On the Automatic Parallelization of Subscripted Subscript Patterns using Array Property Analysis

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

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

ABSTRACT

Parallelizing loops with subscripted subscript patterns at compile-time has long been a challenge for automatic parallelizers. In the class of irregular applications that we have analyzed, the presence of subscripted subscript patterns was one of the primary reasons why a significant number of loops could not be automatically parallelized. Loops with such patterns can be parallelized, if the subscript array or the expression in which the subscript array appears possess certain properties, such as monotonicity. The information required to prove the existence of these properties is often present in the application code itself. This suggests that their automatic detection may be feasible. In this paper, we present an algebra for representing and reasoning about subscript array properties, and we discuss a compile-time algorithm, based on *symbolic range aggregation,* that can prove monotonicity and parallelize key loops. We show that this algorithm can produce significant performance gains, not only in the parallelized loops, but also in the overall applications.

KEYWORDS

monotonicity, aggregation, automatic parallelization.

ACM Reference Format:
Akshay Bhosale and Rudolf Eigenmann. 2021. On the Automatic Parallelization of Subscripted Subscript Patterns using Array Property Analysis . In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447818.3460424

1 INTRODUCTION

Scientific applications such as Molecular Dynamics simulations, Adaptive Mesh Refinement applications and Sparse Matrix computations, are highly irregular. These applications comprise loops with subscripted subscript patterns, wherein an array value appears in the subscript of another array. Current compile-time techniques are ineffective in parallelizing such loops. In a comparative study of applications parallelized by the Cetus [11] compiler and hand-parallelized versions, we have found that such patterns are the main impediment of more successful automatic parallelization. Similar experiments with other compilers or translators, such as Rose [27], Intel's ICC compiler [17] and the PGI [26] compiler yielded the same result. Parallelizing subscripted subscript patterns would require an ability of compilers to determine possible values of the subscript arrays and formulate subscript array properties. What's more, we found that the information required to do so is often present in the application code itself, specifically in loops that modify the content of the subscript array. In this paper, we present a compile-time algorithm capable of analyzing this information and detecting *monotonicity* of the subscript array, which in turn can be used to prove the absence of data dependencies in loops that involve subscript arrays. We manually apply the algorithm to example loops from real scientific applications and demonstrate the improvement in performance of the applications with our technique enabled.

```
1: for (i = 0; i < n; i++) {
2:    for(j = ptr[i]; j < ptr[i+1]; j++){
3:        x[j] = a[j] * diag[b[j]];
4:    }
5: }
```

Figure 1: Example loop using subscripted subscripts. Array x is indexed by j, whose value stems from array *ptr*.

```
for(i = 0; i < n; i++){
      ptr[i] = 0;
}
```

(a)

```
for(i = 0; i < n; i++){
   if(condition) ptr[i]++;
}
```

(b)

```
for(i = 1; i < n; i++){
   ptr[i] = ptr[i - 1] + ptr[i];
}
```

(c)

Figure 2: Defining monotonic subscript values in three steps: (a) initialize to zero; (b) conditional increment; (c) sum recurrence.

For the code example in Figure 1, the outer loop on line 1 can be parallelized if array *ptr*, whose values appear at the subscript of array x on line 3, is known to be monotonically increasing. Before this loop begins, array *ptr* is initialized in a sequence of steps producing monotonic values, as shown in Figure 2. In general scientific applications, the subscript array is filled and modified in more complicated loop patterns. But in our analysis we found that, ***in many such programs, the necessary and sufficient information that the involved loops can in fact be parallelized was present in the program code and was not dependent on the input data.*** While investigating this information can be complex,

ICS '21, June 14–17, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460424

the opportunity *exists* to develop compile-time analyses to do this detection. The present contribution achieves this goal.

The idea of using subscript array properties for parallelizing loops with subscripted subscript patterns is not new. Initially McKinley [22] and more recently Mohammadi et al. [23], have successfully analyzed certain properties, but not yet succeeded in fully automating the process of detecting the properties and parallelizing the involved loops. Runtime analysis techniques, using an inspector/executor scheme [24, 30] or using speculative execution [10, 28], are capable of analyzing the access pattern of the subscript array and execute the involved loop in parallel if no cross-iteration dependencies exist. Though powerful, a major drawback of these techniques are the runtime overheads due to the generation and execution of the inspector and executor code (in the case of inspector/executor schemes) and re-executing the loop sequentially if loop-carried dependencies are detected (in the case of speculative run-time parallelization).

By contrast, our approach uses compile-time analysis, *avoiding runtime overheads.* The algorithm follows two key observations:

(1) Monotonic arrays are often generated through recurrences that create the current array element by summing the previous element and a positive value for strict monotonicity or a non-negative value for non-strict monotonicity.
(2) Positive values are often created by starting with a zero and conditionally incrementing it an arbitrary number of times.

We discuss different flavors of the above observations in Section 3.5. Our algorithm proceeds in two steps. Step one symbolically examines the effect of executing an arbitrary loop iteration on the value of the subscript array. Step two aggregates these values across the loop iterations, determining the effect of the entire loop. Key to step two is the use of an aggregation algebra that helps derive the array properties from certain recurrence relationships. To the best of our knowledge, these are the first compile–time techniques capable of parallelizing programs that exhibit subscripted subscripts without the help of run-time techniques, pattern matching or user assertions.

In summary, this paper makes the following contributions:

- We present a compile-time algorithm capable of detecting monotonicity of subscript arrays, which is sufficient for automatically parallelizing a class of programs.
- We introduce an aggregation algebra for reasoning about values and properties of subscript arrays.
- We present the performance results after applying the algorithm by hand to example loops from real scientific applications and discuss the overall impact on the performance of the applications.

The remainder of the paper is organized as follows. Section 2 presents subscript array properties. Section 3 describes the algorithm for deriving properties of arrays that show up as index arrays in to-be-parallelized loops. Section 4 describes the effectiveness of the algorithm in parallelizing example loops from real scientific applications. Section 5 presents the performance results when applying the techniques presented in Section 3. Section 6 discusses related work. Section 7 presents conclusions.

2 ANALYZING ARRAY PROPERTIES

Table 1 shows the benchmark suites we analyzed. We looked at all codes of these suites, except in SPEC CPU 2006, where we considered the seven most promising of the 17 applications. A total of 14 of the 32 inspected codes exhibited subscripted subscript patterns. Common to all these patterns is that the parallel loops contain a single or multiple array write references with subscript expressions that contain the value of another array, but there is no read reference of the written array. To parallelize the loop, the compiler must prove that there is no self output dependence [4]. In doing so the following array properties are of interest:

Benchmark suite	No. of benchmarks analyzed	No. of benchmarks with subscripted subscript patterns
NPB3.3 [3]	10	3
SuiteSparse 5.4.0 [12]	10	7
SPEC CPU 2006 [15]	7	2
The Mantevo Project [16]	5	2

Table 1: Benchmarks analyzed for subscripted subscripts.

(1) **Injectivity:** An array is said to be injective if $a[i] \neq a[j]$, $\forall i \neq j$. The array accesses $x[a[i]]$ and $x[a[j]]$ are independent, if $i \neq j$, in this case.
  (a) **Injective Expression:** In some applications, the subscript array is part of an expression, such as $a[i] + term$. The challenge for the compiler is to prove that the entire expression is injective, i.e., it differs for distinct values of i.
  (b) **Injective Subset:** In some patterns, only a section of the subscript array is injective, but the program accesses only this section. The property of interest is injectivity combined with the applicable array section.

(2) **Monotonicity:** We found three variants of interest:
  (a) **Monotonically increasing or decreasing:** An array is monotonically increasing if $a[i] \leq a[j]$, $\forall i < j$ and monotonically decreasing if $a[i] \geq a[j]$, $\forall i < j$. This implies non-strict monotonicity.
  (b) **Strictly monotonically increasing or decreasing:** An array is strictly monotonically increasing if $a[i] < a[j]$, $\forall i < j$ and strictly monotonically decreasing if $a[i] > a[j]$, $\forall i < j$. Strict monotonicity implies injectivity.
  (c) **Monotonic difference between arrays:** In this case, the difference between consecutive elements of two different subscript arrays is monotonic and the loop index of the inner loop in a nest traverses this difference. Non-strict monotonicity is sufficient for parallelization of the outer loop in this case.

(3) **Simultaneous Monotonicity and/or Injectivity:** Some benchmark applications involve loop nests with arrays consisting of multiple levels of indirection in their subscripts, such as $A[B[C[i]]]$. Both subscript arrays need to possess certain properties for the enclosing loop to be parallelizable.
  (a) **Simultaneous Monotonicity and Injectivity:** The innermost subscript array is monotonic whereas the outermost subscript array is injective and hence the outer

loop becomes parallelizable. This is observed in loop nests wherein the loop index of the inner loop (say i) traverses a monotonic subscript array (say C), whose values appear at the subscript of an injective array (say B).

(b) **Simultaneous Injectivity:** Both subscript expressions are injective. Injectivity of multiple nested subscript expressions ensures that different values of the innermost expression – typically the loop index – imply different values of the outermost expression.

(4) **Disjoint Injective Expressions:** A loop may contain two or more array references, each with a different subscript expression involving another array. Across the loop iterations, all values of all expressions are different, and hence the references are dependence free.

Typically, these array properties are created in multiple steps, with Figure 2 being a representative example. A common intermediate array property is *positive* or *non-negative*, as after Figure 2(b). Our algorithm will capture these two properties as well.

Previous work [5, 23] has mentioned example loops from benchmark applications that can be parallelized due to each of the above mentioned properties. Recall from Section 1 that we found the above properties to be *present in the programs* and not dependent on program input data. An advanced programmer would be able to determine the properties and thus parallelize the containing loops. This paper is about equipping the compiler with these advanced skills.

3 COMPILE-TIME ALGORITHM FOR SUBSCRIPT ARRAY ANALYSIS

3.1 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 with simple subscripts. For the current algorithm, "simple subscript" means index expressions of the form $i+k$, where i is the iteration number and k is a constant. The current algorithm deals with the common case of a single subscripted-subscript array write reference in a loop.

We assume that 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. Phase 1 analyzes the loop body and determines the effect of one iteration on the variable's value. Phase 2 then aggregates this expression across all iterations, determining the effect of the entire loop on the scalar or array and testing for array properties. After Phase 2, the loop is collapsed, that is, it is substituted by a set of expressions representing the effect of the loop. The algorithm then proceeds with the next outer loop.

3.2 Representation

Our representation of the value of variable v is a symbolic expression, R_v, which may contain ranges of the form $[lb : ub]$ (inclusive), where the lower bound lb and upper bound ub are symbolic expressions. Values are *may* ranges, i.e., the actual value may be any one in the range.

$x : [lb : ub]$ – (scalar) variable x has a possible value range between lb and ub at the current program point.

For arrays, the representation includes a subscript element or range. Subscript ranges are *must* ranges, i.e., the indicated value holds for all array elements in that range.

$y[sl : su] : [vl : vu]$ – all elements of array y in the index range sl to su have a value between vl and vu.

The representation makes use of several special symbols, referring to particular values of a variable being analyzed:

- λ refers to the value of a loop variant variable (LVV) at the beginning of the loop iteration being analyzed. This notation is used in Phase 1.
- Λ refers to the value of the variable at the beginning of the loop. This is useful in the aggregation step of Phase 2. Λ is also used in the expression that represents the effect of the loop after collapsing, where it refers to the value before the expression.
- $\perp$ indicates an unknown value, typically, if an expression is too complex for the compiler to analyze or represent.
- Instead of a value (range) the representation may indicate an array property, such as *Monotonic* or *Strictly_Monotonic*.

3.3 Integration in the Cetus Parallelizer

The representation is stored in a Symbolic Value Dictionary (SVD), associated with each statement and each control-flow edge of the program flow graph. The SVD is an extension of the Range Dictionary used by Cetus' Range Analysis Capability [8, 11]. While the Range Dictionary stores symbolic value information for scalars, the SVD also stores the information gathered about arrays by the new algorithm. In the overall compilation process, Range Analysis runs first, initializing the SVD. At each eligible loop nest, the new algorithm then determines symbolic values and properties about arrays, which it adds to the SVD. If the algorithm is able to gather more precise information about scalars, it will add this information to the SVD as well. The new algorithm focuses on loop-variant variables ($LVVs$). Symbolic value information about loop-invariant variables remains unmodified in the SVD, but is important for the symbolic execution that the algorithm performs at each statement of the analyzed loops. For the outermost loop in a loop nest, a final SVD is determined before the loop is collapsed, by substituting in all known values of the variables before the loop (Λ) in the aggregated value expressions.

3.4 Algorithm for Phase 1

Phase 1 determines the effect of executing an arbitrary loop iteration on the value of an LVV (integer scalars and arrays; arrays with loop-invariant subscript expressions are treated as scalars). The algorithm computes the values of $LVVs$ at the end of the loop iteration. For an array, it aims to determine the value for a single *element*, indexed by the subscript expression. The value expression may include the loop index, constants (loop-invariant symbolic terms) and $LVVs$. Our algorithm makes use of the symbolic range propagation scheme [8], which collects and propagates variable ranges through the program.

Definition 1 : Let LG be the sub-graph of the program control flow graph (G) corresponding to a `for` loop. LG consists of $control(LG)$ and $body(LG)$, representing the loop header and body, respectively. $collap(G, LG)$ is a flowgraph with LG collapsed into one node.

Figure 3 shows the algorithm for Phase 1. The algorithm begins by determining a list of all *LVVs* (V) in the loop body and initializes them to λ. The initialized values are stored in the SVD of the first statement (SVD$_{st1}$) of the loop body. To determine the *LVV's* value at the end of the loop iteration, the algorithm then traverses each statement along every path in the loop body sub-graph in program order, updating the current value to reflect the effects of symbolic execution of the statements encountered along the paths.

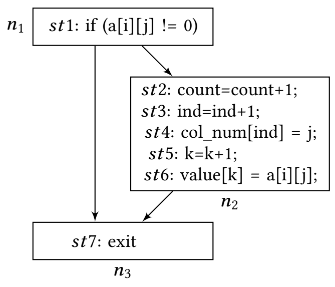
Algorithm 1: Phase 1

```
Input: Loop control flowgraph ($LG$)
Output: Symbolic expression for each LVV, representing the value
  at the end of an iteration
$body(LG) \longleftarrow LG\text{–}control(LG)$
$V \longleftarrow$ LVVs in $body(LG)$
for each $v \in V$ do
    $\text{SVD}_{st1} \longleftarrow (v : \lambda)$
for each $st \in body(LG)$ in program order do
    if ($st$ is an assignment statement) then
        $\text{SVD}_{st} \longleftarrow \cup$ (SVD of immediate predecessors of $st$)
        $v \longleftarrow$ LHS of $st$
        $R_v \longleftarrow$ RHS of $st$
        $R_v \longleftarrow evaluate(v, R_v, \text{SVD}_{st})$
        $\text{SVD}_{st} \longleftarrow (v : R_v)$
```

Figure 3: Algorithm to symbolically evaluate the effect of one loop iteration on *LVVs*.

```
1: for(i = 0; i < ROWLEN; i++){
2:  count = 0;
3:  for(j = 0; j < COLUMNLEN; j++){
4:     if(a[i][j] != 0){
5:      count++;
6:      col_num[ind++] = j;
7:     value[k++] = a[i][j];
8:    }
9:   }
10:  rowsize[i] = count;
11: }
```

(a)



(b)

Figure 4: Example code and flowgraph. (a) loop nest counting the number of non-zero elements in a sparse matrix row; (b) flow sub-graph of the normalized inner loop body – lines 4–9 of (a).

Before analyzing each statement, the incoming values from immediate predecessor nodes are merged conservatively (*may* semantics) using a union ($\cup$) operation. The SVD of the current statement (SVD$_{st}$) is initialized to these values. The value assigned to a variable, i.e. the right-hand-side (RHS) expression of the statement, is evaluated by recursively substituting in all known values of variables appearing in the expression. The substitutions are done conservatively in the subroutine *evaluate*, which retrieves the most recent value of a variable by tracing back to the SVD of a predecessor statement which holds that value. After substitution completes, the value expression is *simplified*, bringing it into a normal form (this function is part of Cetus' symbolic manipulation package, performing constant folding, partial expression evaluation, and canonical representation).

Example: For the loop shown in Figure 4(a), the algorithm determines the symbolic expression for variable *count* for an arbitrary iteration of the inner loop as follows. Before analyzing the first statement ($st1$) in the loop body subgraph in Figure 4(b), the value of count is initialized to λ_{count}. Statement $st1$ has no effect on the value of *count*. Therefore, SVD of statement $st1$ upon exit is SVD$_{st1}$ $\longleftarrow$ $(count : \lambda_{count})$. The algorithm then proceeds with the next statement. For statement $st2$, SVD of $st2$ upon entry becomes:

$$\text{SVD}_{st2} \longleftarrow \cup\,(\text{SVD}_{st1})$$
$$\text{SVD}_{st2} \longleftarrow (count : \lambda_{count})$$

Values for other variables have not been shown. Processing the statement `count = count + 1`, increments the value of *count*, resulting in $R_{count} \longleftarrow (\lambda_{count} + 1)$ and SVD$_{st2}$ $\longleftarrow (count : \lambda_{count} + 1)$ upon exiting statement $st2$. At the beginning of statement $st7$, the union operation determines the combined effect of statements $st1$ and $st6$ on the value of *count* as:

$$\text{SVD}_{st7} \longleftarrow \cup\,(\text{SVD}_{st1}, \text{SVD}_{st6})$$
$$\text{SVD}_{st7} \longleftarrow (count : [\lambda_{count} : \lambda_{count} + 1])$$

Note that the SVD of statement 6 (SVD$_{st6}$) will have the value $(count : \lambda_{count} + 1)$ upon exit, as none of the statements from $st3$ to $st6$ modify the value of count. Therefore, the value of *count* at the end of Phase 1 is:

$$R_{count} \longleftarrow [\lambda_{count} : \lambda_{count} + 1]$$

which is equivalent to:

$$R_{count} \longleftarrow \lambda_{count} + [0 : 1]$$

3.5 Phase 2 Concepts for Proving Monotonicity

In our algorithm, Phase 1 captures the effect of one loop iteration on the value of *LVVs*. It represents this value as a symbolic range expression. Phase 2 – the aggregation phase – extends that value to the full loop iteration space. Aggregation reasons about mathematical relationships of the value of variables at the end of the iteration versus the value at the beginning. Recurrence relationships are of particular interest. We begin by introducing the concepts behind Phase 2. Then we describe the algorithm, which we illustrate with a number of examples.

Our method aims to recognize the following three recurrence classes – Simple Scalar Recurrence (SSR), Scalar Recurrence Array Assignments (SRA) and Array Recurrence. All these classes are flavors of the key observations mentioned in Section 1, where the next value in a sequence is computed from the previous value plus an increment known to be positive or non-negative (*PNN*). Such recurrences create properties of monotonicity or, in simpler cases, *PNN* properties. We use *PNN* as a placeholder for either a Positive (*P*) or Non-negative (*NN*) term in an expression. A sequence where all elements differ by an *NN* term is monotonic; if all differences are *P*, the sequence is strictly monotonic.

Recall that we consider a normalized loop with iteration variable *i*, lower bound 0, and *N* iterations. λ_x and Λ_x refer to the values of variable x at the beginning of the loop iteration and at the beginning of the loop, respectively.

(1) Recurrence Class 1 (Simple Scalar Recurrence, SSR): recognizes the form $sc = sc + k$, where k is a loop–invariant *PNN* value or value range. Phase-2 computes the aggregated value of *sc* as Λ_{sc} + *N*∗*k*. For example, a range $k = [lb : ub]$ leads to the expression Λ_{sc} +*N*∗$[lb : ub]$, which is equivalent to Λ_{sc} + [*N*∗*lb* : *N*∗*ub*][1]. Here $[lb : ub]$ is called the *increment expression*. In many cases, only the final value after the loop matters. For other cases, it is useful to remember that this value was reached through a sequence of monotonic steps. We also remember that the final value is *PNN*, if Λ_{sc} is *PNN*.

(2) Recurrence Class 2 (Scalar Recurrence Array Assignments, SRA): deals with the form $ar[i] = SSR\_expression$. An SSR expression is assigned to the current array element. The SSR expression can be an SSR variable plus a *PNN* term, where the *PNN* term is either Positive or Non-negative.

The MRA notation facilitates the aggregation of piece-wise monotonic array subsections of Class-2 recurrences:

Definition 2: Monotonic Range Assignment (MRA) Given a monotonic scalar k that has a value in the range $[lb : ub]$, the array assignment $ax[sl : su] = k\#MRA$ means that the array ax, in the index range $[sl : su]$, gets assigned values of k in a way that is monotonic, i.e. $ax[i] \leq ax[i+1]$. If k is *strictly monotonic*, $ax[sl : su] = k\#SMRA$ and $ax[i] < ax[i+1]$.

LEMMA 1: *Given a loop that makes the following assignments in iteration 'i':*
- (scalar) k = λ_k+$[lb : ub]$*, where* $[lb : ub]$ *is Non-negative(NN)*
- (array) ax[*range(i)*] = *k#MRA*

then, if the index ranges are consecutive, i.e., range(i) < range(i+1), and contiguous, ax overall (i.e. after the loop) will be monotonic.

Proof: The value range for array ax in each iteration is monotonic (by MRA definition). We need to prove that they do not overlap. Because the ranges are monotonic, it suffices to show that the last element of $ax[range(i)]$ is less than or equal to the first element of $ax[range(i+1)]$. This is indeed the case because k is monotonically increasing from iteration i to $i+1$, per the first statement.

Similarly, one can prove strict monotonicity for array ax, given k is strictly monotonic. A minor extension of this class is the assignment to an array element $ar[i+c]$, where c may be -1 or +1.

[1]See [8] for algebraic rules for range expressions.

After aggregation, per LEMMA 1, the array range $ar[0 : N{-}1]$ is known to be monotonic, with values in the range Λ_{sc} + [*N*∗*lb* : *N*∗*ub*]+*constant*, where *constant* is a loop–invariant *PNN* value, referring to the terms used in Class 1. Using Definition 2, we can also express the same as:

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

This notation retains the fact that the values of *ar* were assigned from variable *sc*; this fact is lost in Class 1, but is needed in some cases. Substituting the aggregated value of *sc* in equation (1) results in:

$$ar[0 : N-1] = \begin{cases} (\Lambda_{sc} + [N * lb : N * ub])\#MRA + constant, \\ (\Lambda_{sc} + [N * lb : N * ub])\#SMRA + constant, \end{cases} \quad (2)$$

If Λ_{sc} is *PNN*, then $ar[0 : N-1]$ is *PNN* as well.

LEMMA 2: *Given a stride–1 loop with iteration variable 's' and range [0 : N-1] and given an array Y: Let Y[s′] be the value of Y at the end of an iteration s′ and Y[s″] be the value of Y at the end of the immediately preceding iteration s″. If Y[s′] ≥ Y[s″], then elements of array Y in the subscript range [0:N-1] assume monotonic values. If Y[s′] > Y[s″], then Y[0 : N-1] is strictly monotonic.*

LEMMA 3: *Given a loop and scalar LVV X; let* X_i *be the value of X at the end of any loop iteration i, where* $i \in [0 : N-1]$*. If* $X_i \geq \lambda_X$*, then X assumes monotonic values across iterations of the loop. If* $X_i > \lambda_X$*, then X is Strictly Monotonic.*

(3) Recurrence Class 3: (Array recurrence) of the form $ar[i] = ar[i-1]$ + *PNN* term. Class 3 is similar to Class 2, but the recurrence is built directly in the array, rather than via a scalar variable. We will handle the same minor extension as in Class 2, e.g., $ar[i+1] = ar[i]$ + *PNN* term. The recurrence distance (the difference of the LHS and RHS array subscript) must be 1, in all cases.

A special case of a *PNN* value is in the Class 3 recurrence $ar[i] = ar[i-1] + ar[i]$, where the entire array *ar* has *PNN* property before the loop. Note that the LHS and $ar[i-1]$ terms are loop variant, while the RHS $ar[i]$ term is loop-invariant, referring to the incoming *PNN* values. After aggregation, the array range $ar[0 : N-1]$ can be:

$$ar[0 : N{-}1] = \begin{cases} Strictly\_Monotonic, & \text{if } PNN \text{ term is } Positive \\ Monotonic, & \text{if } PNN \text{ term is } Non\text{-}negative \\ \perp, & \text{otherwise} \end{cases} \quad (3)$$

In addition to these three classes of recurrences, we have found the following non-recurrent expression in our applications: $ar[i + k] = constant$, where *constant* refers to a constant range expression. Here, the result of aggregation for *ar* is $ar[k : N+k-1] = constant$.

3.6 Algorithm for Phase 2

Figure 5 shows the algorithm for Phase 2. We use elements of the data flow framework presented by Tu and Padua [31] in their technique for automatic array privatization.

Algorithm 2: Phase 2

```
Input:  1. Loop control flowgraph ($LG$)
        2. SVD of the final statement in $LG$ (SVD$_{stn}$) after
           Phase 1
        3. Loop information ($Idx$, $N$, $LIR$)
Output: 1. Aggregated symbolic expression for each $LVV$
        2. Node representing the collapsed Loop
           flowgraph ($collap(G, LG)$)

SVD$_{stn}$ $\longleftarrow$ $determine\_eval\_order$(SVD$_{stn}$)
Add ($Idx : LIR$) to SVD$_{stn}$
$List\_SSR\_vars \longleftarrow (Idx, Strictly\_Monotonic)$

for each $(v, R_v) \in SVD_{stn}$ do
    $(class\_type, expr) \longleftarrow$
      $identify\_recurr\_class(v, R_v, List\_SSR\_vars, Idx)$
    switch ($class\_type$) do
        case Class 1: do
            $R_v \longleftarrow (\Lambda_v + N * expr)$
            $List\_SSR\_vars \longleftarrow v$
        case Class 2: do
            $ssr\_var \longleftarrow Find\_SSR\_var(R_v, List\_SSR\_vars)$
            if ($ssr\_var$ is $Strictly\_Monotonic$) then
                $R_v \longleftarrow expr\#SMRA$
            else
                $R_v \longleftarrow expr\#MRA$
            $R_v \longleftarrow evaluate(ssr\_var, R_v,$ SVD$_{stn})$
        case Class 3: do
            $R_v \longleftarrow eval\_PNN(expr)$
        otherwise do
            $R_v \longleftarrow evaluate(v, R_v,$ SVD$_{stn})$
            if ($R_v$ is not constant) then
                $R_v \longleftarrow \perp$
    if ($v$ is an array) then
        $s \longleftarrow$ Subscript expression of $v$
        $s \longleftarrow determine\_index\_range(s,$ SVD$_{stn})$
        if ($s$ is $\perp$) then
            $R_v \longleftarrow R_v \cup$ ($R_v$ before $LG$)
    SVD$_{stn}$ $\longleftarrow (v : R_v)$
Determine final SVD$_{stn}$ if $LG$ is outermost
for each $(v, R_v) \in SVD_{stn}$ do
    $collap(G, LG) \longleftarrow \{v = R_v\}$
Replace $LG$ with $collap(G, LG)$
```

Figure 5: Algorithm to determine aggregated symbolic expressions (overall effect of the loop) for $LVVs$.

The algorithm has two parts – (a) Identifying the recurrence class of an LVV and (b) determining its aggregated symbolic expression. The key input to the algorithm is the result of Phase 1, represented by the SVD of the final statement in the loop body (SVD$_{stn}$), which includes the symbolic expression of each LVV (R_v). The loop flowgraph (LG), loop index variable (Idx), loop iteration count (N) and loop index range (LIR) are the remaining inputs. Subroutine *determine_eval_order* sets an order in which symbolic expressions of variables are to be processed. For each LVV v_1 in SVD$_{stn}$, if R_{v1} contains another LVV v_2, then R_{v2} is processed first.

Algorithm 3 in Figure 6 determines the recurrence class to which R_v belongs. A scalar variable v can have a Class 1 recurrence (SSR) relationship. The algorithm symbolically subtracts from R_v the value of v at the beginning of the loop iteration (λ_v). If the resulting expression (*inc_expr*) is *PNN*, then v is an SSR variable and the aggregated value of v is $\Lambda_v + N * inc\_expr$ (shown under **case** *Class 1* in Figure 5). Variable v is then added to the list of SSR variables (*List_SSR_vars*). The loop index variable is a special case of an SSR variable and is known to be strictly monotonic.

Algorithm 3: *identify_recurr_class*

```
Input:  1. An $LVV$ ($v$)
        2. Value for $v$ after Phase 1 ($R_v$)
        3. List of SSR variables in the loop ($List\_SSR\_vars$)
        4. Loop index ($Idx$)
Output: 1. Recurrence class of $v$
        2. An Expression

if ($v$ is a scalar) then
    $inc\_expr \longleftarrow R_v - \lambda_v$
    if ($is\_PNN(inc\_expr)$) then
        return (Class 1, $inc\_expr$)          ▷ increment expression
    else
        return ($\perp$, $\perp$)
else
    $s \longleftarrow$ Subscript expression of $v$
    $ssr\_var \longleftarrow Find\_SSR\_var(R_v, List\_SSR\_vars)$
    if (($ssr\_var$) && $is\_simple\_subscript(s, Idx)$) then
        $remainder \longleftarrow R_v - ssr\_var$
        if ($is\_PNN(remainder)$) then
            return (Class 2, $R_v$)           ▷ returns SSR expression
    else if (($R_v$ contains $v[s-1]$)&&
      $is\_simple\_subscript(s, Idx)$) then
        $E \longleftarrow R_v - v[s-1]$
        if ($is\_PNN(E)$) then
            return (Class 3, $E$)             ▷ returns PNN term
    else
        return ($\perp$, $\perp$)
    return ($\perp$, $\perp$)
```

Figure 6: Algorithm to recognize the recurrence class of an LVV.

Array recurrence relationships can belong to either Class 2 or Class 3. For Class 2 recurrences, Algorithm 3 determines if the value of an SSR expression is assigned to an array element. As described in Section 3.5, an SSR expression has the form of an SSR variable plus a *PNN* term. Subroutine *Find_SSR_var* returns the SSR variable (*ssr_var*) in R_v. Subroutine *is_simple_subscript* checks if an array subscript expression is a simple subscript as described in Section 3.1. If an SSR variable exists and if *is_simple_subscript* returns true, then the SSR variable is symbolically subtracted from R_v and the resulting expression must be *PNN*. In this case, the aggregated value expression is *ssr_var#SMRA+ PNN* or *ssr_var#MRA+PNN*, depending on the property (strict or non-strict monotonicity) of *ssr_var* (shown under **case** *Class 2* in Figure 5). Subroutine *evaluate* substitutes in the range expression for *ssr_var*, so that the aggregated expression is of the form: $\Lambda_{ssr\_var}+[N*lb : N*ub]\#SMRA+ constant$ or $\Lambda_{ssr\_var}+[N*lb : N*ub]\#MRA+constant$ as shown in equation (2) in Section 3.5.

To identify a Class 3 recurrence on array v, Algorithm 3 symbolically subtracts $v[s-1]$ from R_v, where s is the subscript expression of array v, and is a simple subscript. The resulting expression must be *PNN*. Subroutine *eval_PNN* (shown under **case** *Class 3* in Figure 5) evaluates the range expression of the *PNN* term i.e. determines if the *PNN* term is *Positive* or *Non-negative*, and returns the corresponding property. Subroutine *determine_index_range* aggregates array subscript expressions of the form $i + k$, where i is the loop index and k is a loop-invariant scalar, by replacing i and k with their known symbolic ranges. For all other types of subscript expressions, the algorithm returns $\perp$.

Our algorithm is capable of handling all recurrence relationships described in Section 3.5. If an R_v does not belong to any of the recurrence classes and R_v is not constant, then the aggregated value of R_v is $\perp$. After Phase 2, the loop is collapsed and replaced by a single node (*collap*(*G, LG*)) containing a sequence of assignment statements, representing the effect of the loop on each *LVV*. The RHS of the statements are the aggregated value expressions, corresponding to each *LVV* on the left-hand side (LHS).

4 EXAMPLES

We demonstrate the effectiveness of our techniques by applying them to example loops that fill and modify the subscript array from three real scientific applications: Sparse Matrix Scaling [12, 18], GROMACS [32] (from the SPEC CPU 2006 [15] benchmark suite) and Supernodal Cholesky Factorization [9]. Upon determining the required properties, key loops in these applications where the subscript array is used can be parallelized. The applications represent an important class of sparse computations.

4.1 Example 1 - from Sparse Matrix Scaling

The outermost j–loop (on line 2) of the loop nest shown in Figure 7 can be parallelized, if array *Ap* whose values appear at the subscript of array *Ax* on line 6 is *Monotonic*. Array *Ap* is defined in the loop nest shown in Figure 8.

```
1 #pragma omp parallel for private(j, k)
2 for(j=0; j<ncol; j++)
3 {
4     for(k=Ap[j]; k<Ap[j+1]; k++)
5     {
6         Ax[k]=s[j]*Ax[k]*s[Ai[k]];
7     }
8 }
```

Figure 7: Example loop to parallelize. This loop performs symmetric scaling of a sparse matrix A by a scaling factor s, where s is a diagonal vector.

The analysis begins with the inner loop on line 5 of Figure 8. For this loop, the algorithm determines for scalar variable p:

Phase 1(loop on line 5): $(p : [\lambda_p : \lambda_p + 1])$

Values for other variables have not been shown. After aggregation, the value of p becomes:

Phase 2(loop on line 5): $(p : [\Lambda_p : \Lambda_p + nrow])$

Note that Λ_p in the above expression is the value of p at the beginning of the inner loop on line 5. This loop is then collapsed and replaced with the aggregated expressions of its *LVVs*, including the expression for p. For the outermost j–loop on line 2, the Phase 1 algorithm determines for *Ap* and p respectively:

Phase 1(loop on line 2): $(Ap[j] : \lambda_p)$, $(p : [\lambda_p : \lambda_p + nrow])$

```
1  p = 0;
2  for(j=0; j<ncol; j++)
3  {
4     Ap[j] = p;
5     for(i=0; i<nrow; i++)
6     {
7        if(Xx[i+j*d]!=0.)
8        {
9           Ai[p]=i;
10          if(values)
11          {
12            Ax[p]=Xx[i+j*d];
13          }
14          p++;
15       }
16    }
17 }
```

Figure 8: Loop that initializes the value of subscript array *Ap*. An element of *Ap* is assigned the value of p on line 4 in an iteration of the j–loop. Array *Ap* is filled with monotonically increasing values in the j–loop.

λ_p now represents the value of p at the beginning of an iteration of the j–loop. Since the expression for array *Ap* after Phase 1 contains the value of an *LVV* i.e. p, p is evaluated before *Ap* in Phase 2. The Phase 2 algorithm determines that p is an SSR variable. Therefore, the recurrence relationship for array *Ap* is of type Class 2. The aggregated expressions for p and *Ap* are as follows:

Phase 2 (loop on line 2):
$(p: [\Lambda_p : \Lambda_p + nrow * ncol])$, $(Ap[0 : ncol - 1]:[\Lambda_p : \Lambda_p + nrow * ncol]\#MRA)$

Before collapsing the j–loop, final values for p and *Ap* are determined by substituting in the value for Λ_p (value of p before the loop) i.e. $\Lambda_p = 0$ in the Phase 2 expressions above to give:

Phase 2 (loop on line 2):
$(p : [0 : nrow * ncol])$, $(Ap[0 : ncol - 1] : [0 : nrow * ncol]\#MRA)$

Therefore, consecutive elements of array *Ap* from $Ap[0]$ to $Ap[ncol - 1]$ are assigned monotonically increasing values in the range $[0 : nrow*ncol]$. This enables parallelization of the outermost loop in Figure 7.

4.2 Example 2 - from GROMACS

The outer loop on line 2 of the loop nest shown in Figure 9 can be parallelized, if array *blnr* whose values appear at the subscript of array *blm* on line 10 is *Monotonic*. The array *blnr* is defined in the loop nest shown in Figure 10. The analysis begins with the inner loops, on lines 6 and 10, in that order.

For each of these loops, the Phase 1 algorithm determines that $blnr[i+1]$ is a scalar LVV. For the loop on line 6, the algorithm determines for array $blnr$:

Phase 1(loop on line 6) : $(blnr[i+1] : \lambda_{blnr[i+1]} + [0:1])$
Phase 2(loop on line 6) : $(blnr[i+1] : \Lambda_{blnr[i+1]} + [0 : X[a1 - start]])$

```
1  #pragma omp parallel for private (b, n, tmp0, tmp1, tmp2, i,j, k,mvb)
2  for(b=0; b<ncons; b++){
3    tmp0 = r[b][0];
4    tmp1 = r[b][1];
5    tmp2 = r[b][2];
6    i = bla1[b];
7    j = bla2[b];
8    for(n=blnr[b]; n<blnr[b+1]; n++){
9     k = Z[n];
10    blm[n]= blcc[n]*(tmp0*r[k][0]+tmp1*r[k][1]+tmp2*r[k
          ][2]);
11   }
12   mvb=q[b]*(tmp0*(xp[i][0]-xp[j][0])+
13          tmp1*(xp[i][1]-xp[j][1])+
14          tmp2*(xp[i][2]-xp[j][2])-bllen[b]);
15   rhs1[b]=mvb;
16   sol[b]=mvb;
17 }
```

Figure 9: Loop to parallelize in GROMACS subroutine *clincs*.

The aggregated expression above results due to variable $blnr[i+1]$ being an SSR variable and is applicable for all positive values of $X[a1 - start]$. The value of $blnr[i+1]$ is further incremented in the loop on line 10. Applying the algorithm to this loop yields:

Phase 1(loop on line 10) : $(blnr[i+1] : \lambda_{blnr[i+1]} + [0:1])$
Phase 2(loop on line 10) : $(blnr[i+1] : \Lambda_{blnr[i+1]} + [0 : X[a2 - start]]$

```
1  blnr[0] = 0;
2  for(i=0; i<ncons; i++){
3     a1 = iatom[3*i+1];
4     a2 = iatom[3*i+2];
5     blnr[i+1] = blnr[i];
6     for(k=0; k<X[a1-start]; k++){
7        if(Y[a1-start][k] != i)
8           Z[blnr[i+1]++] = Y[a1-start][k];
9     }
10    for(k=0; k<X[a2-start]; k++){
11       if(Y[a2-start][k] != i)
12          Z[blnr[i+1]++] = Y[a2-start][k];
13    }
14 }
```

Figure 10: Simplified version of the code that fills in the subscript array *blnr* from GROMACS.

Note that the values $X[a1 - start]$ and $X[a2 - start]$ in the aggregated expressions above are the upper bounds of the loops on lines 6 and 10 respectively. For the outermost loop on line 2, Phase 1 determines for $blnr$:

Phase 1(loop on line 2) :
$(blnr[i+1] : \lambda_{blnr[i]} + [0 : X[iatom[3*i+1] - start] + X[iatom[3*i+2] - start]])$

The Phase 1 symbolic expression for array $blnr$ above is a Class 3 recurrence relation. Therefore, Phase 2 yields:

Phase 2 (loop on line 2) : $(blnr[1 : ncons] : Monotonic)$

The outer loop on line 2 of Figure 9 can now be parallelized. The reader may notice a subtlety for this loop: the first iteration is a special case, as the values of elements of array $blnr$ in the subscript range $[1 : ncons]$ are *Monotonic* as determined by our algorithm. Such cases could be handled by peeling the iteration. However, symbolic analysis can do better and prove that there is no overlap even in the first iteration.

4.3 Example 3 - from Cholesky Factorization

In the loop nest shown in Figure 11, the subscript expression of array Lx is $2*q$ on line 14 and $2*q+1$ on line 15, where q is $Map[i] + psx + (k - k1) * nsrow$. The outermost k–loop can be parallelized, if the maximum value of the expression $2*q+1$ in any loop iteration k is less than the minimum value of the expression $2*q$ in the next iteration i.e. $k+1$, so that the values appearing at the subscript of array Lx on lines 14–15 do not overlap across iterations of the loop.

```
1  #pragma omp parallel for private (p, pf, i, k, q, fjk)
2  for(k=k1; k<k2; k++)
3  {
4    for(pf=Fp[k]; pf<Fp[k+1]; pf++)
5    {
6       fjk[0] = Fx[2*pf];
7       fjk[1] = Fx[2*pf+1];
8       for(p=Ap[Fi[pf]]; p<Ap[Fi[pf]+1]; p++)
9       {
10         i = Ai[p];
11         if(i>=k && Map[i]>=0)
12         {
13            q = (Map[i]+psx+(k-k1)*nsrow);
14            Lx[2*q] += Ax[2*p]*fjk[0]-Ax[2*p+1]*fjk[1];
15            Lx[2*q+1] += Ax[2*p+1]*fjk[0]-Ax[2*p]*fjk[1];
16         }
17      }
18   }
19 }
```

Figure 11: Example loop to parallelize in the subroutine *cholmod_super_numeric* from the Supernodal module in CHOLMOD from the SuiteSparse benchmark suite [12].

In order for this condition to be satisfied, the values of $Map[i]$ appearing in the subscript expression of Lx must be in the range $[0 : nsrow - 1]$. As we will see, the values of Map are indeed in this range, except for some elements that are negative. The statement on line 11 makes sure that these elements will not get used.

```
1: for(i = 0; i < n; i++)
2: {
3:   Map[i] = -1;
4: }
```

(a)

```
1: for(k = 0; k < nsrow; k++)
2: {
3:   Map[Ls[psi + k]] = k;
4: }
```

(b)

Figure 12: Loops from the subroutine *cholmod_super_numeric* that initialize and modify the content of array *Map*.

The loop shown in Figure 12(a), initializes the value of array Map to -1. For this loop, the Phase 1 and Phase 2 algorithms determine:

Phase 1 (Loop in Fig. 12(a)): $(Map[i] : [-1 : -1])$
Phase 2 (Loop in Fig. 12(a)): $(Map[0 : n-1] : [-1 : -1])$

Array *Map* is further modified in the loop shown in Figure 12(b). The Phase 1 algorithm determines for this loop:

Phase 1 (Loop in Fig. 12(b)): $(Map[Ls[psi + k]] : \lambda_k)$

The subscript expression of *Map*, $Ls[psi + k]$ is not a known term ($\perp$). Hence, *Map* does not belong to any of the recurrence classes described in Section 3.5. Phase 2 yields:

Phase 2 (Loop in Fig. 12(b)): $(Map[\perp] : [-1 : nsrow - 1])$

Since the subscript expression of *Map* is $\perp$, the union operation in the Phase 2 algorithm (in Figure 5), merges the values $[0 : nsrow - 1]$ and $[-1 : -1]$, resulting in the above expression. Since array *Map* is only partially modified after being initialized, the union operation accurately captures the combined effect of both the loops shown in Figure 12 on the value of array *Map*. Therefore, the upper bound of the range of values for elements of array *Map*, as determined by our algorithm is $nsrow - 1$. This enables parallelization of the outermost k−loop of the loop nest shown in Figure 11.

For the examples discussed above, the technique of Section 3 finds a *monotonically increasing* property of the subscript array. Based on this information, the data dependence test can prove non-overlap of the iteration ranges of the to-be-parallelized loop. Current Range Analysis [8] in Cetus can determine range expressions for scalar *LVVs*. To implement our algorithm, this capability will need to be extended for array variables. In addition we are extending the Range Test [7], which symbolically computes the array ranges being accessed in the iterations of the loop being analyzed, and then tests if these ranges overlap. If they do not, the iterations are independent and thus the loop can be parallelized.

```
1   From - UA benchmark( NPB 3.3 ):
2   #pragma omp parallel for private (j, iel, ntemp, mielnew)
3   for (j = 0; j < nelt; j++) {
4       iel = aid[j];
5       if (ich[iel] == 4) {
6          ntemp = (front[j]-1)*7;
7          mielnew = j + ntemp;
8       } else {
9          ntemp = front[j]*7;
10         mielnew = j + ntemp;
11      }
12      mt_to_id[mielnew] = iel;
13      ref_front_id[iel] = nelt + ntemp;
14  }
```

Figure 13: In this example, two subscript expressions can appear at the subscript of array *mt_to_id* on line 12. The two expressions, one on line 7 where *ntemp* is $(front[j] - 1) * 7$ and on line 10 where *ntemp* is $front[j] * 7$ produce sets of values which are strictly monotonic as well as mutually exclusive. Hence unique values appear at the subscript of *mt_to_id* on line 12 in every loop iteration. In addition, array *aid* on line 4 is injective.

The key step in which the extended Range Test looks for overlap of the subscript expression ranges is a comparison of the range accessed in an iteration i and the successor iteration $i + 1$. Such comparisons can be performed by the symbolic analysis techniques in Cetus if the subscript expressions contain scalar variables. Extensions to the symbolic analysis capabilities in Cetus to handle array subscript expressions and prove non-overlap will be discussed in a forthcoming contribution. Furthermore, we will be discussing advanced symbolic analyses required to analyze and automatically parallelize more complex subscripted subscript patterns such as the j−loop shown in Figure 13.

5 PERFORMANCE RESULTS

5.1 Experimental Setup

We have applied the array analysis techniques presented in Section 3 to the Numerical Supernodal Sparse Cholesky Factorization and the Symmetric Sparse Matrix Scaling codes from the CHOLMOD package of the latest SuiteSparse benchmark suite v5.4.0 [12] by hand. The Cholesky factorization code contains two loops with subscripted subscript patterns. The loop of Figure 11 is one of them. The code also spends time calling BLAS [6] and LAPACK [2] functions. The Sparse Matrix Scaling application spends substantial time in I/O operations, such as reading in the input sparse matrix, in addition to the actual computation. We report the performance of the actual computation, which includes the loop shown in Figure 7.

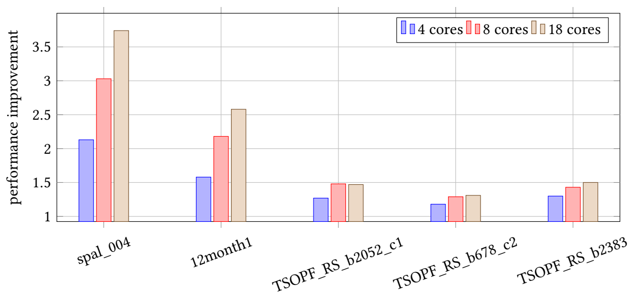
Input Matrix	Serial execution time of the application	Serial execution time of the subscripted subscript loop parallelizable using aggregation (time/%)	Serial execution time of the BLAS and LAPACK routines (time/%)
spal_004	20.26 s	10.1 s/49.85%	10.16 s/50.15%
12month1	28.64 s	11.22 s/39.17%	17.32 s/60.47%
TSOPF_RS_b2052_c1	96.72 s	22.2 s/22.95%	47.85 s/49.47%
TSOPF_RS_b678_c2	287.04 s	45.99 s/16.02%	159.97 s/55.73%
TSOPF_RS_b2383	372.91 s	98.83 s/26.5%	182.57 s/48.95%

Table 2: Serial execution time of the Supernodal Sparse Cholesky factorization application, showing overall time and time of the parallelizable parts.

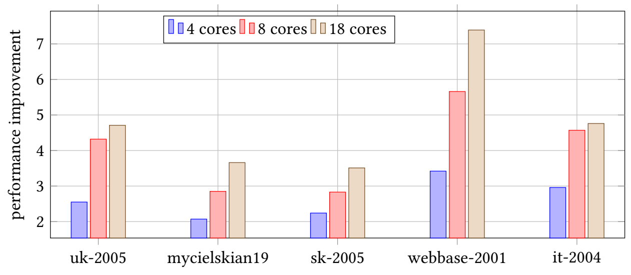
We used ten non-symmetric sparse matrices from the University of Florida Sparse Matrix collection [13] as inputs for our experiments. The number of non-zero elements in these matrices ranges between 7.3e-6% to 1.4%. Table 2 shows the serial execution time taken by the Cholesky factorization application for the five input matrices. Each of these matrices satisfy the constraints on the dimensions for an input sparse matrix mentioned in the application code. The susbscripted subscript loop parallelizable using our technique plus the BLAS and LAPACK routines take between 71.75–100% of the total application execution time. The remaining time is taken up by another loop that also exhibits subscripted subscripts, but is not yet parallelizable using our technique. The execution times for both applications were recorded on a compute node with an 18 core intel E5-2695v4 (broadwell) processors in a dual socket configuration, with a processor base frequency of 2.1 GHz, 45MB cache and we used upto 128GB of DDR4 memory. We compiled the application code using GCC v4.8.5 with the -O3 optimization flag enabled on CentOS v7.4.1708 and we report the mean of 10 application runs. We use one thread per core.

5.2 Results

Figures 14(a) and 14(b) show the performance results of the Supernodal Cholesky factorization and the Sparse Matrix Scaling application codes for the input matrices. Performance improvement is defined as the execution time without versus with the key loops of Figure 11 and Figure 7 parallel, the latter being enabled by our technique. The figures show the performance on 4, 8 and 18 cores. Our technique leads to an overall application performance improvement of up to 383% for the Supernodal Cholesky factorization code and upto 739% for the computational part of the Sparse Matrix Scaling code.



(a) Improvement in performance of the Supernodal Cholesky factorization application observed after applying our technique.



(b) Improvement in performance of the computational part of the Sparse Matrix Scaling code observed after applying our technique.

Figure 14: Performance results for the parallel applications.

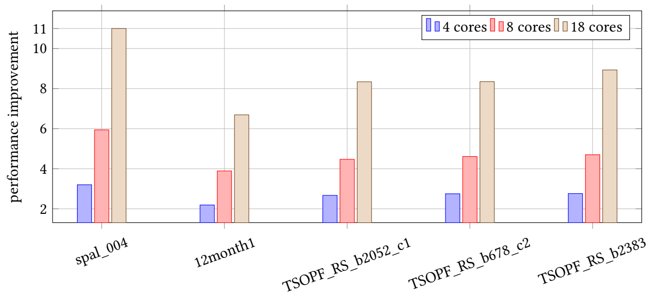


Figure 15: Performance of the parallelized k–loop in Figure 11.

Figure 15 presents the performance of the parallel version of the loop shown in Figure 11. The maximum parallelization efficiency achieved for this loop is 80% on 4 cores, 74.25% on 8 cores and 61.11% on 18 cores. For the loop in Figure 7, we achieved a maximum efficiency of 85.5% on 4 cores, 70.75% on 8 cores and 41.05% on 18 cores. The reduction in the efficiency of the loops is in part due to imbalanced assignment of loop iterations to threads, resulting from the sparsity pattern. This result can be improved by dynamic loop scheduling, as shown in Figure 16, increasing the parallel efficiency by 3.75% on 4 cores, 5.5% on 8 cores and 5.55% on 18 cores. Also, the maximum performance achieved for the application increases by 9% from 374% to 383%.

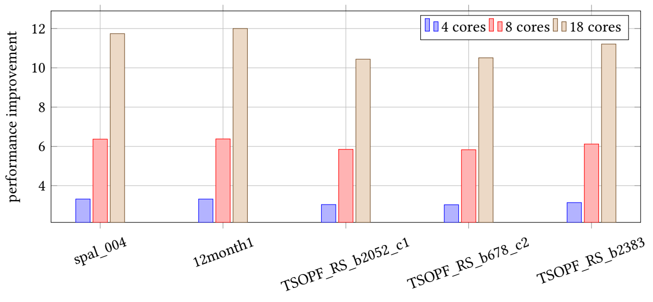


Figure 16: Performance of the parallelized k–loop in Figure 11 with dynamic scheduling enabled.

For two of the used input matrices (TSOPF_RS_b678_c2 and TSOPF_RS_b2383), parallelizing the remaining, serial loop would further improve the performance of the Supernodal Cholesky factorization application by up to 75%. We will discuss extensions to our algorithm to analyze the involved complex subscripted subscript patterns and automatically parallelize this loop, in a future contribution.

The key evaluation result, for this paper, is that the parallel execution of loops with subscripted subscripts, which our new technique enables, can yield significant, overall application performance.

6 RELATED WORK

The importance of monotonicity for subscript array analysis was recognized early on by McKinley [22]. Gutierrez et al. [14] presented simple run-time techniques to detect monotonicity of the subscript array and parallelize loops where the subscript array is used in applications such as sparse matrix computations and irregular reductions. Static analysis techniques for identifying loop monotonic statements presented by Spezialetti and Gupta [29] are capable of detecting monotonicity for scalar variables. The techniques however are insufficient for detecting monotonicity of subscript arrays at compile-time. Lin and Padua [19–21] presented a compile-time technique to analyze the content of index arrays and automatically parallelize loops. They used a form of demand-driven, interprocedural query propagation to analyze array properties, such as injectivity, closed-form distance, closed-form values and closed-form bounds. Their technique has limitations on the type of loops that can be analyzed to detect important properties. For example, their technique could determine injectivity of the subscript array only in index gathering loops, wherein the subscript array is assigned the values of the loop index variable. They also use pattern matching to detect properties such as closed form distance in two specific loop patterns. Their technique is insufficient to reason about subscript array properties in complex loops, such as the loops with recurrence relationships presented in sections 4.1 and 4.2. By contrast, our technique makes use of symbolic range aggregation and manipulation to analyze general classes of loops that define subscript array values. From this information, the technique derives subscript array properties that are sufficient for eventual parallelization.

Hybrid analysis techniques proposed by Oancea and Rauchwerger [25] make use of static interprocedural and run-time techniques to analyze memory access patterns for proving monotonicity of subscript expressions. Their techniques can prove independence of loops at compile-time by generating and analyzing predicates that represent sufficient conditions for parallelization. Predicates that are too complex to analyze at compile-time are evaluated during program execution. They could prove monotonicity for subscript arrays where, the value of the subscript array can be expressed in the form of a closed form expression at compile-time. Their techniques do not suffice in detecting monotonicity for the subscript arrays in the examples discussed in Section 4. Work related to our analysis methods include abstract interpretation and loop aggregation methods. Ammarguellat and Harrison [1] presented a method for automatically recognizing recurrence relations in loops at compile-time. Their method uses abstract interpretation to summarize the net effect of the loop body upon each variable assigned in the loop, referred to as the *abstract store*. The presence of a recurrence relation is determined by matching the *abstract store* with one of nine pre-defined recurrence templates. Their techniques can recognize recurrence relations of type Class 1 and Class 3 described in Section 3.5, but are insufficient in recognizing complex recurrence expressions such as the Class 2 recurrence.

Our algorithm also builds on a method applied by Tu and Padua in their array privatization technique [31] to analyze array sections that are defined and used. This method contrasts with abstract interpretation and is able to precisely capture the effect of recurrence relationships, which allows us to gather such array properties as monotonicity. Understanding these effects is the key to extending the work by Lin and Padua, mentioned above, for successfully recognizing parallel loops.

7 CONCLUSIONS

We have presented a novel compile-time analysis method for subscripted subscripts, which can symbolically analyze the content of subscript arrays to successfully parallelize an important class of programs exhibiting sparse matrix patterns. The method finds that certain arrays that are used in the subscripts of other arrays are monotonic. This property provides sufficient information to data-dependence analysis, allowing it to detect that the enclosing loops are parallel. The Supernodal Cholesky factorization and Sparse Matrix Scaling codes from the latest version of the SuiteSparse benchmark suite represent this class of patterns. Applying the techniques by hand to these codes, yields a parallel program that improves the performance by as much as 383% for the Supernodal Cholesky factorization code and 739% for the Sparse Matrix Scaling code, compared to the best alternative, demonstrating that the subscript patterns show up in key program sections. Our techniques are the first compile-time only techniques that can gather the requisite information for automatically parallelizing loops containing subscripted subscript patterns without the need for user assertions or pattern matching.

REFERENCES

[1] Zahira Ammarguellat and Williams Ludwell Harrison III. 1990. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation.* 283–295.

[2] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK Users' guide.* SIAM.

[3] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[4] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. 1993. Automatic Program Parallelization. *Proc. IEEE* 81, 2 (1993), 211–243. http://engineering.purdue.edu/paramnt/publications/BENP93.pdf

[5] A. Bhosale and R. Eigenmann. 2020. Compile-time Parallelization of Subscripted Subscript Patterns. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 317–325.

[6] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[7] William Blume and Rudolf Eigenmann. 1994. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proceedings of Supercomputing '94, Washington D.C.* (Nov. 1994), 528–537.

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

[9] Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)* 35, 3 (2008), 1–14.

[10] Francis Dang, Hao Yu, and Lawrence Rauchwerger. 2002. The R-LRPD test: Speculative parallelization of partially parallel loops. In *Proceedings 16th International Parallel and Distributed Processing Symposium.* IEEE, 10–pp.

[11] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. 2009. Cetus: A source-to-source compiler infrastructure for multicores. *Computer* 42, 12 (2009), 36–42.

[12] Timothy A. Davis. 2006. *Direct Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718881 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898718881

[13] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[14] Eladio Gutiérrez, Rafael Asenjo, O Plata, and Emilio L. Zapata. 2000. Automatic parallelization of irregular applications. *Parallel Comput.* 26, 13-14 (2000), 1709–1738.

[15] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[16] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. *Improving Performance via Mini-applications.* Technical Report SAND2009-5574. Sandia National Laboratories.

[17] Intel. 2011. *Automatic Parallelization with Intel Compilers.* Intel. https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers , visited 10-21-2019.

[18] Philip A Knight, Daniel Ruiz, and Bora Uçar. 2014. A symmetry preserving algorithm for matrix scaling. *SIAM journal on Matrix Analysis and Applications* 35, 3 (2014), 931–955.

[19] Yuan Lin and David Padua. 1999. Demand-driven interprocedural array property analysis. In *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 303–317.

[20] Yuan Lin and David Padua. 2000. Analysis of irregular single-indexed array accesses and its applications in compiler optimizations. In *International Conference on Compiler Construction.* Springer, 202–218.

[21] Yuan Lin and David Padua. 2000. Compiler Analysis of Irregular Memory Accesses. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00).* ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/349299.349322

[22] Kathryn S. McKinley. June 1991. *Dependence Analysis of Arrays Subscripted by Index Arrays.* Technical Report. Rice University. TR91-162.

[23] Mahdi Soltan Mohammadi, Kazem Cheshmi, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. 2018. Extending index-array properties for data dependence analysis. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 78–93.

[24] 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)*. ACM, New York, NY, USA, 594–609. https://doi.org/10.1145/3314221.3314646

[25] Cosmin E Oancea and Lawrence Rauchwerger. 2011. A hybrid approach to proving memory reference monotonicity. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 61–75.

[26] PGI. 2018. *PGI Compiler User's Guide*. Nvidia. https://www.pgroup.com/resources/docs/18.4/openpower/pgi-user-guide/index.htm

[27] Dan Quinlan and Chunhua Liao. 2011. The rose source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.

[28] 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.

[29] Madalene Spezialetti and Rajiv Gupta. 1995. Loop monotonic statements. *IEEE Transactions on Software Engineering* 21, 6 (1995), 497–505.

[30] 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.

[31] Peng Tu and David Padua. August 12-14, 1993. Automatic Array Privatization. In *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.), Vol. 768. 500–521.

[32] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E Mark, and Herman JC Berendsen. 2005. GROMACS: fast, flexible, and free. *Journal of computational chemistry* 26, 16 (2005), 1701–1718.