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

--Manuscript Draft--

Manuscript Number:	TOPC-D-13-00023
Full Title:	Automatic Parallelization of a Class of Irregular Loops for Distributed Memory Systems
Article Type:	Original Article
Abstract:	Many scientific applications spend significant time within loops that are parallel, except for dependencies from associative reduction operations. However these loops often 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 formalize the computations that are targeted by this approach and develop algorithms to detect such computation. We describe in detail, algorithms to generate a parallel inspector which performs the run-time analysis, and a parallel executor. The effectiveness of the approach is demonstrated on several benchmarks and a real-world applications. We measure the inspector overhead and also evaluate the benefit of optimizations applied during the transformation.

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. However these loops often 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 formalize the computations that are targeted by this approach and develop algorithms to detect such computation. We describe in detail, algorithms to generate a parallel inspector which performs the run-time analysis, and a parallel executor. The effectiveness of the approach is demonstrated on several benchmarks and a real-world applications. We measure the inspector overhead and also evaluate the benefit of optimizations applied during the transformation.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors - Code generation, compilers, optimization

General Terms: Performance

Additional Key Words and Phrases: Distributed-Memory systems, inspector-executor

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]

This work is supported by 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.

Author's addresses: M. Ravishankar and J. Eisenlohr and A. Rountev and P. Sadayappan, Computer Science and Engineering, Ohio State University. Louis-Noel Pouchet, Computer Science, University of California, Los Angeles. J. Ramanujam, Electrical and Computer Engineering Division, School of Electrical Engineering and Computer Science, Louisiana State University.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

```
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...
}
```

Listing 1. Sequential conjugate gradient computation.

approach. The I/E approach uses (1) an *inspector* 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* 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 9) 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 that are targeted for transformation
— It presents algorithms to automatically detect such computations.
— 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 experimental results comparing the performance of the automatically generated distributed memory code with manual MPI impementations. We evaluate the overhead of the run-time analysis. Experimental results are also presented to demonstrate the importance of maintaining contiguity of accesses from the original computation.

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 4 provides details of the approach to generate computation partitions using a hypergraph that models the affinity of loop iterations to data elements accessed. Section 3 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, with Section 6 describing the overall code-generation process. Section 7 elaborates on how the need for inspector code can be optimized away for portions of the input code that are strictly affine. Experimental results using four kernels and one significant application code are presented in Section 8. Related work is discussed in Section 9 and conclusions stated in Section 10.

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`. In loop `i`, vector `y` is computed by multiplying matrix `A` and vector `x`. Here `A` uses the Compressed

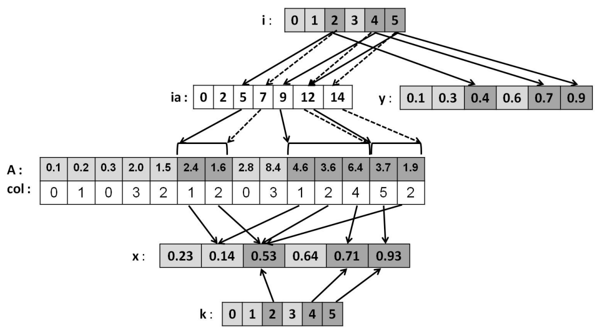


Fig. 1. Control flow and data-access patterns of iteration 2, 4, and 5, of loops i and k mapped to process 0.

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 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 `if`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 `ia`, and accesses to `x` depend on `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

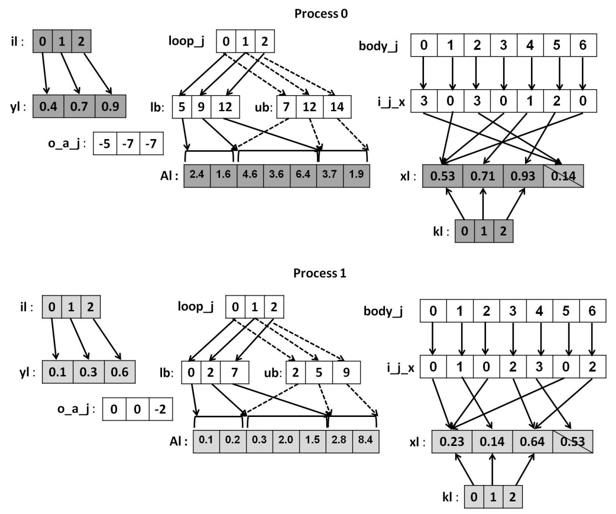


Fig. 2. Transformed iteration and data view.

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 3 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 4) 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

```
/* Inspector code to generate o_a_j, i_x_j, lb,*/
/* ub, xl, yl, Al and to compute nl; not shown */
while( !converged ){
  //...Other computation not shown...
  //...Update Values of ghosts for arrays read in loop...
  body_k = 0;
  for( kl = 0 ; kl < nl ; kl++ ){
    xl[kl] = ...;
    body_k++;
  }
  //...Update values of owner of elements of xl...
  //...Other computation not shown...
  //...Update values of ghosts for Al and xl...
  body_i = 0; loop_j = 0 ; body_j = 0;
  for( il = 0 ; il < nl ; il++ ) {
    offset_a_j = o_a_j[loop_j] - lb[loop_j];
    for( j = lb[loop_j] ; j < ub[loop_j] ; j++ ){
      yl[il] += Al[j+offset_a_j]*xl[i_x_j[body_j]];
      body_j++;
    }
    loop_j++; body_i++;
  }
  //...Update values of owners of yl...
  //...Other computation; not shown...
 }
```

Listing 2. Parallel conjugate gradient computation.

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, yl, Al, and xl 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 "+=", 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 xl is the local copy of data array x on each process, all elements of x accessed by a process are represented in xl. To access the correct elements in xl, 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 `o_a_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, since iterations of a partitionable loop mapped to a process are not necessarily contiguous.

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 `xl` can be accessed by using iterator `kl` if `xl[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 `nl` that are mapped to a process. The loop bounds of loop `j` are read from arrays `lb` and `ub`. Accesses to local data arrays `xl` and `Al` are determined by values in arrays `i_x_j` and `o_a_j`. In addition, communication calls are inserted to satisfy the producer-consumer relationship due to array `x`.

3. 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

$$
\begin{array}{rcl}
\langle\text{Start}\rangle & ::= & \langle\text{ElementList}\rangle \\
\langle\text{ElementList}\rangle & ::= & \langle\text{Element}\rangle\texttt{;}\langle\text{ElementList}\rangle \mid \langle\text{Element}\rangle \\
\langle\text{Element}\rangle & ::= & \langle\text{Assignment}\rangle \mid \langle\text{Loop}\rangle \mid \langle\text{If}\rangle \\
\langle\text{Loop}\rangle & ::= & \texttt{for (}\langle\text{Iterator}\rangle\texttt{=}\langle\text{Expr}\rangle\texttt{ ; }\langle\text{Iterator}\rangle\texttt{<}\langle\text{Expr}\rangle\texttt{ ; }\langle\text{Iterator}\rangle\texttt{++)}\ \{\langle\text{ElementList}\rangle\} \\
\langle\text{If}\rangle & ::= & \texttt{if (}\langle\text{Expr}\rangle\texttt{)}\ \{\langle\text{ElementList}\rangle\}\ \texttt{else}\ \{\langle\text{ElementList}\rangle\} \\
\langle\text{Assignment}\rangle & ::= & \langle\text{LHSExpr}\rangle\langle\text{AssignOp}\rangle\langle\text{Expr}\rangle \\
\langle\text{Expr}\rangle & ::= & \text{Side-effect-free expression of } \langle\text{BasicExpr}\rangle \\
\langle\text{LHSExpr}\rangle & ::= & \langle\text{Scalar}\rangle \mid \langle\text{Array}\rangle\texttt{[}\langle\text{Expr}\rangle\texttt{]} \\
\langle\text{BasicExpr}\rangle & ::= & \langle\text{Scalar}\rangle \mid \langle\text{Iterator}\rangle \mid \langle\text{Array}\rangle\texttt{[}\langle\text{Expr}\rangle\texttt{]} \\
\langle\text{AssignOp}\rangle & ::= & \texttt{=} \mid \texttt{+=} \mid \texttt{*=} \mid \ldots
\end{array}
$$

Fig. 3. Syntactic structure of targeted computations.

only by the loop increment expression (with unit increment). An assignment could use the standard assignment operator, or an update operator `op=` where `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 $arr[expr]$, the index expression $expr$ can itself contain array access expressions. As discussed earlier, the arrays whose elements are used to compute this $expr$ are indirection arrays. The grammar allows for multiple levels of indirection.

Algorithm 1 takes as input an AST corresponding to a ⟨Loop⟩ from this grammar, and determines whether the loop is partitionable. Section 3.1 discusses the analysis of detect indirection arrays and scalars; this analysis corresponds to lines 2–37 in the algorithm. Section 3.2, 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 3.3.

3.1. 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.

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 code-generation scheme where the executor code does

ALGORITHM 1: CheckForPartitionableLoop($\mathcal{A}$)

```
Input  :  $\mathcal{A}$ : AST of loop satisfying the grammar in Figure 3
Output:  is_partitionable : Boolean flag set to true if $\mathcal{A}$ is partitionable
          $I_A$ : Set of indirection arrays
          $I_S$ : Set of indirection scalars
          $I$ : Set of loop iterators
1  begin
2      $I_S = \emptyset$ ; $I_A = \emptyset$ ; $I = \mathcal{A}$.Iterator ; is_partitionable = true ;
3      foreach  $s \in$ GetLoopStmts($\mathcal{A}$.Body) do
4          $I = I \cup s$.Iterator ;
5      foreach  $s \in$ GetLoopStmts($\mathcal{A}$.Body) do
6          $l = s$.LowerBound ; $u = s$.UpperBound ;
7          $I_A = I_A \cup$ GetAllArrayVariables($l$) $\cup$ GetAllArrayVariables($u$) ;
8          $I_S = I_S \cup$ ( GetAllScalars($l$) $\cup$ GetAllScalars($u$) $- I$ ) ;
9      foreach  $s \in$ GetIfStmts($\mathcal{A}$.Body) do
10         $c = s$.Cond ;
11         $I_A = I_A \cup$ GetAllArrayVariables($c$) ;
12         $I_S = I_S \cup$ ( GetAllScalars($c$) $- I$ ) ;
13     foreach  $s \in$ GetAssignStmts($\mathcal{A}$.Body) do
14         foreach  $e \in$ GetArrayRefExprs($s$) do
15             $c = e$.IndexExpr ;
16             $I_A = I_A \cup$ GetAllArrayVariables($c$) ;
17             $I_S = I_S \cup$ ( GetAllScalars($c$) $- I$ );
18     worklist = $I_S$ ;
19     while $\neg$IsEmpty(worklist) do
20         $v$ = RemoveElement(worklist) ;
21         foreach $s \in$ GetAssignStmts($\mathcal{A}$.Body) do
22             if  IsScalar($s$.LHS) $\wedge$ $s$.LHS == $v$ then
23                 $c = s$.RHS ;
24                 $I_A = I_A \cup$ GetAllArrayVariables($c$) ;
25                 $S$ = GetAllScalars($c$) $- I$ ;
26                 foreach  $u \in S$ do
27                     if  $u \notin I_S$ then
28                         $I_S = I_S \cup \{u\}$ ; AddElement(worklist,$u$) ;
29     foreach  $s \in$ GetAssignStmts($\mathcal{A}$.Body) do
30         if  $\neg$IsScalar($s$.LHS) $\vee$ $s$.LHS $\notin I_S$ then
31             $c = s$.RHS ;
32             $S$ = GetAllScalars($s$.RHS,IGNORE_INDEX_EXPR);
33             if  $S \cap I_S \neq \emptyset$ then
34                 is_partitionable = false ;
35             $S$ = GetAllArrays($s$.RHS,IGNORE_INDEX_EXPR) ;
36             if  $S \cap I_A \neq \emptyset$ then
37                 is_partitionable = false ;
38     is_partitionable = CheckForParallelism($\mathcal{A}$.Body,$I_S$,$I_A$,$I$) ;
39     return [is_partitionable,$I_S$,$I_A$,$I$] ;
```

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 3.4 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

```
       ⟨Start⟩ ::= ⟨ElementList⟩
 ⟨ElementList⟩ ::= ⟨Element⟩;⟨ElementList⟩ | ⟨Element⟩
     ⟨Element⟩ ::= ⟨IAssignment⟩ | ⟨DAssignment⟩ | ⟨Loop⟩ | ⟨If⟩
        ⟨Loop⟩ ::= for (⟨Iterator⟩=⟨IExpr⟩ ; ⟨Iterator⟩<⟨IExpr⟩ ; ⟨Iterator⟩++) {⟨ElementList⟩}
          ⟨If⟩ ::= if (⟨IExpr⟩) {⟨ElementList⟩} else {⟨ElementList⟩}
 ⟨IAssignment⟩ ::= ⟨IScalar⟩⟨AssignOp⟩⟨IExpr⟩
 ⟨DAssignment⟩ ::= ⟨BasicDExpr⟩⟨AssignOp⟩⟨DExpr⟩
       ⟨IExpr⟩ ::= Side-effect-free expression of ⟨BasicIExpr⟩
       ⟨DExpr⟩ ::= Side-effect-free expression of ⟨BasicDExpr⟩
  ⟨BasicIExpr⟩ ::= ⟨IScalar⟩ | ⟨Iterator⟩ | ⟨IArray⟩[⟨IExpr⟩]
  ⟨BasicDExpr⟩ ::= ⟨DScalar⟩ | ⟨DArray⟩[⟨IExpr⟩]
    ⟨AssignOp⟩ ::= = | += | *= | ...
```

Fig. 4. Syntactic structure of partitionable loops

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.

3.2. 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

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 ⟨DArray⟩ can appear either only on the right-hand side of ⟨DAssignment⟩s, or only on the left-hand side, but not both
— A ⟨DArray⟩ can appear on the left-hand side of multiple ⟨DAssignment⟩s, but all those statements must use the same ⟨AssignOp⟩

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 ⟨DArray⟩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 ⟨IScalar⟩).

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.

3.3. 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 ⟨ElementList⟩, i.e., a sequence of loops, conditionals, and assignments. Within this element list, each maximal sequence of consecutive ⟨Loop⟩ 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 empty, then these two loops can not 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 ⟨ElementList⟩ (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 8, only the sequences that were closest to the top level of the AST were transformed.

3.4. Possible Generalizations

One of the restrictions described in Section 3.1 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 3.1 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.

4. Partitioning the Computation

The computation is partitioned by considering the iteration-data affinity. To model this affinity, we use a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ where $\mathcal{V}$ 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 $i \in \mathcal{V}$. 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 achieve the following:

— partition the iteration of each of the partitionable loops in a load-balanced manner, and
— minimize the communication required to satisfy the producer-consumer relationships between partitionable loops.

Each partition created represents the computation that is performed by one process.

4.1. Achieving Load Balancing

Each vertex is associated with a vector of weights $\vec{w_i}$ 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. W_P represent the weight of a partition, P and is computed as follows

$$W_P = \sum_{i \in P} \vec{w_i} \tag{1}$$

If P_n (where $n \in [0, N)$) are the partitions generated, load balance for every partitionable loop is achieved by applying the following constraint

$$P_n \leq P_{avg}(1 + \epsilon) \tag{2}$$

where $P_{avg} = P_{\mathcal{V}}/N$; ϵ is the maximum load imbalance tolerated.

4.2. Minimizing Communication

Each net in $\mathcal{N}$ 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 *internal* nets, I_n and nets with pins in other partitions are *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 λ_j that have a ghost for the data element represented by $j \in \mathcal{N}$. This is achieved by minimizing the *cut-size* Π_n for each partition defined by Equation 3.

$$\Pi_n = \sum_{j \in E_n} c_j(\lambda_j - 1) \tag{3}$$

The hypergraph is subjected to a min-cut partitioning, under the load-balance constraints specified above [Catalyurek and Aykanat 2009].

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 `[kstart,kend)` for loop `k` and `[istart,iend)` 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` for 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,vi,true)` adds vertex `vi` 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 *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, `shadow_xindex` is the shadow scalar corresponding to `xindex`. It is set to false if the value of `col[j]` is not known on a process. Since this value is used by `AddPin(id_x_array,xindex,vi,false)`, this statement is guarded with a check of the state of `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 4, 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.

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 ⟨IAssignment⟩ 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 `get_elem`. This AST modification is performed by function *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 *GenGuards*.

The generation of the guard expression constructed by *GenGuards* is described in Algorithm 3. The algorithm takes as input an ⟨IExpr⟩. For expressions that are references to ⟨IScalar⟩, the con-

```
 1 do{
 2   for( k = kstart ; k < kend ; k++ ) {
 3     vk = AddVertex(id_k_loop);
 4     AddPin(id_x_array,k,vk,true); ...;
 5   }
 6   for( i = istart ; i < iend ; i++ ) {
 7     vi = AddVertex(id_i_loop);
 8     if(is_known(id_ia_array,i) && is_known(id_ia_array,i+1))
 9       for( j = get_elem(id_ia_array,i) ; j < get_elem(id_ia_array,i+1) ; j++ ) {
10         if( is_known(id_col_array,j) ){
11           xindex = get_elem(id_col_array,j);
12           shadow_xindex = true;
13         }
14         else
15           shadow_xindex = false;
16         AddPin(id_y_array,i,vi,true);
17         AddPin(id_A_array,j,vi,true);
18         if( shadow_xindex )
19           AddPin(id_x_array,xindex,vi,false);
20       }
21   }
22 }while( DoneGraphGen() );
```

Listing 3. Phase I of the inspector.

dition checks if the corresponding shadow scalar is set to true. For expressions that are ⟨IArray⟩ references, the condition contains two parts. The first part is a call to function `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 *and* operator. Since the latter condition has to be checked before the former one, it is set as the first operand in the *and* expression. The short-circuit evaluation of C/C++ ensures that the `is_known` function is called only when the index expression can be evaluated.

Since a ⟨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 ⟨BasicIndexExpr⟩, all children of the ⟨IExpr⟩ are recursively evaluated and their guards are combined with the *and* operator.

Algorithm 2 sets the conditional expression returned by *GenGuards* as the condition of the guard statements. The statements to set the shadow scalar associated with the ⟨IScalar⟩ 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 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

ALGORITHM 2: CodeGenHyperGraph($\mathcal{H}$,ss)

```
Input  : $\mathcal{H}$ : Hypergraph object
         $ss$ : Sequence of statements in the original AST
Output: $\mathcal{A}_H$ : AST of the inspector code to generate hypergraph
         $I$ : $\langle$IScalar$\rangle$s defined in $ss$
1  begin
2      $\mathcal{A}_H = \emptyset$ ; $I = \emptyset$ ;
3      foreach $s \in ss$ in order of appearance do
4          if IsIAssignment(s) then
5              $b$ = ConvertArraysToFunctions($s$.RHS) ;
6              $l_H$ = NewIfStmt() ; $l_H$.Cond = GenGuards($s$.RHS) ;
7              $s_H$ = NewAssignmentStmt($s$.LHS,$b$) ;
8              $s_H$.Append(SetShadowScalar($s$.LHS,true)) ;
9              $I_L = \{ s$.LHS $\}$ ; $l_H$.Then = $s_H$ ;
10         else if IsDAssignment(s) then
11             $l_H = \emptyset$ ;
12             foreach $d \in$ GetDataArrayRefExprs(s) do
13                 $b$ = ConvertArraysToFunctions($d$.IndexExpr) ;
14                 $s_H$ = NewIfStmt() ; $s_H$.Cond = GenGuards($d$.IndexExpr) ;
15                 $t$ = GenAddPinFn($\mathcal{H}$,$d$.Array,$b$,IsUnitStride($d$.IndexExpr)) ;
16                 if $s_H$.Cond $== \emptyset$ then
17                     $s_H$.Then = $t$ ; $l_H$.Append($s_H$) ;
18                 else
19                     $l_H$.Append($t$) ;
20         else if IsLoop(s) then
21             $l$ = $s$.LowerBound ; $u$ = $s$.UpperBound ;
22             $b_l$ = ConvertArraysToFunctions($l$) ;
23             $b_u$ = ConvertArraysToFunctions($u$) ;
24             $s_H$ = NewLoopStmt($s$.Iterator,$b_l$,$b_u$) ;
25             [$s_H$.Body,$I_L$] = CodeGenHyperGraph($\mathcal{H}$,$s$.Body) ;
26             $l_H$ = NewIfStmt() ; $l_H$.Then = $s_H$ ;
27             $l_H$.Cond = NewAndCond(GenGuards($l$),GenGuards($u$)) ;
28         else
29             $c_H$ = ConvertArraysToFunctions($s$.Cond) ;
30             $s_H$ = NewIfStmt() ; $s_H$.Cond = $c_H$ ;
31             [$s_H$.Then,$I_1$] = CodeGenHyperGraph($\mathcal{H}$,$s$.Then) ;
32             [$s_H$.Else,$I_2$] = CodeGenHyperGraph($\mathcal{H}$,$s$.Else) ;
33             $l_H$ = NewIfStmt() ; $l_H$.Cond = GenGuards($s$.Cond) ;
34             $I_L = I_1 \cup I_2$ ; $l_H$.Then = $s_H$ ;
35         foreach $v \in I_L$ do
36             $l_H$.Else.Append(SetShadowScalar($v$,false)) ;
37         $\mathcal{A}_H$.Append($l_H$) ; $I = I \cup I_L$ ;
38     return [$\mathcal{A}_H$,$I$] ;
```

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.

ALGORITHM 3: GenGuards(e)

```
Input  : $e$ : ⟨IExpr⟩ to be guarded
Output: $c$ : Condition to be used for the guard statement
1  begin
2      if IsIScalar($e$) then
3          $c$ = NewCheckEquality($e$,true) ;
4      else if IsArrayRefExp($e$) then
5          $c_l$ = GenGuards($e$.IndexExpr) ;
6          $c_r$ = GenerateIsKnownFn($e$.Array,$e$.IndexExpr) ;
7          $c$ = NewAndCond($c_l$,$c_r$) ;
8      else
9          $c = \emptyset$ ;
10         foreach $d \in e$.Children do
11             $c$ = NewAndCond($c$,GenGuards($d$)) ;
12     return $c$ ;
```

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 `yl`, `xl` and `Al` 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.

```
do{
  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 ){
      if( is_known(id_ia_array,i) && is_known(id_ia_array,i+1)){
        for( j = get_elem(id_ia_array,i) ; j < get_elem(id_ia_array,i+1) ; j++ ){
          if( is_known(id_col_array,j) )
            xindex = get_elem(id_col_array,j);
          body_j++;
        }
        loop_j++;
      }
      body_i++;
    }
}while( DoneCounters() );
```

Listing 4. Phase II of the inspector.

Listing 4 shows the code for this phase of the inspector for the running example. Each process analyzes only those iterations that are mapped to it after the partitioning process. The number of invocations of inner loop `j` is tracked via counter `loop_j`. Counters `body_*` 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, since the iterations mapped to a process may be different from those analyzed while building the hypergraph, this phase of the inspector also ensures that all values of indirection arrays needed for subsequent analysis have been prefetched. Therefore, as in Section 5.1, all inner loops, conditionals, and assignments are guarded to check if the values of indirection scalars and indirection array elements are known. Again, this phase is completed only after all levels of indirections have been resolved.

Based on the values of these counters, the access arrays that would be needed to recreate the control flow and data accesses patterns are allocated in the local memory of every process.

5.3.2. Code Generation at Compile Time The code generation for this phase is similar to the approach from Phase I. Only statements that affect the control-flow and array-access patterns are considered. The differences from Phase I are (1) the bounds of the partitionable loop are same as the original computation and its body is enclosed in an `if` statement that checks if the iteration is to be executed on the current process, (2) statements to increment counters `loop_*`, `body_*`, `if_*`, `then_*` and `else_*` are introduced, and (3) there statements that add pins and vertices to the hypergraph are removed.

5.4. Phase III: Initializing Local Access Arrays

5.4.1. Run-time Functionality After allocation, the access arrays are initially populated with the sequence of values of the corresponding expression in the original computation. Listing 5 shows the code to do so, for the example in Listing 1. Each process again analyzes the iterations mapped to it after partitioning. For expressions that are unit-stride with respect to a surrounding inner loop, the index of the element accessed by the first iteration of every invocation of the loop is stored in the array that represents the index expression. For all other expressions, the values for all iterations are stored in arrays. The values of the loop bounds of all inner loops, as well as the results of conditionals, are also stored in arrays in this phase of the inspector.

5.4.2. Code Generation at Compile Time To generate the code for this phase, once again each generated loop uses an `if` statement to analyze only the iterations mapped to the current process. The body of each new loop is generated by applying Algorithm 4 on the statements in the body of the corresponding partitionable loops from the original program.

```
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.

As was done for Phases I and II, ⟨IAssignment⟩s are replicated in the inspector AST with references to indirection arrays on the right-hand side of the original statement replaced with calls to function `get_elem`. For example, the right-hand side of `xindex=col[j]` in Listing 1 is modified to `get_elem(id_col, j)`. There is no need for any guards in this phase since the previous phase ensured that all values needed have been prefetched.

For ⟨DAssignment⟩s, all expressions used to access data array elements are considered. For every such expression, an assignment is generated to store the array index accessed. The array that is to be used to store the value of the index expression is retrieved by function *GetAccessArray*. If the index expression used is an unit-stride expression with respect to a surrounding loop, *GetCounterVariable* returns the `loop_*` counter associated with that loop. If not, it returns the `body_*` counter associated with the outermost loop for which the expression is not loop invariant. In the presence of conditional statements between the ⟨Loop⟩ AST node for that loop and the AST node for the ⟨DAssignment⟩, the function returns the `then_*` or `else_*` counter associated with this intervening conditional statement, depending on whether the ⟨DAssignment⟩ 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.

Upon encountering a ⟨Loop⟩ 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. ⟨If⟩ 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.

ALGORITHM 4: CodeToInitializeArrays(ss,L,$\mathcal{C}$)

```
Input  : $ss$ : Sequence of statements in the original AST
         $L$ : Access arrays for index expressions, loop bounds, and conditional values
         $\mathcal{C}$ : Counter variables
Output: $\mathcal{A}_P$ : AST of inspector code to populate the access arrays
begin
    $\mathcal{A}_P$ = $\emptyset$ ;
    foreach $s \in ss$ in order of appearance do
        if IsIStatement(s) then
            $b$ = ConvertArraysToFunctions($s$.RHS) ;
            $l_P$ = NewAssignmentStmt($s$.LHS,$b$) ;
        else if IsDStatement(s) then
            $l_P$ = $\emptyset$ ;
            foreach $d \in$ GetDataArrayRefExprs(s) do
                $a$ = GetAccessArray($L$,$d$.Array,$d$.IndexExpr) ;
                $c$ = GetCounterVariable($\mathcal{C}$,$d$.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($L$,$s$) ; $a_u$ = GetUpperBoundArray($L$,$s$) ;
            $c$ = GetLoopCounterVariable($\mathcal{C}$,$s$) ;
            $e_l$ = 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,$L$,$\mathcal{C}$) ;
            $c_b$ = GetBodyCounterVariable($\mathcal{C}$,$s$) ; $s_p$.Body.Append(NewIncrementStmt($c_b$)) ;
            $l_P$.Append($s_P$) ;
            $l_P$.Append(NewIncrementStmt($c$)) ;
        else if IsIf(s) then
            $c_P$ = ConvertArraysToFunctions($s$.Cond) ;
            $a$ = GetConditionalArray($L$,$s$) ;
            $c$ = GetConditionalCounterVariable($\mathcal{C}$,$s$) ;
            $e$ = NewArrayRefExpr($a$,$c$) ; $l_P$ = NewAssignmentStmt($e$,$c_P$) ;
            $s_P$ = NewIfStmt() ; $s_P$.Cond = $e$ ;
            $s_P$.Then = CodeToInitializeArrays($s$.Then,$L$,$\mathcal{C}$) ;
            $c_b$ = GetThenCounterVariable($\mathcal{C}$,$s$) ; $s_P$.Then.Append(NewIncrementStmt($c_b$)) ;
            $s_P$.Else = CodeToInitializeArrays($s$.Else,$L$,$\mathcal{C}$) ;
            $c_b$ = GetElseCounterVariable($\mathcal{C}$,$s$) ; $s_P$.Else.Append(NewIncrementStmt($c_b$)) ;
            $l_P$.Append($s_P$) ;
            $l_P$.Append(NewIncrementStmt($c$)) ;
        $\mathcal{A}_P$.Append($l_P$) ;
    return $\mathcal{A}_P$ ;
```

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.

ALGORITHM 5: GenerateExecutor(D,L,$\mathcal{C}$,ss)

```
Input :  $D$ : Local data arrays
         $L$ : Access arrays for index expressions, loop bounds, and conditional values
         $\mathcal{C}$ : Counter variables
InOut:   $ss$ : Sequence of statements in the original AST
1  begin
2      foreach $s \in ss$ in order of appearance do
3          if IsDAssignment(s) then
4              foreach $d \in$ GetDataArrayRefExprs(s) do
5                  ReplaceWithLocalArray($d$.Array,$D$) ;
6                  $c$ = GetCounterVariable($\mathcal{C}$,$d$.IndexExpr) ;
7                  $a$ = GetAccessArray($L$,$d$.Array,$d$.IndexExpr) ;
8                  $e$ = NewArrayRefExpr($a$,$c$) ;
9                  if IsUnitStride($d$.IndexExpr) then
10                     $l$ = GetLoopStatement($c$) ; $o$ = NewVariable() ;
11                     $e_l$ = NewArrayRefExpr(GetLowerBoundArray($L$,$l$),$c$) ;
12                     $s_E$ = NewAssignmentStmt($o$,NewSubtractExpr($e$,$e_l$)) ;
13                     InsertBefore($l$,$s_E$) ;
14                     ReplaceExpression($d$.IndexExpr,NewAddExpr($l$.Iterator,$o$));
15                 else
16                     ReplaceExpression($d$.IndexExpr,$e$) ;
17         else if IsLoop(s) then
18             $l$ = $s$.LowerBound ; $u$ = $s$.UpperBound ;
19             $a_l$ = GetLowerBoundArray($L$,$s$) ; $a_u$ = GetUpperBoundArray($L$,$s$) ;
20             $c$ = GetLoopCounterVariable($\mathcal{C}$,$s$) ;
21             $e_l$ = NewArrayRefExpr($a_l$,$c$) ; ReplaceExpression($s$.LowerBound,$e_l$) ;
22             $e_u$ = NewArrayRefExpr($a_u$,$c$) ; ReplaceExpression($s$.UpperBound,$e_u$) ;
23             GenerateExecutor($D$,$L$,$\mathcal{C}$,$s$.Body) ;
24             $c_b$ = GetBodyCounterVariable($\mathcal{C}$,$s$) ; $s_P$.Body.Append(NewIncrementStmt($c_b$)) ;
25             $s$.Append(NewIncrementStmt($c$)) ;
26         else
27             $a$ = GetConditionalArray($L$,$s$) ;
28             $c$ = GetConditionalCounterVariable($\mathcal{C}$,$s$) ;
29             $e$ = NewArrayRefExpr($a$,$c$) ; ReplaceExpression($s$.Cond,$e$) ;
30             GenerateExecutor($D$,$L$,$\mathcal{C}$,$s$.Then) ;
31             $c_b$ = GetThenCounterVariable($\mathcal{C}$,$s$) ; $s_P$.Then.Append(NewIncrementStmt($c_b$)) ;
32             GenerateExecutor($D$,$L$,$\mathcal{C}$,$s$.Else) ;
33             $c_b$ = GetElseCounterVariable($\mathcal{C}$,$s$) ; $s_P$.Else.Append(NewIncrementStmt($c_b$)) ;
34             $s$.Append(NewIncrementStmt($c$)) ;
```

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. Communica-

ALGORITHM 6: CodeGenInspectorExecutor($\mathcal{P}$)

```
InOut  :  $\mathcal{P}$ : Sequence of partitionable loop ASTs
Output:  $\mathcal{A}_I$ : Code for the inspector
1  begin
2      $\mathcal{H}$ = InitHyperGraph() ; $\mathcal{A}_I$ = $\emptyset$ ; $\mathcal{A}_H$ = $\emptyset$ ;
3      foreach $p \in \mathcal{P}$ in order do
4          [$l_H$,$u_H$] = BlockIterationBounds($p$.LowerBound,$p$.UpperBound) ;
5          $P_1$ = NewLoopStmt($p$.Iterator,$l_H$,$u_H$) ;
6          $P_1$.Body = GenAddVertex($\mathcal{H}$,$p$.Iterator) ;
7          [$L_1$,$S_1$] = CodeGenHyperGraph($\mathcal{H}$,$p$.Body) ;
8          $P_1$.Body.Append($L_1$) ; $\mathcal{A}_H$.Append($P_1$) ;
9      $A_H$ = NewDoWhile("DoneGraphGen()",$A_H$); $A_I$.Append($A_H$) ;
10     $\mathcal{A}_I$.Append(CodeToPartitionIterations($\mathcal{H}$)) ;
11     $\mathcal{D}$ = CodeToAllocateLocalData($\mathcal{H}$) ; $\mathcal{A}_I$.Append($D$) ;
12     $\mathcal{C}$ = DeclareCounterVariables($\mathcal{P}$) ; $\mathcal{A}_I$.Append($\mathcal{C}$) ;
13     $\mathcal{A}_C$ = $\emptyset$ ;
14     foreach $p \in \mathcal{P}$ in order do
15         $P_2$ = NewLoopStmt($p$.Iterator,$p$.LowerBound,$p$.UpperBound) ;
16         $L_2$ = NewIfStmt() ; $L_2$.Cond = GenerateIsHome() ;
17         $L_2$.Then = CodeToGetAccessArraySizes($p$.Body,$\mathcal{C}$) ;
18         $P_2$.Body = $L_2$ ; $A_C$.Append($P_2$) ;
19     $A_C$ = NewDoWhile("DoneCounters()",$A_C$) ; $\mathcal{A}_I$.Append($\mathcal{A}_C$) ;
20     $\mathcal{I}$ = CodeToAllocateAccessArrays($\mathcal{C}$) ; $\mathcal{A}_I$.Append($\mathcal{I}$ ) ;
21     $\mathcal{A}_P$ = $\emptyset$ ;
22     foreach $p \in \mathcal{P}$ in order do
23         $P_3$ = NewLoopStmt($p$.Iterator,$p$.LowerBound,$p$.UpperBound) ;
24         $L_3$ = NewIfStmt(); $L_3$.Cond = GenerateIsHome() ;
25         $L_3$.Then = CodeToInitializeArrays($p$.Body,$\mathcal{I}$,$\mathcal{C}$) ;
26         $P_3$.Body = $L_3$; $\mathcal{A}_P$.Append($P_3$) ;
27     $A_P$.Append(CodeToRenumberAccessArrays($\mathcal{C}$,$\mathcal{D}$,$\mathcal{I}$)) ;
28     $\mathcal{A}_I$.Append($A_P$) ;
29     foreach $p \in \mathcal{P}$ in order do
30         GenerateExecutorCode($\mathcal{D}$,$\mathcal{I}$,$\mathcal{C}$,$p$.Body) ;
31         InsertCommunicationCode($\mathcal{D}$,$p$) ;
32     return $\mathcal{A}_I$ ;
```

tion 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 3.3. 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`.

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 *GenerateIsHome*. The true branch of this conditional statement is generated by analyzing the body of the partitionable loops using function *CodeToGetAccessArraySizes*, 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 `do-while` loop with the conditional being a call to function `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 *CodeToRenumberAccessArrays* 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, followed 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. In this section, we provide an overview of the method that could be used to parallelize such computations.

A regular distribution of the iterations of partitionable loops (such as block, cyclic, or block-cyclic) is used instead of the distribution suggested by the hypergraph partitioning scheme. Although the actual bounds of the partitionable loop to be executed by each processor can only be resolved at run time, (since it is a non-linear function of the problem size parameters such as N, and the number of processors P), code can be generated using standard polyhedral machinery by setting

```
/* Original affine computation */
for( i = 0 ; i < n ; i++ )
  for( j = 0 ; j < n ; j++ )
    y[i] += A[i][j] * x[j];

/* Transformed parallel computation */
// ... Update values of ghost locations for Al and xl
for( i = 0 ; i < nl ; i++ )
  for( j = 0 ; j < n ; j++ )
    yl[i] += Al[i][j] * xl[j];
// ... Update values of owners for yl
```

Listing 6. Affine conjugate gradient computation (dense matrix).

the number of iterations executed on a process as a parameter. For example, if N is the number of iterations of a partitionable loop, for P processes, N/P can be treated treated as single parameter within the polyhedral framework.

Local copies of the data arrays can be computed as footprints of the partitioned iterations. Ghosts can be computed as the intersection of the process footprints. For all affine expressions used to access 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. Loop bounds of inner loops also use affine expressions, 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 while using the block or block-cyclic partitioning scheme. As before, the inspector also creates new 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 and purely static techniques can be used to generate the executor code. Listing 6 shows the code obtained for the executor by following this approach.

8. Evaluation

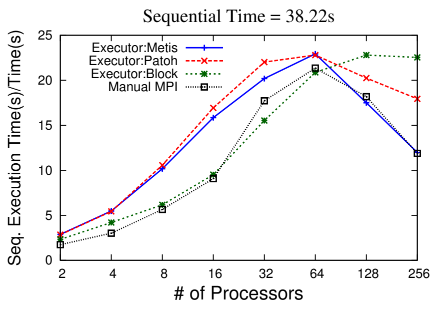
For our experimental evaluation we used a cluster with Intel Xeon E5630 processors with 4 cores per node and a clock speed of 2.67GHz, with an Infiniband interconnect. MVAPICH2-1.7 was used for MPI communications, along with Global Arrays 5.1 for the ARMCI one-sided communications. All benchmarks/applications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

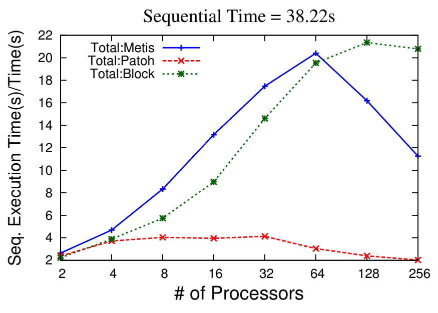
For all benchmarks and applications, all functions were inlined, and arrays of structures were converted to structures of arrays for use with our prototype compiler which implements the transformations described earlier. The compiler was developed in the ROSE infrastructure [ROSE]. Further, the performance of the generated code for each application was compared against manual MPI implementations.

8.1. Benchmarks

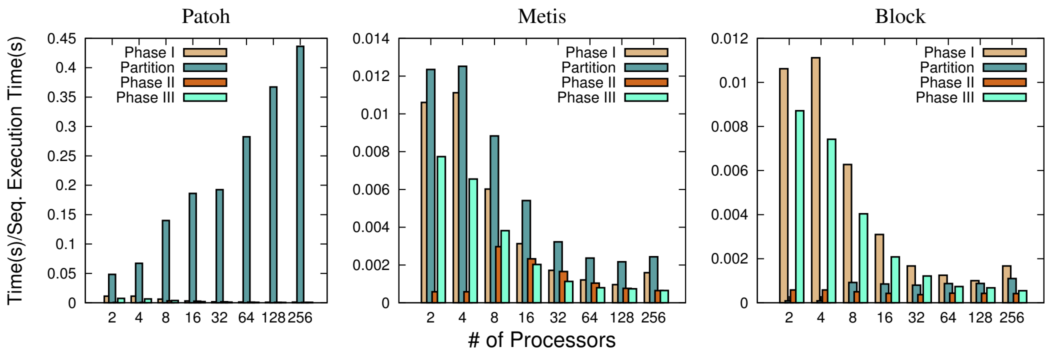
For evaluation purposes, we used benchmarks with data-dependent control-flow and array-access patterns. Each benchmark has a sequence of partitionable loops enclosed within an outer sequential (time or convergence) loop, with the control-flow and array-access pattern remaining the same for



(a) Executors, manual-MPI times



(b) Inspector + Executor times



(c) Inspector Overhead Breakdown

Fig. 5. 183.equake with *ref* input size

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.

8.1.1. 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. We also developed an MPI implementation of this benchmark for evaluation purposes. Figure 5a shows that the performance of the generated executor code for all three partitioning schemes 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.

Figure 5c shows execution times for the graph partitioniner and for each of the phases described in Section 5. As expected, PaToH being a sequential partitioner, the graph partitioning overhead increases logarithmically with the number of partitions. Using ParMetis, significantly reduces this cost.

8.1.2. 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. While *hood* has 220542 rows and 9895422 non-zero elements, *tmt_sym* has more (726713) rows and lesser (5080961) non-zero elements. The structure of the latter is such that the non-zero elements fall along diagonals of the matrix.

Figures 6a and 6d show that the executor code achieves good scaling overall with better than ideal 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*. Due to the structure of 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 6e. This cost could be further amortized in cases where the linear system of equations 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 executors perform on par with the manual implementation for all three partitioning schemes (Figure 6d) upto 128 processes.

The break down of the inspector overheads for each of the matrices with different partitioning schemes are shown in Figures 6c and 6f. It is interesting to note that the execution time of Phase III of the inspector is lower when using Metis than with Block. Since this phase populates the indirection arrays needed by the executor, lesser ghosts results in lesser computation. At the same time, the block partitioning scheme has lesser overhead for Phase II of the inspector since the iterations of the partitionable loop analysed in this phase are same as those analyzed in Phase I. The values of indirection arrays have already been prefetched, reducing the amount of communication required for this phase.

8.1.3. P3-RTE Benchmark [Ravishankar et al. 2010]

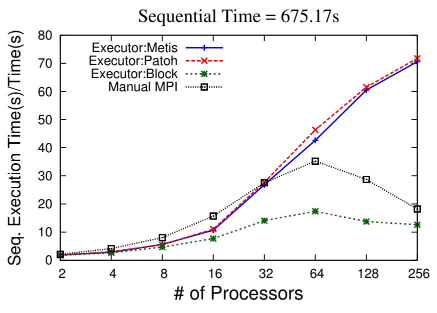
This benchmark solves the radiation transport equation (RTE) [Modest 2003] approximated using spherical harmonics on an unstructured physical grid of 164540 triangular cells. Finite-Volume Method is used for discretizing the RTE with Jacobi method used for solving the system of equations at each cell center. 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. Since the partitioning scheme used by the auto-generated code groups together iterations that touch the same data elements, the generated partitions are similar to those generated by the domain decomposition scheme used by the manual MPI implementation. As a result, the executor code while using PaToH or ParMetis achieves performance comparable to the manual MPI implementation up to 32 processes. Since the block-partitioning scheme does not do this, the performance of the corresponding executor code achieves poor performance.

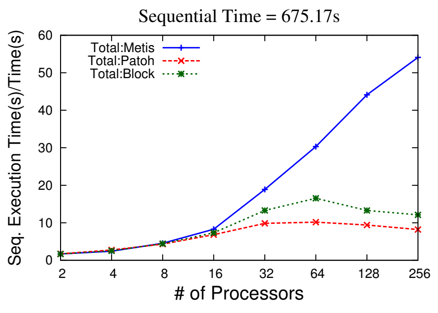
Past 32 processes, the manual implementation continues to achieve scalable performance by replicating some of the computation on multiple processes, significantly reducing the communication costs. Since our scheme strictly partitions the computation across processes without any replication, the generated executor code has to perform five collective communications instead of two such communications performed by the manual implementation. Automatically identifying computations which when replicated significantly reduce communication costs would be an interesting avenue to explore. Figure 7b shows that the inspector overhead is negligible even when using the sequential PaToH partitioner.

8.1.4. miniFE-1.1 [Heroux et al. 2009]

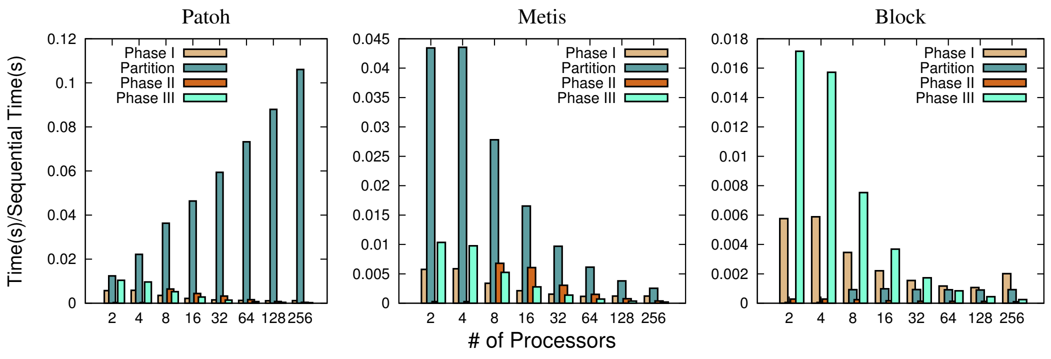
This is a mini-application from the Mantevo suite developed by Sandia National Laboratories [Mantevo]. It uses an implicit finite-element method over an unstructured 3D mesh. A problem size of 100 points along each axes was used for the evaluation. The suite also provides a manual MPI implementation of the computation which was used for comparison.



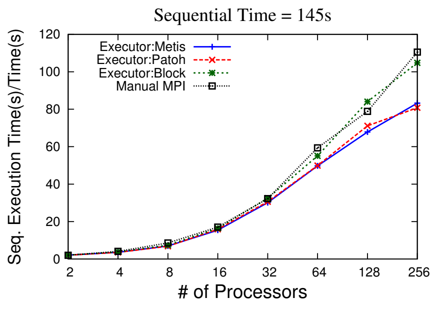
(a) Executors times - *hood.rb*



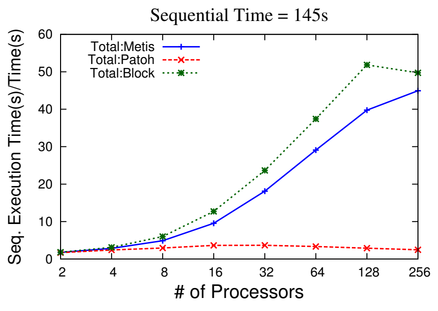
(b) Total times - *hood.rb*



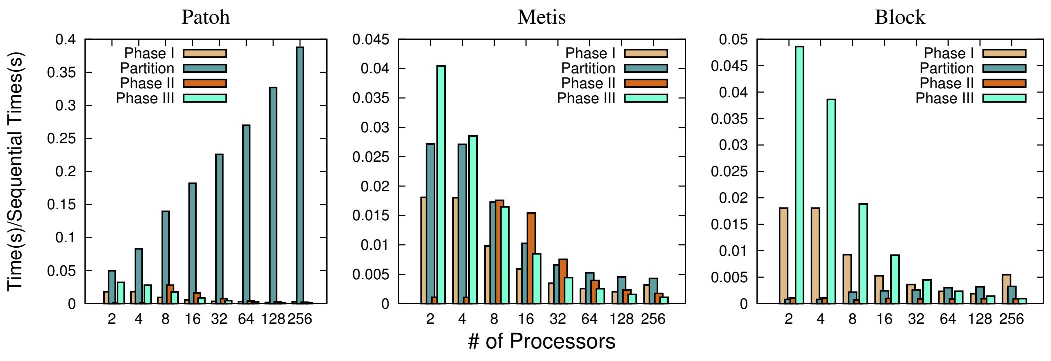
(c) Inspector Overhead Breakdown - *hood.rb*



(d) Executor times - *tmt_sym.rb*

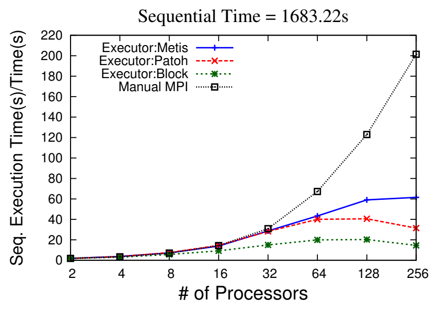


(e) Total times - *tmt_sym.rb*

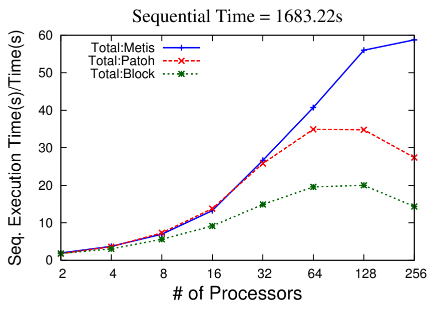


(f) Inspector Overhead Breakdown - *tmt_sym.rb*

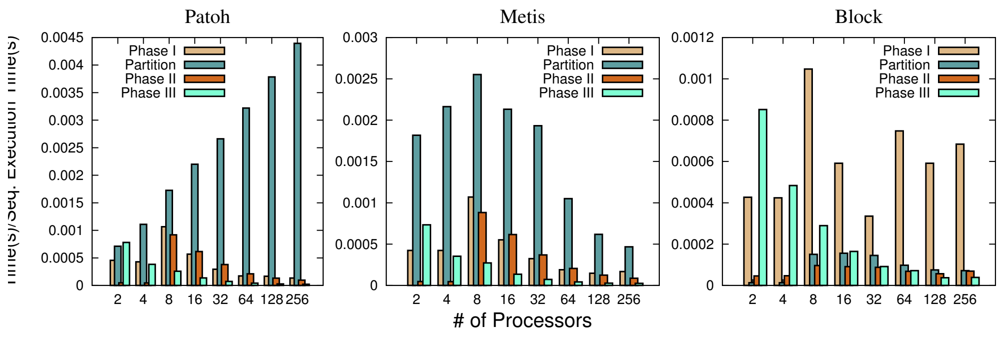
Fig. 6. CG Kernel with *hood.rb* and *tmt_sym.rb*



(a) Executors, manual-MPI times



(b) Inspector + Executor times



(c) Inspector Overhead Breakdown

Fig. 7. P3-RTE on unstructured mesh

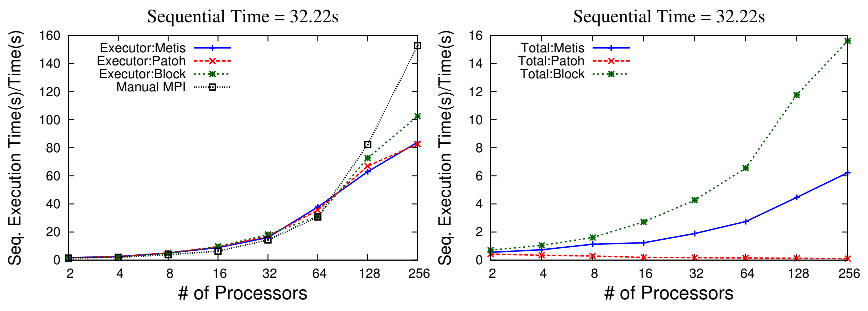
Figure 8 compares the running times for the executors (using ParMetis, PaToH, and block-partitioning) with the execution time of the manual MPI implementation. Up to 128 processes, the performance of the auto-generated executor is on par with the manual implementation. The communication costs in the manual implementation are reduced by overlap of communication and computation. This aspect can be incorporated into the current code-generation scheme by static analysis of the def-use relationship of arrays between the different partitionable loops.

Figure 8b and 8c show the impact of the inspector overheads on the total speedup achieved and a break down of the inspector execution time. Since the actual running time of the application is not very significant even for the large problem size, the cost of the inspector dominates the overall running time. It should be noted that this mini-application doesnt capture the behaviour of unsteady Finite Element applications that have an additional outer-time loop. Hoisting the inspector out of the time loop would further amortize the inspector overheads.

8.2. OLAM

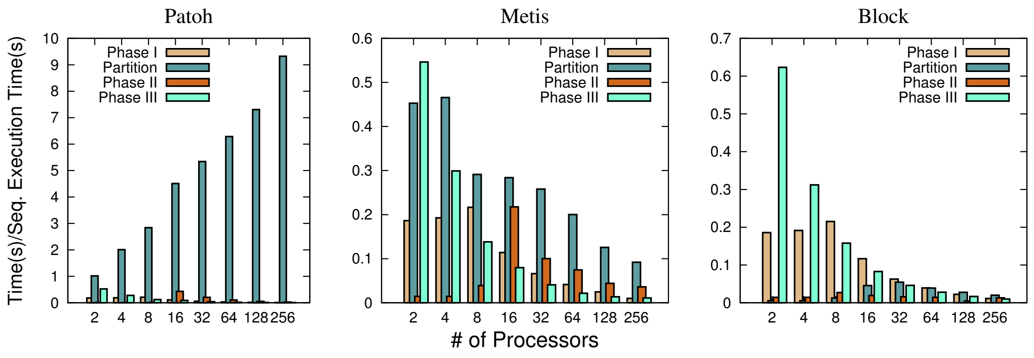
[Walko and Avissar 2008]

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. We applied these techniques to parallelize a real-world application called OLAM (Ocean, Land, and Atmosphere Modeling) used for climate simulations of the entire planet written in Fortran 90. It employs finite-volume methods of discretization to solve for physical quantities such as pressure, temperature, and wind velocity over an 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.



(a) Executors, manual-MPI times

(b) Inspector + Executor times



(c) Inspector Overhead Breakdown

Fig. 8. miniFE-1.1 with 100x100x100 grid

Since our current compiler implementation targets C codes and does not do any inter-procedural analysis, it was not possible to automatically generate the distributed memory code for this application. Therefore for evaluation purposes, we manually implemented the code that would be generated by a full-fledged compiler using the approach described in this paper. We focussed on the atmospheric model simulation which consists of 13 partitionable loops enclosed within a sequential time loop. While the outer loop typically executes hundreds of thousands of iterations, Figure 9 shows data for 30000 iterations. Since the inspector is hoisted out of this timeloop, the overhead of the inspector is almost negligible, even when using the sequential hypergraph partitioner for generating the partitions. Therefore, the result shown in this section were obtained using PaToH.

We compared the performance of our implementation with a reference MPI implementation developed by domain experts. The latter uses an efficient domain decomposition scheme to partition the computation across the MPI processes. Figure 9 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 better than ideal scaling achieved by the manual implementation. Note that the I/E version still achieves ideal scaling.

8.3. Impact of Exploiting Contiguity

The code-generation scheme described in this paper takes special care to preserve the contiguous accesses present in the original code within the generated executor code. The advantage of this

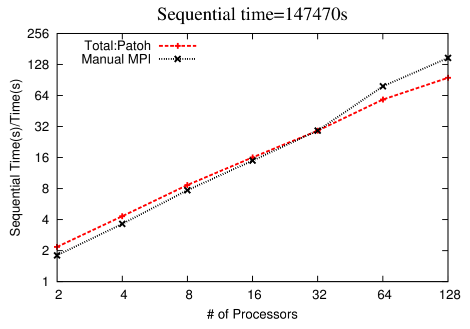


Fig. 9. OLAM atmospheric model.

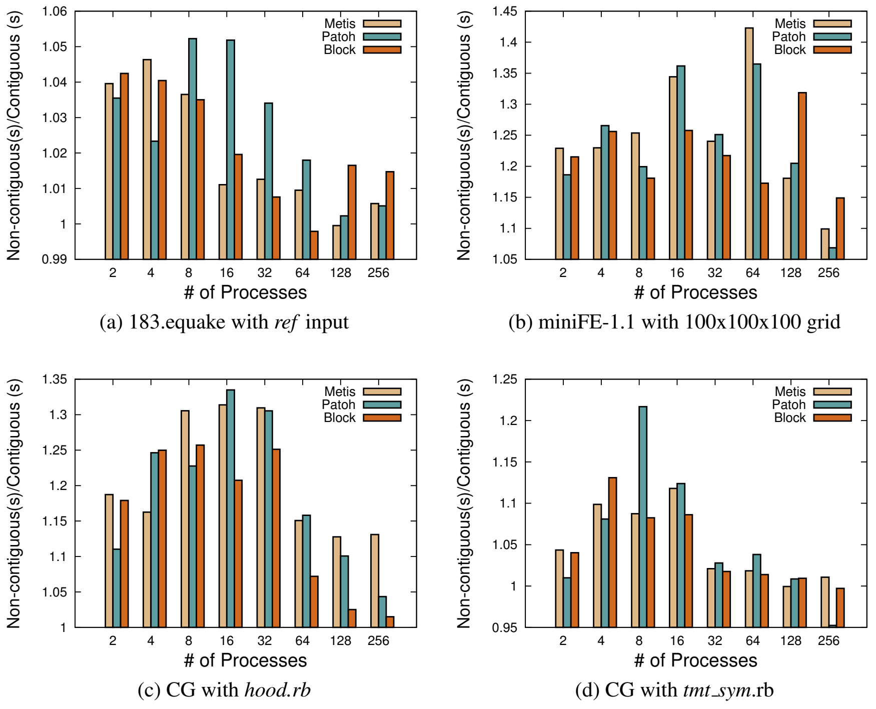


Fig. 10. Impact of exploiting contiguity

— At most one read from an indirection array is needed for every set of contiguous access to data arrays reducing the total memory request made by the executor code
— The size of the indirection array needed is significantly reduced resulting in a small footprint of the executor
— The resulting array access expression might enable future compiler optimizations like memory prefetching which could further improve the executor code performance.

Figure 10 shows the improvements in total run-time of the executor with and without this optimization applied to inner-loops alone for all the bechmarks where this optimization was applied. The benefit of the optimization is especially important in benchmarks that have a high foot-print, like the CG kernel with *hood.rb* and miniFE-1.1 (Figures 10c and 10b), resulting in a 25% reduction in running-times. Since the footprint for CG kernel with *tmt_sym.rb* is almost half of that with *hood.rb*, the maximum benefit for the former occurs at 8 processes and for the latter at 16 processes. As the number of processes used increases, the footprint of the computation on a process fits in some level of cache, reducing the benefit of the optimization. Similarly, for *183.equake* which has a small footprint to begin with, the benefit is within 5%.

9. 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, but could handle only a single level of indirection. 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[j]` in Listing 1). While their approach could handle multiple levels of indirection, it used the owner-computes paradigm and utilized the `EXECUTE-ON-HOME` directive of HPF to partition the computation. Such techniques are inapplicable to codes such as 183.equake and P3-RTE, where multiple elements of an array are updated within an inner loop.

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; Strout et al. 2012] 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; Griebl 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 Zahorjan 1993; Rus et al. 2002; Yu and Rauchwerger 2000; Oancea and Rauchwerger 2012]. 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.

10. 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.

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. The benefit of exploiting contiguity of data accesses within inner loops was also demonstrated. Future work would focus on optimizing the communication layer to achieve better scalability of the generated code.

11. 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

AGRAWAL, G., SALTZ, J., AND DAS, R. 1995. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *PLDI*.

BALAY, S., BROWN, J., BUSCHELMAN, K., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., SMITH, B. F., AND ZHANG, H. 2012. PETSc Web page. http://www.mcs.anl.gov/petsc.

BAO, H., BIELAK, J., GHATTAS, O., KALLIVOKAS, L. F., O'HALLARON, D. R., SHEWCHUK, J. R., AND XU, J. 1998. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering 152*, 85–102.

BASKARAN, M. M., RAMANUJAM, J., AND SADAYAPPAN, P. 2010. Automatic C-to-CUDA code generation for affine programs. In *CC*.

BASUMALLIK, A. AND EIGENMANN, R. 2006. Optimizing irregular shared-memory applications for distributed-memory systems. In *PPoPP*.

BERRYMAN, H., SALTZ, J., AND SCROGGS, J. 1991. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience 3*, 159–178.

BONDHUGULA, U., GUNLUK, O., DASH, S., AND RENGANARAYANAN, L. 2010. A model for fusion and code motion in an automatic parallelizing compiler. In *PACT*.

BONDHUGULA, U., HARTONO, A., RAMANUJAN, J., AND SADAYAPPAN, P. 2008. A practical automatic polyhedral program optimization system. In *PLDI*.

CATALYUREK, U. V. AND AYKANAT, C. 2009. *PaToH: Partitioning Tool for Hypergraphs*.

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

DAS, R., SALTZ, J., AND VON HANXLEDEN, R. 1993. Slicing analysis and indirect access to distributed arrays. Tech. Rep. CRPC-TR93319-S, Rice University.

DAVIS, T. A. 1994. University of Florida sparse matrix collection. *NA Digest*.

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

ESWAR, K., SADAYAPPAN, P., AND HUANG, C.-H. 1993. Compile-time charactirization recurrent patterns in irregular computations. *ICPP 2*, 148–155.

FEAUTRIER, P. 1992. Some efficient solutions to the affine scheduling problem - Part I: One-dimensional time. *IJPP 21*, 313–347.

GRIEBL, M. 2004. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. FMI, University of Passau.

HALL, M., CHAME, J., CHEN, C., SHIN, J., RUDY, G., AND KHAN, M. M. 2010. Loop transformation recipes for code generation and auto-tuning. In *LCPC*.

HAN, H. AND TSENG, C.-W. 2006. Exploiting locality for irregular scientific codes. *IEEE Trans. Parallel Distrib. Syst. 17*, 606–618.

HEROUX, M. A., DOERFLER, D. W., CROZIER, P. S., WILLENBRING, J. M., EDWARDS, H. C., WILLIAMS, A., RAJAN, M., KEITER, E. R., THORNQUIST, H. K., AND NUMRICH, R. W. 2009. Improving performance via mini-applications. Tech. Rep. SAND2009-5574, Sandia National Laboratories.

IRIGOIN, F. AND TRIOLET, R. 1988. Supernode partitioning. In *POPL*.

LAIN, A. 1996. Compiler and run-time support for irregular computations. Ph.D. thesis, University of Illinois at Urbana-Champagne.

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

LAMIELLE, A. AND STROUT, M. 2010. Enabling code gen. with sparse polyhedral framework. Tech. Rep. CS-10-102, Colorado State University.

LEUNG, S.-T. AND ZAHORJAN, J. 1993. Improving the performance of runtime parallelization. In *PPoPP*.

LIM, A. W. AND LAM, M. S. 1997. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL*.

Mantevo. Mantevo project. `https://software.sandia.gov/mantevo`.

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

MODEST, M. F. 2003. *Radiative Heat Transfer*. Academic Press.

NEIPLOCHA, J., TIPPARAJU, V., KRISHNAN, M., AND PANDA, D. K. 2006. High performance remote memory access communication: The ARMCI approach. *Int. J. High Performance Computing Applications 20*, 233–253.

OANCEA, C. E. AND RAUCHWERGER, L. 2012. Logical inference techniques for loop parallelization. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI '12.

Par4All 2012. Par4all. `www.par4all.org`.

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

RAMANUJAM, J. AND SADAYAPPAN, P. 1992. Tiling multidimensional iteration spaces for multicomputers. *JPDC 16,* 2, 108–230.

RAUCHWERGER, L. AND PADUA, D. 1995. The LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization. In *PLDI*.

RAVISHANKAR, M., MAZUMDER, S., AND KUMAR, A. 2010. Finite-volume formulation and solution of the p3 equations of radiative transfer on unstructured meshes. *Journal of Heat Transfer 132*, 0123402.

ROSE. Rose compiler infrastructure. `www.rosecompiler.org`.

RUS, S., PENNINGS, M., AND RAUCHWERGER, L. 2002. Sensitivity analysis for automatic parallelization on multi-cores. In *ICS*.

SALTZ, J., CROWLEY, K., MIRCHANDANEY, R., AND BERRYMAN, H. 1990. Run-time scheduling and execution of loops on message passing machines. *J. Parallel Distrib. Comput. 8*, 303–312.

SALTZ, J. H., BERRYMAN, H., AND WU, J. 1991. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*.

SCHLOEGEL, K., KARYPIS, G., AND KUMAR, V. 2002. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience 14*, 219–240.

STROUT, M. M., CARTER, L., AND FERRANTE, J. 2003. Compile-time composition of run-time data and iteration reodrderings. In *PLDI*.

STROUT, M. M., CARTER, L., FERRANTE, J., FREEMAN, J., AND KREASECK, B. 2002. Combining performance aspects of irregular Gauss-Seidel via sparse tiling. In *LCPC*.

STROUT, M. M., GEORGE, G., AND OLSCHANOWSKY, C. 2012. Set and relation manipulation for the sparse polyhedral framework. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*.

STROUT, M. M. AND HOVLAND, P. D. 2004. Metrics and models for reordering transformations. In *MSP*.

VON HANXLEDEN, R., KENNEDY, K., KOELBEL, C., DAS, R., AND SALTZ, J. 1993. Compiler analysis for irregular problems in Fortran D. *LCPC*.

WALKO, R. L. AND AVISSAR, R. 2008. The Ocean-Land-Atmosphere Model (OLAM). Part I: Shallow-Water Tests. *Monthly Weather Review 136*, 4033–4044.

YU, H. AND RAUCHWERGER, L. 2000. Techniques for reducing the overhead of run-time parallelization. In *CC*.

ZHUANG, X., EICHENBERGER, A. E., LUO, Y., O'BRIEN, K., AND O'BRIEN, K. 2009. Exploiting parallelism with dependence-aware scheduling. In *PACT*.

Code Generation for Parallel Execution of a Class of Irregular Loops on Distributed Memory Systems

Mahesh Ravishankar*, John Eisenlohr*, Louis-Noël Pouchet*, J. Ramanujam†, Atanas Rountev*, P. Sadayappan*
*The Ohio State University, †Louisiana State University
Email: *{ravishan,eisenloh,pouchet,rountev,saday}@cse.ohio-state.edu,†jxr@ece.lsu.edu

***Abstract*—Parallelization and locality optimization of affine loop nests has been successfully addressed for shared-memory machines. However, many large-scale simulation applications must be executed in a distributed-memory environment, and use irregular/sparse computations where the control-flow and array-access patterns are data-dependent.**

In this paper, we propose an approach for effective parallel execution of a class of irregular loop computations in a distributed-memory environment, using a combination of static and runtime analysis. We discuss algorithms that analyze sequential code to generate an *inspector* and an *executor*. The inspector captures the data-dependent behavior of the computation in parallel and without requiring complete replication of any of the data structures used in the original computation. The executor performs the computation in parallel. The effectiveness of the framework is demonstrated on several benchmarks and a climate modeling application.

I. Introduction

Automatic parallelization and locality optimization of affine loop nests have been addressed for shared-memory multiprocessors and GPUs with good success [4], [7], [8], [16], [29], [30]. 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 is inapplicable in such cases.

We propose a framework for automatic parallelization and distributed-memory code generation for an extended class of affine computations that allow some forms of indirect array accesses. The class we address is prevalent in many scientific/engineering domains and the paradigm for its parallelization is often called the inspector/executor (I/E) [38] approach. The I/E approach uses (1) a so-called *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) to construct distributed data structures and computation partitions, and (2) an *executor* code that uses data structures generated by the inspector to execute the desired computation using parallelism.

The I/E approach has been well known in the high-performance computing community, since the pioneering work of Saltz and coworkers [38] 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 Sec. VII) 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 transformation system for effective automatic parallelization and distributed-memory code generation for an extended class of affine programs;
- It develops an efficient approach for generating parallel code with a lower degree of indirect array access than any previously proposed algorithms for the class of computations handled; and
- It presents experimental results on the use of the approach to develop a complex parallel application, with performance approaching that of manual parallelization implemented by expert application developers.

The rest of the paper is organized as follows. Section II describes the class of extended affine computations that we address, along with a high-level overview of the approach to code transformation. Section III provides details of the approach to generate computation partitions using a hypergraph that models the affinity of loop iterations to data elements accessed. The algorithms for generation of inspector and executor code are provided in Section IV. Section V elaborates on how the need for inspector code can be optimized away for portions of the input code that are strictly affine. Experimental results using four kernels and one significant application are presented in Section VI. Related work is discussed in Section VII and conclusions stated in Section VIII.

II. 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`. 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

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00©2012 IEEE

```
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 */
 }
```

Listing 1. Sequential conjugate gradient computation.

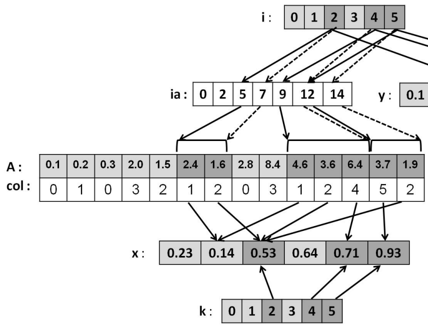


Fig. 1. Control flow and data-access patterns of iteration 2, 4, and 5, of loops i and k mapped to process 0.

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 referred to as *indirection arrays*. All other arrays will be referred to as *data arrays* (x, y, and A).

The goal is to parallelize the computation by partitioning the iterations of loops i and k among a 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 latter for the running example.

Targeted computations. We target a class of computations that are more general than *affine* computations. In affine codes, the loop bounds, conditionals of ifs, and array-access expressions

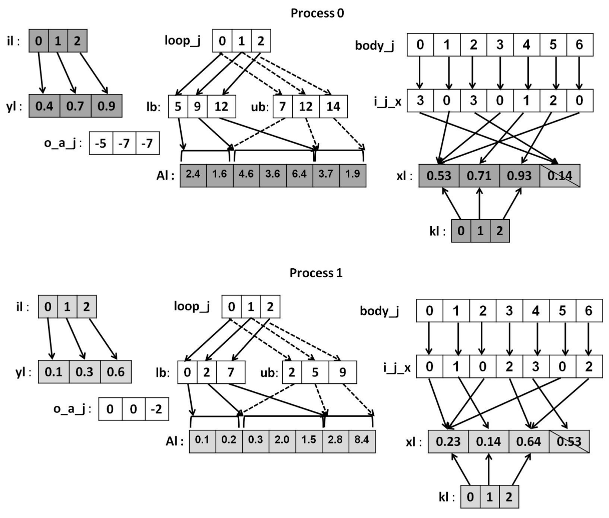


Fig. 2. Transformed iteration and data view.

are affine functions of loop iterators and 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 ia, and accesses to x depend on 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 affine functions of iterators, parameters, and values stored in indirection arrays. Further, values in these indirection arrays may themselves be accessed through other indirection arrays.

Within this extended class, we target loops that are parallel, except for loop-carried dependences due to reductions of scalars or array elements using operators which are associative and commutative. For any scalar or array element participating in such reductions, its value should not be used for any other computation within the loop body. Values of indirection arrays must not be modified within the loop. Loops that satisfy these properties will be referred to as *partitionable loops*. Loops i and k in Listing 1 are partitionable. For code transformations, only partitionable loops not nested within each other are considered. A formal definition of partitionable loops can be found in [34].

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.

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 III) may

```
/* Inspector code to generate o_a_j, i_x_j, lb,*/
/* ub, xl, yl, Al and to compute nl; not shown */
while( !converged ){
  /* Other computation; not shown */
  /* Update ghosts */
  body_k = 0;
  for( kl = 0 ; kl < nl ; kl++ ){
    xl[kl] = ...; body_k++; }
  /* Update owners */
  /* Other computation; not shown */
  /* Update ghosts */
  body_i = 0; loop_j = 0 ; body_j = 0;
  for( il = 0 ; il < nl ; il++ ) {
    offset_a_j = o_a_j[loop_j];
    for( j = lb[loop_j] ; j < ub[loop_j] ; j++ ){
      yl[il] += Al[j+offset_a_j]*xl[i_x_j[body_j]];
      body_j++;    }
    loop_j++; body_i++;  }
  /* Update owners */
 }
```

Listing 2. Parallel conjugate gradient computation.

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).

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 loop on process 0 (shown as `lb` and `ub` in Figure 2). Conditionals of `if`s are handled similarly, by storing their values in local arrays.

Partitioning the data. Once the iterations have been partitioned, the data is 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, `yl`, `Al`, and `xl` are the local arrays on process 0 for `y`, `A`, and `x`.

The same 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 other processes 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 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 arrays whose values are updated within the loop are initialized to the identity element of the update operator (0 for "+=", 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.

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 `xl` is the local copy of `x` on each process, all elements of `x` accessed by a process are represented in `xl`. To access the correct elements in `xl`, 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` executed 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]`.

Optimizing accesses from inner loops. The procedure 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 `o_a_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.

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 xl can be accessed by using iterator kl if xl[0-2] correspond to x[2], x[4], and x[5]. The same could be done for y[i] in Listing 1 since there are no other accesses to it.

If the same array is accessed by an 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 the accesses the loops must be partitioned in a similar way in order to obtain a consistent ordering of the array elements (Section IV-A).

Executor code. To execute the original computation on each process, 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 nl that are mapped to a process. The loop bounds of loop j are read from arrays lb and ub. Accesses to local arrays xl and Al are determined by values in arrays i_x_j and o_a_j. Also, communication calls are inserted to satisfy the producer-consumer relationship due to array x.

III. Partitioning the Computation

The computation is partitioned by considering the iteration-data affinity. To model this affinity, we use a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ where $\mathcal{V}$ is the set of vertices and $\mathcal{N}$ is the set of nets [43]. Iterations of all partitionable loops are represented by separate vertices $i \in \mathcal{V}$. 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.

Achieving load balancing. Each vertex is associated with a vector of weights $\vec{w_i}$ 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 $W_P = \sum_{i \in P} \vec{w_i}$. 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 $P_{avg} = P_{\mathcal{V}}/N$; ϵ is the maximum load imbalance tolerated.

Minimizing communication. Each net in $\mathcal{N}$ 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 *internal* nets, I_n and nets with pins in other partitions are *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 λ_j that have a ghost for the data element represented by $j \in \mathcal{N}$. This is achieved by minimizing the *cut-size* Π_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 [9].

IV. Inspector: Functionality and Code Generation

This section describes the run-time inspector analysis required to create the iteration partitions (along with the required data) for each process. The process has 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.

A. Phase I: Hypergraph Generation

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 contains only statements from the original computation that affect the control flow and array accesses.

The inspector (for the sake of analysis) starts with the assumption that all arrays are block-partitioned across the processes. Each process analyzes a block-partitioned subset of the original iteration (represented by [kstart,kend) for loop k and [istart,iend) 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. For every data array element that is accessed by this iteration, the vertex is added as a pin to the corresponding net. For example, AddPin(id_y,i,vi,1) adds vertex vi as a pin to the net for the i-th element of array y (id_y is a unique id for y). The last argument specifies that the element is accessed by an expression with a unit stride.

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

```
do{  for( k = kstart ; k < kend ; k++ ){
       AddVertex(id_k,vk);
       AddPin(id_x,k,vk,1); ...}
     for( i = istart ; i < iend ; i++ ){
       AddVertex(id_i,vi);
       if(is_known(id_ia,i) && is_known(id_ia,i+1))
         for( j = get_elem(id_ia,i) ;
            j < get_elem(id_ia,i+1) ; j++ ){
           AddPin(id_y,i,vi,1); AddPin(id_A,j,vi,1);
           if( is_known(id_col,j) )
               xindex = get_elem(id_col,j);
           if( is_known(id_xindex) )
             AddPin(id_x,xindex,vi,0);      }
     }while( DoneGraphGen() );
```

Listing 3. Phase I of the inspector.

of the outer blocked-partitioned loop, as shown in Listing 3.

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 III 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.

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. To analyze the control-flow and array-access patterns of the original computations, all loops and conditional statements are included in the inspector code with certain modifications, as discussed below. Since scalars might be involved in loop bounds, conditionals, and index expressions of arrays, all assignments to such scalars are also included. For example, `xindex` in Listing 1 is used to access array `x`. Therefore, statement `xindex=col[j]` has to be executed by the inspector to capture the elements of `x` accessed. Scalars that are used (directly or transitively) to determine control flow or array-access expressions inside a partitionable loop (referred to as *inspected scalars*) are handled in this manner.

To ensure that values not yet known on a process are not used, for all array elements and inspected scalars, boolean state variables are maintained. Loops/branches are executed only if all data accessed in the bounds/conditionals are known on a process, as illustrated by the calls to `is_known`. Similarly, inspected scalars are assigned values only if all elements on the right-hand side are known.

To support the optimizations of accesses from partitionable loops as discussed in Section II, it might be necessary to ensure that multiple partitionable loops are partitioned the same way. To enforce this, the loop bounds of all such partitionable loops are checked at compile time. If they are the same, `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 are not optimized.

B. Initializing Local 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 ghosts locations for elements owned by other processes. This is done for all data arrays in the computation. For example, arrays `yl`, `xl` and `Al` of Listing 2 are allocated at this time.

A compile-time analysis determines expressions that result in unit-stride accesses to a data array due to a surrounding loop, with the conflict between accesses from a partitioned loop and from an inner loop resolved statically. Elements accessed by such expressions (known at run time using 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 in the transformed code and within partitionable loops when they do not conflict with the former.

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

Run-time functionality. The next step is to determine the sizes of arrays used to (1) store loop bounds of inner loops, (2) store the result of conditionals, and (3) store the indices of accessed data array elements. The size of these arrays depends on the expressions they represent. Every array-access expression is analyzed at compile time to determine if the access is unit-stride with respect to a surrounding loop. For such an expression, the size of the array needed would be the same as the number of invocations of the corresponding 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 this loop across all iterations of the partitioned loop.

The sizes 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_*` track the total number of times a loop body is executed. In addition, 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 hypergraph. As in Section IV-A, all inner loops, conditionals, and statements are guarded to check if the values of inspected scalars and indirection array elements have been determined. Again, this phase is completed only after all levels of indirections have been resolved.

Code generation at compile time. The code generation for this phase is similar to Phase I. Only statements that affect the control-flow and array-access patterns are considered. The

```
do{
  body_i = 0 ; loop_j = 0 ; body_j = 0; body_k = 0;
  for( k = 0 ; k < n ; k++ )
    if( home(id_k,k) == myid )
      body_k++;
  for( i = 0 ; i < n ; i++ )
    if( home(id_i,i) == myid ){
      if( is_known(id_ia,i) && is_known(id_ia,i+1)){
        for( j = get_elem(id_ia,i) ;
             j < get_elem(id_ia,i+1) ; j++ ){
          if( is_known(id_col,j) )
            xindex = get_elem(id_col,j);
          body_j++;                                   }
        loop_j++;                                }
      body_i++;                      }
}while( DoneCounters() );
```

Listing 4. Phase II of the inspector.

```
body_i = 0 ; loop_j = 0 ; body_j = 0; body_k = 0;
for( k = 0 ; k < n ; k++ )
  if( home(id_k,k) == myid )
    body_k++;
for( i = 0 ; i < n ; i++ )
  if( home(id_i,i) == myid ){
    lb_j[loop_j] = get_elem(id_ia,i);
    ub_j[loop_j] = get_elem(id_ia,i+1);
    for( j = lb_j[loop_j] ; j < ub_j[loop_j] ; j++){
      xindex = get_elem(id_col,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.

differences from Phase I are (1) the bounds of the partitionable loop are same as the original computation and its body is enclosed in an `if` statement that checks if the iteration is to be executed on the current process, and (2) statements to increment counters `loop_*` and `body_*` are introduced.

D. Phase III: Initializing Local Access Arrays

Run-time functionality. After allocation, the access arrays are initially populated with the sequence of values the corresponding expression evaluates to in the original computation. Listing 5 shows the code to do so, for the example in Listing 1.

For expressions that are unit-stride with respect to a surrounding inner loop, the element accessed by the first iteration of every invocation of the loop is stored in the array that represents the index expression. For all other expressions, the values for all iterations are stored in arrays. The value of the loop bounds of all inner loops and results of conditionals are also stored in arrays in this phase of the inspector.

Code generation at compile time. To generate the code for this phase, once again the partitionable loop is replicated as is, with the loop body enclosed within an `if` statement to analyze only the iterations mapped to the current process. The body of this new loop is generated by traversing the statements of the original partitionable loops, as shown in Algorithm 1.

Assignment Statements: On encountering such statements in the original AST, a corresponding statement is added to the AST of this phase of the inspector by Algorithm 1. Statements that are assignments to inspected scalars are replicated with references to indirection arrays replaced with calls to `get_elem` by *CovertArraysToFunctions*. For example, the right-hand side of `xindex=col[j]` in Listing 1 is modified

Algorithm 1: CodeToInitializeAccessArrays(s,A,$\mathcal{C}$,S,$\mathcal{A}_P$)

```
Input : $s$ : Statement in the original AST; $A$ : Access Arrays
        $\mathcal{C}$ : Counter Variables; $S$ : Inspected Scalars
InOut: $\mathcal{A}_P$ : AST of code to populate the access arrays
begin
  if $s.LHS \notin S$ then
    foreach $d \in GetDataArrayRefExp(s)$ do
      $a$ = AccessArray($A$,$d$.Array,$d$.IndexExpr) ;
      $c$ = CounterVariable($\mathcal{C}$,$d$.IndexExpr) ;
      $l_P$ = CreateStoreAccessArray($a$,$c$,$d$.IndexExpr) ;
      if IsUnitStride(d.IndexExpr) then
        $l$ = GetLoop($c$) ; $l_P$ = CreateIfFirstIter($c$,$l_P$) ;
      $\mathcal{A}_P$.Append($l_P$) ;
  else
    $b$ = ConvertArraysToFunctions($s$.RHS) ;
    $l_p$ = AssignmentStatement($s$.LHS,$b$) ;
    $\mathcal{A}_P$.Append($l_P$) ;
```

to `get_elem(id_col,j)`. There is no need for any guards in this phase since the previous phase ensured that all values needed are known on the current process.

For all other assignment statements, function *GetDataArrayRefExp* returns the set of all expressions used to access data array elements. For every such expression, an assignment is generated by function *CreateStoreAccessArray* to store the value of the corresponding index expression, with all references to indirection arrays replaced with calls to `get_elem`.

Further, if the index expression is unit-stride with respect to a surrounding loop, the statement is enclosed within an `if` statement, generated by *CreateIfFirstIter*, which checks if the value of the iterator of the loop is 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.

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 arrays being accessed. For access arrays that represent expressions that are unit-stride with respect to a loop, the modified values are element-wise subtracted with the values stored in the lower-bound array of the loop. Adding the loop iterator value to this would point to the correct location.

Loops and conditionals: Corresponding to inner loops within the original partitionable loop, statements to store the value of the current upper/lower bounds of the loop are inserted in the AST. The loop itself is inserted after these statements, with the bounds modified to use these stored values. Conditional statements are treated similarly.

E. 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 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.

Assignment Statements: For such a statement in the original code, the corresponding statement in the executor is generated as follows. Statements that write to inspected scalars are not inserted in the executor, since the control flow and array accesses are handled explicitly through arrays. For all other assignment statements, accesses to original data arrays are replaced with accesses to corresponding local arrays. The generated index expressions depend on the original index expressions. For those that are unit-stride with respect to a loop, the index expression is the sum of the iterator value and the value stored in the corresponding access array. For example, in Listing 2, the index to array `Al` is obtained by adding the offset stored in `o_a_j` to `j`. This index expression ensures unit-stride access to the array. This is important to enable subsequent SIMD optimizations and good spatial locality and cache prefetching. To the best of our knowledge, no previously proposed compiler approaches for I/E code generation ensure this highly-desirable property. The read from the access array is hoisted out of the corresponding loop (loop `j` for the example) since all iterations use the same offset.

For all other accesses, the access arrays store the index of the data array element that is to be accessed. In the executor, the original expression is replaced with a read from the corresponding access array.

Loops and conditionals: Loops and conditionals from the original computation are inserted into the AST of the executor, with loop bounds and conditional expressions modified to read from the arrays populated by Phase III of the inspector. Similar to Phases II and III of the inspector, counters are inserted into the AST of the executor to step through the values stored in the access arrays.

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 used is described below.

F. Communication Between Processes

Elements of local data arrays consist of both owned and ghosts 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 ghosts cells of arrays that are updated are communicated to the owner after the loop execution. For cases where partitionable loops assign values to array elements instead of updating them, the value of the ghost location from the process which executes the last iteration of the original computation is used to overwrite the value at the owner. The ID of the process can be computed by the inspector code while partitioning the computation.

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 [28].

Algorithm 2: CodeGenInspectorExecutor($\mathcal{P}$)

```
   Input  : $\mathcal{P}$ : partitionable loop AST
   Output: $\mathcal{A}_I$: Code for the inspector; $\mathcal{A}_E$: Code for the executor
1  begin
2      $S$ = GetInspectedScalars($\mathcal{P}$) ; $A_I = \phi$ ; $A_E = \phi$ ;
3      $\mathcal{H}$ = CodeToCreateHypergraph($\mathcal{P}$,$S$); $\mathcal{A}_I$.Append($\mathcal{H}$) ;
4      $\mathcal{A}_I$.Append(CodeToPartitionIterations($\mathcal{H}$)) ;
5      $\mathcal{D}$ = CodeToAllocateLocalData($\mathcal{H}$) ;
6      $\mathcal{C}$ = DeclareCounterVariables($\mathcal{P}$); $\mathcal{A}_I$.Append($\mathcal{C}$) ;
7      $\mathcal{A}_C$ = CodeToGetAccessArraySize($\mathcal{P}$,$S$,$\mathcal{C}$) ;
8      $\mathcal{I}$ = CodeToAllocateAccessArrays($\mathcal{P}$); $\mathcal{A}_C$.Append($\mathcal{I}$ );
9      $\mathcal{A}_P$ = CodeToInitializeArrays($\mathcal{P}$,$S$,$\mathcal{C}$,$\mathcal{I}$) ;
10     $\mathcal{A}_P$.Append(CodeToRenumberAccessArrays($\mathcal{C}$,$\mathcal{D}$,$\mathcal{I}$)) ;
11     $\mathcal{A}_I$.Append($\mathcal{A}_C$) ; $\mathcal{A}_I$.Append($\mathcal{A}_P$) ;
12     $\mathcal{A}_E$ = GenerateExecutorCode($\mathcal{P}$,$S$,$\mathcal{C}$,$\mathcal{I}$,$\mathcal{D}$) ;
```

G. Putting Everything Together

The overall code generation scheme is shown in Algorithm 2. ASTs of the original partitionable loops are analyzed to find the inspected scalars. Next, the code to create the hypergraph is generated as described in Section IV-A, followed by the code to partition it. *CodeToAllocateLocalData* generates the code to partition the data arrays, as described in Section IV-B, and allocates local copies for these arrays.

CodeToGetAccessArraySize uses the steps described in Section IV-C to determine the sizes of the access arrays. Code to allocate the access arrays and arrays to store loop-bounds/conditional is then appended to the inspector code. *CodeToInitializeArrays* generates code to initialize these arrays, a part of which is presented in Algorithm 1. The values in the access arrays are then renumbered to point to the correct locations in the local arrays. The executor code is generated as outlined in Section IV-E.

V. 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 since we only partition a single loop, 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 by *CreateStoreAccessArray*

```
/* Original affine computation */
for( i = 0 ; i < n ; i++ )
  for( j = 0 ; j < n ; j++ )
    y[i] += A[i][j] * x[j];

/* Transformed parallel computation */
/* Update ghost read values*/
for( i = 0 ; i < nl ; i++ )
  for( j = 0 ; j < n ; j++ )
    yl[i] += Al[i][j] * xl[j];
/* Communicate ghost write values */
```

Listing 6. Affine conjugate gradient computation (dense matrix).

in Algorithm 1 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 IV. If all arrays are accessed through affine expressions, the inspector is rendered unnecessary and is discarded. Polyhedral techniques can be used to generate the executor code. Listing 6 shows the code obtained for the executor by following this approach.

VI. Evaluation

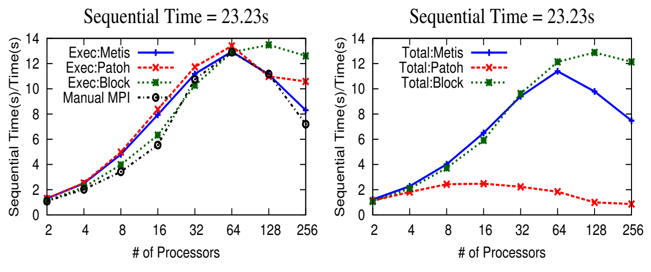
For our experimental evaluation we used a cluster with Intel Xeon E5630 processors with 4 cores per node and a clock speed of 2.67GHz, with an Infiniband interconnect. MVAPICH2-1.7 was used for MPI communications, along with Global Arrays 5.1 for the ARMCI one-sided communications. All benchmarks/applications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

For partitioning hypergraphs, the PaToH hypergraph partitioner [9] 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 [40]. Multi-constraint partitioning (as discussed in Section III) 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 within the ROSE infrastructure [36].

A. Benchmarks and Application

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 and are normalized by the average execution time of the original sequential code.



(a) Executors, manual-MPI times (b) Inspector + Executor times

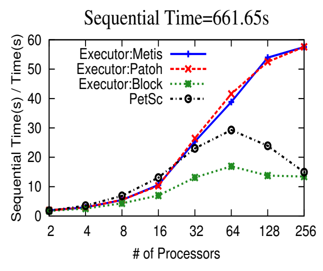
Fig. 3. 183.equake with *ref* input size

183.equake [3] 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 3a 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 3b 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.

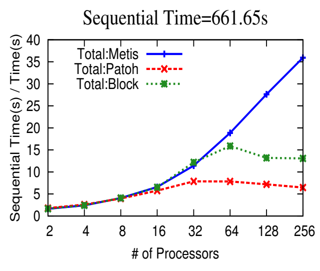
CG kernel The conjugate gradient (CG) method to solve linear system of equations consists of several partitionable loops within a convergence loop. Two sparse matrices, *hood.rb* and *tmt_sym.rb*, from the University of Florida Sparse Matrix Collections [12], stored in CSR format were used as inputs.

Figures 4a, 4c show that the executor code achieves good scaling overall with super-linear scaling between 16 and 64 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 higher 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 4b, 4d. 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.

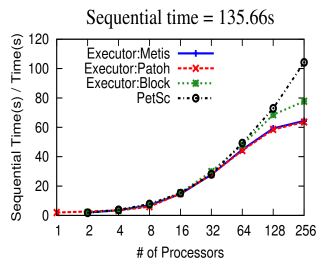
The performance of the executors was also compared to a manual implementation using PETSc [2] which employed a block-partitioning of the rows of the matrix. For *hood*, Figure 4a shows that the performance of the generated executor code while using PaToH and ParMetis out-performs the manual PETSc implementation. The performance of the latter drops off due to the same reason the performance of the block-partitioned scheme drops off. The generated executors



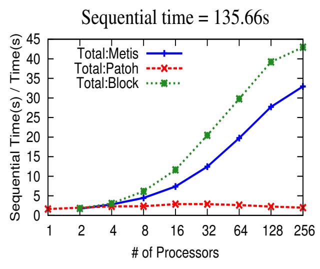
(a) Executors times - *hood.rb*



(b) Total times - *hood.rb*

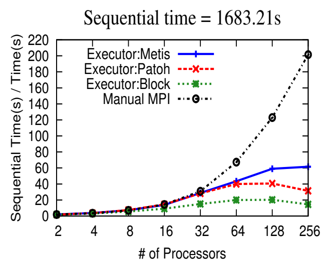


(c) Executor times - *tmt_sym.rb*

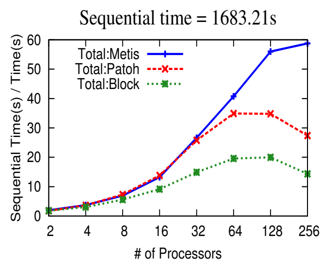


(d) Total times - *tmt_sym.rb*

Fig. 4. CG Kernel with *hood.rb* and *tmt_sym.rb*

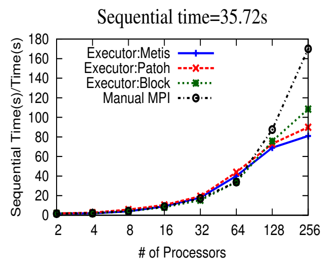


(a) Executors, manual-MPI times

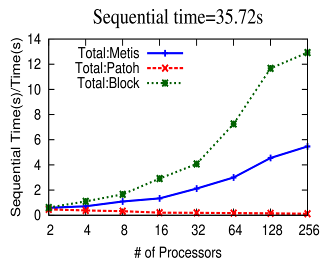


(b) Inspector + Executor times

Fig. 5. P3-RTE on unstructured mesh



(a) Executors, manual-MPI times



(b) Inspector + Executor times

Fig. 6. miniFE-1.1 with 100x100x100 grid

perform better than the manual implementation for *hood* when using PaToH and ParMetis for partitioning. With *tmt_sym*, the generated executor performs on par with the manual implementation for all three partitioning schemes (Figure 4c).

P3-RTE [35] This benchmark solves the radiation transport equation (RTE) [27] approximated using spherical harmonics on an unstructured physical grid of 164540 triangular cells. The Finite-Volume Method is used for discretizing the RTE. Jacobi method is used for solving the system of equations at each cell center. The different partitionable loops iterate over cells, faces, nodes, and boundaries of the domain, and are enclosed within a convergence loop.

Figure 5a 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 higher 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 achieves super-linear scaling since it significantly reduces communication costs by replicating some computation. Using techniques like overlapping communication with computation could improve the performance of the generated executor for higher number of processes. Figure 5b shows that the inspector overhead is negligible even when using the sequential PaToH partitioner.

miniFE-1.1 [18] This is a mini-application from the Mantevo suite from Sandia National Laboratories [25]. It uses an implicit finite-element method over an unstructured 3D mesh. A problem size of 100 points along each axes was used for the evaluation. The suite also provides a manual MPI implementation of the computation. We provide a comparison of the performance of the automatically parallelized version against this manually parallelized version.

Figure 6 compares the running times of the executors (using ParMetis, PaToH, and block-partitioning) with the execution time for 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 since the manual implementation uses an approach similar to block partitioning. Figure 6b 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.

OLAM [45] 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 an 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. The neighborhood information of points is stored in indirection arrays. OLAM is written in Fortran 90. Unlike the other C benchmarks, which could be fully automatically transformed by our compiler, the generation of the inspector/executor code for OLAM required some manual steps in going from the sequential application to the code generated by the compiler.

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 7 shows data for 30000 iterations. The time for inspection, even with a sequential hypergraph partitioner, is several orders of magnitude

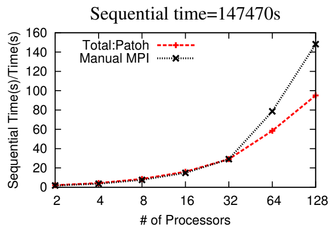


Fig. 7. OLAM atmospheric model.

lower than the executor time. Therefore, the PaToH hypergraph partitioner was used for generating the partitions.

Figure 7 shows that the performance of the code using the I/E framework (including inspector time) is on par with the manual MPI implementation. While the former scales linearly to 128 processes, the latter achieves super-linear scaling due to domain specific optimizations by experts.

VII. Related Work

Saltz and co-workers [10], [11], [38], [39] proposed the inspector-executor (I/E) approach for distributed-memory code generation for scientific applications with irregular access patterns. The PARTI/CHAOS libraries [6], [31] facilitated manual development of parallel message-passing code. Compiler support for optimizing communications within the executor was also explored [1], [44]. An approach to automatic compiler transformation for generation of I/E code via slicing analysis was developed [11], 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[j]` in Listing 1). Most of these approaches could handle only one level of indirection. The approach of Das et al. [11] could handle multiple levels of indirection, but their techniques are inapplicable to codes such as 183.equake and P3-RTE. Lain [20], [21] 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 [13], [17], [26]. Strout et al. [41], [42] 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 [22] 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 [5] presented techniques for translating OpenMP programs with irregular accesses into code for distributed-memory machines, by focusing on exploiting overlap of computation and communication. But 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 [8], [14], [15], [19], [24], [32]. For loops not amenable to static analysis, speculative techniques have been used for run-time parallelization [23], [33], [37], [46]. Zhuang et al. [47] 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.

VIII. Conclusion

Irregular and sparse computations in distributed memory environments are of significant importance in scientific and engineering computing. When control-flow and array-access patterns depend on run-time data, static compiler techniques need to be combined with run-time inspection. We propose an inspector/executor parallelization approach for a class of such applications. The inspector gathers run-time information about control flow and array accesses, partitions the computation, and remaps the data and control structures. Several optimizations are used to exploit contiguity of array accesses. Experimental results show that the generated code comes close to the performance achieved by manual parallelization by domain experts. Future work would focus on optimizing the communication layer to achieve better scalability of the generated code.

IX. 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

[1] G. Agrawal, J. Saltz, and R. Das, "Interprocedural partial redundancy elimination and its application to distributed memory compilation," in *PLDI*, 1995.

[2] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc Web page," 2012, http://www.mcs.anl.gov/petsc.

[3] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and J. Xu, "Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers," *Computer Methods in Applied Mechanics and Engineering*, vol. 152, pp. 85–102, 1998.

[4] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA code generation for affine programs," in *CC*, 2010.

[5] A. Basumallik and R. Eigenmann, "Optimizing irregular shared-memory applications for distributed-memory systems," in *PPoPP*, 2006.

[6] H. Berryman, J. Saltz, and J. Scroggs, "Execution time support for adaptive scientific algorithms on distributed memory machines," *Concurrency: Practice and Experience*, vol. 3, pp. 159–178, 1991.

[7] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan, "A model for fusion and code motion in an automatic parallelizing compiler," in *PACT*, 2010.

[8] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *PLDI*, 2008.

[9] U. V. Catalyurek and C. Aykanat, *PaToH: Partitioning Tool for Hypergraphs*, 2009.

[10] R. Das, P. Havlak, J. Saltz, and K. Kennedy, "Index array flattening through program transformation," in *SC*, 1995.

[11] R. Das, J. Saltz, and R. von Hanxleden, "Slicing analysis and indirect access to distributed arrays," Rice University, Tech. Rep. CRPC-TR93319-S, 1993.

[12] T. A. Davis, "University of Florida sparse matrix collection," *NA Digest*, 1994.

[13] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *PLDI*, 1999.

[14] P. Feautrier, "Some efficient solutions to the affine scheduling problem - Part I: One-dimensional time," *IJPP*, vol. 21, pp. 313–347, 1992.

[15] M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. FMI, University of Passau, 2004.

[16] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *LCPC*, 2010.

[17] H. Han and C.-W. Tseng, "Exploiting locality for irregular scientific codes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, pp. 606–618, 2006.

[18] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[19] F. Irigoin and R. Triolet, "Supernode partitioning," in *POPL*, 1988.

[20] A. Lain, "Compiler and run-time support for irregular computations," Ph.D. dissertation, University of Illinois at Urbana-Champagne, 1996.

[21] A. Lain and P. Banerjee, "Exploiting spatial regularity in irregular iterative applications," in *IPPS*, 1995.

[22] A. LaMielle and M. Strout, "Enabling code gen. with sparse polyhedral framework," Colorado State University, Tech. Rep. CS-10-102, 2010.

[23] S.-T. Leung and J. Zahorjan, "Improving the performance of runtime parallelization," in *PPoPP*, 1993.

[24] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *POPL*, 1997.

[25] "Mantevo project," `https://software.sandia.gov/mantevo`.

[26] N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," in *PACT*, 1999.

[27] M. F. Modest, *Radiative Heat Transfer*. Academic Press, 2003.

[28] J. Neiplocha, V. Tipparaju, M. Krishnan, and D. K. Panda, "High performance remote memory access communication: The ARMCI approach," *Int. J. High Performance Computing Applications*, vol. 20, pp. 233–253, 2006.

[29] "Par4all," `www.par4all.org`.

[30] "Pluto - an automatic parallelizer and locality optimizer for multicores," `pluto-compiler.sourceforge.net`.

[31] R. Ponnusamy, J. H. Saltz, and A. N. Choudhary, "Runtime compilation techniques for data partitioning and communication schedule reuse," in *SC*, 1993.

[32] J. Ramanujam and P. Sadayappan, "Tiling multidimensional iteration spaces for multicomputers," *JPDC*, vol. 16, no. 2, pp. 108–230, 1992.

[33] L. Rauchwerger and D. Padua, "The LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization," in *PLDI*, 1995.

[34] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Code generation for parallel execution of a class of irregular loops on distributed memory systems," The Ohio State University, Tech. Rep. OSU-CISRC-5/12-TR10, 2012.

[35] M. Ravishankar, S. Mazumder, and A. Kumar, "Finite-volume formulation and solution of the p3 equations of radiative transfer on unstructured meshes," *Journal of Heat Transfer*, vol. 132, p. 0123402, 2010.

[36] "Rose compiler infrastructure," `www.rosecompiler.org`.

[37] S. Rus, M. Pennings, and L. Rauchwerger, "Sensitivity analysis for automatic parallelization on multi-cores," in *ICS*, 2002.

[38] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, "Run-time scheduling and execution of loops on message passing machines," *J. Parallel Distrib. Comput.*, vol. 8, pp. 303–312, 1990.

[39] J. H. Saltz, H. Berryman, and J. Wu, "Multiprocessors and run-time compilation," *Concurrency: Practice and Experience*, 1991.

[40] K. Schloegel, G. Karypis, and V. Kumar, "Parallel static and dynamic multi-constraint graph partitioning," *Concurrency and Computation: Practice and Experience*, vol. 14, pp. 219–240, 2002.

[41] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time composition of run-time data and iteration reodrderings," in *PLDI*, 2003.

[42] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck, "Combining performance aspects of irregular Gauss-Seidel via sparse tiling," in *LCPC*, 2002.

[43] M. M. Strout and P. D. Hovland, "Metrics and models for reordering transformations," in *MSP*, 2004.

[44] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz, "Compiler analysis for irregular problems in Fortran D," *LCPC*, 1993.

[45] R. L. Walko and R. Avissar, "The Ocean-Land-Atmosphere Model (OLAM). Part I: Shallow-Water Tests," *Monthly Weather Review*, vol. 136, pp. 4033–4044, 2008.

[46] H. Yu and L. Rauchwerger, "Techniques for reducing the overhead of run-time parallelization," in *CC*, 2000.

[47] X. Zhuang, A. E. Eichenberger, Y. Luo, K. O'Brien, and K. O'Brien, "Exploiting parallelism with dependence-aware scheduling," in *PACT*, 2009.