On the automatic parallelization of sparse and irregular Fortran programs [1]

Yuan Lin * and David Padua
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
E-mail: {yuanlin,padua}@uiuc.edu

Automatic parallelization is usually believed to be less effective at exploiting implicit parallelism in sparse/irregular programs than in their dense/regular counterparts. However, not much is really known because there have been few research reports on this topic. In this work, we have studied the possibility of using an automatic parallelizing compiler to detect the parallelism in sparse/irregular programs. The study with a collection of sparse/irregular programs led us to some common loop patterns. Based on these patterns new techniques were derived that produced good speedups when manually applied to our benchmark codes. More importantly, these parallelization methods can be implemented in a parallelizing compiler and can be applied automatically.

1. Introduction

1.1. Sparse and irregular computation

Sparse matrices are used in many scientific computation problems, such as molecular dynamics simulations, computational fluid dynamics solvers, and climate modeling. Although algorithms for dense matrix operations can be used for sparse matrix, it is convenient and often necessary to take the advantage of the presence of many zeros. Thus, in sparse computations, only the nonzero elements of the matrix are stored (to save memory), and only nonzero elements are operated on (to save computation time). Instead of using the *regular* array to represent dense matrices, a sparse matrix requires more complex data structure to store the value of nonzero elements and the index information indicating their position in the regular matrix. Matrix algorithms are also different because they need to be aware of the sparse matrix storage format. The algorithm should also handle problems that do not arise in dense matrix operation. An obvious example is *fill-in*, which happens when the value of an element changes from zero to nonzero during the computation.

We say that sparse matrix computations are *irregular* in contrast to dense matrix computations which are usually called regular computations.

1.2. Parallelization of sparse computation

Sparse matrix computations usually contain more inherent parallelism than their dense counterparts, but are more difficult to execute with high efficiency on parallel machines [11,7]. The difficulty is partly due to data distribution and task balancing. For example, the normal data distribution schemes for dense matrices, such as block, cyclic and a combination of both, are not efficient for sparse matrices where accessing patterns are usually irregular. Solving this problem with a compiler has been the focus of many recent studies. Several compile-time or run-time solutions have been proposed, and great improvements have been achieved. However, the very first question of parallelization, that is, how to find the parallelism, still remains a problem that, to a large degree, has to be solved manually.

1.3. Automatic parallelization of sparse, irregular programs

Automatic parallelization has been a useful alternative to manual parallelization for regular, dense computations [4]. However, automatic parallelizing techniques for sparse, irregular problems are not well understood. Compilers usually rely on data dependence tests to test parallelism. The effectiveness of the data dependence test is determined by its ability to analyze array subscripts. However, in sparse/irregular

[1]This work is supported in part by Army contract DABT63-95-C-0097; Army contract N66001-97-C-8532; NSF contract MIP-9619351; and a Partnership Award from IBM. This work is not necessarily representative of the positions or policies of the Army or Government.

*Corresponding author.

Scientific Programming 7 (1999) 231–246
ISSN 1058-9244 / $8.00

programs, indirect addressing via pointers or indices stored in auxiliary arrays is often used. As a result, the value of array subscripts can be difficult or impossible to know at compile time. For this reason, automatic parallelization is usually believed to be less effective at exploiting implicit parallelism in sparse codes than in their dense counterparts.

Fig. 1 shows the Compressed Row Storage (CRS) format which is commonly used to store a sparse matrix. Three 1D arrays are used to represent a 2D array. All nonzero elements of a sparse matrix, A, are stored row by row in `data()`. `rptr()` is the row pointer array. The ith element of `rptr()` gives the beginning position within `data()` where row i is stored. Column index array `colind()` has the column number of each element in `data()`. In CRS format, the matrix element `A(i,j)` is in `data(rptr(i)+k)`, where `j=colind(rptr(i)+k)`. A similar format is the Compressed Column Storage(CCS) format, which stores the matrix column by column rather than row by row. Many different storage schemes have been proposed for special problems and for matrices of different structures.

Although it is widely believed that automatic detection of parallelism in sparse computations is difficult or impossible due to the presence of complex subscript array expressions, there is practically no empirical evidence to support this belief. This paper makes three principle contributions. First, by studying a collection of sparse/irregular programs, it shows that automatic parallelism detection techniques can be applied in sparse computations. Second, it describes what kinds of new techniques are required for the automatic parallelism detection in sparse codes and gives a detailed discussion about these techniques. Third, it provides experimental results demonstrating the effectiveness of these techniques.

1.4. The methodology

We started the work reported in this paper by studying a collection of sparse and irregular programs written in Fortran 77 to determine the applicability of automatic parallelization. We found that, despite their irregular memory reference patterns, many loops in the sparse/irregular codes we have studied are parallel. We also found that many of the loops are identified as parallel by traditional compiler analysis techniques. We then derived several new transformation techniques that can be applied automatically by a compiler to parallelize the loops that cannot be handled by existing techniques. Manually applying these techniques to our collection of irregular programs resulted in good speedups. This result strengthened our belief that automatic parallelizing techniques can work on sparse/irregular programs as well as they do on dense and regular codes.

As in our early Polaris study [4], our current work focuses on loop level parallelism in shared memory programs. Although we don't deny the importance of other constructs and parallelism models, we believe focusing on loops and shared memory programming model is an important first step in understanding difficulties in automatic parallelization of sparse/irregular programs.

The remainder of this paper is organized as follows. Section 2 describes the benchmark programs we collected for this work. The most important loop patterns found in our benchmarks are studied in Section 3. Section 4 discusses and proposes the newly identified transformation techniques. The effectiveness of these techniques is evaluated in Section 5. Section 6 compares our work with that of others. And, Section 7 concludes this paper.

2. The benchmark suite

We began our work by studying a collection of sparse/irregular programs written in Fortran 77. Table 1 lists all the codes in this benchmark suite [15]. We chose them for the same reason programs were chosen for the collection of HPF-2 motivating applications: "they include parallel idioms important to full-scale 'sparse/irregular' applications" [8]. The programs in our suite are small. This enabled us to complete the hand analysis in a reasonable time.

3. Loop patterns

3.1. Overview

As we discussed above, sparse/irregular programs usually are considered difficult for automatic parallelization because they use indirectly accessed arrays and the subscript values often are unknown until runtime. However, in order to get an in-depth understanding of this problem, we studied how indirectly accessed arrays are used in our benchmark collection. Our most surprising finding was that efficient methods can be used to automatically analyze and parallelize

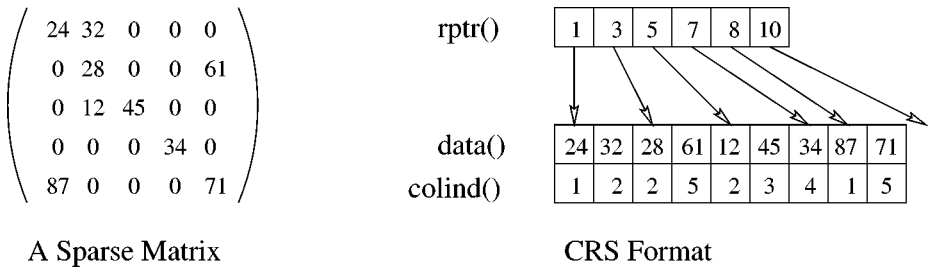


Fig. 1. An example of Compressed Row Storage format.

Table 1
Benchmark codes

	Benchmark	Description	Origin	# lines	Serial Exec (seconds)
1	CHOLESKY	Sparse cholesky factorization	HPF-2	1284	323.50
2	EULER	Conservation equations solved in Eulerian frame	HPF-2	1990	972.14
3	GCCG	Computational fluid dynamics	Univ of Vienna	407	374.27
4	LANCZOS	Eigenvalues of symmetric matrices	Univ of Malaga	269	389.25
5	SpLU	Sparse LU factorization	HPF-2	363	1958
6	SparAdd	Addition of two sparse matrices	[18]	67	2.40
7	ProdMV	Product of a sparse matrix by a column vector	[18]	28	1.28
8	ProdVM	Product of a row vector by a sparse matrix	[18]	31	1.07
9	SparMul	Product of two sparse matrices	[18]	64	3.32
10	SolSys	Solution of system $U^T DUx = b$	[18]	43	4.51
11	ProdUB	Product of matrices U^{-T} and B	[18]	49	3.72

loops with indirectly accessed arrays by examining the loop body[2]. In the benchmark, we also found another class of loops that do not contain indirectly accessed arrays but need new techniques to be parallelized.

Table 2 summarizes the seven most important patterns we found in our benchmarks. A '$\surd$' means this pattern appears in the program. The patterns in columns three through six involve uses of indirectly accessed arrays, while the other three do not. In this paper, we call the array that appears in the subscript of other arrays a *subscript array* and the indirectly accessed array a *host array*.

3.2. Loop patterns containing indirectly accessed arrays

3.2.1. Right-hand side

In this pattern, illustrated in Fig. 2, the indirectly accessed arrays appear only on the right-hand side of assignment statements and are read only throughout the enclosing loops. Thus, this loop pattern does not cause any difficulty for current techniques in the automatic detection of parallelism. The pattern is very popular and appears in GCCG, LANCZOS, ProdMV and SolSys.

```
DO nc = nintci, nintcf
  direc2(nc) =  bp(nc)*direc1(nc)
              - bs(nc)*direc1(lcc(nc,1))
              - bw(nc)*direc1(lcc(nc,4))
              - bl(nc)*direc1(lcc(nc,5))
END DO
```

Fig. 2. An example of *Right-hand Side* pattern.

```
REAL  a(m)
DO i = 1, n
   a(x(i)) = a(x(i)) op expression
END DO
```

Fig. 3. An example of the *Histogram Reduction* pattern.

[2]Sparse/irregular programs written in Fortran 90/95 may have different loop patterns. However, we see no difficulties in extending the techniques in this paper to Fortran 90/95 codes if similar patterns present.

Table 2
Loop patterns

	Benchmark	Indirectly Accessed Array				Others		
		Right-Hand Side Only	Histogram Reduction	Offset and Length	Sparse and Private	Consecutive Written Array	Premature Exit Loop	Array Privatization II
1	CHOLESKY			√			√	√
2	EULER		√					
3	GCCG	√						
4	LANCZOS	√						
5	SpLU			√	√	√		√
6	SparAdd			√	√			
7	ProdMV	√						
8	ProdVM		√					
9	SparMul			√	√			
10	SolSys	√	√					
11	ProdUB		√					

3.2.2. Histogram reduction

An example of the histogram reduction pattern is shown in Fig. 3. There, `op` is a reduction operation, and `expression` does not contain any references to array `a()`. Array `a()` is accessed via the index array `x()`. Because two elements of `x()` could have the same value, loop-carried dependences may exist, but this fact does not preclude parallelization in this case. Histogram reductions are also called irregular reductions [8]. As was the case for the previous pattern, histogram reduction can be identified as parallel by traditional techniques.

Histogram reductions are pervasive in sparse/irregular codes. There are several reasons for this. First, some typical sparse matrix computations, such as the vector/matrix multiplication in the following example, have the form of histogram reduction:

```
DO i = 1, n
   DO k = rowbegin(i), rowend(i)
      c(ja(k)) = c(ja(k)) + b(i) * an(i)
   END DO
END DO
```

Second, a large collection of irregular problems, which can be categorized as general molecular dynamics, accumulate values using histogram reduction. Four programs in the HPF-2 motivation applications (i.e., MolDyn, NBFC, Binz and DSMC) have histogram reductions at their computation core [8]. The third reason is that the computation of structure arrays, like the index arrays used to access sparse matrices and the interaction list in molecular dynamics, often contains histogram reductions. An obvious example is the calculation of the number of nonzero elements for each row in a sparse matrix stored in column-wise form.

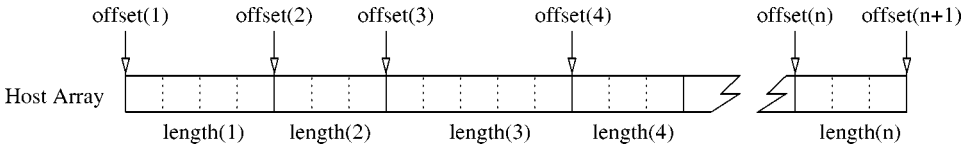
In Section 4.3, we will discuss in detail the techniques that are useful to parallelize a histogram reduction loop.

3.2.3. Offset and length

In this pattern, subscript arrays are used to store offset pointers. The host array is accessed contiguously in each segment. Fig. 4 illustrates the basic pattern. Array `offset()` points to the starting position of each segment, whose length is given in array `length()`. Fig. 4(a) and (b) are two common forms of loop patterns using the offset and length array. They traverse the host array segment by segment.

A typical example of this pattern appears in sparse matrix codes using the CCS/CRS format and traversing the matrix by row or column. The CCS/CRS format is adopted by the Harwell–Boeing Matrix Collection and has become very popular. We also found this pattern in the Perfect Benchmark Codes DYFESM and TRFD. There are several variants of CCS/CRS that reflect the structures of some specific sparse matrices. However, the basic loop patterns found in the codes accessing arrays represented in these variants are similar to those in Fig. 4.

It should be noted that `length(i)` and `offset(i)` in Fig. 4 are loop invariant in the innermost loop. And, we found in our benchmark codes that the array indices in the loop body always are very simple when the loop bounds contain auxiliary array elements. Thus, in this case, even some primitive dependence tests, like the ZIV and SIV tests [1], suffice to




```
    DO 100 i = 1, n
       DO 200 j = 1, length(i)
s3:           data(offset(i)+j-1) = ...
       END DO
    END DO
                  (a)


    DO 100 i = 1, n
       DO 200 j = offset(i), offset(i+1)-1
              data(j) = ...
       END DO
    END DO
                  (b)


s1: offset(1) = 1
    DO i = 2, n+1
s2:    offset(i) = offset(i-1) + length(i-1)
    END DO
                  (c)
```

Fig. 4. Examples of *Offset and Length* pattern.

detect the parallelism of the inner loop. Here, the subscript arrays only cause a problem when detecting the parallelism in the outer loops. The solution will be addressed in Sections 4.2 and 4.4.

The offset and length pattern can be found in serveral important sparse matrix operations, such as addition, multiplication, and rearrangement. In languages, where the array is the basic data type and the loop is the main iterative construct, such as Fortran 77, storing and accessing related data elements contiguously in an array is a natural way to program. Furthermore, this pattern has good spatial cache locality. Thus, the presence of this access pattern in sparse/irregular codes is not surprising.

3.2.4. Sparse and private

In this loop pattern, array elements are accessed in a *totally irregular* manner. However, in this pattern, the array also is used as a working storage which can be privatized.

A typical example of this pattern appears in the scatter and gather operation, as shown in the matrix addition code in Fig. 5, where the working place array `x()` is used to hold the intermediate result.

3.2.5. Summary

Of the four loop patterns, *right-hand side* is trivial to handle; *histogram reduction* can be recognized automatically [19], though its efficient parallelization is not as easy as it first appears; *offset-length* and *sparse and private* can be managed by array property analysis as presented in Section 4.2, or the efficient runtime method as presented in Section 4.4. Thus, we believe that a parallelizing compiler enhanced with the techniques discussed in this proposal should be able to automatically analyze most loops with indirectly accessed arrays.

3.3. Other loop patterns

The loop patterns described in this subsection do not invlove indirectly accessed arrays. However, they appear in our benchmark codes and require some new transformation techniques.

3.3.1. Consecutively written array

In this pattern, array elements are written one after another in consecutive locations given by induction variables. However, there is no closed form ex-

```
DO i = 1, n
   /* scatter */
   DO ip = ic(i), ic(i+1) - 1
      x(jc(ip)) = 0
   END DO
   DO ip = ia(i), ia(i+1) - 1
      x(ja(ip)) = an(ip)
   END DO
   DO ip = ib(i), ib(i+1) - 1
      x(jb(ip)) = x(jb(ip)) + bn(ip)
   END DO
   /* gather */
   DO ip = ic(i), ic(i+1) - 1
      c(ip) = x(jc(ip))
   END DO
END DO
```

$$\forall 1 \leqslant i \leqslant n, \{jc(j) \mid ic(i) \leqslant j \leqslant ic(i+1) - 1\}$$
$$= \{ja(j) \mid ia(i) \leqslant j \leqslant ia(i+1) - 1\}$$
$$\cup \{jb(j) \mid ib(i) \leqslant j \leqslant ib(i+1) - 1\}$$

Fig. 5. Sparse matrix addition.

```
   DO i = 1, n
      WHILE (...) DO
        IF (...) THEN
S1:       a(j) = ...
S2:       j = j+1
        END IF
      END WHILE
   END DO

( a() is write only in this pat-
tern )
```

Fig. 6. An example of the *Consecutively Written* pattern.

pression of the array index because the value of the index is changed conditionally. This pattern is illustrated in Fig. 6.

Current data dependence tests fail to disprove the dependence in this pattern because the index value in each iteration is determined at run-time.

3.3.2. Premature exit loops with reduction operations

A premature exit loop is a loop containing a **goto** or **break** statement that directs the program flow out of the loop before all iterations have been executed. Speculative execution, which is a possible parallelization method for this type of loop, may execute beyond the iteration where the loop exit occurs and cause side-effects. These overshot iterations should be recovered in order to produce the correct result. The need for roll-back makes the parallelization complicated and sometimes impossible.

```
a = initvalue
DO i = lower, upper
   IF (cond(i)) BREAK
   a = reduct(a,i)
END DO
```

Fig. 7. An example of the *Premature Exit Loop with Reduction Operations* pattern.

```
DO i = 1, n
   DO j = 1, m
      t(j) = 0
   END DO
   .. = t(k)
END DO
```

$1 \leqslant k \leqslant m$

Fig. 8. Array privatization of Type I.

However, if the only operation within the premature exit loop is a reduction, as illustrated in Fig. 7, then a simple speculative parallelization transformation is possible. The basic idea is to block schedule the loop, let each processor get its own partial result, and throw away the unneeded part in the cross-processor reduction phase.

3.3.3. Array privatization of type II

Array privatization is one of the most important parallelization techniques. It eliminates data dependences between different occurrences of a temporary variable or an array element across different iterations [9,22, 17]. Blume et al. [6] measured the effectiveness of array privatization on the Perfect Benchmark suite.

Fig. 8 shows a typical access pattern for a privatizable array. Generally, a variable or an array can be privatized if it is defined before it is used in each loop iteration [9,22,17]. This traditionally has been the only kind of data item privatized by existing compliers. However, in our benchmark codes, we found another kind of privatizable array, as shown in Fig. 9, which we will refer to as type II. We call the more traditional privatizable array type I.

In this example, array `t()` is used as a temporary array. In each iteration of LOOP_2, the elements of `t()`, which are modified in `S2`, are always reset to `const` before leaving the iteration. Because the value of `t()` is always the same upon entering each iteration, a private copy for each iteration with the same initial value `const` can be used. Thus, `t()` is a privatizable array.

```
          DO i = 1, m
             t(i) = const
          END DO

LOOP_2:   DO i = 1, n
             k = 0
             ....
             WHILE (...) DO
    S1:          ... = t(...) op express
                 ....
                 k = k+1
    S2:          t(k) = ...
                 ....
             END WHILE
             ....
             DO j = 1, k
    S3:            t(j) = const
             END DO
          END DO
```

Fig. 9. Array privatization of Type II.

It is not surprising that people write code in this way, rather than initializing `t()` to '`const`' at the beginning of each iteration. The reason is twofold. First, the size of `t()` may be very large while the number of elements modified in each iteration is relatively small, thereby creating a high cost to initialize all elements of `t()` in each iteration. Second, when the range of `t()` that will be referenced in each iteration is unknown until the array is actually being written, such as the value of `k` in the example, it is difficult to write initialization statements prior to the references. In this case 'resetting' rather than 'initializing' is more straightforward and allows unnecessary computations to be avoided. Like the pattern of *consecutively written array*, *privatizable array of type II* reflects the dynamic nature of sparse/irregular programs. We found this access pattern in CHOLESKY and SpLU.

4. New techniques

This section describes four techniques that we found important to parallelize sparse/irregular programs. We do not cover the techniques for the patterns of *right-hand side* and of *premature exit loop with reduction operation*. The former is trivial to handle and the latter can be parallelized by using the associative transformation [20].

4.1. Parallelizing loops containing consecutively written arrays

For two reasons, we want to recognize consecutively written arrays.

- A parallelizing compiler should be able to parallelize loops containing this pattern if there are no dependences caused by other statements in the loop.
- The knowledge that all the array elements within a contiguous section are written is important, in some cases, to facilitate array dataflow analysis. To illustrate this, consider the following loop.

```
DO i = 1,  n
   k = 1
   WHILE (..) DO
      a(k) = ..
      k = k+1
   END WHILE
   DO j = 1, k-1
      .. = a(j)
   END DO
END DO
```

Array `a()` is consecutively written from position 1 to k in the `WHILE` loop. Thus, all the elements in array section `a[1:k-1]` are defined before use (in loop j) in each iteration of loop i. Hence, array `a()` can be privatized in loop i.

To recognize consecutively written arrays, we start by selecting arrays with the following characteristics:

(1) the array is write-only in the loop,
(2) the subscripts of all the array occurrences are the same, which is an induction variable, and
(3) all operations on the induction variable are increments of 1, or all are decrements of 1.

Then, we work on the control flow graph of the loop. We call all assignment statements of the candidate array (like S1 in Fig. 6) *black nodes*, and all assignments to the induction variable (like S2) *grey nodes*. The remaining nodes are called *white nodes*. The candidate array is a consecutively written array if and only if on each path between two grey nodes there is at least one black node. We can do the checking in the following way. For each grey node, we do a depth-first search from it. The boundary of this search is black nodes. If any other grey node is found before the search finishes, then the array is not a consecutively written array and the search aborts. If the search finishes for all grey nodes, then a consecutively written array has been

found. Because the working control flow graph is usually rather simple, the recognition phase is fast and can be integrated into the pass for induction variables processing in a parallelizing compiler.

4.1.1. Transformation – array splitting and merging

Because the positions where the elements of a consecutively written array are written in one iteration depend on the previous iteration, there is a loop-carried true dependence. We use a technique called *array splitting and merging* to parallelize this loop.

Array splitting and merging has three phases. First, a private copy of the consecutively written array is allocated on each processor. Then, all processors work on their private copies from position 1 in parallel. After the computation, each processor knows the length of its private copy of the array; hence, the starting position in the original array for each processor can be easily calculated. Finally, the private copies are copied back (merged) to the original array. An example of a program using two processors is shown below, and is illustrated in Fig. 10.

```
Before:
   k = k0
   DO i = 1, n
      WHILE (..) DO
            a(k) = ..
            k = k+1
      END WHILE
   END DO

After:
   pk(1) = 1
   pk(2) = 1
   PARALLEL DO i = 1, n
      WHILE (..) DO
         pa(pk(thread_id), thread_id) = ..
         pk(thread_id) = pk(thread_id) + 1
      END WHILE
   END DO
   PARALLEL SECTION
   SECTION
      DO i = 1, pk(1)-1
         a(k0+i-1) = pa(i,1)
      END DO
   SECTION
      DO i = 1, pk(2)-1
         a(k0+pk(1)+i-2) = pa(i,2)
      END DO
   END PARALLEL SECTION
   k = k0+pk(1)+pk(2)-2
```

4.2. Property analysis of subscript arrays

Symbolic analysis is a very important compiler technique and for this reason, has been the subject of several studies [10,3]. However, most symbolic analysis methods were designed for scalars. As subscript arrays are used extensively in sparse/irregular programs, extending symbolic analysis to subscript arrays is important. It enables the compile-time detection of some of the patterns we found, such as the *offset and length*, and thus extends many parallelization techniques, such as the data dependence test and array privatization, to handle program sections with subscript arrays.

A subscript array is a special kind of array. Our strategy to handle subscript arrays is based on several observations.

(1) Subscript arrays can have only integer values, and the subscripts of subscript arrays are usually simple. This simplifies the analysis.
(2) Subscript arrays tend to be glacial (i.e., it is modified much less frequently than it is referenced).
(3) Subscript array elements within a contiguous array section often share some common properties. Treating an array section rather than an array element as a basic unit makes the analysis more efficient.
(4) And, finally, subscript arrays usually have only one dimension. And, when subscript arrays have multiple dimensions, usually the ranges of the other dimensions are known at compile-time, and the indices for the higher dimensions are constant. As a consequence, we only need to focus on one dimensional subscript arrays.

4.2.1. Property analysis of subscript arrays

The basic strategy for subscript array symbolic analysis is first to derive the properties of the subscript array from where the subscript array is defined and then use the information to analyze the loops that use the subscript array. We call this process *property analysis of subscript arrays*, or simply *subscript array analysis*.

We, as well as several other researchers [5], have found the following four properties of subscript arrays useful:

Monotonicity

Array segment a(m:n) is monotonically increasing, if $a(i) \leqslant a(j)$, $i < j$.

Array segment a(m:n) is monotonically decreasing, if $a(i) \geqslant a(j)$, $i < j$.

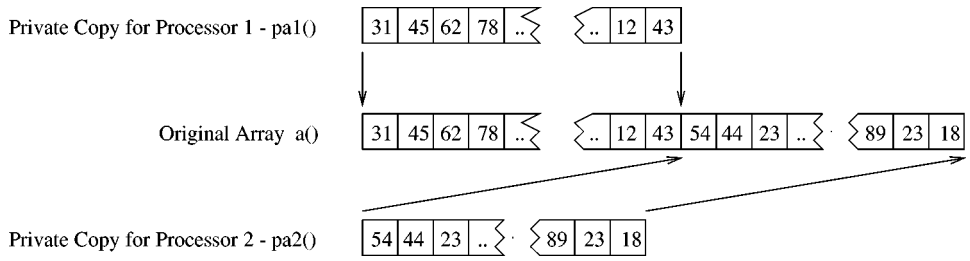


Fig. 10. An example of array splitting and merging.

For example, in Fig. 4(b), if array `offset()` is monotonically increasing, then the outer loop can be parallelized.

Injectivity

Array segment a(m:n) is injective, if a(i) $\neq$ a(j), when i $\neq$ j.

To illustrate the use of this property, consider the array `x()` in Fig. 3. If it is injective, then the loop is parallel and can be simply translated to a DOALL loop. Otherwise, the loop can be transformed into a parallel histogram reduction which, as will be discussed in Section 4.3, could introduce significant overhead in some cases.

Max/Min Value

The Max/Min value of array segment a(m:n) is the maximum/minimum of array element in segment a(m:n).

The use of this property is illustrated in the following loop, where if the Max value of a(1:l) is m and Min value is 1, then array v() can be privatized.

```
DO i = 1, n
   DO j = 1, m
      v(j) = ..
   END DO
   DO j = 1, l
      .. = v(a(j))
   END DO
END DO
```

Regular Distance

We say that an array segment a(m:n) has a regular distance if a(i+1)-a(i) can be represented as a simple expression of i for all i in [m:n]. The expression may contain array elements other than a() indexed by i.

For example, in the loop shown in Fig. 4(a), if array `offset(i)` has a distance of `length(i)`, then the outer loop can be parallelized. In fact, in many programs that follow the pattern of Fig. 4(a), `offset()` is defined following the pattern in Fig. 4(c).

4.2.2. A simple framework for calculating array reaching definitions

For the subscript array analysis to be correct, we must make sure that the definitions can reach the use. For the example in Fig. 4, we want to know whether the values of all the elements of `offset()` read in statement `s3` are provided by statements `s1` and `s2`. In order to get this information, we need *array reaching definition* analysis.

We have developed a simple framework to calculate the array reaching definitions at the statement level [16].

The array reaching definition problem can be described as, "Given an array read occurrence, find all the assignment statements that may provide the value accessed by the read occurrence". Since, in real programs, array regions defined by a single statement tend to have the same property, the statement-wise array reaching definition information is precise enough for our purpose.

Any framework that computes the array reaching defintion has to solve, either explicitly or implicitly, two problems. The first one is Access Order. Suppose two statements S_1 and S_2 write some array elements that are read by another statement S_3. It is necessary to know which statement writes the array elements last. If some writes of S_2 come after some writes of S_1, then those definitions generated by S_1 may be killed by S_2. The second problem is Coverage. Suppose that, in the program region being analyzed, there are several statements writing the array elements read by another statement S_r. If they do not write all the array elements read by S_r, then a definition may come from a statement outside the program region.

In our method, we treat as a unit all the array elements that can be accessed by each array occurrence. For example, in the codes in Fig. 11(a), there are four array occurrences: $O1$, $O2$, $O3$ and $O4$. The first three are read occurrences and the last one is a write occurrence. They can access $\{a(10:90)\}$, $\{a(1:50)\}$, $\{a(51:100)\}$ and $\{a(1,100)\}$, respectively.

```
   DO l=10,90
s0:   a(l) = ...  - O1
   END DO
   DO i=1, 50
s1:   a(i) = ...  - O2
   END DO
   DO j=51, 100
s2:   a(j) = ...  - O3
   END DO
   DO k=1, 100
s3:   ... = a(k)  - O4
   END DO
        (a)
```

```
      DO i=1, 100
s1:  a(i)   = ...    - O1
s2:  a(i-1) = ...    - O2
s3:     ... = a(i-2) - O3
      END DO
              (b)
```

Fig. 11. Code samples.

Input: A read occurrence R, and m write occurrences W_1, W_2, ..., W_m that are executed before R. The m write occurrences are sorted in the order that W_i is executed before W_j if $i < j$.

Output: A subset of the m write occurrences that provide reaching definition for the read occurrence R.

Procedure:

Let $\mathcal{R}(O)$ represent the array region that is accessed by occurrence O.
$S = \phi$, $T = \mathcal{R}(R)$;
for i=m downto 1 do
  if $T \cap \mathcal{R}(W_i) \neq \phi$ then
    $S = S \cup \{W_i\}$;
    $T = T - \mathcal{R}(W_i)$;
  end if
  if $T == \phi$ then return (T, S) ;
end for
return (T, S)

Fig. 12. A simplified high level algorithm of our method.

The coverage problem can be solved by using the set subtract operation. For instance, in the previous example, because

$$(\{[1:100]\} - \{[51:100]\}) - \{[1:50]\} = \phi,$$

$s1$ and $s2$ provide all the definitions used in $s3$.

The set operation has to be applied by taking into account the access order. The previous example illustrates a *straight line coverage* case. Another important case is the *cross iteration coverage*. In the code in Fig. 11(b), an array element written by $s1$ in one iteration will be killed by $s2$ in the following iteration. Since the element is used by $s3$ two iterations later, the definitions in $s1$ do not reach $s3$. In other words, the coverage is performed across the iterations. The access order in this code also is clear. All the elements that can be accessed by both $s1$ and $s2$ first are accessed by $s1$ and then by $s2$. Thus, $a[1:98]$ is defined first by $s1$ and then by $s2$.

The basic idea of our method is to (1) identify all the array elements that an occurrence can access, (2) order the occurrences according to the access pattern, and (3) use set operations to derive the reaching definitions. A simplified high-level algorithm of our method is shown in Fig. 12. We take advantage of the fact that, in real programs, most array accesses are simple and have single or partial single dependence distance. We use this distance to etablish an *execute before* order between these occurences. Based on this order, the array

```
PARALLEL DO i = 1, n
   lock(lck(x(i))
       a(x(i)) = a(x(i)) op express
   unlock(lck(x(i))
END DO
```

Fig. 13. Critical section.

reaching definition can be computed by using set operations in a simple way.

4.3. Histogram reduction

In this section, we describe three existing techniques used to parallelize loops with histogram reduction and propose a new parallelization method.

Despite the many different names used in the literature, the existing parallelization of histogram reduction falls into one of three classes: critical section, data affiliated loops [12] and array privatization/expansion.

4.3.1. Critical section

In this method, illustrated in Fig. 13, the access and update of each array element of `a()` is enclosed by a lock/unlock pair. The operations on different array elements can be executed in parallel. However, this is a general method for updating shared variables in any parallel loop; it does not exploit the parallelism within reductions.

In addition, the overhead due to the **lock** and **unlock** operations could be high if no fast synchronization mechanism is available. Experiments on the SGI PowerChallenge show that the execution time can be 9 times longer than the sequential version if the `test_and_set()` function is used to implement critical section in Fortran 77.

Hence, without fast hardware, critical section is not a good alternative to sequential reduction.

4.3.2. Data affiliated loops

The basic idea of this method is to partition data instead of loop iterations. Each processor traverses all the iterations and checks whether the data referenced in the current iteration belongs to it. If it does, the processor executes the operation; otherwise, it skips the operation. Data affinity scheduling on the SGI Origin 2000 shares the same concept [12]. This approach also is related to the *owner computes rule* used in parallel programming for distributed memory machines.

Suppose we have four processors and $n = 4 * k$. Then, Fig. 14 shows the data affiliated parallel version of the reduction loop.

The advantage of this approach is its potential for good cache locality when block distribution is used. The disadvantage is that this method is difficult to apply when there are several indirectly accessed arrays in the loop. Also, the overhead could be high if each iteration is relatively short.

4.3.3. Array privatization/expansion

In this method, each processor allocates a private copy of the whole reduction array. The parallelized loop has three phases. All the private copies are initialized to the reduction identity in the first phase. In the second phase, each processor executes the reduction operation in its own private copy in parallel. There is no inter-processor communication in this phase. The last phase does cross-processor reduction.

Fig. 15 illustrates the three phases. In this code, a two-dimensional array `pa()` is allocated to accommodate the private copies of the `a()`.

The overhead caused by initialization and cross-processor reduction part may become an obstacle to achieving high speedup. First, these two parts are not scalable with the number of processors used. And, to make matters worse, because of its poor cache locality, the execution time of cross-processor reduction part usually increases with the number of processors on real machines. Second, it requires each private copy to cover the entire range of the global reduction array. When reduction touches only a small portion of the

```
PARALLEL DO j = 1,  4
  DO i = 1, n
     IF ( 4*(k-1)+1 <= x(i) <= 4*(k) ) THEN
        a(x(i)) = a(x(i)) op express
     END IF
  END DO
END DO
```

Fig. 14. Example of data affiliated loop.

```
REAL a(1:m)
REAL pa(1:m,1:num_of_threads)

PARALLEL DO i = 1, num_of_threads
DO j = 1, n
   pa(j,i) = reduction_identity
END DO
END DO

PARALLEL DO i = 1, n
   pa(i, thread_id) = pa(i, thread_id) op express
END DO

PARALLEL DO i = 1, n
   DO j = 1, num_of_threads
        a(i) = a(i) op pa(i,j)
   END DO
END DO
```

Fig. 15. Expansion.

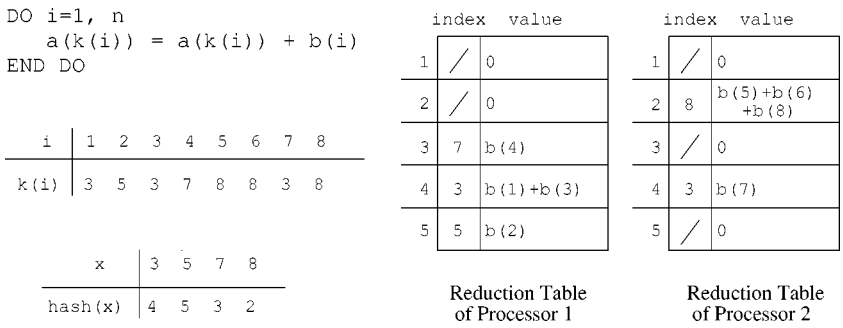


Fig. 16. An example of using reduction table.

global array, copying the whole range arrays back and forth seems unwise. In addition, having multiple copies of large arrays increases the memory pressure. Third, in some programs, the range declared for an array is larger than the range actually used. As the compiler usually cannot detect the real range statically, it has to resort to the declared size of the array, thus introducing more overhead.

4.3.4. Reduction table

The biggest disadvantage of the privatization/expansion approach is having to deal with private copies of the whole array. The *reduction table* method also uses private storage, but the required size is small.

In this method, the private copy is used as a table. The number of entries is fixed. Each entry in the table has two fields: index and value. The value of an entry is accessed by using a fast hash function of the index. Thus, this can be thought of as a hash table. We call this table the *reduction table* because the value of an entry is updated by some reduction operation.

Fig. 16 illustrates how reduction table is used. Processor 1 executes iterations 1 through 4, and Processor 2 executes 5 through 8.

In the reduction table approach, all entries in the reduction tables are initialized to the reduction identity before entering the parallel loop. Then, each processor works on its own reduction table in parallel. Each array element is mapped to a table entry by using a hash function on the array subscript. When a reduction is to be performed, the table is looked up to find an entry that contains the same array index or, if no such entry exists, an empty entry. If the entry is available, then reduction operation is performed on the old value in the entry with the new value. If the entry is not available, which means the table is full, the operation is performed directly on the shared array within a critical section. After the parallel section, all entries whose

values do not equal reduction identity are flushed out to the global array in critical sections.

This reduction table method is a hybrid of the critical section method and the array expansion method. Like the critical section method, it ensures the atomic operation on global shared variables; and, like the expansion method, it takes advantage of reduction operation. The reduction table method does not have to keep private copies of the whole reduction array. The number of table entries can be much smaller than the number of elements in the reduction array. Only those elements that are updated in the loop are kept. This method trades hash table calculations with memory operations. It may be particularly beneficial when used with a simple, fast hash function.

4.4. Loop-pattern-aware run-time dependence test

When static analysis is too complex or the subscript arrays are functions of the input data, a run-time method becomes necessary for parallelization. Some run- time tests, as proposed in [21], use shadow arrays that have the same size as the data arrays. They check and mark every read/write operation on the array elements. This kind of run-time test would introduce high overhead on modern machines where memory access is significantly slower than computation. One solution is to use hardware support [23]. However, in some cases, efficient software methods are available if the information of loop patterns can be used.

Our run-time test is based on the loop patterns we discussed above and is applied to loops with indirectly accessed arrays. Of the four patterns, *right-hand side* and *histogram reduction* always can be handled statically. We, therefore, focus on the last two cases. Our run-time test can be thought of as run-time pattern matching. It relies on the compiler to detect the static syntactic pattern and uses the run-time test to verify variable constraints.

In the case of the *offset and length* pattern, a run-time test would be used to check whether two array segments overlap. Thus, only the first and last positions need to be stored, and the size of the shadow array is greatly reduced. In the case of the *sparse and private* pattern, the marking and analysis phases can be simplified because we only check the validation of privatization for these *totally irregular* arrays. For type I privatization, we mark the shadow array when a write occurs. The test immediately fails once we find a shadow that is not marked when a read occurs. For type II privatization, the shadow is marked when an array element is read or written and is cleared when the element is set to a constant. If there are still some marked elements in the shadow at the end of the inspector phase, then the test fails. In the experiment on CHOLESKY, we found that our simplified run-time test has a run-time overhead of 20% of the parallel execution time on eight processors as compared to 120% overhead of the general run-time test.

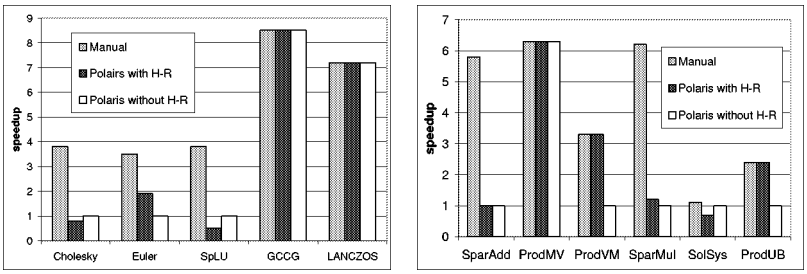
Because the new run-time test is based on the loop pattern, we call it the loop- pattern-aware run-time test. The trade-off is its conservativeness. While the general run-time test [21] can exploit almost all classes of parallelism (even partially parallelizable loop), ours may report dependence where there actually is none. However, for the codes in our benchmark suite, the loop-pattern-aware run-time test suffices.

5. Experimental results

The experimental results are discussed in this section. We compare the speedups obtained by manually applying the new techniques with those by using the current version of Polaris, with and without histogram reduction. SGI's parallelizing compiler, PFA, produced results very similar to Polaris without histogram reduction. We, therefore, do not include its data here. The parallelization technique for histogram reduction loops currently implemented in Polaris is the array expansion method. The programs were executed in real-time mode on an SGI PowerChallenge with eight 195 MHz R10000 processors.

Fig. 17(a) shows the speedups of benchmark codes for the first five programs. Parallelizing Gccg and Lanczos requires no new techniques, and our current version of Polaris does quite well on these two codes. For Cholesky, Euler and SpLU, manual transformation led to good speedups. In Cholesky and SpLU, there are several innermost loops that perform very simple histogram reductions. The overhead of array expansion transformation hinders the performance. However, this transformation works well on Euler, which has large granularity in its loop bodies.

Speedups of the sparse matrix algebra kernels in our benchmark collection are shown in Fig. 17(b). ProvVM and ProdUB achieve speedups because of the parallel histogram reduction transformation. ProvMV is simple and Polaris handles it well. SparAdd and SparMul require the loop pattern aware run-time test and array privatization of type II. These new techniques produce speedups of around 6. SolSys has a



(a) (b)

Fig. 17. Speedups of benchmark codes.

Table 3
Techniques applied to each code

	Code	Techniques
1	Cholesky	Associative Transformation, Loop-pattern-aware Run-time Test
2	Euler	Histogram Reduction(Array Expansion)
3	Gccg	trivial
4	Lanczos	trivial
5	SpLU	Array Splitting and Merging, Loop-pattern-aware Run-time Test
6	SparAdd	Loop-pattern-aware Run-time Test
7	ProdMV	trivial
8	ProdVM	Histogram Reduction(Array Expansion)
9	SparMul	Loop-pattern-aware Run-time Test
10	SolSys	Histogram Reduction(Reduction Table)
11	ProdUB	Histogram Reduction(Array Expansion)

DOALL loop, which Polaris recognized as a histogram reduction loop, thereby causing the slow down.

The techniques manually applied to each code are summarized in Table 3.

6. Related work

To our knowledge, there are very few papers in the literature that directly address the problem of automatically detecting parallelism in sequential sparse/irregular programs.

The closest work we are aware of is that of Keßler's [13]. Keßler had previously developed a pattern-matching algorithm to recognize a set of typical program patterns in sequential numeric codes operating on vector and dense matrices. Currently, they are investigating to what degree their *program comprehension* technique can be generalized to sparse matrix codes. They found some patterns that could be identified in SPARSE-BLAS. Their approach is based on pattern matching and, therefore, depends on the storage scheme and algorithm used by the program. In contrast, we study the loop pattern to find the common ingredients of different algorithms and storage formats. Our approach is more general and even applicable to some 'homemade' codes that do not follow standard algorithms.

Another approach is followed by Bik and Wijshoff [2]. The solution is to generate sparse matrix programs by what they call the sparse compiler. Programs are written as if the matrices are dense, and sparse matrices are specified explicitly. Then the sparse compiler will take the responsibility of selecting the appropriate storage format and generating the sparse program. Because the compiler has the original dense code, in which the data dependencies are easier to detect than in the 'messy' sparse version, it can carefully preserve the independence to produce a final parallel sparse code. This approach opens many opportunities, but has its own limitations and difficulties; and, only a prototype compiler has been written. The major problem is how to generate efficient code. Selecting an appropriate storage format is not easy. Interactive responses from users are required during compilation. However, even with human help, some dense codes cannot be translated into efficient sparse code without breaking the semantic correctness.

Kotlyar, Pingali and Stodghill proposed another interesting solution based on their 'data-centric-compilation' concept [14]. Instead of traversing the iteration space given by the loops in the program, they traverse the space of the working data. And, for each data element, a corresponding operation on that specific element is performed. The transformed code has very good cache spatial locality and is independent of the matrix storage format employed by the program as long as the programmer can provide an enumerate function and a select function. This method requires the enclosing loop to be a parallel DOANY loop; otherwise, the checking of independence will be very expensive. This approach may have good performance on kernel codes, but its applicability on large problems is still unclear. Another disadvantage of both this approach and the sparse compiler is that they cannot handle existing sparse code. Since much effort already has been put into the development of sparse programs, and some complete general-purpose packages are available now, our approach can directly work on those programs to provide better performance.

7. Conclusion

This paper shows that, contrary to common belief, parallelizing compilers can be used to automatically detect the parallelism in sparse/irregular programs. An empirical study of a collection of sparse/irregular codes reported in this paper shows that several important common loop patterns exist in sparse/irregular codes. Based on these patterns automatic parallelism detection can be applied. Some loops can be parallelized by using existing compiler techniques, while some others require new methods. The new techniques are identified and discussed in this paper. We have shown that good speedups can be achieved by applying these techniques to our collection of sparse/irregular programs.

References

[1] J.R. Allen, Dependence analysis for subscripted variables and its application to program transformations, PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[2] A.J.C. Bik and H.A.G. Wijshoff, Automatic data structure selection and transformation for sparse matrix computations, *IEEE Trans. Parallel Distributed Systems* **7**(2) (Febr. 1996), 109–126.

[3] W. Blume and R. Eigenmann, An overview of symbolic analysis techniques needed for the effective parallelization of the perfect benchmarks, in: *Proceedings of the 23rd International Conference on Parallel Processing. Vol.* 2: *Software*, K.C. Tai, ed., CRC Press, Boca Raton, FL, USA, Aug. 1994, pp. 233–238.

[4] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu and S. Weatherford, Polaris: Improving the effectiveness of parallelizing compilers, in: *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua, eds., Lecture Notes Comput. Sci., Ithaca, New York, Aug. 8–10, 1994, Springer, pp. 141–154.

[5] W.J. Blume, Symbolic analysis techniques for effective automatic parallelization, PhD thesis, University of Illinois at Urbana-Champaign, June 1995.

[6] R. Eigenmann and W. Blume, An effectiveness study of parallelizing compiler techniques, in: *Proceedings of the 1991 International Conference on Parallel Processing, Vol.* 2, *Software*, CRC Press, Boca Raton, FL, August 1991, pp. II-17–II-25.

[7] A. Ferreira and J.D.P. Rolim, *Parallel Algorithms for Irregular Problems: State of the Art*, Kluwer Academic Publishers, 1995.

[8] I. Foster, R. Schreiber and P. Havlak, Hpf-2 scope of activities and motivating applications, Technical Report CRPC-TR94492, Rice University, Nov. 1994.

[9] J. Gu, Z. Li and G. Lee, Symbolic array dataflow analysis for array privatization and program parallelization, in: *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, S. Karin, ed., ACM Press and IEEE Computer Society Press, New York, 1995.

[10] M.R. Haghigat, *Symbolic Analysis for Parallelizing Compilers*, Kluwer Academic Press, Boston, 1995.

[11] M.T. Heath, E. Ng and B.W. Peyton, Parallel algorithms for sparse linear systems, *SIAM Review* **33**(3) (Sept. 1991), 420–460.

[12] C. Hogue, W. Ferguson, D. Galgani, G. Ackley and L.R. Sande, *MIPSpro*™ *Fortran 77 Programmer's Guide*, Silicon Graphics Inc., Document Number 007-2361-005, 1997.

[13] C.W. Kessler, Applicability of program comprehension to sparse matrix computations, in: *PROC of 3rd EUROPAR*, Lecture Notes Comput. Sci., Vol. 1300, 1997, pp. 347–359.

[14] V. Kotlyar, K. Pingali and P.V. Stodghill, Compiling parallel code for sparse matrix applications, in: *SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose*, ACM Press and IEEE Computer Society Press, New York/Silver Spring, MD, 1997.

[15] Y. Lin and D. Padua, On the automatic parallelization of sparse and irregular Fortran programs, in: *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers* (*LCR98*), Lecture Notes Comput. Sci., Vol. 1511, Springer, 1998, pp. 41–56.

[16] Y. Lin and D. Padua, A simple framework to calculate the reaching definition of array references and its use in subscript array analysis, in: *Parallel and Distributed Processing*, Lecture Notes Comput. Sci., Vol. 1586, Springer, San Juan, Puerto Rico, 1999, pp. 1036–1045.

[17] D.E. Maydan, S.P. Amarasinghe and M.S. Lam, Array data-flow analysis and its use in array privatization, in: *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, Jan. 10–13, 1993, ACM Press, pp. 2–15.

[18] S. Pissanetzky, *Sparse Matrix Technology*, Academic Press, Orlando, FL, 1984.

[19] B. Pottenger and R. Eigenmann, Idiom recognition in the Polaris parallelizing compiler, in: *Conference Proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3–7*, 1995, ACM Press, New York, pp. 444–448.

[20] W. Pottenger, Theory, techniques, and experiments in solving recurrences in compiler programs, PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[21] L. Rauchwerger, Run-time parallelization: a framework for parallel computation, PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[22] P. Tu and D. Padua, Automatic array privatization, in: *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 12–14, 1993, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua, eds., Lecture Notes Comput. Sci., Springer, pp. 500–521.

[23] Y. Zhang, L. Rauchwerger and J. Torrellas, Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors, in: *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, Febr. 1998, pp. 162–173.