set equal to MAXINT and ready(a(i)) is set equal to NOTDONE. Figure 3 serves to summarize pre and postprocessing required for the preprocessed doacross loop.

2.2 A More Complex Example

In this section, we will examine in some detail how the doacross transformations would be carried out in a slightly more complex case. Following this exposition, experimental results obtained from this example will be presented in Section 3.

In the loop S1 in Figure 4, up to M+1 separate elements of y are read (otice that the inner loop goes from 1 to M). The right hand side elements of y in

```
          Preprocessing

parallel do i=1,N
    iter(a(i)) = i
end parallel do

          Postprocessing

parallel do i=1,N
      iter(a(i)) = MAXINT
      ready(a(i)) = NOTDONE
      yold(a(i)) = ynew(a(i))
end parallel do
```

Figure 3: Pre and Postprocessing Steps

iteration i may or may not have dependency relations with any loop iteration of S1 (including iteration i itself). Any dependency can be either a true dependency or an antidependency. Figure 5 depicts a transformed version of the loop shown in Figure 4. As was described in Section 2.1, `iter(a(i))` is set to i before the parallelized loop is executed. When `iter(b(i)+nbrs(j))` is less than or equal to i, we use ynew, the newly computed value of y (statements S5 and S8). Note that when `iter(b(i)+nbrs(j))` is strictly less than i (statement S3), it is necessary to make sure that the true dependency is satisfied. When `iter(b(i)+nbrs(j))` is equal to i, we do not busy wait because the dependency is within iteration i. Finally, when `iter(b(i)+nbrs(j))` is greater than i, either a reference to y from some later loop iteration is related to `y(b(i)+nbrs(j))` by an antidependency relation or alternately, `y(b(i)+nbrs(j))` is not written to anywhere in the loop nest. In either case, we use the old value of y and do not busy wait (statement S7). After the parallelized loop is completed, postprocessing analogous to that depicted in Figure 3 is carried out.

```
S1 do i=1,N
     do j=1,M
     y(a(i)) = y(a(i)) + val(j)*y(b(i) + nbrs(j))
     end do
   end do
```

Figure 4: Preprocessed Doacross Test Loop

```
S1 parallel    do i=L,N
S2       ynew(a(i)) = y(a(i))
       do j=1,M
         offset = b(i) + nbrs(j)
         check = iter(offset) - i
S3        if(check.lt.0) then
S4           while(ready(offset).ne.DONE)
             endwhile
S5           ynew(a(i)) = ynew(a(i)) + val(j)*ynew(offset)
S6      else if (check.gt.0)
S7            ynew(a(i)) = ynew(a(i)) + vals(j)*y(offset)
        else
S8            ynew(a(i)) = ynew(a(i)) + vals(j)*ynew(offset)
        endif
       end do
     ready(a(i)) = DONE
end parallel do
```

Figure 5: Parallelized Preprocessed Doacross Test Loop

2.3 Further Variants

The transformations we have described in this paper utilize several arrays to schedule the iterations in parallel. These arrays will typically be the size of the index set, resulting in large utilization of memory. There are a number of ways in which the memory used by the preprocessed doacross can be reduced. It is possible to transform the original loop L into a pair of nested loops L_{inner} and L_{outer}. The inner loop L_{inner} would range over contiguous iterations of the original loop L. Loop L_{inner} would be parallelized using the preprocessed doacross methods described above; loop L_{outer} would be carried out in a sequential manner. Preprocessing and postprocessing involving arrays `ready`, `iter` , `ynew` , and `yold` is carried out before and after each set of L_{inner} iterations. This transformation reduces memory requirements because during each iteration of L_{outer} we can reuse `ready` and `iter`.

When the left hand side arrays are indexed by a linear subscript function (i.e. `a(i)` is replaced by some known linear function $c \times i + d$). it is possible to eliminate the execution time preprocessing phase along with the need to allocate storage for array `iter`. For the loop depicted in Figure 4, we can determine whether `y(b(i) + nbrs(j))` can be written to by testing to see whether `(b(i) + nbrs(j) - d` $\underline{mod}$ `c)` is equal to 0. If a write is carried out it occurs during loop iteration $(b(i) + nbrs(j) - d)/c$.

3 Performance of Preprocessed Doacross

In this section, we provide experimental results for the performance of the inspectors and executors described in Section 2. The following timings were done on an Encore Multimax/320 with 13 megahertz APC/02 boards and version 2.1 of the FORTRAN compiler. Parallel efficiency is defined as $T_{seq}/(p * T_{par})$, where T_{seq} is the time required to solve a problem using an optimized sequential version, T_{par} is the time required on the same problem using a parallel code on p processors.

3.1 Preprocessed Test Loop

In this section we report on some experiments to characterize the performance of the *preprocessed doacross* loop construct. We consider Figure 4, where we have initialized arrays `nbrs` and `a`, such that `nbrs(j) = 2j-L`, and `a(i)`

= 2i. We parallelize this loop using the *preprocessed doacross* construct. In the data presented below, we assess the costs of the preprocessing and postprocessing outlined in Section 2.1. So that we may be better able to interpret the test results, in Figure 5, we have chosen to initialize a using a simple linear left hand side array index subscript function. We use the transformations described in Section 2.2.

In Figure 6 we depict parallel efficiencies on 16 processors obtained when we set N equal to 10000, M equal to either 1 or 5, and varied L from 1 to 14. Recall that in loop S1 in Figure 4, up to M + 1 separate elements of y are read. For odd numbered values of L, there are no dependencies between outer loop iterations. The efficiencies we see for those L values reflect the overheads of :

1. performing the runtime preprocessing and postprocessing
2. performing execution time dependency checks

For M equal to 1 and 5 efficiencies observed for odd L values are approximately 33% and 50% respectively.

The efficiencies for even values of L increase monotonically for both values of M. This is understandable because as L increases, the number of outer loop iterations between dependencies also increases.

3.2 Sparse Triangular Solves

We now consider a slightly different test loop which is used to solve sparse triangular systems of equations. Many of the sparse triangular systems we use for model problems arise from incompletely factored matrices obtained from a variety of discretized partial differential equations. The solution of these sparse triangular systems accounts for a large fraction of the sequential execution time of linear solvers that use Krylov methods[1]. The data dependencies between the elements of y are determined by the values assigned to the data structure column during program execution. These dependencies inhibit the parallelization of the outer loop (statement S1, Figure 7). A description of the structure of the triangular systems used in our experiments is found in [1], outlined in the appendix is a brief description of how these systems were generated.

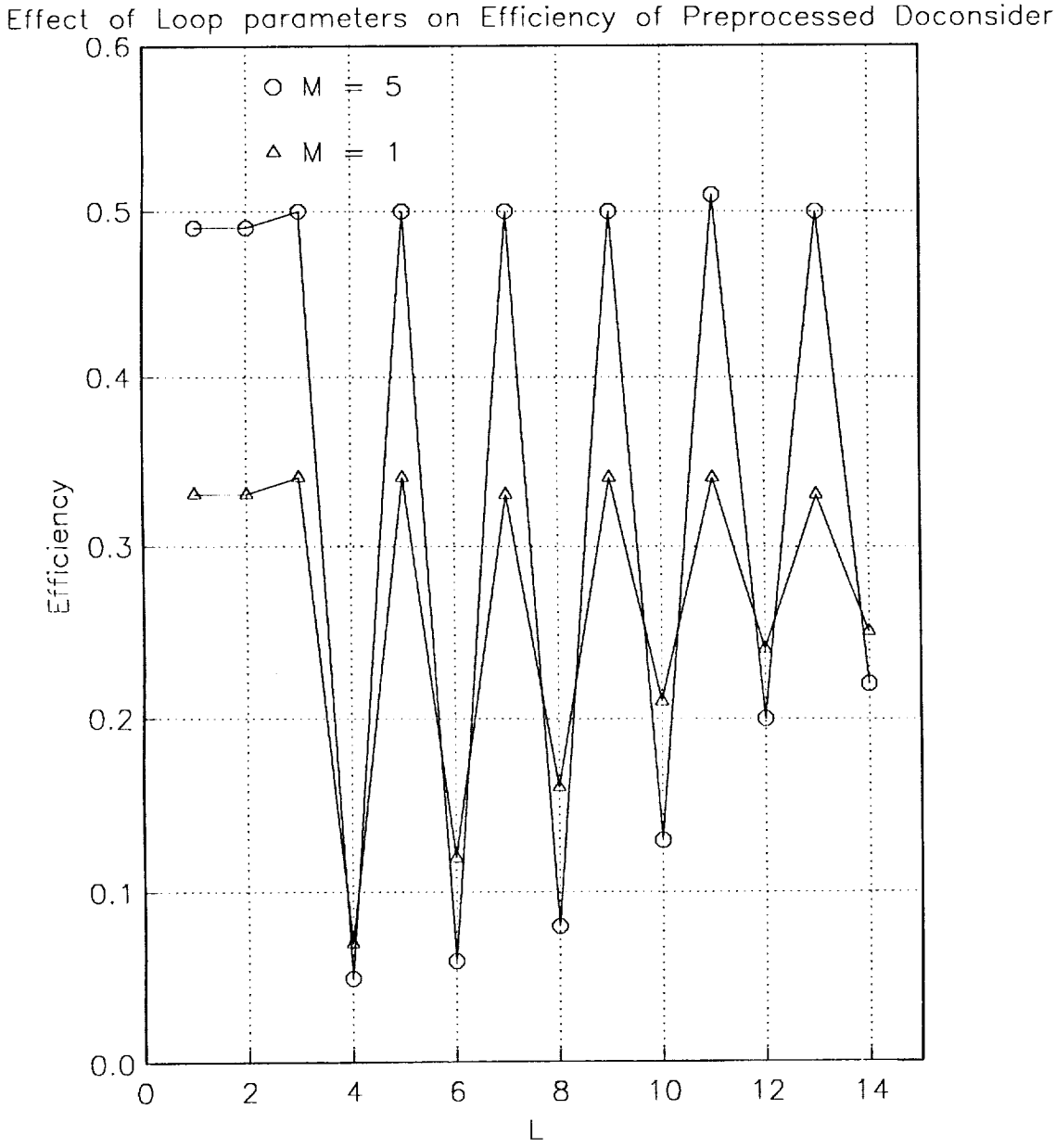


Figure 6: Preprocessed Doacross Efficiencies

```
S1    do i=1,n
        y(i) = rhs(i)
        do j=low(i),high(i)
          y(i) = y(i) - a(j)*y(column(j))
        end do
      end do
```

Figure 7: A Sparse Triangular Solve

Table 1: Preprocessed Doacross Times for Sparse Triangular Matrices

Test Problem	Preprocessed Doacross Time (ms)	Preprocessed Doacross Iterations Rearranged Time (ms)	Sequential Time (ms)
SPE2	34	21	223
SPE5	45	23	241
5-PT	37	19	192
7-PT	84	56	616
9-PT	97	58	698

The loop in Figure 7 was parallelized on 16 processors and the parallelized and sequential times for the test matrices examined are depicted in Table 1. The timings obtained corresponded to parallel efficiencies between 0.32 to 0.46. A modified loop was produced by carrying out the loop iterations in a more advantageous order. This reordering of loop iterations leaves the inter-iteration dependencies unchanged but reduces the effects of these dependencies on performance. The mechanism for carrying out this iteration reordering is described in [4] and is called a *Doconsider* transformation. The resulting loop is parallelized using the preprocessed doacross mechanism and the results are presented below in Table 1. Parallel efficiencies depicted in that table range from 0.63 to 0.75.

4 Conclusion

The *preprocessed doacross* loop is a type of *doacross* loop that allows us to schedule loop iterations onto processors without prior knowledge of inter-iteration dependencies. We have demonstrated that such a loop structure can allow parallelization of loops that would not otherwise be easily parallelized. The overheads required to parallelize loops in this manner can be substantial but should not prevent us from achieving overall performance gains in many cases.

5 Appendix: Definition of Test Triangular Systems

The the triangular systems referred to in Section 3.2 were derived from the following partial differential equation discretizations:

SPE2 This problem arises from the thermal simulation of a steam injection processes. The grid is 6x6x5 with 6 unknowns per grid point, this yields a system with 1080 equations. The matrix is a block seven point operator with 6x6 blocks.

SPE5 This problem arises from a fully-implicit, simultaneous solution simulation of a black oil model. It is a block seven point operator on a 16x23x3 grid with 3x3 blocks yielding 3312 equations.

5-PT The problem is a five point central difference discretization on a 63 x 63 grid; this yields a system with 3969 equations.

7-PT The problem is a seven point central difference discretization on a 20 x 20 x 20 grid; this yields a system with 8000 equations.

9-PT The problem is a nine point box scheme discretization on a 63 x 63 grid; this yields a system with 3969 equations.

References

[1] D. Baxter, J. Saltz, M. Schultz, S. Eisentstat, and K. Crowley. An experimental study of methods for parallel preconditioned krylov methods. In

Proceedings of the 1988 Hypercube Multiprocessor Conference, Pasadena CA, pages 1698,1711, January 1988.

[2] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *The Proceedings of the ICPP, 1986*, pages 836–844, 1986.

[3] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Trans. on Computers*, 29(9):763–776, September 1980.

[4] J. Saltz, R. Mirchandaney, and K. Crowley. The doconsider loop. In *Proceedings of the 1989 ACM International Conference on Supercomputing , Crete, Greece*, pages 29–40, June 1989.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-182056 ICASE Interim Report No. 11		

4. Title and Subtitle	5. Report Date
THE PREPROCESSED DOACROSS LOOP	May 1990
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Joel H. Saltz Ravi Mirchandaney	Interim Report 11
	10. Work Unit No. 505-90-21-01
9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18605
	13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Final Report

16. Abstract

Dependencies between loop iterations cannot always be characterized during program compilation. **Doacross** loops typically make use of a-priori knowledge of inter-iteration dependencies to carry out required synchronizations. We propose a type of **doacross** loop that allows us to schedule iterations of a loop among processors without advance knowledge of inter-iteration dependencies. The method proposed for loop iterations requires us to carry out parallelizable preprocessing and postprocessing steps during program execution.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
dependencies, loops, shared memory, doacross, compilation, runtime, inspector, execution	59 - Mathematical and Computer Sciences (General) 61 - Computer Programming and Software Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	14	A03

NASA FORM 1626 OCT 86