# Recurrence Analysis for Automatic Parallelization of Subscripted Subscripts

Akshay Bhosale
University of Delaware
Newark, Delaware, USA
akshay@udel.edu

Rudolf Eigenmann
University of Delaware
Newark, Delaware, USA
eigenman@udel.edu

## Abstract

Introducing correct and optimal OpenMP parallelization directives in applications is a challenge. To parallelize a loop in an input application code automatically, parallelizing compilers need to disprove dependences with respect to variables across iterations of the loop. Performing such dependence analysis in the presence of index arrays or subscripted subscripts – $a[b[i]]$ – has long been a challenge for automatic parallelizers. Loops with subscripted subscripts can be parallelized if the subscript array is known to possess a property such as monotonicity. This paper presents a compile-time algorithm that can analyze complex recurrence relations and determine irregular or *intermittent* monotonicity of one-dimensional and monotonicity of multi-dimensional subscript arrays. The new algorithm builds on a prior approach that is capable of analyzing simple recurrence relations and determining monotonic one-dimensional subscript arrays. Experimental results show that the proposed array analysis algorithm can substantially improve the performance of ten out of twelve or 83.33% of the benchmarks evaluated, 25-33.33% more than state-of-the-art compile-time automatic parallelization techniques.

***Keywords:*** Subscripted Subscript, Monotonicity, Recurrence.

## 1 Introduction

The solution of sparse linear systems is the most computationally intensive part of many scientific applications [11, 17]. Methods for the direct solution of a sparse linear system rely on matrix factorization. Such algorithms, coupled with the storage format of the sparse matrix in memory, lead to the use of indirection arrays, resulting in subscripted subscript patterns. These program patterns are also observed in deep learning kernels [45], adaptive mesh refinement applications and molecular dynamics simulations [13]. Figure 1 shows an example subscripted subscript loop. In loops with cross-iteration accesses to the host or subscripted array, current dependence testing methods conservatively assume dependences with respect to the host array, as compile-time analysis techniques fail to gather information about the subscript array. The information can be a property or range of possible values of the array and can be determined from the program code itself. We have developed a compile-time algorithm capable of performing such analysis and detecting *monotonicity* of the subscript array, which in turn can be used to prove the absence of data dependences in loops that involve subscript arrays. The algorithm builds on a method introduced in [5][1] and has been fully implemented in the Cetus automatic parallelizer [4], which represents the state of the art in automatic parallelization.

```
1  for (j = 0; j < numPlaced; j++)
2      y[ind[j]] += gamma2[i] * exp(-((xdos[ind[j]]-t)*
3                     (xdos[ind[j]] - t))/sigma2);
```

**Figure 1.** Example Subscripted subscript loop from the EVSL library [19]. Values of array *ind* appear at the subscript of array $y$ on line 2.

In our hand analysis of application codes with subscripted subscript loops we observed that, in a class of programs, the subscript array assumes a property such as monotonicity, when the array is assigned values in statements that formulate recurrence relationships. Example loops with such recurrence relations are shown in Figures 2 and 3. A common characteristic of the sequences created by these relations is that the current value in the sequence is computed by adding a Positive or Non-Negative (*PNN*) value or value range to the immediately preceding value. Two types of loops can assign monotonic values to a one-dimensional array. Loops shown in Figure 2 comprise recurrence relations wherein the subscript array (array $a$) is assigned monotonic values in contiguous loop iterations. The method of [5] is capable of handling such loops and determining monotonicity. The recurrence relation in the loop of Figure 3(a) assigns monotonic values to array $a$ in non-contiguous or intermittent loop iterations. Furthermore, the recurrence relation in the loop of Figure 3(b) leads to monotonic multi-dimensional

*PPoPP'24, March 02–06, 2024, Edinburgh, UK*

[1]Also referred to as the *prior approach* and *Base Algorithm* in this paper.

subscript arrays. This work presents an array analysis algorithm that is capable of determining regular or continuous as well as irregular or intermittent monotonicity of a one-dimensional subscript array. In addition, the algorithm can determine monotonicity of multi-dimensional subscript arrays in complex loop patterns.

```
p = 0;
for(i1=0;i1< n;i1=i1+1){
S1:   a[i1] = p;
   ...
   for(in=0;in< m;in=in+1){
      if(condition){
S2:      p = p + 1;
      }
   }
}
```

(a)

```
a[0] = 0;
for(i1=P1;i1< n;i1=i1+1){
   a[f(i1)] = a[f(i1)-1] + k;
}
```

(b)

**Figure 2.** Generalized forms of loops with recurrence relations that can be handled by the method of [5]. The recurrence statements assign values to index array *a*. For the loop in (b), the subscript expression of array *a*, i.e. $f(i_1)$, is a linear expression and can be either $i_1 + 1$ or $i_1$ depending on the initial value of $i_1$, i.e $P_1$, being 0 or 1.

```
ind = 0;
for(i1=0;i1< n;i1=i1+1){
   if (condition){
      a[ind] = i1;
      ind = ind + 1;
   }
}
```

(a)

```
for(i1=0;i1< n;i1=i1+1){
...
 for(in=0;in< m;in=in+1){
   a[i1]...[in] = α*i1+[rl:ru];
                  //α+rl≥ru
 }
}
```

(b)

**Figure 3.** Generalized forms of loops with recurrence relations that can be handled by the new array analysis algorithm in addition to the recurrences in Figure 2.

Our algorithm proceeds in two phases. Phase-1 determines an expression that represents the effect of symbolic execution of an arbitrary loop iteration on the value of the subscript array. Two important capabilities of the Phase-1 algorithm enable the determination of array properties in Phase-2:

1. Ability to analyze statements that assign values to an array under an *if*-condition and
2. Ability to analyze and simplify symbolic expressions that assign values to elements of a multi-dimensional array.

Phase-2 aggregates the Phase-1 expressions across loop iterations, determining the effect of the loop as well as tests for subscript array properties. The Phase-2 algorithm builds on two concepts that are at the core of the method of [5] – *Simple Scalar Recurrence* (SSR) and *Scalar Recurrence Array Assignments* (SRA). The SSR concept determines monotonic scalar variables (referred to as SSR variables), whereas SRA determines monotonic one-dimensional arrays assigned the values of an SSR variable in continuous loop iterations. Both SSR and SRA can be observed in the recurrence relations in the loop of Figure 2(a) (Statements $S_2$ and $S_1$). Our algorithm builds on these concepts by providing a mechanism to analyze and represent recurrence expressions in SSR variables that (a) assign values to a one-dimensional array in intermittent loop iterations and (b) assign values to a multi-dimensional subscript array. The properties determined by the analysis algorithm are used by an extended data dependence test to prove non-overlap in the loops where the subscript arrays appear. The extended data dependence test will be discussed in a forthcoming contribution.

Various techniques have been proposed in the literature to gather information about subscript arrays and parallelize the loops using them. Compile-time techniques involve the use of pattern matching [1, 20, 21] or user-defined assertions [24–26] and have not been fully automated. Proposed run-time techniques make use of an inspector-executor scheme [40] or speculative execution [8, 36]. Though powerful, a major drawback of these techniques is the involved overhead which adds to the execution time of the application. By contrast, our approach uses compile-time analysis, *avoiding run-time overheads*.

In summary, this paper makes the following contributions:

- We present two novel concepts for analyzing array properties – intermittent monotonicity and monotonic multi-dimensional arrays.
- We discuss a new compile-time algorithm capable of analyzing complex recurrence relations and determining monotonicity of subscript arrays, which is sufficient for automatically parallelizing a class of irregular applications.
- We present experimental results comparing the performance impact of the new array analysis algorithm, the state of the art technique of [5] and the classical automatic parallelization techniques on a set of benchmarks from popular benchmark suites.

## 2 Compile-time Subscript Array Analysis

This section presents a compile-time algorithm capable of determining a property for the subscript array (one-dimensional or multi-dimensional). We start by defining the important subscript array properties observed in application codes. Recall from Section 1 that we found the properties to be *present in the programs* and not dependent on program input data.

### 2.1 Subscript Array Properties

In our analysis of application codes from various benchmark suites we found that loops with subscripted subscript patterns can often be parallelized, if the subscript array is known to be monotonic. In some cases, *Non-Strict Monotonicity* suffices, whereas, in other cases, the subscript array is required to be *Strictly Monotonic* or *Injective*. Monotonicity of one-dimensional arrays is well defined in the literature [5, 25, 43]. We will define the monotonicity property of multi-dimensional arrays.

**Definition 1: Monotonic Multi-Dimensional Arrays**
For an $n$-dimensional array $a$ ($n$>1), if $\{a[i][*][*]...[*]_n = [lb{:}ub]\}$ and $\{a[i'][*][*]...[*]_n = [lb'{:}ub']\}$, and if $[lb{:}ub] \leq [lb'{:}ub']$, $\forall i < i'$, then array $a$ is said to be monotonic w.r.t. the dimension at the first position or dimension indexed by $i$. If $[lb{:}ub] < [lb'{:}ub']$, array $a$ is said to be strictly monotonic. The value range $[lb{:}ub]$ is less than $[lb'{:}ub']$, if $ub{<}lb'$. We also refer to this type of Monotonicity as *Range*-Monotonicity. Note that $(*)$ means any legal value.

## 2.2 Algorithm Overview

Our algorithm determines the properties described in the previous section by proceeding in program order, analyzing the loops in each nest from inside out. At each loop level, two phases analyze the values of the variables of interest. These are: loop-variant integer scalars and integer arrays (arrays with loop-invariant subscript expressions are treated as scalars). Loops containing function calls with side effects (Certain C standard library function calls are considered side-effect free by Cetus [2, 34]) and break statements are considered ineligible for analysis. All eligible loops are normalized, with each statement making at most one assignment, and induction variables having been substituted. Iteration spaces of normalized loops start at 0 and are stride-1. The loop variable represents the iteration number.

## 2.3 The Phase-1 Algorithm

Phase-1 determines the effect of executing an arbitrary loop iteration on the value of a Loop Variant Variable – *LVV*. The algorithm computes the values of *LVVs* at the end of the loop iteration relative to the beginning. It represents this value in the form of a symbolic range expression – $[lb{:}ub]$ (inclusive), where the lower bound $lb$ and upper bound $ub$ are symbolic expressions.

The representation is stored in a *Symbolic Value Dictionary* (*SVD*). The *SVD* is an extension of the Range Dictionary used by Cetus' Range Analysis capability [7] and is a mapping of an *LVV* to its symbolic range expression. The representation can store a set of such ranges, in case more than one expression assign values to the *LVV*. The value expression(s) may include the loop index, constants (loop-invariant symbolic terms) and *LVVs*. Our algorithm makes use of the symbolic range propagation scheme [7], which collects and propagates variable ranges through the program.

The Phase-1 algorithm operates on the *ControlFlowGraph* (*CFG*) of the loop body which is a Directed Acyclic Graph (DAG). Each node in the *CFG* represents a statement in the loop body and the edges represent the control flow. Inner loops are represented by a single, collapsed node, as described in Phase-2. The algorithm performs a forward dataflow traversal of the *CFG* in topological order. At the first node, *LVVs* are initialized to $\lambda$, representing their value at the beginning of the loop iteration being analyzed. At control-flow merge points values take a conservative union of the predecessors (*may* semantics). At control-flow diverge points, the values are tagged with the relevant *if*-condition. Each node updates the current value of an *LVV* to reflect the effects of symbolic execution of the statement it represents. The values are stored in *SVDs* corresponding to each node. The *LVV* values at the last node represent the result of the Phase-1 algorithm.

```
m=0;
for(j=0; j<npts; j++){
 if((xdos[j] - t) < width)
   ind[m++] = j;
}
```

**(a)**

```
m=0;
for (j=0; j<npts; j=j+1) {
  if ((xdos[j] - t) < width){
    _temp_0 = m;
    m = (m+1);
    ind[_temp_0] = j;
  }
}
```

**(b)**

**Figure 4.** Example code, showing (a) the loop to be analyzed; (b) the Cetus-normalized [4] version of the loop.

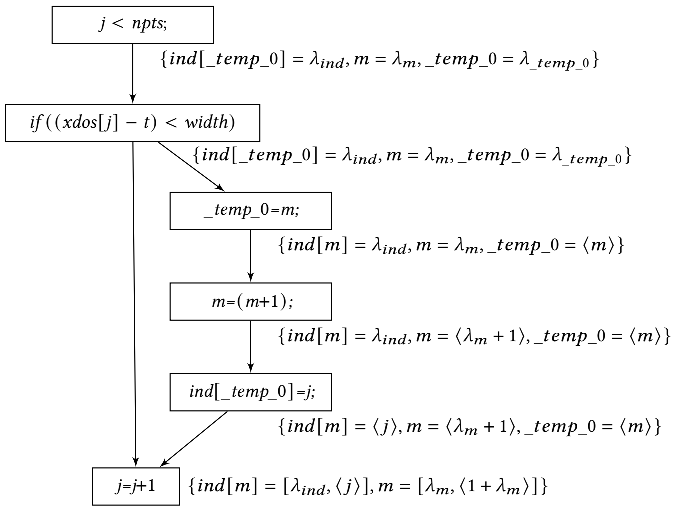

$j < npts;$
$\{ind[\_temp\_0] = \lambda_{ind}, m = \lambda_m, \_temp\_0 = \lambda_{\_temp\_0}\}$
$if((xdos[j] - t) < width)$
$\{ind[\_temp\_0] = \lambda_{ind}, m = \lambda_m, \_temp\_0 = \lambda_{\_temp\_0}\}$
$\_temp\_0{=}m;$
$\{ind[m] = \lambda_{ind}, m = \lambda_m, \_temp\_0 = \langle m \rangle\}$
$m{=}(m{+}1);$
$\{ind[m] = \lambda_{ind}, m = \langle \lambda_m + 1 \rangle, \_temp\_0 = \langle m \rangle\}$
$ind[\_temp\_0]{=}j;$
$\{ind[m] = \langle j \rangle, m = \langle \lambda_m + 1 \rangle, \_temp\_0 = \langle m \rangle\}$
$j{=}j{+}1$
$\{ind[m] = [\lambda_{ind}, \langle j \rangle], m = [\lambda_m, \langle 1 + \lambda_m \rangle]\}$

**Figure 5.** *CFG* of the body of the loop in Figure 4(b) after Phase-1 analysis. The results of Phase-1 are stored in the *SVD* at each node. The representation $\langle expr \rangle$ is used to represent a value expression *expr* tagged with the relevant *if*-condition.

**Example:** For the loop in Figure 4(a), the Cetus-normalized version of the loop is shown in Figure 4(b). Figure 5 shows the *CFG* of the loop body along with the results of Phase-1. The algorithm determines the following symbolic expressions for the *LVVs* for an arbitrary loop iteration $j$:

$$SVD_{stn} \longleftarrow \{ind[m] = [\lambda_{ind}, \langle j \rangle],\ m{=}[\lambda_m, \langle 1 + \lambda_m \rangle]\}$$

where $SVD_{stn}$ is the *SVD* corresponding to the final node in the loop body *CFG* i.e. *j*=*j*+1. The tagged sub-expressions $\langle j \rangle$ and $\langle 1 + \lambda_m \rangle$ represent the values assigned to *ind* and *m* under the *if*-condition on line 3 of Figure 4(b).

## 2.4 Phase-2 Concepts

Phase-2 – the aggregation phase – extends the Phase-1 expressions to the full loop iteration space and tests for array properties. Aggregation reasons about mathematical relationships of the value of variables from one iteration to the next. Recurrence relations are of particular interest. We first introduce the concepts behind the base array analysis algorithm [5] (state of the art concepts) to determine monotonicity of *LVVs*. We then introduce the novel concepts implemented in the new array analysis algorithm and discuss how these concepts build on the capabilities of the base algorithm.

**2.4.1 State of the Art Concepts.** We describe two important concepts that are at the core of the base algorithm – (1) Simple Scalar Recurrence (SSR) and (2) Scalar Recurrence Array Assignments (SRA). These two concepts enable the determination of regular or continuous monotonicity of one-dimensional arrays. The method considers a normalized loop with iteration variable *i*, lower bound 0, and *N* iterations. $\lambda_z$ and $\Lambda_z$ refer to the values of variable *z* at the beginning of the loop iteration and at the beginning of the loop, respectively. The representation *PNN* is used as a placeholder for either a Positive (*P*) or Non-Negative (*NN*) term in an expression. If consecutive elements in a sequence differ by an *NN* term, the sequence is monotonic; if all differences are *P*, the sequence is strictly monotonic.

1. Concept of Simple Scalar Recurrence (SSR): recognizes the form $sc = sc + k$, where $k$ is a loop-invariant *PNN* value or value range. The aggregated value of *sc* in this case is $\Lambda_{sc} + N{*}k$, as computed by Phase-2. *sc* is monotonic if *k* is non-negative. If *k* is positive, *sc* is strictly monotonic.
2. Concept of Scalar Recurrence Array Assignments (SRA): deals with the form $ar[i]$ = *ssr_expr*. The value of an SSR expression is assigned to the current array element. The SSR expression can be an SSR variable plus a *PNN* term. E.g. *ssr_expr* = *sc* + *const*, where *sc* is the SSR variable and *const* is a *PNN* term.
The MA notation facilitates the aggregation of monotonic array sections of type SRA:

**Definition 2: Monotonic Assignment (MA)**
Given an SSR variable *sr*, the array assignment *ap*[0:*N*-1] = *sr#MA* means that array *ap* in the index range [0:*N*-1] gets assigned values of *sr* in a way that is monotonic, i.e. $ap[i] \leq ap[i{+}1]$. If *sr* is strictly monotonic *ap*[0:*N*-1] = *sr#SMA* and, $ap[i] < ap[i{+}1]$.

Using Definition 2, the aggregated expression for *ar* can be expressed as:

$$ar[0:N-1] = \begin{cases} sc\#MA + const, & \text{if } sc \text{ is Monotonic} \\ sc\#SMA + const, & \text{if } sc \text{ is Strictly Monotonic} \\ \perp, & \text{otherwise} \end{cases} \quad (1)$$

where $\perp$ refers to an unknown value or value range. Substituting the aggregated value of *sc* in equation (1) results in:

$$ar[0:N-1] = \begin{cases} (\Lambda_{sc} + N * [lb_k : ub_k])\#MA + const \\ (\Lambda_{sc} + N * [lb_k : ub_k])\#SMA + const \end{cases} \quad (2)$$

where the range $[lb_k{:}ub_k]$ is the value range for *k*.

**2.4.2 Novel Concepts.** The new array analysis algorithm builds on the concepts mentioned above and implements two novel concepts – intermittent monotonicity of one-dimensional arrays and monotonicity of multi-dimensional subscript arrays.

1. **Intermittent Monotonic Sequence**
An Intermittent Monotonic Sequence ($IM(S_n)$) is a sequence that takes on values of a regular (or base) Monotonic Sequence ($S_n$) at irregular intervals.

   Given the base Monotonic sequence,
$S_n = \{a_0, a_1, a_2, a_3, a_4, a_5...\}$, *where* $a_{j+1} > a_j$, $\forall j \in \mathbf{N}$ *then,* $(IM(S_n)) = \{a_j, a_{j+p}, a_{j+p+q}, a_{j+p+q+r}, ...\}$, *where* $\{p, q, r, ...\} \in \mathbf{N}$ *is an intermittent sequence over that base and* $(IM(S_n)) \subset S_n$.

   An intermittent sequence is typically generated by a loop containing a loop-variant *if*-condition, under which the value of $S_n$ is assigned to a variable (typically an array) holding the intermittent sequence. The same *if*-condition may also serve to count the element number of the intermittent sequence. The base sequence may be formed by an induction variable (often the loop index) or a closed-form expression.

   **LEMMA 1:** *Given a loop with N iterations that makes the following assignments under a loop-variant if-condition:*
- (*array*) *inseq*[*ic*] = *j*
- (scalar) $ic = \lambda_{ic} + 1$

   *then, if j is an SSR variable, array inseq will assume monotonic values but at irregular intervals (intermittent sequence). If j is strictly monotonic, array inseq will be strictly monotonic. The intervals are defined by the iterations in which the if-condition is true. Scalar ic counts the element number in the intermittent sequence and assumes the contiguous range* $[0 : ic_{max}]$, *where* $ic_{max}$ *represents the value of ic after the loop and* $0 <= ic_{max} <= N$.

   After aggregation, per LEMMA 1, array *inseq* is known to be monotonic. Using Definition 2, the aggregated expression for *inseq* can be expressed as:

$$inseq[0 : ic_{max}] = \begin{cases} j\#MA, & \text{if } j \text{ is Monotonic} \\ j\#SMA, & \text{if } j \text{ is Strictly Monotonic} \\ \perp, & \text{otherwise} \end{cases} \quad (3)$$

Substituting the value range for $j$ in the above expressions yields:

$$inseq[0 : ic_{max}] = \begin{cases} [lb_j : ub_j]\#MA, & \text{if } j \text{ is Monotonic} \\ [lb_j : ub_j]\#SMA, & \text{if } j \text{ is Strictly Monotonic} \\ \perp, & \text{otherwise} \end{cases} \quad (4)$$

2. **Monotonic Multi-dimensional Arrays**
   **LEMMA 2:** *Given a loop with N iterations that makes the following assignments to an n-dimensional array ax in iteration 'i':*
   *- (array)* $ax[i][*][*]...[*]_n = \alpha * i + [rl{:}ru]$
   *where $\alpha$ is a loop-invariant scalar,*
   *then, if the range [rl:ru] is a PNN range and if $\alpha$+rl ≥ ru, array ax will be monotonic w.r.t. the first dimension or dimension indexed by i. If $\alpha$+rl > ru, ax will be strictly monotonic. The same holds if the dimension indexed by i is in any other than the first position.*

   The aggregated expression for array *ax* in this case will be: $ax[0{:}N\text{-}1][*][*]...[*]_n$ =

$$\begin{cases} \alpha * [0 : N-1]\#(MA; DIM) + [rl : ru], & \text{if } \alpha + rl \geq ru. \\ \alpha * [0 : N-1]\#(SMA; DIM) + [rl : ru], & \text{if } \alpha + rl > ru. \\ \perp, & \text{otherwise} \end{cases} \quad (5)$$

   where *DIM* is an integer value and refers to the position of the dimension w.r.t. which monotonicity exists.

## 2.5 The Phase-2 Algorithm

This section presents an overview of the Phase-2 algorithm for determining monotonicity of *LVVs*. The algorithm implements both the state of the art and novel concepts discussed in the previous section. We describe the implementation of the novel concepts in the new algorithm.

Figure 6 shows the driver algorithm for Phase-2. The algorithm determines the monotonicity property of an *LVV* ($v$) by analyzing $R_v$ (expression for $v$ after Phase-1). For $R_v$'s comprising both tagged and untagged sub-expressions, the algorithm considers only the tagged expressions for analysis (lines 9-10). If $v$ is an SSR variable (lines 12-14), the algorithm adds $v$ to the list of SSR variables (*List_SSR_vars*). The loop index variable is known to be a strictly monotonic SSR variable. To determine monotonicity of an array *LVV*, the algorithm calls upon the function *is_Mono_Array* which implements the novel concepts presented in Section 2.4.2. Upon determining a property, the algorithm determines the aggregated symbolic expression for $v$ (lines 13 and 17).

Function *is_Mono_Array* is shown in Figure 7. The function detects the monotonicity property of one-dimensional (intermittent) and multi-dimensional input arrays by analyzing $R_v$'s comprising SSR variables. Statements on lines 10 to 16 determine intermittent monotonic arrays. The important step in this case is the analysis of the *if*-conditions tagged to $R_s$ (expression of $s$ – the array subscript) and $R_v$ for equality and loop variance (lines 13-15). The function returns *true* if the tagged *if*-conditions are equal and loop variant. Function *Aggregate* on line 17 of the driver algorithm determines the aggregated symbolic expression for $v$ depending on the property Strict or Non-strict monotonicity of $R_v$. The aggregated expression is [0:*N*-1]#*MA* if $R_v$ is monotonic or [0:*N*-1]#*SMA* if $R_v$ is strictly monotonic as per equation (4) in Section 2.4.2.

**Algorithm 1:** Phase-2

```
1  Input: 1. Loop control flowgraph ($LG$)
          2. SVD of the final statement in $LG$ ($SVD_{stn}$) after Phase-1
          3. Loop index ($Idx$), Iteration count ($N$), Range of $Idx$ ($LIR$)
4  Output: 1. Aggregated symbolic expression for each $LVV$
           2. Collapsed loop flowgraph ($collap(LG)$)
6  Add ($Idx : LIR$) to $SVD_{stn}$
7  $List\_SSR\_vars \longleftarrow (Idx)$
8  for each ($v, R_v$) ∈ $SVD_{stn}$ do
9      if ($R_v$ contains tagged sub-expression) then
10         $R_v \leftarrow$ Tagged sub-expression of $R_v$
11     if $v$ is a Scalar then
12         if $is\_SSR(v, R_v)$ then
13             $R_v \leftarrow Aggregate(v, R_v, N)$
14             $List\_SSR\_vars \longleftarrow (v)$
15     else
16         if $is\_Mono\_Array(v, R_v, List\_SSR\_vars, SVD_{stn})$ then
17             $R_v \leftarrow Aggregate(v, R_v, N)$
18         else
19             $R_v \leftarrow$ Simplified $R_v$ after substituting for all $LVVs$
20     $SVD_{stn} \leftarrow (v : R_v)$
21 Determine final $SVD_{stn}$ if $LG$ is outermost
22 for each ($v, R_v$) ∈ $SVD_{stn}$ do
23     $collap(LG) \leftarrow \{v = R_v\}$
24 Replace $LG$ with $collap(LG)$
```

**Figure 6.** Phase-2 algorithm to determine aggregated symbolic expressions for *LVVs*.

For multi-dimensional arrays, a symbolic range expression is used to assign values to elements of the array. The expression is of the form $\alpha$∗*i*+[*rl*:*ru*] where *i* is the loop index. Statements on lines 21 to 29 of function *is_Mono_Array* analyze such expressions and determine monotonicity of a multi-dimensional array. The function returns *true* if the inequality $\alpha$+*rl* ≥ *ru* is satisfied as discussed in **LEMMA 2**. Note that the analysis is performed only if the array subscript expression is a *simple subscript* (expressions of the form *i*+*k*; where *i* is the loop index and *k* is a loop invariant scalar). The aggregated expression in this case is – $\alpha$∗[0:*N*-1]#(*SMA;DIM*)+[*rl*:*ru*], **if** $\alpha$+*rl* > *ru* or $\alpha$∗[0:*N*-1]#(*MA;DIM*)+[*rl*:*ru*], **if** $\alpha$+*rl* ≥ *ru* as per equation

(5) in Section 2.4.2. After Phase 2, the loop is collapsed and replaced by a single node (*collap*(*LG*)) containing a sequence of assignment statements, representing the effect of the loop on each *LVV* (lines 22-24 of the driver algorithm).

**Algorithm 2:** *is_Mono_Array*

```
Input: 1. An array LVV ($v$)
       2. Value for $v$ after Phase-1 ($R_v$)
       3. List of SSR variables in the loop (List_SSR_vars)
       4. SVD of the final statement in LG ($SVD_{stn}$) after Phase-1
Output: Boolean true or false indicating if $v$ is monotonic.

ssr_var ← SSR variable in $R_v$
if ssr_var does NOT exist then
    return false
$s$ ← Subscript expression of $v$
if ($v$ is one dimensional) then
    $R_s$ ← Tagged expression of $s$ from $SVD_{stn}$
    if ($s$ is scalar && $R_s$ is incremented by 1) then
        $Tag_s$ ← If Condition tagged to $R_s$
        $Tag_v$ ← If Condition tagged to $R_v$
        if ($Tag_s$ and $Tag_v$ are equal and loop variant) then
            /* Intermittent Monotonic array detected */
            return true
    else if $s$ is simple subscript then
        return true
    else
        return false
else if ($v$ is multi-dimensional && $s$ is simple subscript) then
    $\alpha$ ← coefficient of ssr_var
    remainder ← $R_v$ - $\alpha$*ssr_var
    if remainder is NOT PNN then
        return false
    rl ⟵ lowerbound of remainder
    ru ⟵ Upperbound of remainder
    if ($\alpha + rl \geq ru$) then
        /* Monotonic Multi-dimensional array detected */
        return true
else
    return false
return false
```

**Figure 7.** Algorithm to determine intermittent monotonic arrays as well as monotonic multi-dimensional arrays.

## 3 Examples

We demonstrate the effectiveness of our analysis techniques by applying them to three scientific applications: Algebraic Multi-Grid Solver (AMG) [22, 33, 44], the Sampled Dense-Dense Matrix Multiplication (SDDMM) [28, 45] and the Unstructured Adaptive (UA) Benchmark from the NAS Parallel Benchmark (NPB) Suite v3.3.1 [12, 14].

### 3.1 Example 1- from AMG

The outermost *i*-loop of the loop nest shown in Figure 8 can be parallelized, if array *A_rownnz*, whose values appear at the subscript of array *y_data* on lines 4 and 7, is known to be injective. Array *A_rownnz* is filled in the loop shown in Figure 9. The Phase-1 algorithm analyzes the loop of Figure 9 and determines the following expressions for *A_rownnz*, *irownnz* and *adiag* for one loop iteration:

Phase-1 (loop on line 2):
$\{A\_rownnz[irownnz]=[\lambda_{A\_rownnz},\langle i\rangle],$
$irownnz=[\lambda_{irownnz},\langle 1+\lambda_{irownnz}\rangle],\ adiag=A\_i[i+1]\text{-}A\_i[i]\}$

The algorithm tags the expressions $i$ and $1+\lambda_{irownnz}$ with the *if*-condition on line 4 and analyzes these expressions in Phase-2.

Variable $i$ is an SSR variable and the value of $\lambda_{irownnz}$ is incremented by 1. In addition, the tagged *if*-conditions are equal and loop variant. Therefore, the algorithm determines an intermittent monotonic relationship for array *A_rownnz*. Since $i$ is known to be strictly monotonic, array *A_rownnz* is also strictly monotonic. The aggregated expressions for *A_rownnz* and *irownnz* are as follows:

```
1  #pragma omp parallel for if(-1+num_rownnz<=irownnz_max)
       private(i, jj, tempx,m)
2  for (i = 0; i < num_rownnz; i++){
3    m = A_rownnz[i];
4    tempx = y_data[m];
5    for (jj = A_i[m]; jj < A_i[m+1]; jj++)
6          tempx +=  A_data[jj] * x_data[A_j[jj]];
7      y_data[m] = tempx;
8  }
```

**Figure 8.** Subscripted subscript loop to parallelize from the AMGmk mini-application [22]. The loop multiplies a sparse matrix with a dense vector.

```
1  irownnz = 0;
2  for (i=0; i < num_rows; i++){
3     adiag = A_i[i+1]-A_i[i];
4     if(adiag > 0)
5        A_rownnz[irownnz++] = i;
6  }
```

**Figure 9.** Loop that fills in the subscript array *A_rownnz* from the AMGmk mini-application.

Phase-2 (loop on line 2):
$\{A\_rownnz[0{:}irownnz_{max}]=i\#SMA,$
$irownnz=[\Lambda_{irownnz}{:}\Lambda_{irownnz}+num\_rows],adiag=\perp\}$

Here, $irownnz_{max}$ in the aggregated subscript expression of array *A_rownnz* represents the value of *irownnz* after the loop. Substituting in the aggregated expression for $i$ and the value of $\Lambda_{irownnz}$ (value of *irownnz* before the loop) i.e. $\Lambda_{irownnz}$=0, in the above expressions yields:

Phase-2(loop on line 2):
$\{A\_rownnz[0{:}irownnz_{max}]=[0{:}num\_rows\text{-}1]\#SMA,$
$irownnz=[0{:}num\_rows],adiag=\perp\}$

Therefore, array *A_rownnz* in the subscript range 0 to $irownnz_{max}$ is strictly monotonic and hence injective. In the to-be parallelized loop of Figure 8, elements of array

*A_rownnz* in the subscript range – [0:*num_rownnz*-1] are accessed. Since the values of $irownnz_{max}$ and *num_rownnz* are not known at compile-time, a run-time check (-1+*num_rownnz* <=$irownnz_{max}$) is inserted by the Cetus Dependence Test [6], ensuring that only strictly monotonic values of array *A_rownnz* are referenced in the to-be parallelized loop.

### 3.2 Example 2 - from SDDMM

For the loop nest shown in Figure 10, the outermost *r*-loop can be parallelized if array *col_ptr* whose values appear at the subscript of array *p* on line 8 is monotonic. Non-strict monotonicity suffices in this case. Array *col_ptr* is filled in the loop shown in Figure 11. For this loop, the Phase-1 algorithm determines for one loop iteration:

```
 1  #pragma omp parallel for if(-1+n_cols <=holder_max) private(ind,
          r, sm, t)
 2  for (r = 0; r < n_cols; ++r){
 3    for (ind = col_ptr[r]; ind<col_ptr[r+1]; ++ind){
 4      sm=0;
 5      for (t = 0; t < k; ++t){
 6        sm += W[r*k + t]*H[row_ind[ind]*k + t];
 7      }
 8      p[ind] = sm * nnz_val[ind];
 9      }
10  }
```

**Figure 10.** Subscripted subscript loop to parallelize from the SDDMM application [28].

```
1  holder=1; col_ptr[0]=0; r=col_val[0];
2  for(i=0; i < nonzeros; i++){
3     if(col_val[i] != r){
4        col_ptr[holder++] = i;
5        r = col_val[i];
6     }
7  }
```

**Figure 11.** Loop that fills in the subscript array *col_ptr* from the SDDMM application.

Phase-1 (loop on line 2):
$\{col\_ptr[holder]=[\lambda_{col\_ptr},\langle i\rangle],holder=[\lambda_{holder},\langle 1+\lambda_{holder}\rangle], r=[\lambda_r,\langle col\_val[i]\rangle]\}$

The algorithm tags the expressions – $i$, $1+\lambda_{holder}$ and *col_val*[*i*] that assign values to variables *col_ptr*, *holder* and *r* under the loop variant *if*-condition on line 3.

The Phase-2 algorithm analyzes the tagged sub-expressions and determines an intermittent monotonic relationship and strict monotonicity for array *col_ptr*. The aggregated expressions for *col_ptr* and *holder* after the appropriate substitutions and simplification are as follows:

Phase-2(loop on line 2):
$\{col\_ptr[0{:}holder_{max}]=[0{:}nonzeros\text{-}1]\#SMA,\ holder=[0{:}nonzeros],\ r=\perp\}$

Similar to Example 1, the Cetus Dependence Test inserts a run-time check i.e. (-1+*n_cols* <= $holder_{max}$) ensuring that only strictly monotonic values of array *col_ptr* are accessed in the to-be parallelized outermost loop of Figure 10.

### 3.3 Example 3 - from UA (NPB v3.3.1)

The UA Benchmark from the NAS Parallel Benchmarks consists of subscripted subscript loops with multi-dimensional subscript arrays. One such loop is present in the kernel *transf* in the serial version of the benchmark. Due to page constraints, the loop is not shown here. We refer the reader to the benchmark source code [3]. In this loop, values of array *idel* appear at the subscript of another array *tx*. Array *idel* is initialized in the loop shown in Figure 12. The analysis begins with the innermost loop on line 4 of Figure 12. For this loop, the Phase-1 algorithm determines for array *idel*:

Phase-1 (loop on line 4):
$\{idel[iel][0:5][j][i]=[\lambda_{ntemp}+ i*5 + j*25+4,\ \lambda_{ntemp}+ i*5 + j*25,\ \lambda_{ntemp}+ i + j*25+ 20,\ \lambda_{ntemp}+ i + j*25,\ \lambda_{ntemp}+ i + j*5+ 100,\ \lambda_{ntemp}+ i + j*5]\}$

```
 1  for(iel = 0; iel < LELT; iel++) {
 2    ntemp = 125*iel;
 3    for(j = 0; j < 5; j++) {
 4      for(i = 0; i < 5; i++) {
 5        idel[iel][0][j][i] = ntemp+ i*5 + j*25 + 4;
 6        idel[iel][1][j][i] = ntemp+ i*5 + j*25;
 7        idel[iel][2][j][i] = ntemp+ i + j*25 + 20;
 8        idel[iel][3][j][i] = ntemp+ i + j*25;
 9        idel[iel][4][j][i] = ntemp+ i + j*5 + 100;
10        idel[iel][5][j][i] = ntemp+ i + j*5;
11      }
12    }
13  }
```

**Figure 12.** Loop that fills in the subscript array *idel* from the kernel *transf* from the UA Benchmark from the NAS Parallel Benchmarks v3.3.1. [14].

At this loop level, the value of variable *ntemp* is not known and therefore, no property can be determined by the Phase-2 algorithm for array *idel*. In this case, the algorithm first attempts to simplify the expressions and deduce a single expression that represents the range of values assigned. Since a simplified expression cannot yet be determined, the Phase-2 algorithm produces the following aggregated expressions for array *idel* by substituting in the range of *i* :

Phase-2 (loop on line 4):
$\{idel[iel][0{:}5][j][0{:}4]=[(4+25*j+\Lambda_{ntemp} : 24+25*j+\Lambda_{ntemp}),\ (25*j+\Lambda_{ntemp} : 20+25*j+\Lambda_{ntemp}),\ (20+25*j+\Lambda_{ntemp} : 24+25*j+\Lambda_{ntemp}),\ (25*j+\Lambda_{ntemp} : 4+25*j+\Lambda_{ntemp}),(100+5*j+\Lambda_{ntemp} : 104+5*j+\Lambda_{ntemp}),\ (5*j+\Lambda_{ntemp} : 4+5*j+\Lambda_{ntemp})]\}$

Note that $\Lambda_{ntemp}$ in the above expressions is the value of *ntemp* at the beginning of the innermost loop on line 4. The loop is then collapsed and replaced with the aggregated expressions of its *LVVs*. For the *j*-loop on line 3, the Phase-1 algorithm determines for array *idel*:

Phase-1 (loop on line 3):
$\{idel[iel][0{:}5][j][0{:}4]=[(4+25*j+\lambda_{ntemp} : 24+25*j+\lambda_{ntemp}),\ (25*j+\lambda_{ntemp} : 20+25*j+\lambda_{ntemp}),\ (20+25*j+\lambda_{ntemp} : 24+25*j+\lambda_{ntemp}),\ (25*j+\lambda_{ntemp} : 4+25*j+\lambda_{ntemp}),(100+5*j+\lambda_{ntemp} : 104+5*j+\lambda_{ntemp}),\ (5*j+\lambda_{ntemp} : 4+5*j+\lambda_{ntemp})]\}$

$\lambda_{ntemp}$ now represents the value of *ntemp* at the beginning of an iteration of the $j$-loop. As no property can be determined from any of the Phase-1 expressions above, the Phase-2 algorithm proceeds to simplify the expressions. In this case, simplification is successful and the following aggregated expression is determined for *idel* after the $j$-loop:

Phase-2 (loop on line 3):
$\{idel[iel][0:5][0:4][0:4]{=}[\Lambda_{ntemp} : 124{+}\Lambda_{ntemp}]\}$

The algorithm collapses the $j$-loop and begins the analysis of the outermost loop. For the $iel$-loop, the Phase-1 algorithm determines for one loop iteration:

Phase-1 (loop on line 1):
$\{idel[iel][0:5][0:4][0:4]{=}125{*}iel{+}[0:124],\ ntemp{=}125{*}iel\}$

In the above expressions, the range [0:124]=[*rl*:*ru*] is a *PNN* range and 125+0>124 ($\alpha{+}rl{>}ru$). Therefore, the Phase-2 algorithm determines the strict monotonicity property for array *idel.* The final aggregated expression for *idel* is:

Phase-2 (loop on line 1):
$\{idel[0:LELT\text{-}1][0:5][0:4][0:4]{=}[0:125{*}(LELT\text{-}1)]\ \#(SMA;0){+}[0:124], ntemp{=}[0:125{*}(LELT\text{-}1)]\}$

The representation $[0{:}125{*}(LELT\text{-}1)]\#(SMA;0)$, indicates strict monotonicity for *idel* w.r.t. the dimension at position 0 (first dimension). Strict monotonicity of array *idel* is sufficient to disprove any cross-iteration dependences in the to-be parallelized loop in the kernel *transf* of the UA Benchmark.

## 4 Evaluation

This section presents performance results after applying the proposed array analysis algorithm on scientific applications, including the applications discussed in the previous section. The results show that the algorithm is successful in improving the performance of 83.33% of the benchmarks evaluated, 25% more than the method of [5] and 33.33% more than classical automatic parallelization techniques.

### 4.1 Experimental Setup and Methodology

We have implemented the array analysis techniques described in Section 2 in the Cetus automatic parallelizer [4] and evaluated them by performing two experiments. Experiment 1 measures the performance impact of the new techniques on three applications – the Algebraic Multi-Grid mini application (AMGmk v1.0) [22], the SDDMM application [28] and the kernel *transf* from the Unstructured Adaptive (UA) Benchmark from the NAS Parallel Benchmarks v3.3.1 [12, 14]. These three application codes are representative of the benchmarks comprising subscript arrays with intermittent monotonic sequences and monotonic multi-dimensional subscript arrays in code sections critical for parallelization. Experiment 2 compares the performance impact of the new analysis techniques with the technique of [5] and classical automatic parallelization. The evaluation is performed using 12 applications from popular benchmark suites.

Table 1 shows the benchmark suite used for our experiments. The suite is a collection of fundamental benchmarks (regular and irregular) chosen from popular benchmark suites. Table 1 also shows the serial application execution time for the input datasets used. In Experiment 1, multiple datasets were used as inputs to the AMGmk, SDDMM and UA applications. In Experiment 2, a dataset was chosen at random from the available datasets in the source suite and used as input for each of the 12 application codes. We used MATRIX2, dielFilterV2clx and CLASS A as input datasets for the AMGmk, SDDMM and UA applications respectively in Experiment 2.

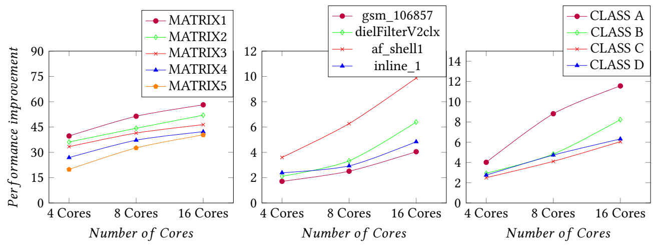
| Benchmark | Source | Input Dataset | Serial Execution time |
|---|---|---|---|
| AMGmk | CORAL suite [22] | MATRIX1 | 1.44 s |
| | | MATRIX2 | 3.112 s |
| | | MATRIX3 | 8.04 s |
| | | MATRIX4 | 14.5 s |
| | | MATRIX5 | 28.66 s |
| CHOLMOD Supernodal | SuiteSparse [10] | spal_004* | 12.35 s |
| SDDMM | Nisa et al. [28] | gsm_106857* | 1.394 s |
| | | dielFilterV2clx* | 1.17 s |
| | | af_shell1* | 0.755 s |
| | | inline_1* | 1.60 s |
| UA(transf) | NPB3.3 [14] | CLASS A | 1.44s |
| | | CLASS B | 9.28 s |
| | | CLASS C | 43.66 s |
| | | CLASS D | 874.22 s |
| CG | NPB3.3 | CLASS B | 40.51 s |
| heat-3d | PolyBench-4.2 [23] | EXTRALARGE | 27.85 s |
| fdtd-2d | PolyBench-4.2 | EXTRALARGE | 22.83 s |
| gramschmidt | PolyBench-4.2 | EXTRALARGE | 17.14 s |
| syrk | PolyBench-4.2 | EXTRALARGE | 7.53 s |
| MG | NPB3.3 and SPECOMP2012 [27] | CLASS B | 4.8 s |
| IS | NPB3.3 | CLASS C | 7.662 s |
| Incomplete Cholesky (C version) | Sparselib++ [35] | crankseg_1* | 27.59 s |

**Table 1.** Benchmarks and input data used. Asterisks indicate data from the SuiteSparse Matrix Collection [16]. Other input datasets are built into the benchmarks. Serial execution times are shown.

Since our technique operates intraprocedurally, we performed inline expansion, so that the to-be parallelized subscripted subscript loops appear in the same subroutine as the loops that define the subscript array. The inline-expanded versions of the application codes have been made available [3]. The execution times for the application codes were recorded on a compute node with a 20-core Intel Xeon Gold 6230 processors in a dual socket configuration, with a processor base frequency of 2.1 GHz, and 27.5MB cache. We used upto 8GB of DDR4 memory. The application codes were compiled using GCC v4.8.5 with the -O3 optimization flag enabled on CentOS v7.4.1708 and we report the mean of 5 application runs. We observed an average run-to-run variation of 1.45% and we used one thread per core.

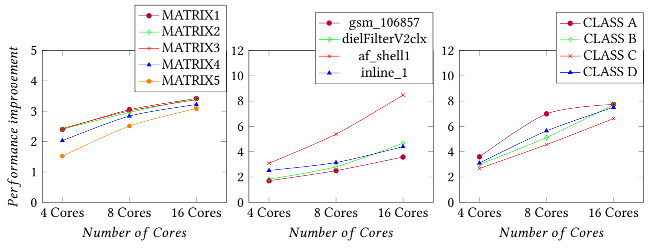
## 4.2 Results of Experiment 1

Figure 13 and Figure 14 show the performance results for the three parallel applications – AMGmk, SDDMM and the kernel *transf* from the UA benchmark – which comprise the subscripted subscript loops parallelizable using our techniques. Performance improvement is defined as the execution time of the Cetus-parallelized codes without versus with our technique. In addition, we measure the improvement in the performance of the Cetus parallelized codes (with our technique applied) versus the serial versions. The figures show the performance on 4, 8 and 16 cores.



**(a)** AMGmk **(b)** SDDMM **(c)** UA (transf)

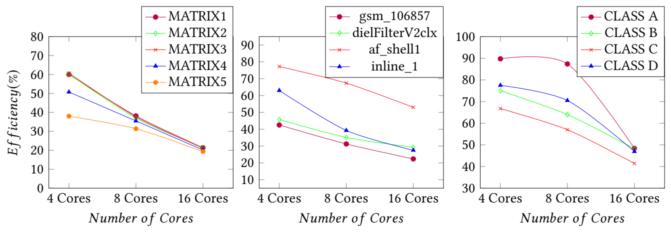
**Figure 13.** Overall improvement in the performance of the parallel application codes with v/s without subscripted subscript analysis applied on 4, 8 and 16 cores.



**(a)** AMGmk **(b)** SDDMM **(c)** UA (transf)

**Figure 14.** Overall improvement in the performance of the parallel application codes with subscripted subscript analysis applied v/s the serial versions on 4, 8 and 16 cores.

When compared against the Cetus parallel code without subscripted subscript analysis (Figure 13), our technique leads to an overall maximum performance improvement of 58× for the AMGmk application, 9.87× for the SDDMM application and 11.56× for the *transf* kernel in the UA benchmark. The reason for this anomaly is that without subscripted subscript analysis applied, classical techniques detect parallelism at the level of the inner loops of the kernel loop nests. The substantial fork-join overhead due to the creation and termination of threads for each iteration of the outer loop leads to a degradation in performance. When compared against the serial versions of the application codes (Figure 14), our technique shows a performance improvement of up to 3.43× for the AMGmk application, 8.48× for the SDDMM application and 7.741× for the *transf* kernel.



**(a)** AMGmk **(b)** SDDMM **(c)** UA (transf)

**Figure 15.** Parallel efficiency of the applications with increasing number of cores.

Figure 15 shows the parallel efficiency (speedup / number of cores) for each of the applications. The decline in efficiency with increasing core count is in part due to imbalanced assignment of loop iterations to threads, resulting from the sparsity pattern of the input matrices and irregular memory accesses. Addressing this issue by implementing better load balancing, synchronization and scheduling schemes in Cetus may further increase the gain due to our technique. For example, Figure 16 shows the impact of dynamic versus (default) static scheduling on the performance of the SDDMM application. The key computational loop in this application (Figure 10) operates on the nonzeros in each column of the input sparse matrix. On average, dynamic outperforms static scheduling by 1.24× on 4 cores, 1.548× on 8 cores and 1.82× on 16 cores for 3 out of the 4 input matrices. For af_shell1, static scheduling performs better due to a fairly balanced distribution of nonzeros across the columns of the matrix.

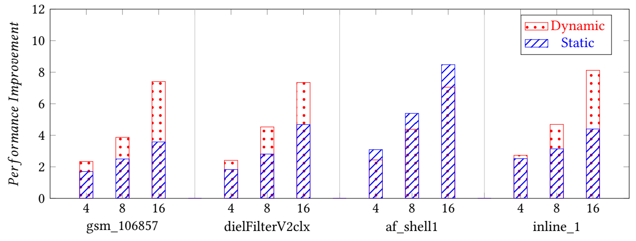


**Figure 16.** Dynamic vs. static scheduling (Cetus' default) for SDDMM on 4, 8 and 16 cores. Showing performance improvement over serial execution.

## 4.3 Results of Experiment 2

Figure 17 shows the performance impact of the new array analysis technique, the method of [5] and the classical automatic parallelization techniques in Cetus on each of the benchmarks shown in Table 1. We recorded the performance of the parallel applications on 16 cores with the performance of the serial versions of the codes chosen as the baseline. The classical automatic parallelization techniques in Cetus

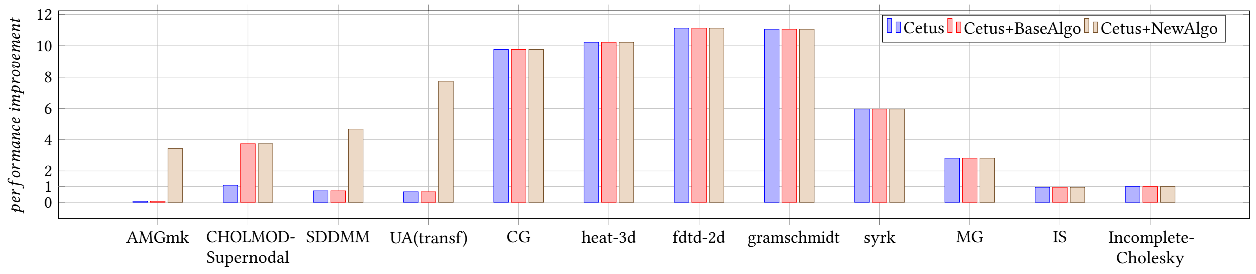


**Figure 17.** Performance comparison of the new array analysis algorithm with the state of the art compile-time automatic parallelization techniques on a set of 12 fundamental benchmarks from popular benchmark suites. Here, *Cetus* refers to the classical automatic parallelization techniques in the Cetus compiler [4], *BaseAlgo* refers to the method of [5] and *NewAlgo* refers to the presented new algorithm. *Cetus* + *NewAlgo* achieves good speedups for 10 out of the 12 benchmarks evaluated.

achieve performance improvement in 6 (CG, heat-3d, fdtd-2d, gramschmidt, syrk, MG) out of the 12 benchmarks. The combination of classical techniques and the method of [5] improves the performance of one additional benchmark (CHOLMOD Supernodal). Classical techniques along with the new techniques presented in this paper achieve performance improvements in 3 additional applications (AMGmk, SDDMM and UA(transf)), with a total of 10 out of 12 or 83.33% of the evaluated benchmarks.

For two benchmarks – IS and Incomplete Cholesky, none of the techniques achieved performance improvement. The IS benchmark comprises subscripted subscript patterns that are too complex to be analyzed at compile-time. The Incomplete Cholesky benchmark includes subscript arrays whose values depend on the program input data. To determine the necessary subscript array properties and parallelize the involved loops, low overhead run-time techniques can be developed that take advantage of the characteristics of the input data and complement the compile-time analyses.

## 5 Related Work

Lin and Padua [20, 21] presented a compile-time demand-driven interprocedural analysis technique for determining index array properties. Their technique uses pattern matching to determine properties such as injectivity, closed form value and closed form bound. Their techniques however are insufficient to determine index array properties in complex loop patterns such as loops that define multi-dimensional subscript arrays, discussed in Section 3.3. Compile-time dependence testing methods have been proposed [6, 29, 43] that take advantage of the monotonicity properties of scalar induction variables appearing in array subscripts instead of computing a closed form expression. These techniques can automatically parallelize loops wherein a closed form expression cannot be determined for the induction variables such as the loops of Figures 9 and 11.

Saltz et al. [39] and Strout et al. [26, 40–42] presented techniques that perform run-time inspection of the index arrays and can determine the relevant index array properties. This information is then used by the executor code, which is a transformed version of the source code loop structures, to execute the computation in an efficient manner. Generating efficient inspectors with low algorithmic complexity is a big challenge. Mohammadi et al. [26] proposed dependence simplification techniques for reducing the cost of inspection. Even with simplified inspectors, the executor code needs to be run 40-60 times to amortize the cost of inspection and gain performance. In general, inspector-executor techniques are not well suited for computational kernels with small workloads such as the ones evaluated in this paper.

Speculative execution techniques [8, 9, 15, 30, 36, 37] execute a sequential loop as parallel and apply a fully parallel data dependence test to determine any cross-iteration dependences. If the run-time test fails, then the loop is re-executed sequentially. The technique performs the analysis and transformations for each invocation of the target kernel, incurring additional overheads. Lazcano et al. [18] proposed a run-time multi-versioning approach that generates several optimized versions of a target kernel and executes the one that performs the best. The approach works entirely at run-time and is capable of handling memory references with indirections.

To reduce the run-time overheads, Hybrid Analysis techniques [31, 32, 38] were proposed that can extract at compile-time the necessary assertions or monotonicity predicates which are then evaluated at run-time to prove the independence of array accesses. The techniques are limited in the type of subscripted subscript patterns that can be analyzed and can incur substantial run-time overheads especially in cases where the generated predicates are too complex.

## 6 Conclusions

In this paper we have presented a compile-time array analysis technique that can analyze complex recurrence relations and derive monotonicity properties of subscript arrays. The algorithm builds on a prior approach and introduces two novel concepts – intermittent monotonicity of one-dimensional and monotonicity of multi-dimensional subscript arrays. We have evaluated the performance impact of the implemented array analysis techniques on the AMGmk and SDDMM applications, and the kernel *transf* of the UA Benchmark from the NAS Parallel Benchmarks. Applying the techniques to these codes yields parallel programs that improve the performance by as much as 3.43× for the AMGmk application, 8.48× for the SDDMM application and 7.74× for the *transf* kernel, compared to the best alternative. Furthermore, when evaluated on a set of fundamental benchmarks from popular benchmark suites, automatic parallelizers equipped with our new analysis techniques can substantially improve the performance of more than 80% of the benchmarks, 25-33.33% more than possible with state-of-the-art techniques. Our techniques are the first fully automated compile-time only techniques capable of determining the monotonicity properties of one-dimensional and multi-dimensional subscript arrays, sufficient for automatically parallelizing a class of irregular applications.

## References

[1] Zahira Ammarguellat and W. L. Harrison. 1990. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. *SIGPLAN Not.* 25, 6 (jun 1990), 283–295. https://doi.org/10.1145/93548.93583

[2] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P Midkiff. 2013. The cetus source-to-source compiler infrastructure: overview and evaluation. *International Journal of Parallel Programming* 41 (2013), 753–767. https://doi.org/10.1007/s10766-012-0211-z

[3] Akshay Bhosale. 2023. *Artifact-SubSubAnalysis*. https://github.com/akshay9594/Artifact-SubSubAnalysis

[4] Akshay Bhosale, Parinaz Barakhshan, Miguel Romero Rosas, and Rudolf Eigenmann. 2022. Automatic and Interactive Program Parallelization Using the Cetus Source to Source Compiler Infrastructure v2. 0. *Electronics* 11, 5 (2022), 809.

[5] Akshay Bhosale and Rudolf Eigenmann. 2021. On the Automatic Parallelization of Subscripted Subscript Patterns Using Array Property Analysis. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 392–403. https://doi.org/10.1145/3447818.3460424

[6] W. Blume and R. Eigenmann. 1994. The range test: a dependence test for symbolic, non-linear expressions. *Supercomputing '94:Proceedings of the 1994 ACM/IEEE Conference on Supercomputing* (1994), 528–537. https://doi.org/10.1109/SUPERC.1994.344316

[7] William Blume and Rudolf Eigenmann. 1995. Symbolic Range Propagation. In *the 9th International Parallel Processing Symposium*. IEEE, California, USA, 357–363. citeseer.nj.nec.com/blume95symbolic.html

[8] Marcelo Cintra and Diego R. Llanos. 2003. Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/781498.781501

[9] Francis H. Dang and Lawrence Rauchwerger. 2000. Speculative Parallelization of Partially Parallel Loops. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, Sandhya Dwarkadas (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 285–299.

[10] Timothy A. Davis. 2006. *Direct Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, USA. https://doi.org/10.1137/1.9780898718881

[11] Timothy A. Davis, Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. *Acta Numerica* 25 (2016), 383–566. https://doi.org/10.1017/S0962492916000076

[12] Huiyu Feng, Rob F Van der Wijngaart, Rupak Biswas, and Catherine Mavriplis. 2004. Unstructured adaptive (UA) NAS parallel benchmark, version 1.0. *NASA Technical Report NAS-04* 6 (2004), 1–17.

[13] Hwansoo Han, Chau-Wen Tseng, et al. 2001. A comparison of parallelization techniques for irregular reductions. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, Vol. 15. IEEE, San Francisco, CA, USA, 27–35. https://doi.org/10.1109/IPDPS.2001.924963

[14] Hao-Qiang Jin, Michael Frumkin, and Jerry Yan. 1999. The OpenMP implementation of NAS parallel benchmarks and its performance. (1999). Preprint on webpage at https://ntrs.nasa.gov/citations/20000102377.

[15] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. 2012. Speculative Separation for Privatization and Reductions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 359–370. https://doi.org/10.1145/2254064.2254107

[16] Scott Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. 2019. The SuiteSparse Matrix Collection Website Interface. *Journal of Open Source Software* 4, 35 (2019). https://doi.org/10.21105/joss.01244

[17] Daniel Langr and Pavel Tvrdik. 2015. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on parallel and distributed systems* 27, 2 (2015), 428–440.

[18] Raquel Lazcano, Daniel Madroñal, Eduardo Juarez, and Philippe Clauss. 2020. Runtime Multi-Versioning and Specialization inside a Memoized Speculative Loop Optimizer. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/3377555.3377886

[19] R. Li, Y. Xi, L. Erlandson, and Y. Saad. 2019. The Eigenvalues Slicing Library (EVSL): Algorithms, Implementation, and Software. *SIAM Journal on Scientific Computing* 41, 4 (2019), C393–C415. https://doi.org/10.1137/18M1170935 arXiv:https://doi.org/10.1137/18M1170935

[20] Yuan Lin and David Padua. 2000. Analysis of Irregular Single-Indexed Array Accesses and Its Applications in Compiler Optimizations. In *Compiler Construction*, David A. Watt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 202–218.

[21] Yuan Lin and David Padua. 2000. Demand-Driven Interprocedural Array Property Analysis. In *Languages and Compilers for Parallel Computing*, Larry Carter and Jeanne Ferrante (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 303–317.

[22] Lawrence Livermore National Laboratory (LLNL). 2014. CORAL Benchmark Codes. Available at https://asc.llnl.gov/coral-benchmarks.

[23] Tomofumi Yuki Louis-Noel Pouchet. 2018. *PolyBench 4.2.1.* https://sourceforge.net/projects/polybench/

[24] Kathryn McKinley. 1991. Dependence analysis of arrays subscripted by index arrays. Technical Report available at http://www.crpc.cs.rice.edu/softlib/TR_scans/CRPC-TR91153throughTR91187/CRPC-TR91163.PDF.

[25] Mahdi Soltan Mohammadi, Kazem Cheshmi, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. 2019. Extending Index-Array Properties for Data Dependence Analysis. In *Languages and Compilers for Parallel Computing*, Mary Hall and Hari Sundar (Eds.). Springer International Publishing, Cham, 78–93.

[26] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 2019. Sparse

Computation Data Dependence Simplification for Efficient Compiler-Generated Inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 594–609. https://doi.org/10.1145/3314221.3314646

[27] Matthias S. Müller, John Baron, William C. Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, Pavel Shelepugin, Matthijs van Waveren, Brian Whitney, and Kalyan Kumaran. 2012. SPEC OMP2012 — An Application Benchmark Suite for Parallel Systems Using OpenMP. In *OpenMP in a Heterogeneous World*, Barbara M. Chapman, Federico Massaioli, Matthias S. Müller, and Marco Rorro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 223–236.

[28] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P. Sadayappan. 2018. Sampled Dense Matrix Multiplication for High-Performance Machine Learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, Bengaluru, 32–41. https://doi.org/10.1109/HiPC.2018.00013

[29] Cosmin E. Oancea and Lawrence Rauchwerger. 2015. Scalable conditional induction variables (CIV) analysis. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015 (Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015)*. Institute of Electrical and Electronics Engineers Inc., United States, 213–224. https://doi.org/10.1109/CGO.2015.7054201 Publisher Copyright: © 2015 IEEE.; 2015 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015 ; Conference date: 07-02-2015 Through 11-02-2015.

[30] Cosmin E. Oancea, Alan Mycroft, and Tim Harris. 2009. A Lightweight In-Place Implementation for Software Thread-Level Speculation. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada) *(SPAA '09)*. Association for Computing Machinery, New York, NY, USA, 223–232. https://doi.org/10.1145/1583991.1584050

[31] Cosmin E. Oancea and Lawrence Rauchwerger. 2012. Logical Inference Techniques for Loop Parallelization. *SIGPLAN Not.* 47, 6 (jun 2012), 509–520. https://doi.org/10.1145/2345156.2254124

[32] Cosmin E. Oancea and Lawrence Rauchwerger. 2013. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Languages and Compilers for Parallel Computing*, Sanjay Rajopadhye and Michelle Mills Strout (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–75.

[33] OSTI. 2013. Algebraic Multigrid Benchmark, Version 00. https://www.osti.gov/biblio/1231933

[34] Phillip James Plauger. 1992. *The standard C library* (1st ed.). Prentice-Hall, Inc., USA.

[35] Roldan Pozo, K Remington, and A Lumsdaine. 1996. Sparselib++ v. 1.5 Sparse Matrix Class Library Reference Guide.

[36] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. 2010. Speculative Parallelization Using Software Multi-Threaded Transactions. *SIGPLAN Not.* 45, 3 (mar 2010), 65–76. https://doi.org/10.1145/1735971.1736030

[37] Lawrence Rauchwerger and David A Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (1999), 160–180.

[38] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2002. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. In *Proceedings of the 16th international conference on Supercomputing* (New York, New York, USA) *(ICS '02)*. Association for Computing Machinery, New York, NY, USA, 11 pages. https://doi.org/10.1145/514191.514229

[39] Joel H Saltz and Ravi Mirchandaney. 1990. *The preprocessed doacross loop.* Technical Report. Institute for Computer Applications in Science and Engineering, Hampton,VA, USA.

[40] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proc. IEEE* 106, 11 (2018), 1921–1934.

[41] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An approach for code generation in the sparse polyhedral framework. *Parallel Comput.* 53 (2016), 32–57.

[42] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. 2014. Non-Affine Extensions to Polyhedral Code Generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 185–194. https://doi.org/10.1145/2581122.2544141

[43] Peng Wu, Albert Cohen, Jay Hoeflinger, and David Padua. 2001. Monotonic Evolution: An Alternative to Induction Variable Substitution for Dependence Analysis. In *Proceedings of the 15th International Conference on Supercomputing* (Sorrento, Italy) *(ICS '01)*. Association for Computing Machinery, New York, NY, USA, 78–91. https://doi.org/10.1145/377792.377809

[44] Ulrike Meier Yang. 2006. Parallel Algebraic Multigrid Methods — High Performance Preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, Are Magnus Bruaset and Aslak Tveito (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–236.

[45] Tong Zhou, Ruiqin Tian, Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, and Vivek Sarkar. 2023. ReACT: Redundancy-Aware Code Generation for Tensor Expressions. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3559009.3569685