Automatic Parallelization of a Class of Irregular Loops for Distributed Memory Systems

Mahesh Ravishankar, Ohio State University
John Eisenlohr, Ohio State University
Louis-Noel Pouchet, University of California, Los Angeles
J. Ramanujam, Louisiana State University
Atanas Rountev, Ohio State University
P. Sadayappan, Ohio State University

Many scientific applications spend significant time within loops that are parallel, except for dependencies from associative reduction operations, and usually contain data-dependent control-flow and array-access patterns. Traditional optimizations that rely on purely static analysis fail to generate parallel code.

This paper proposes an approach for automatic parallelization for distributed memory environments, using both static and run-time analysis. We develop algorithms to detect such loops, generate a parallel inspector which performs the run-time analysis, and a parallel executor. Extensions to the transformation scheme enables parallelization of codes that use external solvers. The effectiveness of the approach is demonstrated on real-world applications.

1. Introduction

Automatic parallelization and locality optimization of affine loop nests have been addressed for shared-memory multiprocessors and GPUs with good success (e.g., [Bondhugula et al. 2008; Bondhugula et al. 2010; Baskaran et al. 2010; Hall et al. 2010; Par4All 2012]). However, many large-scale simulation applications must be executed in a distributed memory environment, using irregular or sparse computations where the control-flow and array-access patterns are data-dependent. A common form of sparsity and unstructured data in scientific codes is via indirect array accesses, where elements of one array are used as indices to access elements of another array. Further, multiple levels of indirection may be used for array accesses. Virtually all prior work on polyhedral compiler transformations for affine codes is not applicable in such cases.

We propose a framework for automatic detection and distributed memory code generation for an extended class of affine computations that allow some forms of indirect array accesses. The class of applications targeted by the framework is prevalent in many scientific/engineering domains and the paradigm for its parallelization is often called the inspector/executor (I/E) [Saltz et al. 1990] approach. The I/E approach uses (1) an inspector code that examines some data that is unavailable at compile time but is available at the very beginning of execution (e.g., the specific inter-connectivity of the unstructured grid representing an airplane wing’s discretized representation) and is used to construct distributed data structures and computation partitions, and (2) an executor code that uses data structures generated by the inspector to achieve parallel execution of the application code.

The I/E approach has been well known in the high-performance computing community, since the pioneering work of Saltz and coworkers [Saltz et al. 1990] in the late eighties. The approach is routinely used by application developers for manual implementation of message-passing codes for unstructured grid applications. However, only a very small number of compiler efforts (that we detail in Section 10) have been directed at generation of parallel code using the approach. In this paper, using the I/E paradigm, we develop an automatic parallelization and code generation infrastructure for an extended class of affine loops, targeting a distributed memory message passing parallel programming model. This paper makes the following contributions:

— It presents a description of a class of extended affine computations which allow for data-dependent control flow and irregular data access patterns. This description is used to automatically detect such loops within the input program AST.
It presents a detailed description of the algorithms used to generate the parallel inspector that analyzes the computation at run-time, and the executor which performs the original computation in parallel.

— It presents a methodology by which applications that interface with external solvers in the original sequential code, can be parallelized with little or no manual intervention.

— It presents experimental results on the use of the approach to develop a complex parallel application, with performance approaching that of manually developed parallel code by expert application developers.

The rest of the paper is organized as follows. Section 2 describes the class of extended affine computations that we address, along with a high-level overview of the approach to code transformation. Section 3 provides details of the approach to generate computation partitions using a hypergraph that models the affinity of loop iterations to data elements accessed. Section 4 describes the first step in the automatic parallelization process - the detection of partitionable loops in the input sequential program. The algorithms for generation of inspector and executor code for the partitionable loops are provided in Section 5. Section 7 elaborates on how the need for inspector code can be optimized away for portions of the input code that are strictly affine. Section 8 addresses the crucial issue of automatic parallelization of sequential codes that utilize libraries from frameworks such as PETSc [Balay et al. 2012] or Trilinos [Heroux et al. 2003] that provide highly optimized distributed memory implementations of the libraries. Experimental results using four kernels and one significant application code are presented in Section 9. Related work is discussed in Section 10 and conclusions stated in Section 11.

2. Overview

This section outlines the methodology for automatic parallelization of the addressed class of applications. Listing 1 shows two loops from a conjugate-gradient iterative sparse linear systems solver, an example of the class of computations targeted by our approach. Loop $k$ computes the values of $x_i$. In loop $i$, vector $y$ is computed by multiplying matrix $A$ and vector $x$. Here $A$ uses the Compressed Sparse Row (CSR) format, a standard representation for sparse matrices. For a sparse matrix with $n$ rows, array $ia$ is of size $n+1$ and its entries point to the beginning of a consecutive set of locations in $A$ that store the non-zero elements in row $i$. For $i$ in $[0,n-1]$, these non-zero elements are in $A[ia[i]], ..., A[ia[i+1]-1]$. Array $col$ has the same size as $A$, and for every element in $A$, $col$ stores its column number in the matrix.

Figure 1 shows sample values for all arrays in the computation. The bounds of loop $j$ depend on values in $ia$, and the elements of $x$ accessed for any $i$ depend on values in $col$. Such arrays that affect the control-flow and array-access patterns are traditionally referred to as indirection arrays. All other arrays will be referred to as data arrays ($x$, $y$, and $A$ in the example). Similarly, scalars can be classified as data scalars or indirection scalars. The latter are those whose values are directly or indirectly used to compute loop bounds, conditionals, or index expressions of arrays. All other scalars (apart from loop iterators referenced in the loop body) are treated as data scalars. A key property of the code in Listing 1 is that the values of indirection arrays and indirection scalars can

Listing 1. Sequential conjugate gradient computation.

```c
while (!converged ) {
  //...Other computation not shown...
  for ( k = 0 ; k < n ; k++ )
    x[k] = ...;
  //...Other computation not shown...
  for ( i = 0 ; i < n ; i++ )
    for ( j = ia[i] ; j < ia[i+1] ; j++ ){
      xindex = col[j];
      y[i] += A[j]*x[xindex];
    }
  //...Other computation not shown...
}
```
be computed ("inspected") by an inspector component before any data arrays or scalars are read or updated. This property holds for all computations targeted by our approach, as discussed later.

The goal is to parallelize the computation by partitioning the iterations of loops $i$ and $k$ among the set of given processes. Suppose that iterations 2, 4, and 5 (shown in dark gray) are chosen to be executed on process 0, and the remaining ones (shown in light gray) on process 1. As discussed below, the choice of this partitioning is done at run time with the help of a hypergraph partitioner. Figure 2 illustrates the details of this partitioned execution; these details will be elaborated shortly.

We present a source-to-source transformation scheme that (1) generates code to analyze the computation at run time for partitioning the iterations, as well as data, among processes, (2) generates local data structures needed to execute the partitioned iterations in a manner consistent with the original computation, and (3) executes the partitions on multiple processes. The code that performs the first two steps is commonly referred to as an inspector, with the final step performed by an executor. Listing 2 shows the executor for the running example.

2.1. Targeted Computations

We target a class of computations that are more general than affine computations. In affine codes, the loop bounds, conditionals of $i \in \mathbb{R}s$, and array-access expressions are affine functions of loop iterators.
and read-only program parameters. For such codes, the control-flow and data-access patterns can be fully characterized at compile time.

Consider loop \( i \) in Listing 1. The bounds of loop \( j \) depend on \( i_a \), and accesses to \( x \) depend on \( \text{col} \). During analysis of loop \( i \), affine techniques have to be conservative and over-approximate the data dependences and control flow. We target a generalized class of computations, in which loop bounds, if conditionals, and array-access expressions are arbitrary functions of iterators, parameters, and values stored in read-only indirection arrays. Further, values in these indirection arrays may themselves be accessed through other indirection arrays. The control-flow and data-access patterns of such computations can be determined at run time by an inspector, before the actual computation is performed.

Within this class, we target loops that are parallel, except for loop-carried dependences due to reductions of data scalars or data array elements using operators which are associative and commutative. Such loops will be referred to as partitionable loops—they can be partitioned for parallel execution on distributed memory systems. Section 4 presents a detailed definition of the partitionable loops transformed by our approach, along with the scheme used to detect such loops. Loops \( i \) and \( k \) in Listing 1 are examples of partitionable loop. We identify and parallelize a set of disjoint partitionable loops. If a partitionable loop is nested inside another partitionable loop, only the outer loop is parallelized.

The proposed framework is well suited for computations that have a sequence of partitionable loops enclosed within an outer sequential loop (usually a time-step loop or a convergence loop), such that the control-flow and array-access patterns are not modified within the sequential loop. Such computations are common in many scientific and engineering domains. Furthermore, with this code structure, the inspector can be hoisted out of the outer loop.

### 2.2. Partitioning the Iterations

In Listing 1, there exists a producer-consumer relationship between the two loops due to array \( x \). In a parallel execution of both loops, communication would be required to satisfy this dependence. The volume of communication depends on the partitioning of the iterations. The process of computing these partitions (Section 3) may result in the iterations mapped to each process not being contiguous. They will be renumbered to be a contiguous sequence starting from 0. For example, iterations 2, 4,
and 5 of loop i, when assigned to process 0, are renumbered 0–2 (shown as the values of local iterator il in Figure 2).

2.3. Bounds of Inner Loops

The control flow in the parallel execution needs to be consistent with the original computation. As discussed earlier, the loop bounds of inner loops depend on values stored in read-only indirection arrays, loop iterators, or fixed-value parameters. Therefore, these bounds can be precomputed by an inspector and stored in arrays in the local data space of each process. The sizes of these arrays would be the number of times an inner loop is invoked on that process. For example, in Figure 1, for iterations mapped to process 0, inner loop j is invoked once in every iteration of loop i. Two arrays of size 3 would be needed to store the bounds of the j loop on process 0 (shown as lb and ub in Figure 2). Conditionals of ifs are handled similarly, by storing their inspected values in local arrays.

2.4. Partitioning the Data

Once the iterations have been partitioned, the data arrays are partitioned such that each process has local arrays to store all the data needed to execute its iterations without any communication within the loop. In Figure 2, y1, A1, and x1 are the local arrays on process 0 for data arrays y, A, and x.

The same data array element may be accessed by multiple iterations of the partitionable loop, which might be executed on different processes. Consider Figure 1, where x[1] and x[2] are accessed by both processes and are replicated on both, as shown in Figure 2. One of the processes is chosen as the owner of the data, and the location of the data on another process is treated as a ghost location. For example, x[2] is owned by process 0, but process 1 has a ghost location for it. The ghost locations and owned locations together constitute the local copy of a data array on a process.

Ghost elements for data arrays that are only read within the partitionable loop are set to the value at the owner before the start of the loop. Ghost locations for data arrays whose values are updated within the loop are initialized to the identity element of the update operator (0 for “+” and 1 for “*”). After the loop, these elements are communicated to the owner where the values from all ghost locations are combined. Therefore, the computation model is not strictly owner-computes. Since all update operations are associative and commutative, all iterations of the loop in the transformed version can be executed without any communication.

2.5. Data Accesses in the Transformed Code

The data-access patterns of the original computation need to be replicated in the transformed version. Consider expression col[j] used to access x in Listing 1. Since x1 is the local copy of data array x on each process, all elements of x accessed by a process are represented in x1. To access the correct elements in x1, array col could be replicated on each process, and a map could be used to find the location that represents the element col[j] of x. Such an approach would need a map lookup for every memory access and would be prohibitively expensive.

Similar to loop bounds, array-access expressions depend only on values stored in read-only indirection arrays, loop iterators, and constant parameters. Values of these expressions can be inspected and stored in arrays allocated in the local memory of each process. Further, the values stored are modified to point to corresponding locations in the local data arrays. The size of the array would be the number of times the expression is evaluated on a particular process. For example, the value of col[j] in Listing 1 is evaluated for every iteration of loop j. From Figure 1, for iterations of i mapped to process 0, the total number of iterations of loop j is 2 + 3 + 2 = 7. Therefore, an array i\_x\_j of size 7 on process 0 is used to “simulate” the accesses to x due to expression col[j].

2.6. Optimizing Accesses from Inner Loops

The approach described earlier would result in another array (of the same size as i\_x\_j) to represent the access A[j]. To reduce the memory footprint, we recognize that access expression j results in contiguous accesses to elements of A, for every execution of loop j. If the local array is such that elements that were accessed contiguously in the original data space remain contiguous in the local
data space of each process, it would be enough to store (in an additional array) the translated value of the access expression for only the first iteration of the loop. The rest of the accesses could be derived by adding to this value the value of the iterator, subtracted by the lower bound. The size of the array to hold these values is the number of times the loop is invoked. For example, for A[j], an array Ωa,j of size 3 is used on process 0 to store the accesses from the first iterations of the 3 invocations of loop j.

This optimization is applicable to all array-access expressions that are unit-stride with respect to a surrounding loop that is not a partitionable loop. Accesses from iterations of a partitionable loop mapped to a process are not necessarily contiguous with respect to the original computation.

2.7. Optimizing Accesses from the Partitionable Loop

For cases where elements of an array were accessed at unit-stride with respect to the partitionable loop in the original computation, it is desirable to maintain unit-stride in the transformed code as well. This can be achieved by placing contiguously in local memory all elements of the array accessed by the successive iterations on a process. For example, if iterations 2, 4, and 5 of loop k are mapped to process 0, elements of array x1 can be accessed by using iterator x1 if x1[0-2] correspond to x[2], x[4], and x[5]. The same could be done for y[i] in Listing 1.

If the same array is accessed elsewhere by another expression that is unit-stride with respect to an inner loop (as described earlier), the ordering of elements required to maintain the unit-stride in the transformed code may conflict with the ordering necessary to maintain unit-stride with respect to a partitionable loop. In such cases, the accesses from the partitionable loop are not optimized. If multiple partitionable loops access an array with unit-stride, to optimize all accesses it is necessary to partition the loops in a similar way in order to obtain a consistent ordering of the array elements (Section 5.1).

2.8. Executor Code

To execute the original computation on each process, the code is transformed such that the modified code accesses local arrays for all data arrays, and uses values stored in local arrays for loop bounds, conditionals, and array-access expressions. Listing 2 shows the modified code obtained from Listing 1. Loop bounds of partitionable loops are based on the number of iterations n that are mapped to a process. The loop bounds of loop j are read from arrays lb and ub. Accesses to local data arrays x1 and A1 are determined by values in arrays i,x,j and Ωa,j. In addition, communication calls are inserted to satisfy the producer-consumer relationship due to array x.

3. Partitioning the Computation

The computation is partitioned by considering the iteration-data affinity. To model this affinity, we use a hypergraph \( H = (V, \mathcal{N}) \) where \( V \) is the set of vertices and \( \mathcal{N} \) is the set of nets [Strout and Hovland 2004]. Iterations of all partitionable loops are represented by separate vertices \( i \in V \). Each accessed element of a data array is represented by a net \( j \in \mathcal{N} \), whose pins are the iterations that access the data element. The hypergraph is subjected to a multi-constraint partitioning to (1) partition the iteration of each of the partitionable loop in a load-balanced manner, and (2) minimize communication required for producer-consumer relationships between partitionable loops.

3.1. Achieving Load Balancing

Each vertex is associated with a vector of weights \( w_i \) of size equal to the number of partitionable loops. A vertex that represents an iteration of partitionable loop \( k \) has the \( k \)-th element as 1, with all other elements being 0. The weight of a set of vertices \( P \) is defined as \( W_P = \sum_{i \in P} w_i \). If \( P_n \) (where \( n \in [0, N) \)) are the partitions generated, load balance for every partitionable loop is achieved by applying the constraint \( P_n \leq P_{avg}(1 + \epsilon) \), where \( P_{avg} = P_V / N \); \( \epsilon \) is the maximum load imbalance tolerated.
3.2. Minimizing Communication

Each net in \( N \) is also associated with a weight \( c_j \), whose value is the same as the size of the data represented by the net. For each partition \( P_n \), the set of nets that have pins in it can be divided into two disjoint subsets. Nets that have pins only in \( P_n \) are \textit{internal} nets, \( I_n \), and nets with pins in other partitions are \textit{external} nets, \( E_n \). Each external net represents a data element that is accessed by more than one process. One of the partitions is the owner of the data, and the other partitions have corresponding ghosts. To minimize communication, the number of ghost cells needs to be minimized, along with the number of partitions \( \lambda_j \) that have a ghost for the data element represented by \( j \in N \). This is achieved by minimizing the \textit{cut-size} \( \Pi_n \) for each partition defined as \( \Pi_n = \sum_{j \in E_n} c_j (\lambda_j - 1) \). The hypergraph is subjected to a min-cut partitioning, under the load-balance constraints specified above \cite{CatalyurekAykanat2009}.

4. Identifying Partitionable Loops

This section describes the structure and automatic detection of partitionable loops in a given computation. The analysis assumes that the input code is consistent with the grammar described in Figure 3. Computations can consist of loops, conditional statements, and assignment statements. Each loop must have a unique iterator, which is referenced only in the loop body and is modified only by the loop increment expression (with unit increment). An assignment could use the standard assignment operator, or an update operator \( \text{op=} \) where \( \text{op} \) is a commutative and associative binary operator. Loop bounds, conditional expressions, right-hand sides of assignments, and index expressions are side-effect-free expressions built from iterators, scalar variables, array access expressions, and various operators (e.g., arithmetic and boolean) and functions (e.g., from libraries). In an array access expression \( \text{arr} \[ \text{expr} \] \), the index expression \( \text{expr} \) can itself contain array access expressions. As discussed earlier, the arrays whose elements are used to compute this \( \text{expr} \) are indirection arrays. The grammar allows for multiple levels of indirection.

Algorithm 1 takes as input an AST corresponding to a \( \langle \text{Loop} \rangle \) from this grammar, and determines whether the loop is partitionable. Section 4.1 discusses how to perform analysis of indirection arrays and scalars; this analysis corresponds to lines 2–37 in the algorithm. Section 4.2, corresponding to line 38 in the algorithm, checks the parallelism of the loop. The overall approach for identifying a sequence of maximal partitionable loops (by applying Algorithm 1 several times) is described in Section 4.3.

4.1. Indirection Arrays and Indirection Scalars

The first step in the analysis of partitionable loops is to determine the variables corresponding to indirection arrays and indirection scalars. As mentioned before, these are variables whose values (or in case of arrays, the values stored in them) are, directly or indirectly, used to compute loop bounds, values of conditional expressions, or index expressions of array access expressions.
Algorithm 1: CheckForPartitionableLoop($A$)

Input: $A$: AST of loop satisfying the grammar in Figure 3
Output: is\_partitionable: Boolean flag set to true if $A$ is partitionable
$I_A$: Set of indirection arrays
$I_S$: Set of indirection scalars
$I$: Set of loop iterators

begin

$I_A = \emptyset \; ; I = A.\text{Iterator} \; ; \text{is\_partitionable} = \text{true} \; ;$

foreach $s \in \text{GetLoopStmts}(A.\text{Body})$ do

$I = I \cup \; s.\text{Iterator} \; ;$

foreach $s \in \text{GetLoopStmts}(A.\text{Body})$ do

$I_A = I_A \cup \text{GetAllArrayVariables}(l) \cup \text{GetAllArrayVariables}(u) \; ;$

$I_S = I_S \cup \{ \text{GetAllScalars}(l) \cup \text{GetAllScalars}(u) - I \} \; ;$

endforeach

endforeach

foreach $s \in \text{GetIfStmts}(A.\text{Body})$ do

$c = s.\text{Cond} \; ;$

$I_A = I_A \cup \text{GetAllArrayVariables}(c) \; ;$

$I_S = I_S \cup \{ \text{GetAllScalars}(c) - I \} \; ;$

endforeach

foreach $s \in \text{GetAssignStmts}(A.\text{Body})$ do

foreach $e \in \text{GetArrayRefExprs}(s)$ do

$c = e.\text{IndexExpr} \; ;$

$I_A = I_A \cup \text{GetAllArrayVariables}(c) \; ;$

endforeach

endforeach

worklist = $I_S \; ;$

while $\neg \text{IsEmpty}(\text{worklist})$ do

$v = \text{RemoveElement}(\text{worklist}) \; ;$

foreach $s \in \text{GetAssignStmts}(A.\text{Body})$ do

if $\text{IsScalar}(s.\text{LHS}) \land s.\text{LHS} == v$ then

$c = s.\text{RHS} \; ;$

$I_A = I_A \cup \text{GetAllArrayVariables}(c) \; ;$

$S = \text{GetAllScalars}(c) - I \; ;$

endforeach

foreach $u \in S$ do

if $u \notin I_S$ then

$I_S = I_S \cup \{ u \} \; ; \text{AddElement}(\text{worklist}, u) \; ;$

endforeach

foreach $s \in \text{GetAssignStmts}(A.\text{Body})$ do

if $\neg \text{IsScalar}(s.\text{LHS}) \lor s.\text{LHS} \notin I_S$ then

$c = s.\text{RHS} \; ;$

$S = \text{GetAllScalars}(s.\text{RHS}, \text{IGNORE_INDEX_EXPR}) \; ;$

if $S \cap I_S \neq \emptyset$ then

is\_partitionable = false \; ;$

$S = \text{GetAllArrays}(s.\text{RHS}, \text{IGNORE_INDEX_EXPR}) \; ;$

if $S \cap I_A \neq \emptyset$ then

is\_partitionable = false \; ;$

endforeach

endforeach

is\_partitionable = \text{CheckForParallelism}(A.\text{Body}, I_S, I_A, I) \; ;$

return [is\_partitionable, I_S, I_A, I] \; ;$

end

To identify such variables, all loop bounds, conditionals, and array index expressions are analyzed. All array variables appearing in them are added to the set of indirection array variables. All scalars appearing in such expressions are added to the set of indirection scalar variables. In addition, such scalars are also added to a worklist.

Following this, for every scalar in the worklist, the right-hand side expression of each assignment statement which assigns to this scalar is analyzed. All arrays appearing on this right-hand side are added to the set of indirection arrays. All scalars that appear on the right-hand side and are not
already in the set of indirection scalars are added to that set and are also added to the worklist. Once all statements that assign to the scalar have been analyzed, it is removed from the worklist. These steps are repeated until the worklist becomes empty. The algorithm is guaranteed to terminate since a scalar is added to the worklist at most once, and for every iteration, an item is removed from the worklist. Upon termination, all indirection scalars and indirection arrays within the loop have been determined.

The remaining variables, apart from loop iterators, are categorized as data scalars and data arrays. For every such variable in the original computation, there is a corresponding variable in the transformed code that represents the local copy to be used by the executor. To simplify the presentation, we describe (in Section 5.5) a scheme where the executor code does not contain any indirection arrays/scalars that appear in the original code. Since the executor must compute the same values for data arrays/scalars as the original code, the loops targeted for transformation should not use values of indirection arrays/scalars to compute values stored in data arrays/scalars. Similarly, iterator values should not affect (directly or indirectly) values written to data arrays/scalars. Section 4.4 outlines how to handle cases where this property does not hold.

To check for this property, every assignment in the target loop body is examined. If the left-hand side is a data scalar or a data array access expression, the right-hand side expression is analyzed further. Ignoring all index expressions (i.e., \( expr \) in \( arr[expr] \)), any reference to a loop iterator, an indirection scalar, or an indirection array indicates that the property does not hold.

The above analysis ensures that scalars and arrays within the computation can be separated into two disjoint categories, one whose values completely determine the control-flow or data-access patterns, and another which contains variables whose values are the inputs and outputs of the computation. The grammar presented in Figure 3 can be modified to reflect this separation, and is presented in Figure 4. This new grammar defines the syntactic structure of partitionable loops and is used by the algorithms for code analysis/generation described in the rest of the paper.

4.2. Parallelism of the Target Loop

The final property to be checked is that the target loop is parallel, except for dependences due to reductions. Since the transformation scheme generates a parallel inspector, the control-flow and data-access pattern for a given iteration of the target loop must not depend on any of the previous iterations. This can be ensured if

- All indirection arrays are read-only within the loop
- For any indirection scalar \( s \) modified within the loop body, any use of \( s \) reads a value that was written to \( s \) in the same iteration

Fig. 4. Syntactic structure of partitionable loops
The second condition ensures that there are no inter-iteration dependences arising due to indirection scalars. Both properties can be easily checked statically.

To ensure that there are no dependences due to data arrays

- A data array variable \( \langle \text{DArray} \rangle \) can appear either only on the right-hand side of \( \langle \text{DAssignment} \rangle \)s, or only on the left-hand side, but not both
- A \( \langle \text{DArray} \rangle \) can appear on the left-hand side of multiple \( \langle \text{DAssignment} \rangle \)s, but all those statements must use the same \( \langle \text{AssignOp} \rangle \)

Finally, it has to be ensured that there is no dependence due to any data scalar. If a data scalar satisfies the same condition as the one for \( \langle \text{DArray} \rangle \)s listed above, then the inter-iteration dependence caused by updates to this scalar can be handled by the transformation scheme described later. Scalars that do not satisfy this property, might still not result in an inter-iteration dependence, if each use of the scalar reads a value that was assigned to it in the same iteration of the loop (similar to the property satisfied by \( \langle \text{IScalar} \rangle \)).

These constraints ensure that the only inter-iteration dependences are either output dependences due to data scalars/arrays assigned to by multiple iterations of the loop, or updates to such variable using associative (and commutative) update operators. The following sections provide descriptions of how these dependences are handled in a parallel execution of the loop.

A loop which satisfies all these properties is valid input for the transformations described in subsequent sections.

### 4.3. Finding Sequences of Maximal Partitionable Loops

Algorithm 1 takes as input one loop from the grammar in Figure 3, and decides whether the loop is partitionable. Suppose we are given an AST based on this grammar, with several loops (disjoint and/or nested within each other). It is desirable to identify partitionable loops that are as large as possible, as well as sequences of such loops that can be optimized together (by inserting communications calls between pairs of consecutive loops).

Suppose that the input program AST is an \( \langle \text{ElementList} \rangle \), i.e., a sequence of loops, conditionals, and assignments. Within this element list, each maximal sequence of consecutive \( \langle \text{Loop} \rangle \) nodes can be identified. For each of those loops, Algorithm 1 decides if it is amenable to transformation. Based on the results of the analysis, each maximal sequence of partitionable loops can be determined. Finally, it is checked if there is an intersection between the set of indirection arrays of one loop in the sequence with the set of data arrays of another loop. If the intersection is not \( \phi \), then these two loops cant be part of the sequence of partitionable loops, since the inspectors of a sequence of loops are executed in sequence too (as shown in Section 6). A similar check is performed to ensure that there is no intersection between the set of data scalars for a loop and the set of indirection scalars of another. This sequence is used as input to the code generation scheme described in Section 5 and Section 6.

To identify sequences of partitionable loops that are not at the top level of the input AST, the branches of conditional statements and the bodies of non-partitionable loops at the top level are analyzed recursively. These branches and bodies themselves are \( \langle \text{ElementList} \rangle \) (recall the grammar in Figure 3). Clearly, partitionable loops identified by this approach are not nested within each other. While in general it is possible to transform all such sequences of partitionable loops, for the benchmarks and applications described in Section 9, only the sequences that were closest to the top level of the AST were transformed.

### 4.4. Possible Generalizations

One of the restrictions described in Section 4.1 was that an assignment statement which assigns to a data scalar or data array element must not contain a reference to an indirection scalar or indirection array outside of index expressions. To relax this constraint, for every such occurrence, a new array is used which stores the value of the expression involving the indirection array/scalar. The inspector evaluates the part of the right-hand side that references the indirection scalar/array and stores it in a
temporary array. This new array is treated as a read-only data array and used to replace the expression involving the indirection array/scalar in the executor code. Substituting the same expression in the original computation with a read from this new data array would result in the loop having a syntactic structure as shown in Figure 4. Therefore, all algorithms discussed later in paper would still be applicable.

A similar restriction from Section 4.1 was that the value of a data array/scalar should not depend on the iterator value for the partitioned loop. This constraint could be removed, with the help of a code generation scheme similar to the one outlined above.

5. Inspector and Executor: Functionality and Code Generation

After a loop has been classified as a partitionable loop, the code for the inspector and the executor is generated by analyzing its loop body. This section describes the tasks performed at run time by the inspector and the executor, along with the compile-time algorithms to generate the corresponding inspector/executor code for a single partitionable loop.

The inspector executes in three phases:
— Phase I: Build and partition the hypergraph by analyzing the data elements touched by the iterations of all partitionable loops; allocate local copies for all data arrays based on the iterations assigned to each process.
— Phase II: Compute the sizes of the arrays needed to replicate the control-flow and array-access patterns.
— Phase III: Populate these arrays with appropriate values.
Each of these phases are elaborated below.

5.1. Phase I: Hypergraph Generation

5.1.1. Run-time Functionality

The inspector analyzes the computation and generates the corresponding hypergraph. For Listing 1, a portion of the inspector that generates the hypergraph is shown in Listing 3. The inspector code for this phase contains only the (IAssignment)s, (Loop)s and (If)s from the original computation.

At the start of the computation, all arrays are block-partitioned across the processes. (The approach can be easily adapted to other partitioning schemes for the initial data—e.g., cyclic and block-cyclic.) Each process analyzes a block-partitioned subset of the original iterations (represented by \([k_{start}, k_{end})\) for loop \(k\) and \([i_{start}, i_{end})\) for loop \(i\)) and therefore computes only a part of the hypergraph. For each iteration of the partitionable loop executed on a process, a vertex is added to represent it in the hypergraph, by calling AddVertex. This function takes as input a compile-time integer identifier which uniquely identifies the partitionable loop being analyzed, (e.g., \(id_{k}\) loop \(k\)) and returns a handle to the vertex added to the hypergraph.

For every data array element that is accessed by this iteration, the vertex returned previously is added as a pin to the corresponding net. For example, AddPin\((id_{y\ array}, i, v_{i}, true)\) adds vertex \(v_{i}\) as a pin to the net for the \(i\)-th element of array \(y\). The last argument, of boolean type, specifies that the element is accessed by an expression with a unit stride. Here \(id_{y\ array}\) is a unique compile-time integer identifier for array \(y\).

Since arrays are block-partitioned, it might not be possible to evaluate each array-access expression since values in indirection arrays might not be local to the process. Thus, every access to an indirection array is guarded by the function \(is\_known\) which returns true if the value needed is known on the current process and false otherwise, with the element flagged as being requested. After the block of iterations have been analyzed, all outstanding requests are serviced. On re-analyzing these iterations, \(is\_known\) for those elements would evaluate to true, and the value can be obtained via function get_elem. Repeated analysis is performed until \(is\_known\) returns true for all accessed elements. In this phase, there is no communication due to the values of the data array elements, since these values are not used to index other arrays. Multiple levels of indirection are handled through successive execution of the outer block-partitioned loop, as shown in Listing 3.
Since values assigned to indirection scalars might depend on indirection arrays, these values might not be known on a process either. A \textit{shadow scalar} is associated with every indirection scalar. If the right-hand side of an assignment to an indirection scalar cannot be computed on a process, the shadow scalar associated with the scalar referenced on the left-hand side is set to false. Statements which use the values of these indirection scalar are guarded to check the state of the corresponding shadow scalar. For example, \texttt{shadow\_xindex} is the shadow scalar corresponding to \texttt{xindex}. It is set to false if the value of \texttt{col[j]} is not known on a process. Since this value is used by \texttt{AddPin(id\_x\_array,xindex,\texttt{vi},false)}, this statement is guarded with a check of the state of \texttt{shadow\_xindex}.

The portions of the hypergraph built by each process are combined to compute the complete iteration-to-data affinity. The hypergraph is partitioned \(P\) ways as described in Section 3, where \(P\) is the number of processes. Each process is assigned a unique partition representing the iterations to be executed on it. The iterations are renumbered such that they form a contiguous set on each process, while maintaining the relative ordering of the iterations mapped to that process.

\subsection*{5.1.2. Code Generation at Compile Time}

The inspector code that achieves the functionality described above (e.g., the code in Listing 3) is generated automatically by the compiler. The compiler algorithm to generate the code to build the hypergraph is shown in Algorithm 2. It traverses the statements within the body of each partitionable loop.

For an \langle IAssignment \rangle in the original AST, an assignment statement is added to the inspector AST with the right-hand side modified to convert all array references to calls to function \texttt{get\_elem}. This AST modification is performed by function \texttt{ConvertArraysToFunctions}. In addition, the statement is guarded by a conditional statement to check that the values of all array elements or scalars are known on the current process. The expression to be used by the conditional is returned by function \texttt{GenGuards}.

The generation of the guard expression constructed by \texttt{GenGuards} is described in Algorithm 3. The algorithm takes as input an \langle IExpr \rangle. For expressions that are references to \langle IScalar \rangle, the condition checks if the corresponding shadow scalar is set to true. For expressions that are \langle IArray \rangle references, the condition contains two parts. The first part is a call to function \texttt{is\_known}, with the arguments being the unique identifier for the array variable (computed at compile time), and the index expression. The second part is generated by recursively analyzing the index expression, to ensure that the index expression itself can be computed. These two parts are combined using a logical \texttt{and} operator. Since the latter condition has to be checked before the former one, it is set as the first
Algorithm 2: CodeGenHyperGraph(\(H, ss\))

```plaintext
Input: \(H\): Hypergraph object
       \(ss\): Sequence of statements in the original AST
Output: \(A_H\): AST of the inspector code to generate hypergraph
       \(I\): \(\langle\text{IScalar}\rangle\)s defined in \(ss\)

begin
  \(A_H = \emptyset ; I = \emptyset\);
  foreach \(s \in ss\) in order of appearance do
    if IsAssignment(s) then
      \(b = \text{ConvertArraysToFunctions}(s.RHS)\);
      \(l_H = \text{NewIfStmt}() ; l_H . Cond = \text{GenGuards}(s.RHS)\);
      \(s_H = \text{NewAssignmentStmt}(s.LHS,b)\);
      \(s_H . Append(\text{SetShadowScalar}(s.LHS,\text{true}))\);
      \(I_L = \{s.LHS\} ; I_H . Then = s_H\);
    else if IsDAssignment(s) then
      \(l_H = \emptyset\);
      foreach \(d \in \text{GetDataArrayRefExprs}(s)\) do
        \(b = \text{ConvertArraysToFunctions}(d.\text{IndexExpr})\);
        \(s_H = \text{NewIfStmt}() ; s_H . Cond = \text{GenGuards}(d.\text{IndexExpr})\);
        \(t = \text{GenAddPinFn}(H,d.\text{Array},b,\text{IsUnitStride}(d.\text{IndexExpr}))\);
        if \(s_H . Cond == \emptyset\) then
          \(s_H . Then = t ; l_H . Append(s_H)\);
        else
          \(l_H . Append(t)\);
      end
    else if IsLoop(s) then
      \(l = s.\text{LowerBound} ; u = s.\text{UpperBound}\);
      \(b_l = \text{ConvertArraysToFunctions}(l)\);
      \(b_u = \text{ConvertArraysToFunctions}(u)\);
      \(s_H = \text{NewLoopStmt}(s.\text{Iterator},l,u)\);
      \([l_H . Body,I_L] = \text{CodeGenHyperGraph}(H,s.\text{Body})\);
      \(l_H = \text{NewIfStmt}() ; l_H . Then = s_H\);
      \(l_H . Cond = \text{NewAndCond}(\text{GenGuards}(l),\text{GenGuards}(u))\);
    else
      \(c_H = \text{ConvertArraysToFunctions}(s.\text{Cond})\);
      \(s_H = \text{NewIfStmt}() ; s_H . Cond = c_H\);
      \([s_H . Then,J_1] = \text{CodeGenHyperGraph}(H,s.\text{Then})\);
      \([s_H . Else,J_2] = \text{CodeGenHyperGraph}(H,s.\text{Else})\);
      \(l_H = \text{NewIfStmt}() ; l_H . Then = s_H\);
      \(I_L = I_L \cup J_2 ; l_H . Then = s_H\);
    end
    foreach \(v \in I_L\) do
      \(l_H . Else . Append(\text{SetShadowScalar}(v,\text{false}))\);
      \(A_H . Append(l_H) ; I = I \cup I_L\);
  end
end
return \([A_H,I]\);
```

operand in the \textit{and} expression. The short-circuit evaluation of C/C++ ensures that the \textit{is\_known} function is called only when the index expression can be evaluated.

Since a \textit{Loop} is executed only when the bounds are known, the loop iterator is always known within the loop body. Algorithm 3 returns \(\emptyset\) for such an expression. For expressions that are not \textit{BasicIndexExpr}, all children of the \textit{IExpr} are recursively evaluated and their guards are combined with the \textit{and} operator.

Algorithm 2 sets the conditional expression returned by \textit{GenGuards} as the condition of the guard statements. The statements to set the shadow scalar associated with the \(\langle\text{IScalar}\rangle\) to true is added to the true branch of the guard statement at line 8 of Algorithm 2, along with a statement to set it
Algorithm 3: GenGuards(e)

Input : e : ⟨IEExpr⟩ to be guarded
Output : c : Condition to be used for the guard statement

begin
if IsIScalar(e) then
    c = NewCheckEquality(e, true);
else if IsArrayRefExp(e) then
    cl = GenGuards(e.IndexExpr);
    cr = GenerateIsKnownFn(e.Array, e.IndexExpr);
    c = NewAndCond(cl, cr);
else
    c = ∅;

foreach d ∈ e.Children do
    c = NewAndCond(c, GenGuards(d));

return c;

to false in the false branch (at line 36 of the algorithm). Lines 10–15 of Listing 3 contain the code generated for the ⟨IStatement⟩ at line 8 of Listing 1.

For ⟨DAssignment⟩s in the original AST, for every reference to a data array, a call to function AddPin is generated by GenAddPinFn. Such a call takes as input (1) a compile-time integer identifier for the data array, (2) the index expression, and (3) a boolean flag which indicates whether the current expression is unit-stride with respect to a surrounding loop. (The optimized handling of unit-stride accesses is discussed in Sections 2.6 and 2.7.) The index expression used in the original AST is modified to convert all arrays references (all of which are ⟨IArray⟩s) to calls to get elem. Every statement is guarded to check that the index expression can be evaluated on a process. This guard is again generated by GenGuards. Lines 16–19 of Listing 3 are the statements generated for the ⟨DStatement⟩ at line 9 of Listing 1.

Upon encountering a ⟨Loop⟩, the statement is replicated in the inspector AST, with references to indirection arrays in the bounds replaced with calls to get elem. The loop body is generated by a recursive call to analyze the loop body in the original AST. This loop should be executed by the inspector only when the loop bounds can be computed on a process. Therefore, the loop statement is guarded by conditional statements to check for this (generated by GenGuards). A similar approach is employed for ⟨If⟩ statements: references to indirection arrays in the conditional expression are replaced by calls to get elem, and the branches are generated recursively. The statement is enclosed within guards to check that the conditional expression can be evaluated on a process.

It is possible that an indirection scalar is modified within an inner loop or within branches of conditional statements and used later within the partitionable loop. Such uses must be avoided when the loop/conditional statements were not executed due to the guards. Therefore, for all indirection scalars modified within the inner loop bodies or within branches of conditional statements, the shadow scalar must be set to false when the guard evaluates to false. The set of such scalars is returned by the recursive call that builds the loop body or the branches of the conditional statement. The statements to set these variables to false are added to the false branch of the guard statement for the corresponding loop or conditional statement, at lines 35–36 of Algorithm 2.

To support the optimizations of accesses from partitionable loops as discussed in Section 2.7, it might be necessary to ensure that multiple partitionable loops are partitioned the same way. To enforce this, the loop bounds of all such loops are checked at compile time. If they are the same, a single compile-time identifier is used to represent all of them. Therefore, at run time, AddVertex would map corresponding iterations of all these loops to the same vertex. In cases where the loop bounds are not the same, the accesses to data arrays that are unit-stride with respect to the partitionable loops are not optimized.
5.2. Initializing Local Data Arrays

After partitioning the hypergraph, Phase I of the inspector partitions the data. For a net that has all its pins in the same partition, the corresponding data element is assigned to the same process. If a net has pins in different partitions, the element is assigned to the process that executes the majority of the iterations that access this data. All other processes have a ghost location for that element. The local copy of the array consists of the elements that a process owns and the ghost locations for elements owned by other processes. This is done for all data arrays in the computation. For example, arrays $y_l$, $x_l$ and $A_l$ of Listing 2 are allocated at this time.

As described in Section 5.1, expressions that result in unit-stride accesses to a data array are identified at compile time. Elements accessed by such expressions (known at run time using the value of the last argument of `AddPin`) are laid out first in increasing order of their original position, followed by all other elements of the array accessed. This scheme maintains the contiguity of accesses within inner loops and partitionable loops in the transformed code, as outlined in Sections 2.6 and 2.7.

5.3. Phase II: Computing the Sizes of Local Access Arrays

5.3.1. Run-time Functionality

The next step is to determine the sizes of access arrays: arrays that are used to (1) store loop bounds of inner loops, (2) store the results of conditionals, and (3) store the indices of accessed data array elements. The sizes of these arrays depend on the expressions they represent. Array-access expressions that are unit-stride with respect to a surrounding loop would need an array of size equal to the number of invocations of that loop. For expressions that are not unit-stride with respect to any surrounding loop, loop-invariant analysis is performed to determine the innermost loop with respect to which the value of the expression changes. In the worst case, this might be the immediately surrounding loop. The size of the array needed to represent these expressions is the total number of iterations of that loop across all iterations of the partitioned loop mapped to a process. The size of arrays that store the bounds of an inner loop are the same as the number of invocations of the loop. For an array needed to store the values of a conditional, the number of times the `if` statement is executed should be counted.

Listing 4 shows the code for this phase of the inspector for the running example. The number of invocations of inner loop $j$ is tracked via counter `loop_j`. Counters `body_i` track the total number of times a loop body is executed. For conditional statements, `then_*` and `else_*` counters track the number of times the true or false branch of the statements are taken, with `if_*` counting the number of times the conditional is evaluated.

In addition to this counting, this phase ensures that a process has all values of indirection arrays needed to analyze the iterations mapped to it. Each process executes only the iterations mapped to it after partitioning, which may be different from those analyzed by this process when building the
body_i = 0; loop_j = 0; body_j = 0; body_k = 0;
for( k = 0 ; k < n ; k++ )
if( home(id_k_loop,k) == myid )
body_k++;
for( i = 0 ; i < n ; i++ )
if( home(id_i_loop,i) == myid ){
    lb_j[loop_j] = get_elem(id_ia_array,i);
    ub_j[loop_j] = get_elem(id_ia_array,i+1);
    for( j = lb_j[loop_j] ; j < ub_j[loop_j] ; j++ )
        xindex = get_elem(id_col_array,j);
    if( j == lb_j[loop_j] )
        o_a_j[loop_j] = j;
    i_x_j[body_j] = xindex;
    body_j++;
}
loop_j++;
body_i++;
}

Listing 5. Phase III of the inspector.
Algorithm 4: CodeToInitializeArrays(ss, L, C)

Input: ss: Sequence of statements in the original AST
L: Access arrays for index expressions, loop bounds, and conditional values
C: Counter variables

Output: AP: AST of inspector code to populate the access arrays

begin

A_P = \emptyset;

for each s \in ss in order of appearance do

if IsIStmt(s) then

b = ConvertArraysToFunctions(s.RHS);

l_P = NewAssignmentStmt(s.LHS, b);

else if IsDStmt(s) then

l_P = \emptyset;

foreach d \in GetDataArrayRefExprs(s) do

a = GetAccessArray(L.d.Array, d.IndexExpr);

c = GetCounterVariable(C.d.IndexExpr);

b = ConvertArraysToFunctions(d.IndexExpr);

e = NewArrayRefExpr(a, c);

s_P = NewAssignmentStmt(e, b);

if IsUnitStride(d.IndexExpr) then

l = GetLoop(c); s_P = GenIfFirstIter(c, s_P);

l_P.Append(s_P);

else if IsLoop(s) then

l = s.LowerBound; u = s.UpperBound;

b_l = ConvertArraysToFunctions(l); b_u = ConvertArraysToFunctions(u);

a_l = GetLowerBoundArray(L, l);

a_u = GetUpperBoundArray(L, u);

c = GetLoopCounterVariable(C, s);

e_l = NewArrayRefExpr(a_l, c);

e_u = NewArrayRefExpr(a_u, c);

l_P = NewAssignmentStmt(e_l, b_l);

l_P.Append(NewAssignmentStmt(e_u, b_u));

s_P = NewLoopStmt(s.Iterator, e_l, e_u);

s_P.Body = CodeToInitializeArrays(s.Body, L, C);

c_l = GetBodyCounterVariable(C, s);

s_P.Body.Append(NewIncrementStmt(c_l));

l_P.Append(s_P);

else if IsIf(s) then

l_P = ConvertArraysToFunctions(s.Cond);

a = GetConditionalArray(L, s);

c = GetConditionalCounterVariable(C, s);

e = NewArrayRefExpr(a, c);

s_P = NewIfStmt(); s_P.Cond = \emptyset;

s_P.Then = CodeToInitializeArrays(s.Then, L, C);

c_l = GetThenCounterVariable(C, s);

s_P.Then.Append(NewIncrementStmt(c_l));

s_P.Else = CodeToInitializeArrays(s.Else, L, C);

c_l = GetElseCounterVariable(C, s);

s_P.Else.Append(NewIncrementStmt(c_l));

l_P.Append(s_P);

l_P.Append(NewIncrementStmt(c_l));

else

return A_P;

end

the function returns the then_s or else_s counter associated with this intervening conditional statement, depending on whether the \( \langle \text{DA} \rangle \) is in the true or the false branch, respectively.

Further, if the index expression is unit-stride with respect to a surrounding loop, the statement is enclosed within an if statement (generated by GenIfFirstIter) which checks if the value of the loop iterator is the same as that stored in the lower-bound array. For example, in Listing 5, o_a, j stores the value of expression j used to access array a, and is enclosed within an if statement that is true for the first loop iteration.
Algorithm 5: GenerateExecutor($D, L, C, ss$)

Input: $D$: Local data arrays
$L$: Access arrays for index expressions, loop bounds, and conditional values
$C$: Counter variables

InOut: $ss$: Sequence of statements in the original AST

begin
foreach $s \in ss$ in order of appearance do
  if IsDAssignment($s$) then
    foreach $d \in GetDataArrayRefExprs(s)$ do
      ReplaceWithLocalArray($d$.Array, $D$)
      $c = GetCounterVariable(C, d$.IndexExpr$)$;
      $a = GetAccessArray(L, d$.Array$).IndexExpr$);
      $e = NewArrayRefExpr(a, c)$;
      if IsUnitStride($d$.IndexExpr) then
        $l = GetLoopStatement(c)$;
        $o = NewVariable()$;
        $e_l = NewArrayRefExpr(GetLowerBoundArray(L, l), c)$;
        $s_E = NewAssignmentStmt(o, NewSubtractExpr(e, e_l))$;
        InsertBefore($l$, $s_E$);
        ReplaceExpression($d$.IndexExpr, NewAddExpr(l.Iterator, o));
      else
        ReplaceExpression($d$.IndexExpr, $e$);
    end if
  end if
  if IsLoop($s$) then
    $l = s$.LowerBound$; u = s$.UpperBound$;
    $a_l = GetLowerBoundArray(L, s)$; $a_u = GetUpperBoundArray(L, s)$;
    $c = GetLoopCounterVariable(C, s)$;
    $e_l = NewArrayRefExpr(a_l, c)$; ReplaceExpression(s.LowerBound, e_l)$;
    $e_u = NewArrayRefExpr(a_u, c)$; ReplaceExpression(s.UpperBound, e_u)$;
    GenerateExecutor($D, L, C, s$.Body$)$;
    $c_0 = GetBodyCounterVariable(C, s)$; $s_p$.Body.Append(NewIncrementStmt(c_0)$);
    $s$.Append(NewIncrementStmt(c))$;
  end if
  else
    $a = GetConditionalArray(L, s)$;
    $c = GetConditionalCounterVariable(C, s)$;
    $e = NewArrayRefExpr(a, c)$; ReplaceExpression(s.Cond, e)$;
    GenerateExecutor($D, L, C, s$.Then$)$;
    $c_0 = GetThenCounterVariable(C, s)$; $s_p$.Then.Append(NewIncrementStmt(c_0)$);
    GenerateExecutor($D, L, C, s$.Else$)$;
    $c_0 = GetElseCounterVariable(C, s)$; $s_p$.Else.Append(NewIncrementStmt(c_0)$);
    $s$.Append(NewIncrementStmt(c))$;
  end if
end for

Upon encountering a $\langle$Loop$\rangle$ node, statements to store the current lower/upper bounds of the loop in arrays are added to the inspector AST. Following this, a loop statement is added, with bounds modified to read from the array locations which were assigned to. The body of the loop is generated by processing recursively the loop body in the original computation. $\langle$If$\rangle$ statements are handled similarly. Statements to store the value of the conditional are added to the inspector AST, followed by a new conditional statement whose branches are computed by recursively traversing the true and false branches in the original code.

Having populated all access arrays with the original values of the expressions they represent, these values are now modified to point to the corresponding locations in the local copies of the data arrays being accessed.

5.5. Executor Code

After all phases of the inspector, the loop iterations and data arrays have been partitioned among the processes. All access arrays have been initialized with values that point to the appropriate locations in the local data arrays.
The executor code is similar to the original code. All counter variables are first reset to 0. The lower and upper bounds of the partitioned loops are set to 0 and the number of assigned iterations, respectively. The body of the executor is generated by Algorithm 5. The AST of the original loop is traversed. ⟨IStatement⟩s are not replicated in the executor since the control-flow and array-access patterns are explicitly represented through access arrays. For ⟨DStatement⟩s, all accesses to data arrays are replaced with accesses to the corresponding local data arrays. The index expressions used to access these arrays are also modified as necessary.

For expressions that are unit stride with respect to an inner loop, the index expression is the sum of the loop iterator and the value stored in an offset variable. This offset variable is initialized to the value stored in the access array associated with the index expression, subtracted with the lower bound of the loop. Since the access array stores the location of the first element of the array accessed within the loop, adding the iterator value to this offset allows accessing the local arrays in a manner consistent with the original computation, and in a contiguous manner. Such an expression allows for subsequent optimizations such as vectorization and prefetching, which rely on this property. To the best of our knowledge, no previously proposed compiler approaches for I/E code generation ensure this highly-desirable property.

Upon encountering ⟨Loop⟩s and ⟨If⟩s, a corresponding loop or conditional statement is added to the executor AST, with the bounds/conditionals modified to read from arrays that were populated in Phase III of the inspector. The loop bodies and the branches of conditionals for the newly created statements are generated recursively.

Once the executor code has been generated for all partitionable loops, communication calls to update the ghosts used within a loop are inserted before that loop in the executor AST. Communication calls to update the owner with values in all ghosts location are also inserted after the loop. The communication scheme is described below.

5.6. Communication Between Processes

Elements of local data arrays consist of both owned and ghost locations. Phase I of the inspector initializes them to their original values. To maintain correctness of the parallel execution, ghost cells for arrays that are read within a partitionable loop are updated before the start of the loop, and ghost cells of arrays that are updated are communicated to the owner after the loop execution. For cases where a partitionable loop assigns values to array elements instead of updating them (i.e., the initial array values are not read in the loop), the ID of the process that owns the last iteration of the partitionable loop which writes to the data element is computed by the inspector. The value of the ghost location from this process is used to overwrite the value at the owner.

The communication pattern used to update ghosts is similar to the MPI_Alltoallv collective. As the number of partitions increases, every process has to communicate with only a small number of other processes. Therefore, the communication costs are reduced by using one-sided point-to-point communication APIs provided by ARMCI [Neiplocha et al. 2006].

6. Overall Code Generation Approach

While the previous section described the algorithms to generate the three phases of the inspector and the executor for a single partitionable loop, this section outlines the complete approach used to generate the inspector/executor code for a sequence of partitionable loops, identified as outlined in Section 4.3. Algorithm 6 describes this process.

The first step is to generate the code to initialize the hypergraph, using function InitHyperGraph. Following this, the Phase I code for all loops in the sequence is generated and appended to the inspector AST. A new loop is created, with the bounds from the original loop modified to be block-partitioned. A call to function AddVertex is generated by function GenAddVertex and added to the new loop body. The rest of the loop body is generated by traversing the AST of the original partitionable loop using Algorithm 2. After all partitionable loops have been processed, all the generated inspector loops are enclosed within a do-while loop with the conditional being a call to DoneGraphGen.
Algorithm 6: CodeGenInspectorExecutor(\(\mathcal{P}\))

\[
\begin{array}{l}
\text{InOut} : \mathcal{P} : \text{Sequence of partitionable loop ASTs} \\
\text{Output} : A_I : \text{Code for the inspector} \\
\text{begin} \\
\quad \mathcal{H} = \text{InitHyperGraph}() ; A_I = \emptyset ; A_H = \emptyset ; \\
\quad \text{foreach } p \in \mathcal{P} \text{ in order do} \\
\quad \quad \{ [L_1,S_1] = \text{BlockIterationBounds}(p, \text{LowerBound}, \text{UpperBound}) ; \\
\quad \quad \quad L_1.\text{Body} = \text{GenAddVertex}(\mathcal{H}, p, \text{Iterator}) ; \\
\quad \quad \quad S_1.\text{Body} = \text{CodeGenHyperGraph} (\mathcal{H}, p, \text{Body}) ; \\
\quad \quad \quad A_H.\text{Body}.\text{Append} (L_1) ; A_H.\text{Body}.\text{Append} (S_1) ; \\
\quad \quad \quad A_H.\text{Body}.\text{Append} (\text{DoWhile} (\text{DoneGraphGen}(), A_H) ; A_I.\text{Body}.\text{Append} (A_H) ; \\
\quad \quad \quad A_I.\text{Body}.\text{Append} (\text{CodeToPartitionIterations} (\mathcal{H})) ; \\
\quad \quad \quad \mathcal{D} = \text{CodeToAllocateLocalData} (\mathcal{H}) ; A_I.\text{Body}.\text{Append} (\mathcal{D}) ; \\
\quad \quad \quad \mathcal{C} = \text{DeclareCounterVariables} (\mathcal{P}) ; A_I.\text{Body}.\text{Append} (\mathcal{C}) ; \\
\quad \quad \quad A_C = \emptyset ; \\
\quad \quad \text{foreach } p \in \mathcal{P} \text{ in order do} \\
\quad \quad \quad L_2 = \text{NewLoopStmt} (p, \text{Iterator}, \text{LowerBound}, \text{UpperBound}) ; \\
\quad \quad \quad L_2.\text{Body} = \text{CodeGenHyperGraph} (\mathcal{H}, p, \text{Body}) ; \\
\quad \quad \quad A_C.\text{Body}.\text{Append} (L_2) ; \\
\quad \quad \quad L_3 = \text{NewIfStmt} () ; L_3.\text{Cond} = \text{GenerateIsHome} () ; \\
\quad \quad \quad L_3.\text{Then} = \text{CodeGenHyperGraph} (\mathcal{H}, p, \text{Body}) ; \\
\quad \quad \quad A_P.\text{Body}.\text{Append} (L_3) ; \\
\quad \quad \quad A_P.\text{Body}.\text{Append} (\text{DoWhile} (\text{DoneCounters}(), A_C) ; A_I.\text{Body}.\text{Append} (A_C) ; \\
\quad \quad \quad \mathcal{I} = \text{CodeToAllocateAccessArrays} (\mathcal{C}) ; A_I.\text{Body}.\text{Append} (\mathcal{I}) ; \\
\quad \quad \quad A_P = \emptyset ; \\
\quad \quad \text{foreach } p \in \mathcal{P} \text{ in order do} \\
\quad \quad \quad L_3 = \text{NewIfStmt} () ; L_3.\text{Cond} = \text{GenerateIsHome} () ; \\
\quad \quad \quad L_3.\text{Then} = \text{CodeGenHyperGraph} (\mathcal{H}, p, \text{Body}) ; \\
\quad \quad \quad A_P.\text{Body}.\text{Append} (L_3) ; \\
\quad \quad \quad A_P.\text{Body}.\text{Append} (\text{DoWhile} (\text{DoneCounters}(), A_C) ; A_I.\text{Body}.\text{Append} (A_C) ; \\
\quad \quad \quad \mathcal{I} = \text{CodeToAllocateAccessArrays} (\mathcal{C}, \mathcal{D}, \mathcal{I}) ; \\
\quad \quad \quad A_I.\text{Body}.\text{Append} (\mathcal{I}) ; \\
\quad \quad \text{foreach } p \in \mathcal{P} \text{ in order do} \\
\quad \quad \quad \text{CodeGenHyperGraph} (\mathcal{H}, p, \text{Body}) ; \\
\quad \quad \quad \text{InsertCommunicationCode} (p) ; \\
\quad \quad \quad \text{return} A_I ; \\
\end{array}
\]

Next, the code to partition the hypergraph and to allocate local arrays is appended to the inspector AST. Following this, the counter variables required within all elements of \(\mathcal{P}\) are initialized to 0. For every partitionable loop, a corresponding loop in the inspector AST is generated, with the same loop bounds as the original loop. The body for this loop is a conditional statement to check whether the current iteration is local to a process. The conditional is generated by function \textit{GenerateIsHome}. The true branch of this conditional statement is generated by analyzing the body of the partitionable loops using function \textit{CodeGenHyperGraph}, which implements the functionality described in Section 5.3.2. Once all partitionable loops have been processed, the loops generated in this phase are enclosed within a \texttt{do-while} loop with the conditional being a call to function \textit{DoneCounters}.

At this stage, the code to allocate all access arrays is appended to the inspector AST. Following that, Phase III code for all partitionable loops are generated using Algorithm 4 (at line 25 of Algorithm 6). Finally, function \textit{CodeGenHyperGraph} generates the code to modify the values stored in access arrays that are used to access elements of local data arrays. This code is also appended to the inspector AST.

The ASTs of the original loops are modified in place to create the executor code, as shown in Algorithm 5. Following this, the code to perform the communications of ghost values (described in Section 5.6) is inserted before and after every partitionable loop.

In addition to the in-place code modifications that create the executor code, the algorithm produces the inspector code (line 32). This inspector performs Phase I for all partitionable loops, fol-
lowed by Phase II for those loops, and finally Phase III. The generated inspector code is, by default, placed just before the executor code. To improve the performance of the generated code, it may be useful to amortize the inspector cost by hoisting it out of surrounding loops. The analysis required to decide the optimal placement of inspector code has been described previously elsewhere [Eswar et al. 1993].

7. Optimizations for Affine Code
When some partitionable loops in a program are completely affine, i.e., loop bounds and array-access expressions are strictly affine functions of surrounding loop iterators (and program parameters), the code generation described earlier is correct but introduces unnecessary overhead. For such loops, inspector code is unnecessary since control-flow and array-access patterns can be characterized statically. For example, if loop $i$ in Listing 1 were of the form shown in Listing 6, where matrix $A$ is dense and hence not stored in the CSR format, the computation can be analyzed statically.

A regular distribution of the iterations of partitionable loops (such as block, cyclic, or block-cyclic) is used instead of the hypergraph partitioning scheme. For data arrays accessed only through affine expressions, local copies can be computed as footprints of the partitioned iterations. Polyhedral code generation can be used as usual, with the number of iterations on a given process treated as a parameter. Ghosts can be computed as the intersection of the process footprints. For all expressions used to access such arrays, the statements generated at line 13 of Algorithm 4 are removed from the inspector AST. The corresponding expressions in the executor code are the affine expressions used in the original code. If loop bounds of inner loops are also affine expressions, these would be used in the executor instead of storing the loop bounds in arrays.

For all other arrays (accessed through non-affine expressions), the inspector code is used to compute a partitioning of the arrays and to create arrays that replicate the data-access patterns, as explained in Section 5. If all arrays are accessed through affine expressions, the inspector is rendered unnecessary and is discarded. In this case, polyhedral techniques can be used to generate the executor code. Listing 6 shows the code obtained for the executor by following this approach.

8. Interfacing with External Parallel Libraries
A common model for development of engineering applications is to first develop a sequential version of the code and then manually develop a parallel version. The previous sections have described how automatic parallelization can be achieved for a class of irregular loop computations when the entire code is written by the user. However, many application codes invoke libraries for commonly used functionality such as the solution of a sparse linear system of equations. In this section we address such codes and develop an approach to parallelization using parallel library frameworks such as PETSc [Balay et al. 2012] and Trilinos [Heroux et al. 2003], with minimal user intervention.

These libraries implement a range of library functions with highly optimized parallel implementations for shared and distributed memory systems. They also provide abstract interfaces that isolate the end user from details of the internal representations of data structures and are primarily targeted at users who develop MPI-based parallel applications. The typical model of usage is that the parallel MPI processes of the user application provide the needed input data to the libraries in a distributed
fashion by every process invoking interface functions (for example, the non-zero elements of a matrix, with each process providing the data for a subset of elements) to initialize values of a global data structure. Then, all processes invoke a collective call to a library function (for example, call a sparse linear system solver). To extract the results, each process uses the interface provided by the library to copy data from the global data structures to local data structures used by the application code.

In developing the automatic parallelization approach, we utilize a key property of the library APIs used by frameworks such as PETSc: they provide a uniform user interface whether the invoking application is sequential or parallel, with a global view of data structures. For example, although internally the nonzero elements of a PETSc sparse matrix are partitioned and distributed among parallel MPI processes and locally indexed within the library code, the user interface is in terms of a global row/column indexing of the elements.

We use an example to illustrate the approach. Consider the loop shown in Listing 7. It is representative of one of the loops from an application to solve radiative heat-transfer problems over a 2D physical domain using an unstructured mesh [Ravishankar et al. 2010]. The differential equations to be solved are discretized using Finite-Volume method into a set of linear equations which can be represented as \( \mathbf{A} \bar{\mathbf{x}} = \bar{\mathbf{b}} \), where \( \bar{\mathbf{x}} \) and \( \bar{\mathbf{b}} \) are vectors of size equal to number of cells in the unstructured mesh (say \( \text{ncells} \)), and \( \mathbf{A} \) is a matrix of size \( \text{ncells} \times \text{ncells} \). Loop 1 populates the values of matrix \( \mathbf{A} \) and vector \( \bar{\mathbf{b}} \) using the PETSc functions \texttt{MatSetValues} and \texttt{VecSetValues}, respectively. The same function can be used by the sequential or parallel code with a global view of the matrices and vectors. The data structures for these matrices and vectors in a distributed memory environment is managed by PETSc in a manner opaque to the user. The system of equations is then solved by calling function \texttt{KSPSolve}. Finally, the result vector \( \bar{\mathbf{x}} \) is read into the data array \( \text{phic} \) through the call \texttt{VecGetArray}. 

```
Mat mat_a //PETSc Matrix object , Size : ncells x ncells
Vec vec_b,vec_x //PETSc Vector Object : Size : ncells
#pragma block_partition phic
while( residual < tol ){
  //...Computation not shown...
  for( i = 0 ; i < ncells ; i++ ){  
    anbs = 0.0;
    aps = 0.0;
    sc = 0.0;
    for( j = ia_cv[i], j < ia_cv[i+1] ; j++ ){    
      neighb = lcc[j];
      aps += ...
      anbs = ...
      sc += ...
    }
    #pragma recreate_exp i
    #pragma recreate_exp neighb
    MatSetValues(mat_a,1,&i,1,&neighb,anbs,ADD_VALUES);
    aps += ...
    sc += ...
    #pragma recreate_exp i
    MatSetValues(mat_a,1,&i,1,&i,aps,ADD_VALUES);
    #pragma recreate_exp i
    VecSetValues(vec_b,1,&i,sc,INSERT_VALUES);
  }
  KSP_Solve(ksp_solve,mat_a,vec_r,vec_x);
  VecGetArray(vec_x,phic);
  VecRestoreArray(vec_x,phic);
  //...Computation not shown...
}
```

Listing 7. Representative loop for P3 with PETSc
Mat mat_a //PETSc Matrix object , Size : ncells x ncells
Vec vec_b,vec_x //PETSc Vector Object : Size : ncells

while( residual < tol ) {
  //...Computation not shown...
  for( il = 0 ; il < local_cells ; il++ ) {
    local_anbs = 0.0;
    local_aps = 0.0;
    local_sc  = 0.0;
    for( jl = lb_j[loop_j] ; jl < ub_j[loop_j] ; jl++ ){
      local_aps += ...
      local_anbs = ...
      local_sc  += ...
    }
    #pragma recreate_exp i exp_1[body_i]
    #pragma recreate_exp neighb exp_2[body_j]
    MatSetValues(mat_a,1,&exp_1[body_j],1,&exp_2[body_j],local_anbs,ADD_VALUES);
    local_aps += ...
    local_anbs = ...
    #pragma recreate_exp i exp_3[body_i]
    #pragma recreate_exp i exp_4[body_i]
    MatSetValues(mat_a,1,&exp_3[body_i],1,&exp_4[body_i],local_aps,ADD_VALUES);
    #pragma recreate_exp i exp_5[body_i]
    VecSetValues(mat_b,1,&exp_5[body_i],local_sc,INSERT_VALUES);
  }
  #pragma recreate_exp i exp_6[body_i]
  #pragma recreate_exp i exp_7[body_i]
  MatSetValues(mat_a,1,&exp_6[body_i],1,&exp_7[body_i],local_sc,ADD_VALUES);
  #pragma recreate_exp i exp_8[body_i]
  VecSetValues(mat_b,1,&exp_8[body_i],local_sc,INSERT_VALUES);
}

KSP_Solve(ksp_solve,mat_a,vec_r,vec_x);
VecGetArray(vec_x,local_phic);
VecRestoreArray(vec_x,phic);
//...Computation not shown...
8.1. User-defined Array Partitioning

Usually, the interface used to extract results from the external solver also requires an array as input. For example, array phic is used to retrieve the values of the PETSc vector vec\_x in Listing 7. The same interface can be used from within both sequential and parallel application code, with the difference being that in the latter case, the local portion of vec\_x is copied into phic. Array vec\_x used within PETSc determines which elements of phic are owned by which of the processes. This information has to be provided to the transformation framework as an input by the user.

In Listing 7 this is done through the pragma #pragma block_partition at Line 3, which specifies that array phic has to be block-partitioned in the transformed code. This information is used to determine the owner of each data-element instead of using the approach described in Section 5.2. While our current implementation supports only block partitioning for such arrays, it can be easily extended to handle other forms of partitioning.

Since the rest of the computation has been partitioned by the I/E framework, oblivious to the partitioning specified by the user, ghost elements are used to represent non-local array elements accessed by a process. These elements are placed in memory after the elements that are owned by a process. Since the layout of data in local memory is not controlled by the current framework, no optimization of accesses is performed for such arrays.

9. Evaluation

For our experimental evaluation we used a cluster with Intel Xeon E5630 processors with 4 cores per node and a clock speed of 2.67GHz, with an Infiniband interconnect. MVAPICH2-1.7 was used for MPI communications, along with Global Arrays 5.1 for the ARMCI one-sided communications. All benchmarks/applications were compiled using ICC 12.1 at -O3 optimization level.

For partitioning hypergraphs, the PaToH hypergraph partitioner [Catalyurek and Aykanat 2009] was used. While it supports multi-constraint hypergraph partitioning, it is sequential and requires the replication of the hypergraph on all processes. Since the generated inspector is inherently parallel, and parallel graph partitioners are available, an alternative approach was also pursued: convert the hypergraph to a graph, which can be partitioned in parallel. For this conversion, an edge was created between every pair of vertices belonging to the same net. The resulting graph was partitioned in parallel with ParMetis [Schloegel et al. 2002]. Multi-constraint partitioning (as discussed in Section 3) was employed to achieve load balance between processes while reducing communication costs. We also evaluated a third option: block partitioning of the iterations of partitionable loops (referred to as Block), where the cost of graph partitioning can be completely avoided.

For all benchmarks and applications, all functions were inlined, and arrays of structures were converted to structures of arrays for use with our prototype compiler which implements the transformations described earlier. The compiler was developed in the ROSE infrastructure [ROSE].

9.1. Benchmarks

For evaluation purposes, we used benchmarks with data-dependent control-flow and array-access patterns. Each benchmark has a sequence of partitionable loops enclosed within an outer sequential (time or convergence) loop, with the control-flow and array-access pattern remaining the same for every iteration of that outer loop. All reported execution times are averaged over 10 runs. The speed-up reported is with respect to the execution time of the original sequential code.

9.1.1. 183.equake [Bao et al. 1998]

This is a benchmark from SPEC2000 which simulates seismic wave propagation in large basins. It consists of a sequence of partitionable loops enclosed within an outer time loop. The SPEC ref data size was used for the evaluation. Figure 5a compares the performance of the executor code using the three partitioning schemes with a manual MPI implementation by the authors. In all cases, the executor performance is comparable to the manual MPI implementation. After 64 processes, the performance of all executors drops off due to the overhead of communication. Figure 5b shows
that the overhead of the inspector while using ParMetis or block partitioning is negligible, but with PaToH, the sequential nature of the partitioner adds considerable overhead.

9.1.2. CG Kernel
The conjugate gradient (CG) method to solve linear system of equations consists of five partitionable loops within a convergence loop. Two sparse matrices, `hood.rb` and `tmt_sym.rb`, from the University of Florida Sparse Matrix Collections [Davis 1994], stored in CSR format were used as inputs for evaluation.

Figures 6a and 6c show that the executor code achieves good scaling overall with super-linear scaling between 8 and 32 processes, due to the partitions becoming small enough to fit in caches. Using block-partitioning gives good performance with `tmt_sym` but not for `hood`. For the latter, block partitioning results in a larger number of ghosts cells and therefore higher communication costs, demonstrating the need for modeling the iteration-data affinity. The inspector overheads reduce the overall speed-up achieved, as shown in Figures 6b and 6d. This cost could be further amortized in cases where the linear systems represented by the matrices are solved repeatedly, say within an outer time loop, with the same non-zero structure. Such cases are common in many scientific applications.

The performance of the executors was also compared to a manual implementation using PETSc [Balay et al. 2012] which employed a block-partitioning of the rows of the matrix. For `hood`, Figure 6a shows that the performance of the generated executor code while using PaToH and ParMetis out-performs the manual PETSc implementation. The performance of the latter drops off due to the same reason the performance of the block-partitioned scheme drops off. With `tmt_sym`, the generated executor performs on par with the manual implementation for all three partitioning schemes (Figure 6c).

9.1.3. P3-RTE Benchmark [Ravishankar et al. 2010]
This benchmark solves the radiation transport equation (RTE) [Modest 2003] approximated using spherical harmonics on an unstructured physical grid of 164540 triangular cells. The Finite-Volume Method is used for discretizing the RTE. As described in Section 8, the original application uses an external solver package to solve the resulting set of linear equations. For evaluation of proposed automatic transformation scheme, the use of the external solver package was replaced with the Jacobi method for solving the system of equations at each cell center, making the entire application code free of calls to any external solver. The different partitionable loops iterate over cells, faces, nodes, and boundaries of the domain, and are enclosed within a convergence loop.

Figure 7a compares the executor times for the three schemes with a manual MPI implementation which uses domain decomposition of the underlying physical grid to partition the computation. The results once again show that a simple block-partitioning could result in poor performance. Surprisingly, ParMetis performs better than PaToH for larger number of processes. The executor code while using PaToH or ParMetis achieves performance comparable to the manual MPI implementation up
to 64 processes. Past that, the manual implementation continues to achieve scalable performance since the communication costs are reduced significantly by replicating some computation. Using techniques like overlap of communication with computation could improve the performance of the generated executor for larger number of processes. Figure 7b shows that the inspector overhead is negligible even when using the sequential PaToH partitioner.
9.1.4. miniFE-1.1 [Heroux et al. 2009]
This is a mini-application from the Mantevo suite from Sandia National Laboratories [Mantevo]. It uses an implicit finite-element method over an unstructured 3D mesh. A problem size of 100 points along each axis was used for the evaluation. The suite also provides a manual MPI implementation of the computation. The performance of the automatically generated parallel code is compared against this manually parallel implementation.

Figure 8 compares the running times for the executors (using ParMetis, PaToH, and block-partitioning) with the execution time of the manual MPI implementation. Up to 128 processes, the performance of the auto-generated executor is on par with the manual implementation. For 256 processes, the block-partitioned version performs slightly better than the other two approaches (PaToH and ParMetis), since the manual implementation uses an approach similar to block partitioning. Figure 8b shows the speed-up achieved for the total running times. Since the actual running time of the executor is not very significant even for the large problem size, the cost of the inspector dominates the overall running time.

9.2. Applications
The previous section showed the performance obtained from automatic transformations of codes that are representative of the compute-intensive parts of a wide-variety of scientific computing applications. In this section we evaluate the automatic parallelization of two real-world applications. Both these applications are written in Fortran 90. Unlike the C benchmarks discussed earlier, which could be automatically transformed by our prototype compiler, the generation of the inspector/executor code for the applications below required some manual steps. It was found that the inspector costs for both applications were negligible when compared to the overall running time of the executor, even when using the hypergraph partitioner. Therefore, all results shown in this section are obtained while using PaToH for generating the partitions.

9.2.1. OLAM [Walko and Avissar 2008]
OLAM (Ocean, Land, and Atmosphere Modeling) is an application used for climate simulations of the entire planet. It employs finite-volume methods of discretization to solve for physical quantities such as pressure, temperature, and wind velocity over a 3D unstructured grid consisting of 3D prisms covering the surface of the earth. Physical quantities are associated with centers of prisms and prism edges. The input grid contained 155520 prisms.

We report performance on an atmospheric model simulation consisting of 13 partitionable loops enclosed within a sequential time loop. While the outer loop typically executes hundreds of thousands of iterations, Figure 9a shows data for 30000 iterations. The authors of OLAM have also developed an MPI implementation of the application with several domain-specific optimizations.
Figure 9a shows that up to 32 processes, the performance of the code generated by the transformation scheme (including inspector time) is on par, if not better, when compared to the manual MPI implementation. Past that, the efficient domain decomposition scheme used by the manual MPI implementation results in fewer ghosts and therefore, lower space and communication overheads. These factors contribute to a super-linear scaling achieved by the manual implementation. Note that the I/E version still achieves linear scaling.

9.2.2 P3-RTE Full Application As described in Section 9.1.3, the original application that solves the P3 approximation to the radiative transport equation on an unstructured mesh uses the external library, PETSc, to solve the set of linear equations obtained by discretizing the transport equation. The compute-intensive portion of the application contains a sequence of five partitionable loops, one of which sets up the linear equations to be solved using the interface to PETSc, as described in Section 8. Here we evaluate the performance achieved by the transformation scheme that required hints from the user and was described in that section.

The problem size used for evaluation purposes is same as the one used in Section 9.1.3. Figure 9b shows the performance of the generated code. Since the original application has no manual MPI implementation by domain experts, Figure 9b compares the performance with the reference line for linear scaling. Due to the high memory requirements of the original application, on 2 processes the generated code achieves a threefold improvement in performance by reducing the working set of the problem on each process by half, resulting in better cache performance. The almost linear scaling observed on increasing the number of process, shows the benefits of allowing the generated executor code interface to the manually tuned parallel solvers of PETSc.

10. Related Work

Saltz and co-workers [Saltz et al. 1990; Saltz et al. 1991; Das et al. 1995; Das et al. 1993] proposed the inspector-executor (I/E) approach for distributed-memory code generation for scientific applications with irregular access patterns. The PARTI/CHAOS libraries [Ponnusamy et al. 1993; Berryman et al. 1991] facilitated manual development of parallel message-passing code. Compiler support for optimizing communications within the executor was also explored [von Hanxleden et al. 1993; Agrawal et al. 1995]. An approach to automatic compiler transformation for generation of I/E code via slicing analysis was developed [Das et al. 1993], but required indirect access of all arrays in the executor code even when the original sequential code used direct access through inner loop iterators (for example, A[i] in Listing 1). Most of these approaches could handle only one level of indirection. The approach of Das et al. [1993] could handle multiple levels of indirection, but their techniques are inapplicable to codes such as 183.equake and P3-RTE. Lain et al. [1996; 1995] exploited contiguity within irregular accesses to reduce communication costs and inspector overheads. Since the layout of data was not explicitly handled to maintain contiguity, the extent to which this property could be exploited in the executor depended on the partitioning of data.
Some later approaches have proposed the use of run-time reordering transformations [Ding and Kennedy 1999; Mitchell et al. 1999; Han and Tseng 2006; Strout et al. 2003; 2002] proposed a framework for code generation that combined run-time and compile-time reordering of data and computation. The recent work of LaMielle and Strout [LaMielle and Strout 2010] proposes an extended polyhedral framework that can generate transformed code (using inspector/executor) for computations involving indirect array accesses. The class of computations addressed by that framework is more general than the partitionable loops considered here and can handle more general types of iteration/data reorderings. But the generality of the framework, without additional optimizations, can result in code that is less efficient than that generated for partitionable loops here. For example, restricting the order of execution of inner loops within partitionable loops to be the same as that of the original sequential code enables exploitation of contiguity in data access. Arbitrary iteration reordering would require use of indirect access for all expressions in the executor code. Formulation of such domain/context specific constraints within the sparse polyhedral framework so as to generate more efficient code is an interesting open question.

Basumallik and Eigenmann [Basumallik and Eigenmann 2006] presented techniques for translating OpenMP programs with irregular accesses into code for distributed-memory machines, by focusing on exploiting overlap of computation and communication. However, the approach requires partial replication of shared data on all processes.

A large body of work has considered the problem of loop parallelization. Numerous advances in automatic parallelization and tiling of static control programs with affine array accesses have been reported [Irigoin and Triolet 1988; Ramanujam and Sadayappan 1992; Feautrier 1992; Lim and Lam 1997; Griebi 2004; Bondhugula et al. 2008]. For loops not amenable to static analysis, speculative techniques have been used for run-time parallelization [Rauchwerger and Padua 1995; Leung and Zaho 1993; Rus et al. 2002; Yu and Rauchwerger 2000]. Zhuang et al. [Zhuang et al. 2009] inspect run-time dependences to check if contiguous sets of loop iterations are dependent. None of those efforts address distributed memory code generation.

In contrast to prior work, we develop a framework for effective message-passing code generation and effective parallel execution of an extended class of affine computations with some forms of indirect array accesses. We are not aware of any other compiler work on inspector-executor code generation that maximize contiguity of accesses in the generated code, which is important in reducing cache misses as well as enabling SIMD optimizations in later compiler passes.

11. Conclusion

In this paper we have presented techniques for effective automatic parallelization of irregular and sparse computation for distributed memory systems, using the inspector/executor paradigm. Algorithms to automatically detect target loops were described in detail. The transformation scheme described generated a parallel inspector which analyzed the iterations of the loop at run-time to partition both the iterations and the data. The inspector also generated auxiliary data structures that were used by the executor for efficient parallel execution. Optimizations for exploiting contiguity of accesses reduced the memory overhead significantly and also enabled further compiler optimizations. For applications which use external solvers, the framework is extended such that the generated code uses the corresponding parallel solvers, either automatically, or with limited user intervention.

The effectiveness of the approach is demonstrated on several benchmarks and real-world applications. The performance of the transformed code is comparable to the manually parallelized implementations of the same. Future work would focus on optimizing the communication layer to achieve better scalability of the generated code.

12. Acknowledgments

We thank Umit Catalyurek for making available the PaToH software and for his help with formulating the partitioning constraints. We thank Gagan Agrawal and the SC reviewers for their comments and feedback with respect to existing inspector/executor techniques. We also thank Robert L. Walko, the author of OLAM, for his guidance with the software. This material is based upon work supported
by the U.S. National Science Foundation under grants 0811457, 0811781, 0926687, 0926688 and 0904549, by the U.S. Department of Energy under grant DE-FC02-06ER25755, and by the U.S. Army through contract W911NF-10-1-0004.

REFERENCES


DAS, R., HAVLAK, P., SALTZ, J., AND KENNEDY, K. 1995. Index array flattening through program transformation. In SC.


DING, C. AND KENNEDY, K. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. In PLDI.


LAIN, A. AND BANERJEE, P. 1995. Exploiting spatial regularity in irregular iterative applications. In IPPS.


MITCHELL, N., CARTER, L., AND FERRANTE, J. 1999. Localizing non-affine array references. In PACT.


Par4All 2012. Par4All. www.par4all.org.

PONNUSAMY, R., SALTZ, J. H., AND CHAUDHARY, A. N. 1993. Runtime compilation techniques for data partitioning and communication schedule reuse. In SC.


