An Overview of Compiler Techniques for Interprocedural Array Section Analysis

Sung-Eun Choi
Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350 USA

Technical Report 95-11-06
November 1995

Abstract

Dependence analysis of arrays is crucial in the compilation of parallel applications, as an inaccurate summary of array usage may reduce the potential for optimizations. Standard scalar techniques are inadequate for they do not accommodate specific accesses to arrays. *Array section analysis* describes accesses to arrays at a finer granularity than the scalar techniques. More precisely, array section analysis techniques summarize a collection of accesses to a specific array in a procedure. In this paper, we present a summary of existing array section analysis techniques for interprocedural dependence analysis. We identify and compare two classes of such techniques and give suggestions for improving the techniques.

1 Motivation

Many scientific research problems such as material simulation and fluid dynamics model two- or three-dimensional objects that can be mapped onto grid-like structures. These problems lend themselves well to implementations using array data structures. The data sets for these problems can be very large, thus requiring the use of parallelizing techniques.

However, obtaining good performance on parallelized versions of scientific codes is not a simple matter. Parallelization consumes much programmer effort; scientists are now required to be experts in parallel programming as well as their given disciplines. As a result, much of the research in parallel programming focuses on ease of use and portability as well as performance. Usability intensifies the conflict between portability and performance. Good performance is usually achieved by exploiting specific hardware and hardware mechanisms, while portability can be achieved by hiding much of the underlying architecture from the user.

Two approaches to solving the parallel programming problem exist: parallelizing compilers and parallel programming languages. Parallelizing compilers extract parallelism from a sequential program. Parallel programming languages introduce new concepts and constructs to allow the programmer to express parallelism. Though parallelizing compilers and compilers for parallel languages perform somewhat distinct duties, both approaches require information about array accesses to aid in producing high performance applications.

A substantial effort has been concentrated on compiler optimizations for parallel operations on arrays. These arrays are either shared among many threads of control or distributed across many processors such that the entire array may not be directly accessible by any one thread. Given accurate analysis tools, a compiler can make wise decisions about the existence of parallelism, data placement and communication. To illustrate this point, we present two examples that benefit from optimizations using array dependence information.

Array dependence analysis can be used to perform loop optimizations. In the following example, the procedure ShiftVector shifts part of its one-dimensional array parameter, A, by k positions ($k \geq 0$.) ShiftVector calls ShiftElement with parameters A, i and k, and copies the data in A_{i+k} to A_i.

```
ShiftVector (A, lo, hi, k)
    for i ← lo to hi do
        ShiftElement(A, i, k)

ShiftElement (A, i, k)
    A_i = A_{i+k}
```

If at compile time, a compiler can determine that k is greater than $hi - lo + 1$, then each call to ShiftElement can run independently of each other and the loop can be parallelized (see Figure 1.) In fact, even if the condition does not hold true, the loop may be *unpeeled* $(hi - lo + 1) - k$ times and then the rest of the iterations of the loop may be run in parallel.

Dependence information for parallel arrays can also be utilized to exploit task parallelism and in automatic data distribution. In the following example, the procedure InitializeJacobi performs

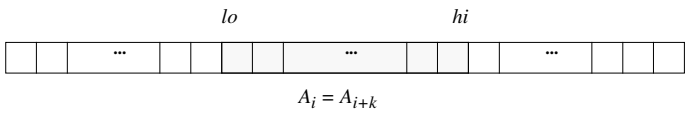

Figure 1: Example of data shifting. If A_{i+k} never falls within the shaded region, the all the iterations can be run simultaneously.

the initialization for the Jacobi iterations. InitializeJacobi initializes the boundaries of its array parameter and then initializes each element of the body.

```
InitializeJacobi (A, ub1, ub2)
    { Initialize the array A for the Jacobi Iterations }

    { Initialize the boundary conditions }
    InitializeRow(A, 0, ub2, 0.0)
    InitializeRow(A, ub1, ub2, 0.5)
    InitializeColumn(A, 0, ub1, 0.75)
    InitializeColumn(A, ub2, ub1, 0.5)

    { Initialize the body }
    for i ← 1 to ub1 − 1 do
        InitializeRow(A, i, ub1, 1.0)

InitializeRow (A, row, ub, value)
    for i ← 1 to ub − 1 do
        A_{row,i} = value

InitializeColumn (A, column, ub, value)
    for i ← 1 to ub − 1 do
        A_{i,column} = value
```

Figure 2 illustrates the initialization of the boundary conditions. InitializeJacobi calls InitializeRow to initialize the top and bottom boundaries and InitializeColumn to initialize the left and right boundaries. Each of these four *tasks* may be run in parallel since none of them interfere with the others. Moreover, the loop that executes the initialization of the body may be parallelized. The execution of the loop may also be considered a task, and all calls in InitializeJacobi may be run simultaneously. With this knowledge, the compiler may choose to distribute the data in such a way as to maximize processor utilization during these tasks.

Clearly, maintaining information about array access patterns across procedure boundaries is vital to the compiler if it is to perform optimizations such as the ones suggested above. In this paper, we investigate the issues involved in summarizing array accesses across procedure boundaries by evaluating the existing techniques. The goals of the paper are threefold: present a set of criteria for evaluating existing techniques, use the criteria to evaluate existing techniques, and suggest a method for improving the accuracy of the techniques.

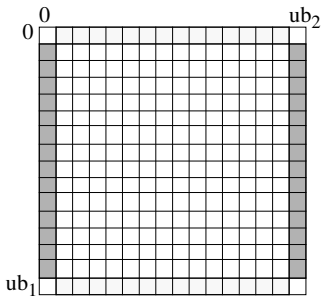

Figure 2: Initialization of boundary conditions. Light shaded areas indicate the areas modified by InitializeRow in the boundary initialization phase and the dark shaded areas are modified by InitializeColumn.

The paper is organized as follows. In Section 2, we give some background and further motivation for the paper. In Section 3, notation and assumptions used throughout the paper are presented. In Section 4, we introduce the criteria for evaluating summary techniques and define *standard* and *shape-based* schemes. In Section 5, we compare existing analysis techniques in the given framework. In Section 6, the techniques are evaluated using our criteria. Finally, in Section 7, we suggest a way to improve the existing techniques.

2 Background

An *array section* is the subset of array elements referred to by an access to the array in a loop or loop nest. The loop induction variables are expected to index into the array and help describe the section. Hence, additional information about the access may be secured from loop bounds and stride information.

*Intra*procedural dependence analysis of array sections is a well studied problem [14, 18]. Consider the following:

$$A_{f()} = \ldots$$
$$\ldots = A_{g()}$$

Determining whether a dependence exists from the assignment of $A_{f()}$ to the use of $A_{g()}$ requires knowledge about the array indexing functions f and g. If f and g are affine in the loop induction variables, a *standard dependence test*[1] can often determine whether f and g may refer to the same location by checking for (symbolic) equality of the two equations. For references nested in loops, many standard dependence tests use a system of linear equations that includes bounds and stride information. If the system of equations has one or more solutions, then a dependence may exist. Dependence is often called *intersection* because we are checking if the solution set of f intersects with that of g.

[1]We refer the reader to other sources for discussion of several standard dependence tests [3, 8, 14, 19].

Array section analysis is complicated by procedure calls. Any implementation of *inter*procedural dependence analysis requires some knowledge of the array accesses made by a call to a particular procedure. For simple scalar variables, access information can consist of as little as two bits of information: a *use* bit and a *mod* bit. The use bit is set to *true* if the variable may be used anywhere in the procedure or procedures called from within it. The mod bit is set to *true* if the variable may be modified anywhere in the procedure or procedures called from within it. Arrays are actually collections of scalar variables. Summarizing accesses to arrays implies comprehending the collection of accesses for each procedure called.

One solution to the problem of summarizing accesses in procedure calls is to use inline expansion [10, 15]. Inline expansion simply replaces the call to a procedure with the procedure body itself. After expansion, array accesses can be analyzed as in the intraprocedural case. Unfortunately, inline expansion is expensive for all but very small or infrequently called procedures. In fact, even in such cases, inline expansions can give unexpected results [6]. In particular, inlined procedures increase code size and compile time. The procedure body is inserted at each call site and that same code must be optimized at these call sites. The code explosion caused by inlining is particularly expensive when the program's call graph is very deep. Furthermore, in the presence of recursion of unknown depth or routines with unavailable source code, inline expansion is impossible.

If compile time and memory were not issues, the correct solution to summarizing array usage would be not to summarize at all. Each reference to an array A is treated as a reference to a specific scalar. For example, a reference to the i^{th} element of A would be treated as a reference to a simple scalar variable, A_i. The following example illustrates the massive time and storage requirements of an exact scheme.

for $i \leftarrow 0$ **to** 999999 **do**
    $A_i = i$

The references to A are described by a set of one million two bit values noting that every position from 0 to 999999 is modified. In addition, testing for dependence between this and any other reference to A requires at least one million tests for intersection, one for each A_i.

Alternatively, any part of an array accessed in a procedure could imply that the *entire* array is accessed. Thus the entire array is treated as a simple scalar variable. A dependence always exists between a summarized access and any other access to the same array. This *scalar approximation* may be insufficient due to the nature of procedures in scientific array codes. Procedures are often used to perform specific computation on regular slices of arrays such as rows, columns, diagonals and triangles. Approximating all accesses to an array by the entire array can limit optimization [5].

Solutions that lie between scalar approximation and the exact method may better fit the needs of scientific applications for which array optimizations are necessary. This paper focuses on four technique that reside in this solution space.

3 Notation and Assumptions

In this section, we present our notation for identifying array accesses within a loop and state some assumptions and simplifications made throughout the paper.

3.1 Notation

In general, an array access can be distinguished by three ordered sets, or *tuples*, of expressions: array index expressions, Φ, loop boundary conditions, B, and loop stride expressions, Σ. For an access to a d-dimensional array, A, nested in l loops, we define the following:

array index expressions
For the ordered set of array index expressions $\Phi(A) = (\phi_1, \phi_2 \ldots \phi_d)$, ϕ_i is the expression used to index into the i^{th} dimension of A.

loop boundary conditions
For the ordered set of loop boundary conditions $B(A) = (\beta_1, \beta_2 \ldots \beta_l)$, β_j is the ordered pair of expressions, (lb_j, ub_j), the lower and upper bounds of the loop induction variable i_j of the j^{th} loop of the loop nest.

loop stride expressions
For the ordered set of loop stride expressions $\Sigma(A) = (\sigma_1, \sigma_2 \ldots \sigma_l)$, σ_j is the expression representing the stride of the j^{th} loop of the loop nest with induction variable, i_j.

An expression may contain any number of program variables, including loop induction variables and other variables that may vary between iterations of any loop in the loop nest. We define the *array access descriptor*, Δ as the triplet, $<\Phi,B,\Sigma>$. The space required to maintain Δ is proportional to the size of each of its components. The size of Φ varies with d, the rank of the array being summarized. The size of B and Σ vary with l, the number of loops in which the array access is nested.

3.2 Assumptions

Though we make no assumption about programming language or the underlying machine architecture, we particularize the following for convenience:

- The rank of an array is d and the number of loops around an access to an array is l. The number of *static* references to a particular array in a particular procedure is m.
- Loops are numbered 1 to l from the outermost loop to the innermost loop. For the ordered set of loop induction variables, $I = (i_1, i_2 \ldots i_l)$, each element i_j is the loop induction variable of the j^{th} loop of the loop nest.
- Arrays are physically allocated in *row-major order* and are indexed from higher dimension to lower dimension. The first element of a d-dimensional array A is $A_{0,\ldots,0}$.
- All expressions that access a particular dimension of an array are assumed to be linear in the loop induction variables. A non-linear index expression for a dimension is handled by assuming dependence in that dimension.
- Only array parameters passed by reference and global array variables are relevant in interprocedural array section analysis.

We also assume that individual accesses to array elements are easily described by any of the techniques presented Section 5. As a result, only those accesses nested in loops are addressed.

4 Comparison Criteria

Prior to this section, use of the term *summary* has been somewhat imprecise. There are actually two types of array summaries: *single-site* and *multi-site*. Single-site summaries represent access information associated with a single reference. Multi-site summaries are the *union* of two or more single-site summaries. We define a *simple* multi-site summary as a union that is represented as a collection of single-site summaries. Alternatively, a *condensed* multi-site summary is a union of two or more summaries that is collapsed into fewer summaries. A *partially condensed* multi-site summary scheme collapses any number of summaries into fewer multi-site summaries. A *completely condensed* summary scheme collapses any number of summaries into a single multi-site summary; thus the size of a completely condensed summary is independent of the number of references summarized.

Recall that interprocedural array section analysis should increase the opportunity for optimizations while maintaining reasonable bounds on the cost in compilation time and memory requirements. Our comparison focuses on three aspects: summary representation, the union operation, the intersection operation.

Any summarization technique should balance accuracy with time and space complexity, where accuracy is determined by how close the summary is to the actual array accesses. Summary representation balances accuracy and space; a high precision representation generally requires more storage space. The union and intersection operations balance accuracy and time; a high precision operation generally requires more time.

Based on the intersection operation used, we distinguish two types of array summarizing schemes: *standard* and *shape-based*. Standard schemes can be thought of as extensions to *intra*procedural array section analysis. Standard schemes use Δ, or a subset of Δ, to describe array accesses and employ one of a number of standard dependence tests to determine intersection. Standard dependence tests come in two flavors: *exact* and *inexact*. The exact tests determine the existence of exact integer solutions to the equations involved in the dependence test. Exact tests are complicated, and as a result, implementations of these tests generally handle very specific and well structured loops and indexing expressions [3]. The inexact tests are less precise and are thus sufficient but not neccessary in determining dependence. Shape-based schemes use the geometric properties of array sections for union and intersection. The intersection operation is always exact since the shape-based scheme uses geometric objects; the intersection of d-dimensional accesses is the intersection of these objects in d space. We use this distinction to present the array summary techniques in the next section.

5 Comparison of Summary Techniques

In this section we compare two standard schemes, *atoms* and *regular sections*, to two shape-based schemes, *convex hulls* and *simple sections*. Of the two standard schemes, the atom technique

uses simple multi-site summary representations, while regular sections use completely condensed representations. Both shape-based schemes use completely condensed representations (see Table 1.)

summary representation	standard	shape-based
simple multi-site	atom	
condensed multi-site	regular section	convex hull simple section

Table 1: Classification of the four techniques.

We use the following example to demonstrate the advantages and limitations of each technique. On line 1 in LowerTriangle, part of a lower triangular region, bounded by ub, of the $n \times n$ array parameter A is written. On line 2, part of row x, bounded by ub, is also modified.

```
LowerTriangle (A, ub)
        { modify the lower triangular region, bounded by ub }
        for i1 ← 0 to ub by 1 do
            for i2 ← 0 to i1 by 1 do
1               A_{i1,i2} = ...

        x = ub/2
        { modify part of row x, bounded by ub }
        for i2 ← 0 to ub by 1 do
2           A_{x,i2} = ...
```

Figure 3 shows the exact triangle, row and combined sections written in LowerTriangle. The array access descriptor for the triangle access is as follows:

$$\begin{aligned} \Phi_1(A) &= (i_1, i_2) \\ B_1(A) &= ((0, ub), (0, i_1)) \\ \Sigma_1(A) &= (1, 1). \end{aligned}$$

The array access descriptor for the row access is as follows:

$$\begin{aligned} \Phi_2(A) &= (x, i_2) \\ B_2(A) &= ((0, ub)) \\ \Sigma_2(A) &= (1). \end{aligned}$$

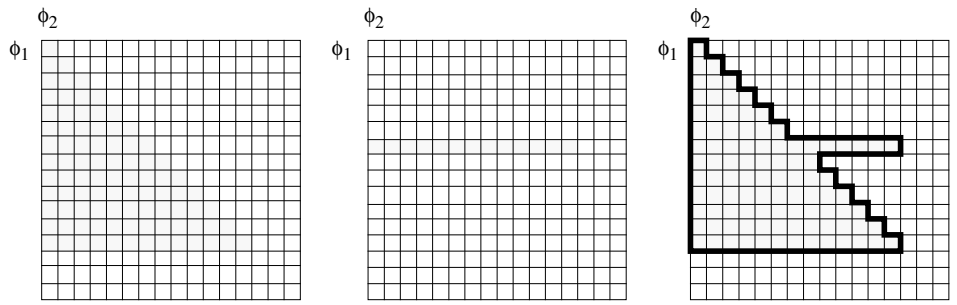

Figure 3: Array sections of A modified by a call to LowerTriangle (the element $A_{0,0}$ is the upper left corner of the array; ϕ_1 increases downward and ϕ_2 increases to the right.)

5.1 Standard Schemes

One reason for using a standard scheme is that standard dependence tests developed for intraprocedural analysis may be directly applied to the representations. Standard dependence tests can be interchanged for different levels of accuracy and cost, and thus are treated as invariants in our comparison.

Atoms. Li and Yew [11, 12, 13] use a data structure called an *atom* for single-site summaries. The multi-site summary is simply a list of atoms; in other words, it is a simple multi-site summary. The union operation can compose the simple multi-site summary in time linear in the number of summaries involved in the union. Though fast and highly accurate, simple multi-site summaries require space proportional to the number of references being summarized. Moreover, the intersection of two simple multi-site summaries requires testing for dependence between every single-site summaries in each multi-site summary.

An atom is essentially a matrix describing the ordered sets Φ and B, with very few restrictions on the expression of Φ and B. An expression can be any combination of the following: a constant, a procedure invariant[2], loop induction variable or a variable of unknown value. The rows of the matrix are the index expressions, Φ, followed by the induction variables, I. The columns include the induction variables I, a column for the constant term, c, and a column to indicate linearity. A position in the first d rows of an atom is the coefficient (or constant value, for column c) of the loop induction variable in the index expression; a position in the last l rows of an atom is the coefficient (or constant value, for column c) of the loop induction variable in the loop boundary condition (see Figures 4 and 5.) Notice that loop boundary condition expressions are not allowed to contain loop induction variables of loops that are nested deeper that the loop for which the boundary condition is being set (these are marked by a $-$ in Figures 4 and 5.) The size of an atom is $(d+l) \times (l+2)$. If l is assumed to be no bigger than d, then an atom's size is proportional to d^2.

[2]A procedure invariant is a variable whose value does not change after the first use of that variable in that procedure.

	Linear	c	i_1	i_2
ϕ_1	Yes	0	1	0
ϕ_2	Yes	0	0	1
i_1	–	ub	–	–
i_2	–	0	1	–

Figure 4: Atom for the use of A in line 1 of LowerTriangle.

	Linear	c	i_2
ϕ_1	Yes	0	1
i_2	–	ub	–

Figure 5: Atom image for the use of A in line 2 of LowerTriangle.

Regular Sections. Regular sections are motivated by observing the common case. Most array accesses do not require the expressibility offered by Li and Yew's atoms. Originally proposed by Callahan [4], regular sections have undergone a few evolutionary changes. *Restricted* regular sections [5] can represent accesses to rows, columns and diagonals, and their higher dimension equivalents. *Bounded* regular sections [9], while disallowing the diagonal, consider bounds and stride information. Both are completely condensed multi-site summaries.

Restricted regular sections are represented by the index expressions, Φ. Each expression, ϕ_i, is either a constant or procedure invariant, α, or an index expression of the form $\pm i_j + \alpha$. Each index position can vary over only one loop induction variable. The size of a restricted regular section summary is $O(d)$. The restricted regular section for the above example is as follows:

$$\begin{aligned} \Phi_1(A) &= (i_1, i_2) \\ \Phi_2(A) &= (x, i_2) \\ \Phi_{1\cup 2}(A) &= (i_1, i_2). \end{aligned}$$

The summary is obtained by the following algorithm. If the i^{th} index position of both of the sections involved in the summary is the same constant, k, then ϕ_i of the union is k. If ϕ_i is not already part of a diagonal, then ϕ_i of the union is the outer-most induction variable over which either section varies. A diagonal is identified by two or more index positions that vary over the same loop induction variable. Consequently, if the previous cases do not hold, for each index position, ϕ_i, every other index position is checked to see if a diagonal exists. If the same diagonal exists in both sections, that diagonal becomes part of the union. Notice that although diagonals are representable, triangles are not representable since bounds information is not preserved. The ability to express diagonal sections forces the union operation to run in $O(d^2)$ time. The shaded area in Figure 6 shows the restricted regular section summary of the accesses to A in LowerTriangle.

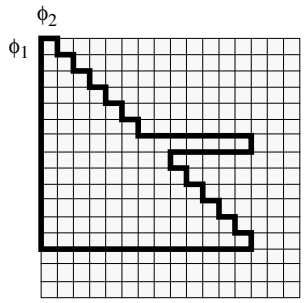

Figure 6: Restricted regular section summary of the accesses to A in LowerTriangle.

Unlike the restricted version, bounded regular sections exclude diagonals, yet maintain loop bounds and stride information. Though the entire array access descriptor, Δ, may be needed, the size of a bounded regular section representation is $O(d)$, due to representation and restrictions on the expressions. An expression may only contain constants or procedure invariants. A bounded regular section can be represented by a single expression or a triplet, $[lb_i : ub_i : \sigma_i]$ for each index position. Diagonals are not describable, since the all loop induction variable information is lost in the triplet. Triangles also cannot be accurately represented since loop bounds are not allowed to include other loop induction variables. The bounded regular section for LowerTriangle is as follows:

$$\begin{aligned} BRS_1 &= ([0 : ub : 1], [0 : ub : 1]) \\ BRS_2 &= (x, [0 : ub : 1]) \\ BRS_{1\cup 2} &= ([0 : ub : 1], [0 : ub : 1]). \end{aligned}$$

where $BRS_{1\cup 2}$ is the bounded regular section of the union of the sections accessed in lines 1 and 2.

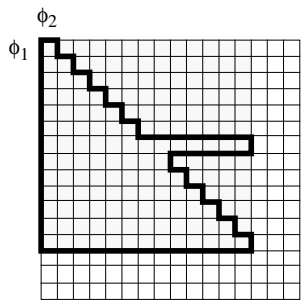

Figure 7: Bounded regular section summary of the accesses to A in LowerTriangle.

The union of two bounded regular sections can be obtained in d steps. The boundary conditions of the union of two sections are obtained by calculating the minimum of the two lower bounds,

lb, and the maximum of the two upper bounds, ub. The stride is obtained by calculating the greatest common divisor of the two stride expressions, s. Figure 7 shows the bounded regular section summary of the accesses to A in LowerTriangle.

5.2 Shape-Based Schemes

Shape-based schemes exploit the geometric characteristics of the access patterns in regular scientific applications. Any array can be thought of as a d-dimensional space. Array accesses often define geometric shapes in this space. While the intersection operation is exact for both techniques, the union operations may introduce areas which are not in either section involved in the union. Both techniques presented below are completely condensed summaries.

Convex Hulls. Triolet et al. [17] suggests using the smallest convex regions covering the array accesses to describe the shape of the accesses. The convex hulls are described by *regions* and *execution contexts*. A region is simply the ordered set of index expressions, Φ, and the execution context is the loop boundary conditions, B. The result is a set of linear equalities and inequalities that describe the convex hull. The union and intersection operators are implemented using solutions from convex hull theory. A multi-site summary is a convex hull obtained by computing the union of the convex hulls involved in the summary. The intersection operation looks for the existence of a convex hull that satisfies both sets of linear equalities and inequalities. The union and intersection operations are rather expensive and grow more so as the number of edges of the convex hull increases. The system of linear equations for the triangle and row accesses in LowerTriangle are shown below and the convex hull describing the union of the two accesses is illustrated in Figure 8.

$$
\begin{aligned}
H_1 &= \{i_1 \geq 0, i_1 \leq ub, i_2 \geq 0, i_2 \leq i_1\} \\
H_2 &= \{i_1 = x, i_2 \geq 0, i_2 \leq ub\} \\
H_{1\cup 2} &= \{i_1 \geq 0, i_1 \leq ub, i_2 \geq 0, i_2 \leq ub, i_1 \leq (\frac{x}{ub})i_2\}
\end{aligned}
$$

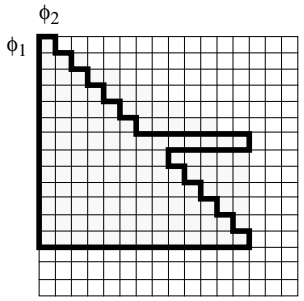

Figure 8: Convex hull defined by the region and execution contexts in LowerTriangle.

Simple Sections. Balasundaram and Kennedy [1, 2] exploit the observation that accesses to arrays usually occur in simple, mathematically describable entities. A *simple section* is any section of the space that can be described by a set of *simple boundaries*. A simple boundary is a line, plane or hyperplane that is either parallel to any axis, or diagonal at a 45° angle to any two axes. A simple section can be described by exactly $2d^2$ simple boundaries: $2d$ for each axis and $2 \times (2 \times \binom{d}{2})$ for each diagonal. Simple sections also maintain stride information. Notice that simple sections are convex regions described by specific edges that are bounded in number.

The following are the simple boundaries for the running example and the union of the two accesses is illustrated in Figure 9. For convenience, we have describe the two boundaries for each axis or diagonal as a single inequality. Each of the $2d^2$ boundaries are maintained, regardless of whether the boundary contributes to the section or not. These boundaries are called redundant and are marked with an asterisk.

$$\begin{aligned} SS_1 &= \{0 \le i_1 \le ub, \\ &\quad 0 \le i_2 \le ub^*, \\ &\quad 0^* \le i_1 + i_2 \le ub + ub^*, \\ &\quad 0 \le i_1 - i_2 \le ub^*\} \end{aligned}$$

$$\begin{aligned} SS_2 &= \{x \le i_1 \le x, \\ &\quad 0 \le i_2 \le ub, \\ &\quad x^* \le i_1 + i_2 \le x + ub^*, \\ &\quad x - ub^* \le i_1 - i_2 \le x^*\} \end{aligned}$$

$$\begin{aligned} SS_{1\cup 2} &= \{0 \le i_1 \le ub, \\ &\quad 0 \le i_2 \le ub, \\ &\quad 0^* \le i_1 + i_2 \le ub + ub^*, \\ &\quad x - ub \le i_1 - i_2 \le ub^*\} \end{aligned}$$

The union of two simple sections is obtained by finding the minimum of each lower bound and the maximum of each upper bound for each inequality. A dependence exists if for all inequalities, the maximum of the lower bounds is less than or equal to the minimum of the upper bounds. Both of these operations run in time proportional to the number of simple boundaries, $2d^2$.

6 Discussion

We divide our discussion into three parts. First, we evaluate the techniques in terms of the three comparison criteria: summary representation, union operation and intersection operation. Next, we present a summary of all the techniques. Finally, we discuss some performance results and examine these results.

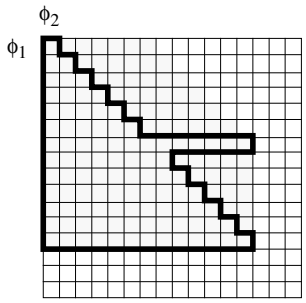

Figure 9: The simple section union of the triangle and row accesses in LowerTriangle.

6.1 Comparison Criteria

In this section, we examine each of the three comparison criteria. For each subsection, the discussion will progress from the *worst* to the *best* techniques, where the worst is the slowest or largest and the best is the fastest or smallest.

Summary Representation. The simplest and smallest summary representation is that of the restricted regular section. It requires space proportional to d, the rank of the array being summarized. In fact, due to the restrictions on the expressions, the space requirement is cd, where c is small. As shown in Figure 6, the lack of bounds information may make the restricted regular section summary approximate a subarray access as an access to the entire array.

Bounded regular sections also require summary information proportional to d, though with a larger constant, as the bounds and stride information are maintained. For LowerTriangle, a bounded regular section results in a slightly more accurate union than that of the restricted version. The additional cost of storing bounded regular sections pays off when procedures work on subsections of the array or use stride values other than one.

The bound on the size of simple section summaries is tight, since even redundant boundaries must be maintained. Every summary requires $2d^2$ simple boundaries. The representations of convex hull summaries can be said to be *minimal* in that they do not contain redundant equations and they describe the smallest convex region covering the array sections. Unfortunately, the number of equations needed to describe a convex hull cannot be precisely bounded.

Since atoms allow an access to vary over more than one loop induction variable, their size is proportional to $l \times (d + l)$, $d + l$ rows and l columns. Recall that Li and Yew's scheme is a simple multi-site summary. Simple multi-site summaries are precise and easy to construct, but are generally not reasonable as the storage requirements and intersection operation do not scale.

Union Operation. The atom technique is the most accurate summary of the schemes presented. The union of any number of atoms is obtained in linear time. We have already determined that the space requirements for this multi-site summary are excessive. A constant time union which does not check for duplicates may magnify the memory requirements. If we assume that only single-site

summaries are merged with multi-site summaries, the duplicate-free union operation requires m atom comparisons, where m is the number of single-site atoms in the multi-site summary. Each atom comparison is actually $l \times (d + l)$ integer comparisons.

For the convex hull scheme, the union always gives the smallest single convex shape that covers every array section. Unfortunately, this may introduce areas that are not in any section. Convex hull operations are dependent on the number of points or inequalities involved in the operation. Therefore, we cannot accurately bound the time for the convex hull operations in terms of d, l or m. Triolet et al. report that their technique generally ran in $O(I^2V)$, where I is the number of inequalities and V is the number of variables in those inequalities.

Simple sections are approximations to convex hulls and thus can be even less precise. Unlike the convex hull algorithms, the bound on the union operation is tight, running in time proportional to the number of simple boundaries, $\Theta(d^2)$. Obtaining the union of two simple sections requires comparing the upper and lower bounds of the d^2 inequalities.

Regular section representations are least accurate. The union of two bounded regular sections runs in $O(d)$ time. Combining restricted regular sections runs is $O(d^2)$, because of diagonal constraints. Both are guaranteed to find the smallest regular section describing the union, though both may introduce areas not in either section being summarized.

Intersection Operation. For the standard schemes, the amount of information available, in terms of Δ, affects the types of standard dependence tests employable. The implementations of exact tests can be unrealistic, while inexact tests are less precise. Linearization [3] has been proposed to increase the precision of standard dependence tests by ensuring *simultaneity* of the index expression. For certain standard dependence tests, an dimension-by-dimension comparison may show dependence between two references, when in fact the accesses are independent due to another dimension's index function. Unfortunately, linearization may also introduce inaccuracies when array references are not contiguous. Stride information may also be sacrificed through linearization.

Since atoms carry very few restrictions on the types of expressions that can be represented, they may make use of any standard dependence test. Unfortunately, the number of *applications* of the dependence test may be unreasonable. A simple multi-site summary requires a pairwise comparison of each single-site summary in each multi-site summary. Intersection of atoms requires $O(m_1 m_2)$ applications of the intersection operator, where m_1 and m_2 are the sizes of each of the multi-site summaries and may be unbounded. Regular sections contain less information, and thus employ less accurate dependence tests. More importantly, since regular sections are completely condensed, the dependence test need only be applied once.

The dependence tests for the shape-based schemes are exact for the array summaries involved. The running times are similar to those for summary union. Again, the time for the convex hull dependence test cannot be accurately bound, as it may depend on the types and number of accesses involved in the summaries. The time bound on the intersection of simple sections is $\Theta(d^2)$.

6.2 Summary

We have examined two types of interprocedural array summary techniques, standard and shape-based. The standard schemes are simpler and can profit from dependence tests developed for

intraprocedural analysis. The shape-based schemes are more precise and have larger memory requirements. Table 2 lists the schemes from most accurate to least accurate and comments on the overall behavior of the scheme.

technique	accuracy	space requirements	time requirements
atom	highest	unbounded	unbounded
convex hull		unbounded	unbounded
simple section		quadratic in d	quadratic in d
bounded regular section		linear in d	linear in d
restricted regular section	lowest	linear in d	quadratic in d

Table 2: Summary of overall behavior of presented techniques.

Of the techniques described, the simple multi-site summary of Li and Yew's atoms is most accurate. Despite this, we dismissed this simple multi-site summary technique, since asymptotically, it is as expensive as performing standard scalar analysis on each array access. Of the condensed schemes considered, Triolet's convex hull method is the next most accurate. Unfortunately, it suffers problems similar to the atom method, as the cost of union and intersection generally grow with the number of accesses summarized.

Regular sections and simple sections have size and running times proportional to the rank of an array being summarized. Asymptotically, bounded regular sections behave up to an order of magnitude better than restricted regular sections and simple sections. Though restricted regular sections allow diagonals, without bound information they seem to be under-specified and gain little, if any, benefit over the bounded regular section implementation. Simple sections are elegant and have a tight bound on their size and operations, though that bound is quadratic in d.

Table 3 shows the size and running times of the discussed schemes. The times for the intersection operation of the standard schemes are marked with an asterisk because they only indicate the number of times the intersection operations must be applied, not the actual time. The actual time for the intersection operation of the standard schemes depends on the standard dependence test used. The union and intersection operations for the convex hull scheme run in time proportional to some function of d, l and m, denoted f_u, for union, and f_i, for intersection.

6.3 Observations

In papers such as this one, pinpointing the *right answer* is almost always dependent on specific details and conditions under which certain assumptions are made (e.g. the problem type, the programming model, the phase of the moon.) With this in mind, we will make some observations and attempt to further evaluate the four array summarization techniques.

technique		Type	Size	Union	Intersection
Standard	atom	simple	$O(m \times l \times (d+l))$	$O(m \times l \times (d+l))$	$O(m_1 m_2)*$
	restricted regular section	condensed	$O(d)$	$O(d^2)$	$O(1)*$
	bounded regular section	condensed	$O(d)$	$O(d)$	$O(1)*$
Shape-Based	convex hull	condensed	$O(m \times (d+l))$	$O(f_u(m,d,l))$	$O(f_i(m,d,l))$
	simple section	condensed	$\Theta(d^2)$	$\Theta(d^2)$	$\Theta(d^2)$

Table 3: Space and time requirements for each summary technique.

Bounds Parameters. Table 3 says nothing about the variables used within it. The first observation is that d, the rank of the array, is generally small. Expressing problems in three dimensions or less is very natural to us because we live in a three dimension world. More concretely, Shen et al. [16] studied array access patterns over a suite of a dozen numerical packages. They found that, of the 18,549 array accesses in this suite, over 99% of them are to arrays of three or less dimensions. In fact, no arrays of rank greater than five are ever accessed. On the other hand, m, the number of static references to a particular array in a particular procedure, is dependent solely on the program being compiled. Consequently, techniques whose behavior varies with d are preferred over those whose behavior varies with m.

Experimental Results. The atoms, bounded regular sections and convex hull techniques were each used to detect parallelizable loops in the presence of procedure calls in LINPACK, a library of linear algebra routines [7]. Of 99 loops containing procedure calls, 27 are parallelizable. The scalar approximation technique is only able to recognize 9 of the loops as parallelizable while each of the three techniques[3] are able to detect all 27.

The LINPACK benchmark consists primarily of basic linear algebra subroutines, or *BLAS*. BLAS are common in many regular scientific applications. The experimental results suggests that for the type of routines in LINPACK, the inexpensive and more general bounded regular section method is sufficient for interprocedural analysis of array accesses.

Context of Work. We should also consider the context in which each technique was developed and also the context in which each technique could be used.

The atoms, bounded regular sections and convex hull techniques are intended to alleviate the problems presented by procedure calls within loops. The granularity of parallelism sought is at the

[3]The simple section technique would certainly perform as well as the bounded regular sections, since simple sections are bounded regular sections with diagonal constraints.

loop-level. Without interprocedural information, a compiler must assume the worst case. Callahan exploited the observation that procedures in BLAS generally perform operations on rows, columns and other easily describable shapes. Those observations proved to be correct, at least for the types of BLAS seen in the LINPACK benchmark.

Simple sections are directed at task parallelism. While an entire procedure may access regions of an array resembling regular sections, portions of the procedure may access more oddly shaped regions, for which simple sections are better suited.

7 Suggestions for Hybrid Representation

Simple multi-site summaries provide high precision, yet the time and space requirements are not scalable. Completely condensed summaries alleviate the time and space demands while sacrificing accuracy. In this section, we suggest a method to convert a completely condensed summary scheme into one that provides a partially condensed hybrid summary.

Creating hybrid summaries should consider three factors: accuracy of analysis, speed of analysis and memory requirements. The accuracy of a hybrid summary is dictated by an acceptance function that decides whether two summaries benefit from condensation. The speed of analysis is affected by the size of the hybrid summary; the number of elements in a hybrid summary determines the number of applications of the dependence test. The memory requirements are directly related to the size of the hybrid summary.

Partial condensation can be reduced to a graph partitioning problem. Each array section is a node; the edges are determined by some locality function that connects nodes that may benefit from condensation. Each node may be weighted with section size information. Each edge can be weighted by quantifying the benefit from condensing two nodes. Using a *general purpose* graph partitioning algorithm is not desirable since this requires precomputing the entire graph, which can be expensive; for every array variable, we must create a complete graph of every section of the array accessed. Consequently, we suggest a heuristic algorithm that constructs parts of the graph *dynamically* as needed. We introduce the *bounding box* acceptance test and two partitioning algorithms, *bucket* and *largest-first*. The algorithms can be applied to two completely condensed techniques that we deemed equitable in the previous section in terms of time, space and precision: bounded regular sections and simple sections.

7.1 Bounding Box Test

An acceptance test should not require significant computation, as this decreases the speed of analysis. Also, the information required for the test should be available from the summary information. In order to illustrate the use of the bounding box test, we use the following example. The procedure TRANSPOSEONE performs a transpose on one column and one row of the given square matrix, A.

```
TRANSPOSEONE (A, lb, ub, x)
    { copy the x^th row }
    for i ← lb to ub do
        temp_i = A_{i,x}
```

```
        { copy the x^th column into the x^th row }
        for i ← lb to ub do
1           A_{i,x} = A_{x,i}

        { copy the x^th row into the x^th column }
        for i ← lb to ub do
2           A_{x,i} = temp_i
```

The bounding box test calculates the area of the bounding box surrounding each of sections and also that of the section produced by merging the two. The test compares the sum of the areas of the two separate sections to the area of the condensed array section. If the condensed summary's area is within some δ of the summed areas, the summaries are condensed. Each bounding box calculation can be performed in $O(d)$ time and the entire test can be performed in $O(d)$ time. The bounding box calculation for TRANSPOSEONE is as follows:

$$\begin{aligned} bb_1(A) &= 1 \times (ub - lb + 1) \\ bb_2(A) &= 1 \times (ub - lb + 1). \\ bb_{1\cup 2}(A) &= (ub - lb + 1) \times (ub - lb + 1). \end{aligned}$$

If δ is $\frac{3}{2}$ and $(ub - lb + 1)^2$ is within $3(ub - lb + 1)$, the two summaries are condensed. Since the area of the bounding box is a single integer, it is reasonable to store the area for every summary.

7.2 Condensation Algorithms

As with the acceptance test, a condensation algorithm should attempt to maintain the asymptotic bound on the running time of collecting condensed multi-site summary information. The union operation for all of the completely condensed schemes is associative, and thus the completely condensed summary can be collected incrementally in a single pass. The bucket algorithm can also run simultaneously with the computation of the single-site summaries. The largest-first algorithm requires a preprocessing pass.

For each array variable, the *bucket* algorithm maintains a set of b partitions called *buckets*. Each bucket contains a completely condensed multi-site summary, and together, these buckets form a partially condensed summary. For each single-site summary visited, the bounding box test is applied to that single-site summary and each bucket. If the test succeeds for only one of the buckets, the single-site summary is added to that bucket, and the new multi-site summary for that bucket is computed. If the test fails for every bucket and the number of buckets is less than b, that single-site summary is placed in a new bucket. If the test fails for every bucket and the number of buckets is equal to b, we must force it into a bucket. The single-site summary should be placed into the bucket for which the bounding box test was closest to succeeding. If the test returns positively for more than one bucket, the single-site summary is added to the bucket that would produce the smallest bounding box area. After the entire procedure is analyzed, the set of buckets become the partially condensed summary.

The *largest-first* algorithm also maintains b buckets, but the buckets are not filled until all of the single-site bounding box areas have been collected. The first pass saves and sorts the summaries by the area of the bounding box around each access. If the number of single-site summaries is less than some small number, $s \leq b$, the summaries are kept as single-site summaries. As an optimization, if any of the single-site summaries completely cover any another, these summaries can be condensed. Otherwise, the original bucket algorithm is run using the sorted list. The larger sections ensure that an unlucky ordering does not place in separate buckets two or more smaller sections that would benefit from being condensed with a larger section.

7.3 Performance Evaluation

The following example is used to demonstrate the bucket algorithm. SplitTranspose performs a single transpose on row x_1 for the square sub array of A starting at $A_{0,0}$ and ending with $A_{s,s}$, and on row x_2 for the square sub array of A starting at $A_{s+1,s+1}$ and ending with $A_{ub,ub}$ (see Figure 10.)

SplitTranspose (A, x_1, s, x_2, ub)
{ Perform a single transpose on the sub array of A starting at 0 bounded by s }
TransposeOne$(A, 0, s, x_1)$

{ Perform a single transpose on the sub array of A starting at $s+1$ bounded by ub }
TransposeOne$(A, s+1, ub, x_2)$

The following assumptions are made. First, the number of buckets, b, is three. Next, the splitter, s, divides either dimension of A into unequal parts and the part from 0 to s is larger than the part from $s+1$ to ub. Next, δ should be small enough so that the bounding box test always fails. The bucket algorithm has left the accesses in TransposeOne uncompressed and we have already bound the summary information in TransposeOne to the environment of SplitTranspose.

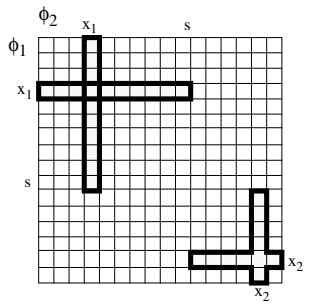

Figure 10: Array sections modified by the procedure SplitTranspose. The dark outlines indicate the partial condensation achieved by the bucket algorithm with bucket size three.

The bucket algorithm processes the following summaries this order: (a) elements 0 to s in row x_1, (b) elements 0 to s in column x_1, (c) elements $s+1$ to ub in row x_2, and (d) elements $s+1$ to ub in column x_2. The summaries (a), (b) and (c) immediately fill the three buckets, since they

fail the bounding box test. Summary (d) also fails, but is forced into the same bucket as (c) since the bounding box around these two accesses is smaller than that of (d) and any other bucket. The partially condensed summary is shown by the dark outlines in Figure 10. Notice that the bucket algorithm is fairly sensitive to b and δ. If we had chosen $b = 2$, due to the order in which the summaries were seen, one bucket would contain (a) and (c) and the other would contain (b) and (d). Even with $b = 2$, a larger δ could have placed (a) and (b) to the same partition and (c) and (d) into the same partition. This actually points to a problem with the largest-first algorithm. The algorithm produces a partition that is as bad as the normal bucket algorithm for groups of elements of nearly the same size. We may want to embed some locality information in our list by using ranges instead of discrete sizes and then ordering the elements of the ranges by some locality metric. Using a locality metric slows down the preprocessing step by the cost of obtaining locality information. Alternatively, all elements of nearly the same size may be considered together. This may increase the time for a single step of the algorithm by a factor of $\frac{(b-1+m)!}{(b-1)!m!}$, if all possible permutations are considered. Both of the two methods increase the running time of the bucket algorithm by more than a constant factor and are not discussed further here.

Limiting the number of buckets to b maintains the asymptotic bound on the running time of the dependence test; an unbounded bucket algorithm may result in a simple multi-site summary. Moreover, b bounds the running time of the partial condensation. A complete condensation takes m total applications of the union operation per array variable, where m is the number of accesses to that array. The bucket algorithm requires at most m union operations and $m * b$ applications of the acceptance test. Since b is a constant, the cost of each union operation increases by at most $O(d)$, the price of the acceptance test.

The bucket algorithm suffers from an ordering problem: collection of summary information is no longer associative. The order in which each of the accesses is visited determines the partition of the summaries. Though the accuracy of such a partially condensed summary is never worse than that of the completely condensed summary, a poor partition increases the running time of the dependence test by up to a factor of b^2.

Largest-first is a two-pass algorithm that corrects the associativity problem by preprocessing. Each step of the first pass runs in time proportional to the cost of the bounding box calculation, $O(d)$, plus insertion into the sorted list $O(\log m_{ave})$, where m_{ave} is the average number of elements in the list. The running time of the new pass is the same as for the bucket algorithm. Effectively, the cost of each union operation is increased by $O(d + \log m_{ave})$.

Recall that we expect d to be not much more than three. With this consideration, the bucket algorithm's asymptotic behavior is the same as for complete condensation. Unfortunately, we cannot bound m_{ave}, and thus the largest-first algorithm always behaves worse than complete condensation. One thing that can be said about m_{ave} is that it is not necessarily a function of program size. A larger program may imply an increase in the *number* of procedures, but it does not necessarily imply an increase in the *size* of those procedures. Since m_{ave} grows logarithmically, the largest-first algorithm seems reasonable.

In summary, partial condensation can be viewed as a graph partitioning problem. Since the graph describing the array sections and their locality is expensive to compute, we have introduced two partitioning algorithms that create the graph dynamically. The bucket algorithm increases the running time of the collection of summary information by $O(d)$. The largest-first algorithm

requires a preprocessing pass and increases the running time by $O(d + \log m_{ave})$. The parameters b and δ enable tuning of accuracy and performance. Neither hybrid summary is less accurate than a completely condensed summary, but the number of applications of the dependence test needed may increase by a factor of b^2.

8 Conclusions

Compiler techniques for interprocedural array section analysis enable optimizations that benefit parallel scientific codes. Both compiler-guided and programmer-guided parallelization take advantage of information gathered about array accesses across procedure boundaries. We have identified two types of interprocedural array section summary techniques, standard and shape-based. While the shape-based schemes cleverly exploit the geometric properties of array accesses and are usually more accurate, the standard schemes can leverage dependence tests developed for intraprocedural analysis. We also noticed that while completely condensed multi-site summaries add inaccuracies, simple multi-site summaries are unreasonable due to their time and space requirements. Consequently, a hybrid summary representation, obtained through partial condensation, may be desirable.

The technique of choice depends on the behavior of the program being analyzed and the type of parallelism sought. While the type of parallelism sought is often a decision of the compiler, the program clearly is not. As is true in most large systems, when the compiler techniques improve, the style of programming changes. Fortunately, two of the techniques discussed (bounded regular sections and simple sections) exploit the common cases that are very natural to the programmer, namely rows, columns, diagonals and their higher dimensional equivalents. Encouraging the programmer to use this model for programming by employing these analysis techniques would inspire either a change for the better or no change at all in programming style.

Acknowledgements. Many thanks go out to Craig Chambers, Susan Eggers, Burton Smith and Larry Snyder for their (repeated) careful readings of this paper and the lessons learned from them.

References

[1] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9:154–170, 1990.

[2] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 41–53, June 1989.

[3] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 162–175, June 1986.

[4] D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, March 1987.

[5] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550, 1988.

[6] K. D. Cooper, M. W. Hall, and L. Torczon. Unexpected side effects of inline substitution: A case study. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, March 1992.

[7] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK: user's guide*. SIAM Publications, Philadelphia, Pennsylvania, 1979.

[8] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 26, pages 15–29, June 1991.

[9] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[10] C. A. Huson. An in-line subroutine expander for parafrase. Master's thesis, University of Illinois at Urbana-Champaign, January 1983.

[11] Z. Li and P.-C. Yew. Efficient interprocedural analysis for program parallelization and restructuring. In *Proceedings of ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 85–99, July 1988.

[12] Z. Li and P.-C. Yew. Interprocedural analysis and program restructuring for parallel progams. Technical Report CSRD Report No. 720, University of Illinois at Urbana-Champaign, Urbana, Illinois, January 1988.

[13] Z. Li and P.-C. Yew. Interprocedural analysis for parallel computing. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 221–228, August 1988.

[14] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[15] R. W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, September 1977.

[16] Z. Shen, Z. Li, and P.-C. Yew. An empirical study on array subscripts and data dependencies. In *Proceedings of the International Conference on Parallel Processing*, pages 145–152, August 1989.

[17] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 176–185, June 1986.

[18] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, October 1982.

[19] A. Yazici and T. Terzioğul. A comparison of data dependence analysis tests. *Parallel Computing and Transputer Applications*, 1:575–583, September 1992.