Extending Index-Array Properties for Data Dependence Analysis

Mahdi Soltan Mohammadi[1], Kazem Cheshmi[2], Maryam Mehri Dehnavi[2], Anand Venkat[3], Tomofumi Yuki[4], and Michelle Mills Strout[1]

[1] University of Arizona, Tucson, USA
{kingmahdi, mstrout}@cs.arizona.edu
[2] University of Toronto, Toronto, Canada
{kazem, mmehride}@cs.toronto.edu
[3] Intel, United States
anand.venkat@intel.com
[4] Univ Rennes, Inria, France
tomofumi.yuki@inria.fr

Abstract. Automatic parallelization is an approach where a compiler analyzes serial code and identifies computations that can be rewritten to leverage parallelism. Many data dependence analysis techniques have been developed to determine which loops in a code can be parallelized. With code that includes indirect array accesses through what are commonly called index arrays, such data dependence analysis is restricted in the conclusions that can be drawn at compile time. Various approaches that use index array properties such as monotonicity have been shown to more effectively find parallel loops. In this paper, we extend the kinds of properties about index arrays that can be expressed, show how to convert loop-carried data dependence relations and relevant index-array properties to constraints that can be provided to the Z3 SMT solver, and evaluate the impact of using such index-array properties on identifying parallel loops in a set of numerical benchmarks.

Keywords: Data Dependence Analysis · Sparse Matrices · Automatic Parallelization · SMT Solvers

1 Introduction

Many numerical computations involve sparse tensors, which are vectors, matrices, and their higher-dimensional analogs, that have so few non-zeros that compressed storage of some kind is used. These compressed formats result in indirect array accesses such as `x[col[i]]` and non-affine loop bounds such as `rowptr[i+1]`. The arrays used to index other arrays are referred to as index arrays as well as subscripted subscripts in the compiler literature [12]. The index arrays cause difficulties for compile-time data dependence analyses used to find parallelizable loops. However, there is an opportunity to use properties about the values in such arrays such as monotonicity to determine at compile time that some of these loops are parallelizable.

```
1 for (i=0; i<nnz_y; i++) {
2     x[ idx[i] ] = x[ idx[i] ] + y[i];
3 }
```

Fig. 1: Addition of a sparse vector with a dense vector.

As an example, Figure 1 contains code that performs a vector add of a dense vector `x` and a sparse vector `y`. Since `y` has been compressed, it only stores non-zero values while the `idx` index array stores what dense index is associated with each value. The values stored in the `idx` index array are all unique since they should indicate dense locations of unique non-zeroes in `y` array. In other words, the `idx` index array has the injectivity property.

To determine whether the `i`-loop in Figure 1 is fully parallelizable, one must show there are no loop-carried dependences on the loop. Since the `idx` index array is injective, this implies that no two values in the `idx` array are the same and therefore each iteration reads and writes to and from different locations of the `x` array with the `x[idx[i]]` access. And, since there is no overlap between the writes and reads to `x[idx[i]]` in different iterations of the loop, there is no loop carried dependence. If the injectivity property is not known, then the compiler has to assume that any two values in `idx[]` might be the same and therefore conservatively assume a loop-carried dependence.

In the past, other research has used index-array properties for finding loop parallelism. McKinley [12] first pointed out index array properties can be used to facilitate data dependence analysis, but automating the approach was left as future work. She detailed how injectivity and monotonicity of index arrays can show lack of flow, anti, and output dependences for some common access patterns. Wonnacott and Pugh [16] represented index arrays as uninterpreted function symbols in their constraint based dependence analysis. That resulted in them inherently utilizing functional consistency property about index arrays. Lin and Padua [11] formulated five index array properties including monotonicity, and four properties about values and bounds of index array being closed-formed under loop iterators. Fuzzy data dependence analysis [5] uses the range of index arrays to conservatively approximate dependences. The Hybrid Analysis research formulated the monotonicity and injectivity properties and used them to find full parallelism in the PERFECT CLUB and SPEC benchmarks [15, 19, 18, 13].

We can express all of the previously studied index-array properties as universally quantified affine constraints on index array values, similar to approaches used in program verification community to formulate properties about general arrays [6, 10, 9]. We use constraint-based data dependence analysis to determine lack of loop-carried data dependences for finding fully parallel loops in number of popular sparse computations that are building blocks of bigger numerical applications. Just like [16], we represent index arrays as uninterpreted functions. For example, the injectivity property for an index array like `idx[]` in Figure 1 can be specified by indicating that for all indices x_1 and x_2 into the index array

where x_1 is not equal to x_2, the values in the index array are also not equal:

$$(\forall x_1, x_2)(x_1 \neq x_2 \implies index(x_1) \neq index(x_2).$$

In this paper, we experiment with more index-array properties that can be expressed with such universally quantified constraints including information about sparse matrix formats, matrix triangularity information, and periodic monotonicity and injectivity. Oancea and Rauchwerger [14] observed that periodic monotonicity could help prove loop parallelism, but did not provide a mechanism for applying the property automatically.

This paper makes the following contributions:

- Specification of more index-array properties, namely triangularity, and periodic monotonicity.
- A constraint-based data dependence analysis that uses an SMT solver to apply the index array properties for finding fully parallel loops.
- Experimental results showing that the index array properties introduced in this paper lead to more loops being parallelized and that those loops when parallelized exhibit improved performance.
- An in-depth comparison of related work based on what index array properties used and what impact on automatic parallelization such work reported.

2 Background: Data-Dependence Analysis

This section reviews how loop carried array dependence analysis can be specified as a constraint problem, and how index arrays found in sparse codes can be represented in these constraints as uninterpreted functions [17].

2.1 Loop-Carried Dependence Constraints for Sparse Codes

A fully parallel loop will not have any loop-carried dependences. A data dependence occurs between two iterations of a loop when both of the iterations access the same memory location and at least one of the accesses is a write. Such data dependence constraints can be expressed in the following generic form:

$$\{I \rightarrow I' | \overbrace{\boldsymbol{I} \prec \boldsymbol{I'}}^{lexicographical\ Ordering} \wedge \underbrace{F(\boldsymbol{I}) = G(\boldsymbol{I'})}_{Array\ Access\ Equality} \wedge \overbrace{Constraints(\boldsymbol{I}) \wedge Constraints(\boldsymbol{I'})}^{Loop\ Bounds\ and\ Conditional\ Constraints\ (if,...)}\}$$

where $\boldsymbol{I}$ and $\boldsymbol{I'}$ are iteration vector instances from the same loop nest, $\boldsymbol{I} \prec \boldsymbol{I'}$ denotes that iteration I happens lexicographically before iteration I', F and G are macro functions that define array index expressions to the same array with at least one of the accesses being a write, $Constraints$ is the macro function that defines conditional expressions, and the loop bounds for the $\boldsymbol{I}$ iteration vectors.

In this paper, the term *dependence relation* is used interchangeably with dependence constraints by viewing them as a relation between $\boldsymbol{I}$ and $\boldsymbol{I'}$. If the constraints in the data dependence relation are shown to be satisfiable, then a loop-carried dependence exists and the loop is not fully parallelizable. Therefore, our goal is to find as many UNsatisfiable data dependence relations as possible.

```
1 for (int j = 0 ; j < n ; j++){
2   x[j] /= Lx[colPtr[j]] ;
3   for(int p = colPtr[j]+1 ; p < colPtr[j+1] ; p++){
4     x[row[p]] -= Lx[p] * x[j];
5 }}
```

Fig. 2: Forward Solve computation assuming matrices are stored in CSC sparse matrix format, code from Cheshmi et al. [7].

2.2 Data Dependence Analysis Example

As an example, Figure 2 shows a forward solve computation implemented for any lower triangular sparse matrix stored in compressed sparse column (CSC) format. In this code the outer loop `j` traverses over compressed column indices in `colPtr` index array, and the `p` loop goes over nonzeros. Also, the `row` index array stores the row indices, while `Lx` and `x` store the values of nonzeros. Consider a read `x[j]`, and a write `x[row[p]]` to array `x`, both in line 4 of Figure 2. The read after write dependence for the `j-loop` has the following constraints:

$$\underbrace{\{[j] \rightarrow [j'] : \exists p, p' : \overbrace{j < j'}^{lexicographical\ Ordering} \wedge \overbrace{row(p) = j'}^{Array\ Access\ Equality} \wedge}_{Loop\ Bounds}$$
$$\underbrace{0 \leq j, j' < n \wedge\ colPtr(j) < p < colPtr(j+1) \wedge\ colPtr(j') < p' < colPtr(j'+1)\}}_{Loop\ Bounds}$$

Also note that, if we define the flow dependence based on the same accesses for the loop carried dependence of inner loop `p` it would be as follows:

$$\underbrace{\{[j, p] \rightarrow [j', p'] : \overbrace{j = j' \wedge p < p'}^{lexicographical\ Ordering} \wedge \overbrace{row(p) = j'}^{Array\ Access\ Equality} \wedge}_{Loop\ Bounds}$$
$$\underbrace{0 \leq j, j' < n \wedge\ colPtr(j) < p < colPtr(j+1) \wedge\ colPtr(j') < p' < colPtr(j'+1)\}}_{Loop\ Bounds}$$

The difference between two dependences is in the lexicographical ordering.

3 Disproving Dependences with Index-Array Properties

With no information about uninterpreted functions, it is important to assume they can take on any value. Nonetheless, the index arrays that uninterpreted functions represent have various properties that can be used to add more constraints to data dependence relations and in some cases can enable showing more dependences are unsatisfiable. In this section, we first give an example illustrating a case where a combination of two index array properties enables showing a dependence is unsatisfiable, show how to implement this approach using an SMT solver, and present the index array properties we have found useful.

```
1  for(i = 0; i < n; i++) {
2    val[colPtr[i]] = sqrt(val[colPtr[i]]);
3
4    for(m=colPtr[i]+1; m<colPtr[i+1]; m++){
5      val[m] = val[m] / val[colPtr[i]];
6    }
7    for(m=colPtr[i]+1; m<colPtr[i+1]; m++){
8      for(k = colPtr[row[m]];
9          k < colPtr[row[m]+1]; k++){
10       for(l = m; l < colPtr[i+1]; l++){
11         if(row[l] == row[k]
12            && row[l+1] <= row[k]){
13            val[k] -= val[m]* val[l];
14 }}}}}
```

$$
\begin{aligned}
&\{[i,m] \rightarrow [i',m'] : \exists k,l,k',l' : \\
&i = i' \wedge m < m' \wedge k = m' \\
&0 \leq i,i' < n \wedge row(l) = row(k) \\
&colPtr(i) < m < colPtr(i+1) \\
&colPtr(row(m)) \leq k \\
&k < colPtr(row(m)+1) \\
&colPtr(i') < m' < colPtr(i'+1) \\
&colPtr(row(m')) \leq k' \\
&k' < colPtr(row(m')+1) \\
&m \leq l < colPtr(i+1) \\
&m' \leq l' < colPtr(i'+1) \\
&row(l') = row(k') \\
&row(l'+1) \leq row(k')\}
\end{aligned}
$$

Fig. 3: Incomplete Cholesky0 implementation from SparseLib++ library [2]

3.1 Example Using New Index Array Properties

For the Incomplete Cholesky code shown in Figure 3 on the left, the data dependence relation due to the write `val[k]`, and a read `val[m]` in Line 13 for m-loop in line 7 is shown on the right side of the figure. Using only original constraints above, it is not possible to prove this dependence unsatisfiable. Nonetheless, using additional constraints instantiated from the triangularity and monotonicity index array properties, explained in Section 3.3, can show us that the dependence is unsatisfiable. Figure 4 depicts the partial ordering between key constraints from the dependence in question, derivation of new constraints from index array properties assertions, and the contradiction that proves unsatisfiability.

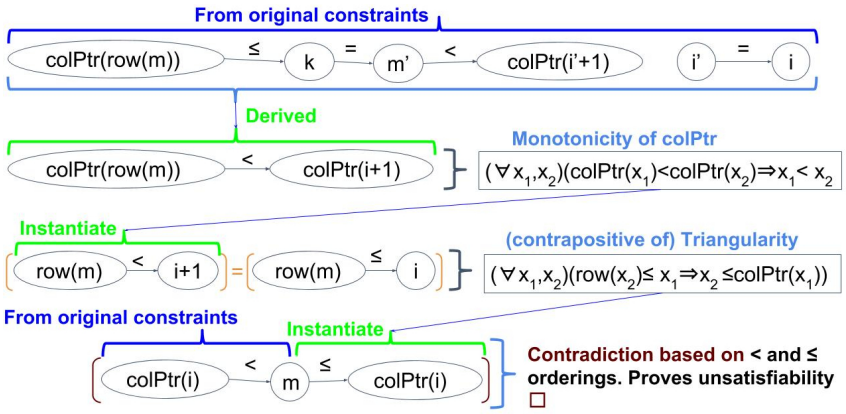


Fig. 4: Proving a dependence of Incomplete Cholesky Unsatisfiable.

3.2 Leveraging SMT Solvers

We can leverage SMT solvers to use index array properties to determine the satisfiability of data dependence relations. A common approach used in verification community to obtain more precise results [6, 10, 9] is to express array properties as universally quantified assertions, combine them with their originally extracted constraints about a program, and ask a SMT solver whether the constraints are satisfiable. We use the same approach by utilizing Z3 SMT solver [3]. The Z3 SMT solver uses the interface format SMTLIB2 [4]. Therefore, we specify all the constraints from the original dependence and all the user asserted index array properties for Z3 SMT solver in SMTLIB format. For instance, the following shows some of the constraints from original dependence used in the unsatisfiability proof in Section 3.1 in SMTLIB format:

Original constraint	**Z3 specification**
$colPtr(i) = k'$	`(assert (= (colPtr i) k') )`
$colPtr(row(m')) <= k'$	`(assert (<= (colPtr (row m')) k') )`
$(\forall x_1, x_2)(x_1 < x_2 \iff colPtr(x_1) < colPtr(x_2))$	`(assert (forall ((e1 Int) (e2 Int)) (=> (< e1 e2) (< (colPtr e1) (colPtr e2)))) )` `(assert (forall ((e1 Int) (e2 Int)) (=> (< (colPtr e1) (colPtr e2)) (< e1 e2))) )`
$(\forall x_1, x_2)(colPtr(x_1) < x_2 \implies x_1 < row(x_2))$	`(assert (forall ((e1 Int) (e2 Int)) (=> (< (colPtr e1) e2) (< e1 (row e2)))) )`

3.3 Index-Array Properties as Universally Quantified Assertions

In this section, we describe such index array properties that can be formulated as universally quantified assertions.

- **Functional Consistency**: If two inputs to a function are equal then the function will return equivalent results.

$$(\forall x_1, x_2)(x_1 = x_2 \Rightarrow f(x_1) = f(x_2))$$

- **Domain and range of index arrays**: We assume the input domain and output range of the index arrays are known and can be expressed as follows:

$$(\forall x)(p \leq x \leq q \Rightarrow LB_{range} \leq f(x) <= UB_{range}).$$

- **Monotonic index arrays**: In several different sparse matrix formats, it is common to see index arrays that are monotonic. There are four variations of the monotonicity property:

`Increasing:` $(\forall x_1, x_2)(x_1 \leq x_2 \Rightarrow f(x_1) \leq f(x_2))$.
`Strictly Increasing:` $(\forall x_1, x_2)(x_1 < x_2 \iff f(x_1) < f(x_2))$.
`Decreasing:` $(\forall x_1, x_2)(x_1 \geq x_2 \Rightarrow f(x_1) \geq f(x_2))$.
`Strictly Decreasing:` $(\forall x_1, x_2)(x_1 > x_2 \iff f(x_1) > f(x_2))$.

For instance the `colPtr` in Figure 3 is monotonically strictly increasing since it stores the starting point of nonzero row indices in `row` and values in `val`, and computation always operates on matrices that have nonzeros in main diagonal, therefore we have $(\forall x_1, x_2)(x_1 < x_2 \iff colPtr(x_1) < colPtr(x_2))$.

– **Injective index arrays**: An index array is injective if none of its values are the same. Index arrays that have strict monotonicity, also have injectivity property. Strict monotonicity is a more informative property. Consequently, in our experimental evaluations, we only utilize strict monotonicity for index arrays that poses them. Note, in the kernels that we have looked for parallelism, we have not come across any index array that only poses injectivity, that is why injectivity property does not show up in our results.

`Injectivity:` $(\forall x_1, x_2)(x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2))$.

– **Periodic injective/monotonic index arrays**: Some index arrays are monotonic or injective in intervals.

$$\texttt{Periodic Property: } (\forall x_1, x_2, x_3)(x_2 \bowtie x_3 \wedge g(x_1) \leq x_2 < g(x_1+1) \\ \wedge\, g(x_1) < x_3 < g(x_1+1) \Rightarrow f(x_1) \bowtie f(x_2)).$$

Where $\bowtie$ can be $\neq$, $\leq$, $<$, $\geq$, and $>$, for injectivity and different forms of monotonicity. For instance the `row` in Figure 3 is periodically monotonically strictly increasing and it is indexed using `colPtr`. This is because `row` stores row index values of nonzeros which are unique for nonzeros in a column. Therefore, we have:

$$(\forall x_1, x_2, x_3)(x_2 < x_3 \wedge colPtr(x_1) < x_2 < colPtr(x_1) \wedge \\ colPtr(x_1) < x_3 < colPtr(x_1) \iff row(x_1) < row(x_2)).$$

– **Triangular Matrix (Triangularity)**: Some numerical computations only operate on lower or upper triangular parts of matrices. We express this property for CSC, Compressed Sparse Row (CSR), and Block CSR (BCSR) formats as following assertions, assuming `f` is the compressed index array of the format and `g` is the non-compressed one:

$$\begin{aligned}
&\texttt{CSC Lower Triangularity:} && (\forall x_1, x_2)(f(x_1) < x_2 \iff x_1 < g(x_2)).\\
&\texttt{CSC Upper Triangularity:} && (\forall x_1, x_2)(f(x_1) > x_2 \iff x_1 > g(x_2)).\\
&\texttt{(B)CSR Lower Triangularity:} && (\forall x_1, x_2)(x_1 < f(x_2) \iff f(x_1) < x_2).\\
&\texttt{(B)CSR Upper Triangularity:} && (\forall x_1, x_2)(x_1 > f(x_2) \iff f(x_1) > x_2).
\end{aligned}$$

For instance the Incomplete Cholesky computation in Figure 3 operates on lower triangular matrices. Considering the `colPtr` compresses the column index arrays in CSC format, and `row` stores row indices explicitly, we have:

$$(\forall x_1, x_2)(colPtr(x_1) < x_2 \implies x_1 < row(x_2))$$

This indicates that the integer value of row indices of nonzeros in columns after column x_1 are greater than the integer value of (column index) x_1.

4 Impact on Finding Full Parallelism

In this section, we study the effect of utilizing index array properties for finding fully parallel loops in number of popular numerical sparse computations. Section 4.1 presents the suite of numerical kernels that we compiled to evaluate our approach, and discusses the automatic driver that utilizes index array properties for finding full parallelism. Section 4.2 presents results for finding fully parallel loops in each benchmark, we also indicate what kind of index array properties were needed to prove that a loop is parallel. We also parallelized the parallel loops found in two of the kernels by hand using simple OpenMP parallel loop pragma. Section 4.3 reports the performance results for hand parallelized kernels while comparing performance to serial versions' performance, and discusses the implications of this results. Following public git repository hosts this paper's artifact, and includes instructions on how to reproduce the evaluation results:

https://github.com/CompOpt4Apps/Artifact-SparseLoopParallelism

The driver depends on CHILL compiler [1], IEGenLib library [22], and Z3 SMT solver. CHiLL is a source-to-source compiler framework for composing and applying high level loop transformations. IEGenLib is a library for manipulating integer sets/relations that contain uninterpreted function symbols.

4.1 Sparse Computation Benchmark Suite and Methodology

Table 1 lists the numerical sparse codes that we have used to evaluate usefulness of utilizing index array properties. The benchmark suite includes the fundamental blocks in several applications: (1) The Cholesky factorization, Incomplete Cholesky0, and sparse triangular solver, which are commonly used in direct solvers and as preconditioners in iterative solvers; (2) sparse matrix vector multiplication and Gauss-Seidel methods, often used in iterative solvers.

We specify user-defined properties about index arrays in JSON files for each code. The driver for finding parallel loops uses CHILL to extract the dependences for different loops in a code that would include loops in different levels and locations of an (im)perfectly nested loop. Then, it converts extracted dependences one at a time alongside related user defined assertions to an input file for Z3, and queries Z3 whether the constraints are satisfiable. If all the dependences of a loop are unsatisfiable we say that loop is parallel.

SMT solvers like Z3 use numerous heuristics to detect unsatisfiable set of constraints quickly. In our experience, Z3 can return with answer for any of unsatisfiable (unsat) dependencies in our benchmark suite in less than 1 second. It also quickly comes back with the answer satisfiable (sat) for some of the dependences. However, for dependences that there is not enough constraints to determine them either as sat or unsat, it could run for a long time while deriving new (not helpful) constraints from universally quantified assertions. Therefore, we use a 2 seconds timeout so if Z3 could not detect a single dependence as sat or unsat, it would return `unknown` after timeout. In such case, we conservatively consider the dependence satisfiable.

Table 1: The code benchmarks that we apply our data dependence analysis on, with formatting and source.

Algorithm name	Format	Source
Forward solve	CSC	[7]
Forward solve	CSR	[25]
Gauss-Seidel solver	CSR	MKL [26]
Gauss-Seidel solver	BCSR	MKL [26]
Sparse MV Multiply	CSR	Common
Incomplete Cholesky	CSC(R)	[2]
Static Left Cholesky	CSC	[7]

Table 2: Input Matrices for parallelized codes from [8]. Sorted in order of Number of Nonzeros per Column

Matrix	Columns	Nonzeros	$\frac{NNZ}{COL}$
G3_circuit	1,585,478	7,660,826	5
af_shell3	504,855	17,562,051	35
bmwcra_1	148,770	10,641,602	72
crankseg_2	63,838	14,148,858	222
nd24k	72,000	28,715,634	399

It takes about 34, 18, and 18 seconds respectively to determine outer most loops of Incomplete Cholesky, Static Cholesky, and Gauss-Seidel BCSR, are not parallel. The dependence analysis of all other loops in all benchmarks takes less than 2.5 seconds. The reason why it takes more for those mentioned three loops is that there are lots of dependences to check for them, some of which exhausts the 2-second timeout we specified for Z3.

We hand parallelized some of the parallel loops that we found in our benchmarks to study the pragmatic impact of our methods, the results are presented in Section 4.3. We ran our experiment on a machine with an Intel(R) Core(TM) i7-6900K CPU, 32GB of 3000MHz DDR4 memory, and Ubuntu 16.04 OS. The CPU has 8 cores, therefore we record performance for 2, 4, and 8 OpenMP threads while the hyper-threading is disabled. We report mean value of 5 runs, though there were no significant variation between runs. All codes are compiled with GCC 5.4.0 with -O3 flag enabled.

Table 2 lists set of five matrices from the University of Florida sparse matrix collection [8] that we used as input to our experiments. The matrices are listed in increasing ordered of average nonzeros per column. Generally speaking, the loops that we are parallelizing usually operate on different nonzeros in a column (or row). Therefore, one can expect the parallelization result getting better for matrices with more nonzero per columns (rows).

4.2 Finding Loop Parallelism

Since we want to study effect of using different index array properties, as well as using all available properties, we have look into what set of index array properties can help us prove a loop to be parallel. We listed 6 properties in Section 3.3. Of those properties, we do not need to specify functional consistency for a SMT solver, since it considers it inherently for any uninterpreted function symbol. We also just specify domain and range properties while considering effect of any combination of properties. Additionally, all the injective index arrays in our benchmarks also have monotonically strictly increasing property.

Table 3: Table lists loops found in our benchmarks, whether they are statically parallel, and whether we needed index array properties to prove them parallel. The third column, ‘‘`Static Par?`’’, is indicating whether the loop can be considered statically parallel, if the loop would require reduction for parallelization it is noted as Reduction. Although, we are not doing dependence analysis for finding reduction operations at this time. The fourth column indicates whether we have found the loop parallel. The fifth column lists what index array property were necessary to detect the loop as parallel, while `Linear` indicates that we did not need any, functional consistency would be used by Z3 for all cases. The shortened names, Mono, PerMono, Tri stand for monotonicity, periodic monotonicity, and triangularity properties respectively.

Algorithm	Loop	Static Par?	Detected?	Helps to Prove Parallel
Forward Solve CSC	j	No	-	-
	p	Yes	Yes	Tri + PerMono
Forward Solve CSR	i	No	-	-
	j	Reduction	-	-
Gauss-Seidel CSR	i	No	-	-
	j	Reduction	-	-
Gauss-Seidel BCSR	i	No	-	-
	$ii1$	Yes	Yes	`Linear`
	j	Reduction	-	-
	$jj1$	Reduction	-	-
	$ii2$	Yes	Yes	`Linear`
	$ii3$	Yes	Yes	`Linear`
	$jj2$	Reduction	-	-
	$ii4$	Yes	Yes	`Linear`
Sparse MV Multiply	i	Yes	Yes	`Linear`
	j	Reduction	-	-
Incomplete Cholesky	i	No	-	-
	$m-4$ (line 4)	Yes	Yes	`Linear`
	$m-7$ (line 7)	Yes	Yes	Tri + Mono + PerMono
	k	Yes	Yes	Tri + Mono
	l	Yes	Yes	PerMono
Static Left Cholesky	$colNo$	No	-	-
	$nzNo$	Yes	Yes	PerMono
	i	No	-	-
	l	Yes	Yes	PerMono
	j	Yes	Yes	PerMono

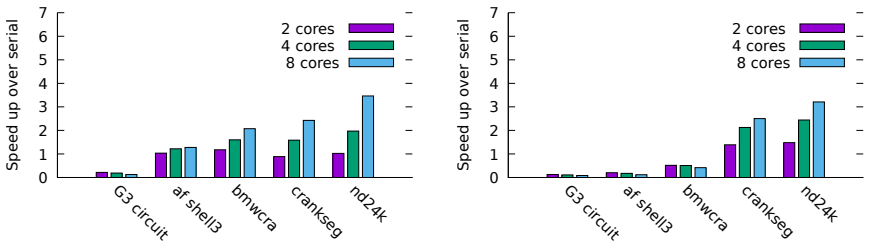
Table 3 presents the result of finding parallel loops while utilizing index array properties in our benchmark suite. The second column in the table lists all the unique loops in each kernel by loop's iterator name. The third column indicates whether considering the algorithmic property of the computation the loop can be statically considered parallel. And, the last column indicates what information has helped us prove the loop as parallel. The `Linear` keyword indicates that all the dependences of the loop can be proved unsatisfiable just by looking at their linear constraints and we would not need using index array property.

The main observation from analyzing Table 3 are: (1) The index array properties are very helpful for finding parallel loops in three codes, namely Forward Solve CSC, Incomplete Cholesky, and Static Left Cholesky; (2) We find all the loops that are parallel considering whether nature of an algorithm allows a loop to be parallel, except for loops that require parallelization with reduction, which was explained earlier; (3) All the three none-trivial index array properties that we have formulated, including Periodic Monotonicity and Triangularity that were not look at in previous works, are being helpful.

4.3 Performance Impact

Results presented in this section will indicate our dependence analysis can indeed improve performance of serial implementations of some of the sparse kernels in our benchmarks. This could be useful in practice, since to the best of our knowledge, library implementations of parallel sparse computations are not common, and we could not find any open source, parallel libraries for our benchmark computation. At this time, there is no generic code generator that can generate transformed code for our sparse benchmarks. Consequently, we only hand parallelized the parallel loops that our analysis founds in Forward Solve CSC and Incomplete Cholesky, simply by using OpenMP parallel pragma's without any further optimization. The parallel loop in Forward Solve computation only has one subtraction and one multiplication operations, hence it is memory-bound. As could be expected, our experiment results showed that parallelization overhead only slow downs the Forward Solve CSC compared to serial version. Nonetheless, we get considerable performance gain by parallelizing Incomplete Cholesky even without any enabling transformation like tiling.

Table 3 shows that our analysis detects 4 loops as fully parallel in Incomplete Cholesky. Nonetheless, we have experimented with parallelizing two of those loops, namely `m-7` (line 7 in figure 3) and `k`. Figure 5 presents relative performance of parallelizing `m-7` and `k`. The result show steady increase in performance gain for parallel version over serial as we increase the number of threads compared. Parallelizing loop `m-7` gives us performance gain of 1.3-3.5$\times$ in 4 out of 5 while utilizing 8 threads, For G3_circuit matrix parallelization overhead makes parallel version slower than sequential version since it has very few nonzeros per columns. Parallelizing the `k` loop has worse results compared to `m-7` loop for three of the matrices, but has better results for two of the matrices. This could be attributed to sparsity structure of matrices that can effect parallel load balancing.



(a) Performance of parallelized m-7-loop. (b) Performance of parallelized k-loop.

Fig. 5: Incomplete Cholesky loop parallelization performance. The serial absolute execution time in order from top are: 1.9, 11.1, 18.1, 202.6, and 725.4 seconds.

5 Related Work

Several previous works have looked into usefulness of index array properties for improving data dependence analysis. Nonetheless, it seem that index array properties have not been taken advantage of in production compilers. In this section, we describe what properties each previous work has look into, how they have derived them, and what impact index array properties had on their results.

5.1 Initial Observation of Index Array Property Utility

McKinley [12] studied how using assertions about index arrays can improve dependence testing precision at compile-time. She observed and discussed 5 common assertions about index arrays that included, injectivity ($index(I) \neq index(J), I \neq J$), monotonically non-decreasing ($index(I) \leq index(I+1)$), monotonically increasing ($index(I) < index(I+1)$), monotonically non-increasing ($index(I) \geq index(I+1)$), and monotonically decreasing ($index(I) > index(I+1)$). We usually refer to later four properties simply as monotonicity. Author then discussed how these properties can effect founding flow, anti, or output independence in some common memory access patterns. The report discussed two loops in MDG from PERFECT club where using monotonicity is necessary to detected the loop as parallel in compile-time. Automating the approach of utilizing index arrays was left as future work.

5.2 Exact Data Dependence Analysis

Pugh and Wonnacott [16] presented a constraint based data dependence analysis that was able to handle nonlinear data access including index arrays. They represented affine parts of the dependences with Presburger Formulas while representing index arrays as uninterpreted function symbols. Authors used Ackerman Reduction procedure that can be applied to Presburger formulas with uninterpreted function calls [20]. By using that procedure they inherently utilized functional consistency for index arrays. Their framework could also come

up with sufficient constraints to prove the dependences unsatisfiable for some of codes that earlier methods could not. Although, solving those constraints required methods that were not discussed or implemented for that work. Authors used their methods to do dependence analysis for PERFECT CLUB benchmarks. They were able to directly find parallel loops in MDG and ARC2D, one in each, and were able to find sufficient conditions for showing one parallel loops in each of TRFD, and ARC2D benchmarks.

5.3 Automating the detection of properties

Lin and Padua [11] looked into two types of irregular array accesses, single-index, and indirect (index array) accesses. The single-index was defined for array index expressions, and included: Consecutively written accesses, and stack accesses. They also formulated tests for detecting 5 index array properties: monotonicity, injectiveness, closed-form distance (CFD), closed-form value (CFV), and closed-form bound (CFB). The closed-formed properties are defined when either difference of two consecutive values (CFD), or all values (CFV), or upper or lower bound (CFB), of an index array can be defined with a closed-form expression over loop iterators. Analyzing benchmarks from PERFECT club, this work showed following usefulness for those properties: CFV and CFD each separately helped to find one parallel in TRFD and DYFESM benchmarks respectively, and CFB helped privatization analysis in two loops inside BDNA and P3M.

5.4 Combining with More General Dependence Analysis

Hybrid analysis (HA) presented over several works is an approach to do general data dependence analysis of generic array accesses for optimization purposes [15, 19, 18, 13]. It gathers dependence constraints with inter-procedural analysis, and represents the gathered summary sets with an intermediate representation called Unified Set Representation (USR). The USRs can include uninterpreted function calls to represent index arrays. HA formulates flow, anti, and output independence with USRs, and uses them for for finding full parallelism, private arrays, and reduction operations. HA also formulates the monotonicity and injectivity properties directly for array access expressions, which could have index arrays, and uses them in facilitation of different dependence analysis. Hybrid analysis works successfully find many parallel loops in PERFECT CLUB and SPEC benchmarks, however, the usefulness of index array properties is not exactly clear in their results, partially because they define the properties for array indexing expression and not index array, and partially because they usually just differentiate between finding a loop parallel either in compile-time or runtime. Nonetheless, their results implies that they do get similar results, for finding parallel loops and analyzing privatization in MDG, BDNA, TRFD, and DYFESM, as McKinley [12], and Lin and Padua [11].

5.5 Other Uses for Index Array Properties

Unlike other works that were either doing general dependence analysis or finding full parallelism, Venkat et al. [23] used index array properties in wavefront parallelization. They were trying to partially parallelize outer most loops in two numerical sparse computation, namely Gauss-Seidel, and Incomplete LU0. And, since the index array values would not be available until runtime, they were using runtime inspectors to generate dependence graph of loop iterations, which could be used to generate wavefronts of iterations, which can be run in parallel. One problem that they addressed was that if one naively generates runtime inspectors for all the compile-time extracted dependence, the overhead of runtime inspection would make parallelization useless. Instead, they used index array properties for two purposes, for one they used them to prove as many dependence as unsatisfiable at compile-time, then they also used index array properties to derive new equality constraints in remaining dependences that made their runtime inspectors faster. They formulated and used two index arrays properties for Incomplete LU0 namely monotonicity, and correlated monotonicity (a relation between two related index arrays in this code). They also formulated and used monotonicity property for index arrays in Gauss-Seidel computation. In follow-on work, Mohammadi et al. [21] formulated these constraints with universally quantified constraints and instantiated them to ISL [24].

6 Conclusion

In this paper, we showed how properties about index arrays in sparse matrix computation can be formulated with universally quantified assertions and automatically utilized in compile-time dependence analysis. Although, previous works have looked into usefulness of index array properties, to our knowledge, none of them has formulated or utilized all the properties formulated in this paper for dependence analysis. Particularly, we are not aware of any previous work that would describe triangularity and periodic monotonicity properties. Our results showed index array properties can help us find several parallel loops in our benchmarks that would have not been possible without them, at least not without runtime analysis. We also discussed results of parallelization for two of the codes in our benchmark where detected parallel loops by our framework had been hand-parallelized. The results showed 1.3-3.5× performance gain for parallelized Incomplete Cholesky over its sequential version.

References

1. Ctop research group webpage at utah (2018), http://ctop.cs.utah.edu/ctop/
2. Sparselib++ homepage (2018), https://math.nist.gov/sparselib++/
3. Z3 git homepage (2018), https://github.com/Z3Prover/z3/wiki
4. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)

5. Barthou, D., Collard, J.F., Feautrier, P.: Fuzzy array dataflow analysis. Journal of Parallel and Distributed Computing **40**(2), 210–226 (1997)
6. Bradley, A.R., Manna, Z., Sipma, H.B.: What's Decidable About Arrays?, pp. 427–442. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
7. Cheshmi, K., Kamil, S., Strout, M.M., Dehnavi, M.M.: Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 13:1–13:13. SC '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3126908.3126936, http://doi.acm.org/10.1145/3126908.3126936
8. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS) **38**(1) (2011)
9. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
10. Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: Proceedings of the 11th International Conference on Foundations of Software Science and Computational Structures. pp. 474–489. FOSSACS'08/ETAPS'08 (2008)
11. Lin, Y., Padua, D.: Compiler analysis of irregular memory accesses. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. vol. 35, pp. 157–168. ACM, New York, NY, USA (May 2000)
12. McKinley, K.: Dependence analysis of arrays subscriptecl by index arrays. Tech. Rep. TR91187, Rice University (1991)
13. Oancea, C.E., Rauchwerger, L.: Logical inference techniques for loop parallelization. In: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation. PLDI '12, ACM, New York, NY, USA (2012)
14. Oancea, C.E., Rauchwerger, L.: A hybrid approach to proving memory reference monotonicity. In: Languages and Compilers for Parallel Computing. pp. 61–75. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
15. Paek, Y., Hoeflinger, J., Padua, D.: Simplification of array access patterns for compiler optimizations. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation. pp. 60–71. PLDI '98, ACM, New York, NY, USA (1998)
16. Pugh, W., Wonnacott, D.: Nonlinear array dependence analysis. In: Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers. Troy, New York (May 1995)
17. Pugh, W., Wonnacott, D.: Constraint-based array dependence analysis. ACM Transactions on Programming Languages and Systems **20**(3) (May 1998)
18. Rus, S.: Hybrid Analysis of Memory References and its Application to Automatic Parallelization. Ph.D. thesis, Texas A&M (2006)
19. Rus, S., Hoeflinger, J., Rauchwerger, L.: Hybrid analysis: static & dynamic memory reference analysis. International Journal Parallel Programming **31**(4) (2003)
20. Shostak, R.E.: A practical decision procedure for arithmetic with function symbols. J. ACM **26**(2), 351–360 (Apr 1979). https://doi.org/10.1145/322123.322137, http://doi.acm.org/10.1145/322123.322137
21. Soltan Mohammadi, M., Cheshmi, K., Gopalakrishnan, G., Hall, M., Mehri Dehnavi, M., Venkat, A., Yuki, T., Strout, M.: Sparse Matrix Code Dependence Analysis Simplification at Compile Time. ArXiv e-prints (Jul 2018)
22. Strout, M.M., LaMielle, A., Carter, L., Ferrante, J., Kreaseck, B., Olschanowsky, C.: An approach for code generation in the sparse polyhedral framework. Parallel Computing **53**(C), 32–57 (April 2016)

23. Venkat, A., Mohammadi, M.S., Park, J., Rong, H., Barik, R., Strout, M.M., Hall, M.: Automating wavefront parallelization for sparse matrix computations. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 41:1–41:12. SC '16 (2016)
24. Verdoolaege, S.: Integer Set Library: Manual (2018), http://isl.gforge.inria.fr
25. Vuduc, R., Kamil, S., Hsu, J., Nishtala, R., Demmel, J.W., Yelick, K.A.: Automatic performance tuning and analysis of sparse triangular solve. ICS (2002)
26. Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., Wang, Y.: Intel math kernel library. In: High-Performance Computing on the Intel® Xeon Phi, pp. 167–188. Springer (2014)