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

MAHESH RAVISHANKAR and JOHN EISENLOHR, Ohio State University
LOUIS-NOËL POUCHET, University of California, Los Angeles
J. RAMANUJAM, Louisiana State University
ATANAS ROUNTEV and P. SADAYAPPAN, Ohio State University

Many scientific applications spend significant time within loops that are parallel, except for dependences 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 in such cases.

This article proposes an approach for automatic parallelization for distributed memory environments, using both static and runtime analysis. We formalize the computations that are targeted by this approach and develop algorithms to detect such computations. We also describe algorithms to generate a parallel inspector that performs a runtime analysis of control-flow and array-access patterns, and a parallel executor to take advantage of this information. The effectiveness of the approach is demonstrated on several benchmarks that were automatically transformed using a prototype compiler. For these, the inspector overheads and performance of the executor code were measured. The benefit on real-world applications was also demonstrated through similar manual transformations of an atmospheric modeling software.

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, parallelization, irregular applications

ACM Reference Format:
Ravishankar, M., Eisenlohr, J., Pouchet, L.-N., Rumanujam, J., Rountev, A., and Sadayappan, P. 2014. Automatic parallelization of a class of irregular loops for distributed memory systems. *ACM Trans. Parallel Comput.* 1, 1, Article 7 (September 2014), 37 pages.
DOI: http://dx.doi.org/10.1145/2660251

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., [Baskaran et al. 2010; Bondhugula et al. 2008, 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 data structures where the control-flow and array-access patterns are data-dependent. A common methodology to handle

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.
Authors' addresses: M. Ravishankar (corresponding author), J. Eisenlohr, A. Rountev, and P. Sadayappan, Computer Science and Engineering, Ohio State University; email: ravishan@cse.ohio-state.edu; L.-N. 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: http://dx.doi.org/10.1145/2660251

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 an approach for automatic detection and distributed memory code generation for an extended class of affine computations that allows some forms of indirect array accesses. The class of applications targeted by the proposed source-to-source transformation scheme is prevalent in many scientific/engineering domains. The paradigm used for the parallelization is often called the inspector/executor (I/E) [Saltz et al. 1990] approach. The I/E approach uses the following.

(1) An *inspector* examines some data that is unavailable at compile time but is available at the very beginning of execution (e.g., the specific interconnectivity of the unstructured grid representing an airplane wing's discretized representation). This analysis is used to construct distributed data structures and computation partitions.
(2) An *executor* 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 colleagues [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 (which we detail in Section 9) have been directed at the generation of parallel code using this approach. In this article, using the I/E paradigm, we develop an automatic parallelization and code generation infrastructure for a class of programs encompassing affine and certain non-affine loops to target a distributed memory message-passing parallel programming model. This article makes the following contributions.

— It presents a description of a class of extended affine computations that allow data-dependent control flow and irregular data access patterns, 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, which analyzes the computation at runtime, 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 implementations. We evaluate the overhead of the runtime analysis. Experimental results are also presented to demonstrate the importance of maintaining contiguity of accesses from the original computation.

The rest of the article is organized as follows. Section 2 describes the class of extended affine computations that we address, along with a high-level overview of the approach to code transformation. Section 3 provides details of the approach for generating computation partitions using a hypergraph that models the affinity of loop iterations to data elements accessed. Section 4 describes the first step in the automatic parallelization process—the detection of partitionable loops in the input sequential program. The algorithms for generation of inspector and executor code for the partitionable loops are provided in Section 5, with Section 6 describing the overall code generation process. Section 7 explains how the need for inspector code can be optimized away for portions of the input code that are strictly affine. Experimental results using

```
1  while( !converged ){
2    //...Other computation not shown...
3    for( k = 0 ; k < n ; k++ )
4      x[k] = ...;
5    //...Other computation not shown...
6    for( i = 0 ; i < n ; i++ )
7      for( j = ia[i] ; j < ia[i+1] ; j++ ){
8        xindex = col[j];
9        y[i] += A[j]*x[xindex];
10     }
11   //...Other computation not shown...
12 }
```

Listing 1. Sequential conjugate gradient computation.

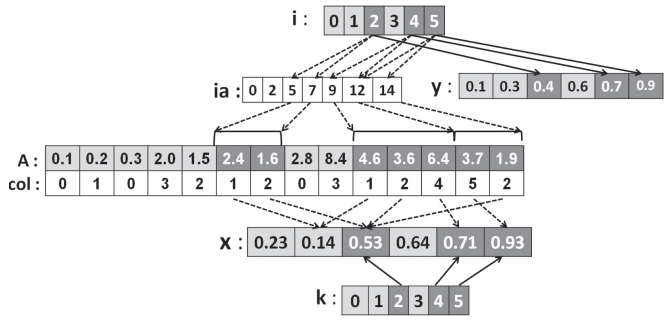


Fig. 1. Control flow and data-access patterns of iteration 2, 4, and 5, of loops i and k mapped to process 0.

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 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 nonzero elements in row `i`. For `i` in `[0,n-1]`, these nonzero 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.

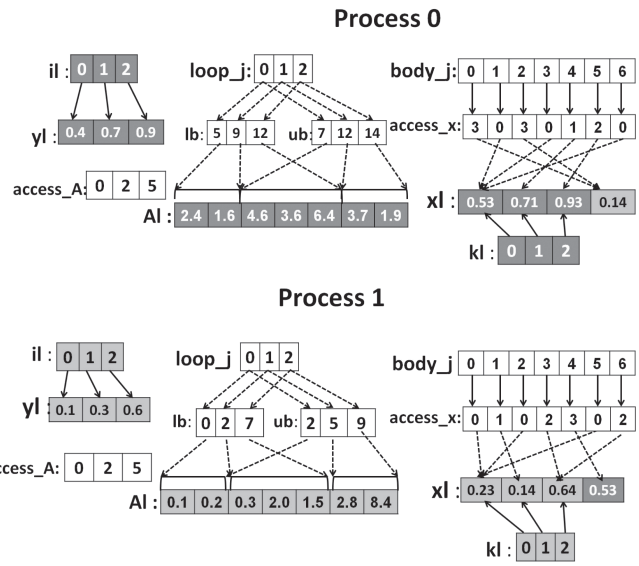


Fig. 2. Transformed iteration and data view.

```
1  /* Inspector code to generate o_a_j, i_x_j, lb,*/
2  /* ub, xl, yl, Al and to compute nl; not shown */
3  while( !converged ){
4    //...Other computation not shown...
5    //...Update values of ghosts for arrays read in loop...
6    body_k = 0;
7    for( kl = 0 ; kl < nl ; kl++ ){
8      xl[kl] = ...;
9      body_k++;
10   }
11   //...Update values of owner of elements of xl...
12   //...Other computation not shown...
13   //...Update values of ghosts for Al and xl...
14   body_i = 0; loop_j = 0 ; body_j = 0;
15   for( il = 0 ; il < nl ; il++ ) {
16     offset_A = access_A[loop_j] - lb[loop_j];
17     for( j = lb[loop_j] ; j < ub[loop_j] ; j++ ){
18       yl[il] += Al[j+offset_A]*xl[access_x[body_j]];
19       body_j++;
20     }
21     loop_j++; body_i++;
22   }
23   //...Update values of owners of yl...
24   //...Other computation not shown...
25 }
```

Listing 2. Parallel conjugate gradient computation.

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 in the following, 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 in Sections 2.2 through 2.8. Listing 2 shows the code to execute these partitions in parallel 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 program parameters (variables whose values do not change during an execution of the affine region). For such codes, the control-flow and data-access patterns can be fully characterized at compile time.

Consider the computation within loop `i` in Listing 1. The bounds of the inner loop `j` depend on `ia`, and accesses to `x` depend on `col`. During analysis of loop `i`, affine techniques have to be conservative and overapproximate the data dependences and control flow [Benabderrahmane et al. 2010]. We target a generalized class of computations, in which loop bounds, 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 read-only indirection arrays. The control-flow and data-access patterns of such computations can be determined at run-time by an inspector, before the actual computation is performed.

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

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

2.3. Bounds of Inner Loops

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

Elements of an array might receive updates from iterations of the partitionable loops executed on different processes. For example, consider a modification to Listing 1, where the statement at Line 9 was replaced with `x[col[j]] += A[j] * y[i];`. This modified computation is still within the scope of targeted computations. The loop `i` is still a partitionable loop, with elements of the array `x` updated by multiple iterations of loop `i`. In the partitioned execution shown in Figure 2, `x[2]` and `x[1]` again are owned by a process and have ghost locations on the other. In the executor, these ghost locations are initialized to the identity element of the update operator (0 for "+=", 1 for "*=") before the loop execution. After the loop, their values are communicated to the owner where the values from all ghost locations are combined. For example, in the preceding modified computation, if process 0 owns `x[2]`, the ghost location for that element on process 1 would be initialized to 0 before the loop execution in the distributed memory code. After the loop, the value at the ghost location on process 1 is communicated to process 0, where it is combined with the value at the owned location. 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, surrounding 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, `access_x` of size 7 on process 0 is used to simulate the accesses to `x` due to expression `col[j]`. The values stored in this array are modified to point to corresponding locations in `xl`.

2.6. Optimizing Accesses from Inner Loops

The approach described in Section 2.5 would result in another array of the same size as `access_x` to recreate 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 layout of 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 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. For example, in Figure 1, for iterations mapped to process 0, the different invocations of the inner loop accesses the elements `A[5:6]`, `A[9:11]` and `A[14:15]`. These correspond to elements `Al[0:1]`, `Al[2:4]` and `Al[5:6]` in the partitioned view shown in Figure 2. The array `access_A` on process 0 stores the index of the first element of `Al` accessed within each sequence *i.e.* 0, 2, and 5. The other elements of the sequence are accessed within the loop by adding `(j - lb[loop_j])` to this value, where `loop_j` counts the number of invocations of loop `j` executed on a process. In Listing 2, this is achieved by setting the variable `offset_A` to the value `(access_A[loop_j] - lb[loop_j])` before the loop. The expression `Al[offset_A + j]` would be consistent with `A[j]` from the original computation.

Using such an approach reduces the size of the array needed to recreate the data access pattern for unit-stride index expressions to the number of invocations of the inner loop (3 in Figure 2), instead of the total number of iterations of the loop executed on a process.

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 of 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, 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. For example, in Figure 1 iterations 0, 1, and 3 of the `i`-loop access the elements `A[0:1]`, `A[2:4]`, and `A[7:8]` using the unit-stride expression, `A[j]` . In the partitioned view shown in Figure 2, these iterations are mapped to process 1. To maintain the contiguity of accesses, `A[0:1]`, `A[2:4]`, and `A[7:8]` have to be piecewise-contiguous in the local array `Al` on that process. Suppose that this loop also contained the expression `A[i]`. Since the iterations of the `i`-loop would be renumbered to be contiguous, to maintain contiguity of accesses from this loop, elements corresponding to `A[0]`, `A[1]`, and `A[3]` would have to be contiguous in memory as well. This conflicts with the layout needed to maintain contiguous access from loop `j`. In such cases, the accesses from the partitionable loop are not optimized, since using a separate array to recreate the access from the partitionable loop (as described in Section 2.5) would have a smaller footprint.

A potential conflict could also arise if multiple partitionable loops access an array with unit-stride. To optimize all accesses it is necessary to partition all participating loops in a similar way 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 `access_x` and `access_A`. In addition, communication calls are inserted to satisfy the producer-consumer relationship due to array `x`.

3. PARTITIONING THE COMPUTATION

In order to achieve efficient distributed-memory execution, the main factors to be considered are the following.

— The computation must be partitioned such that each partitionable loop is executed in a load-balanced manner.
— The amount of communication between processes has to be minimized.

Due to the use of ghost locations, communication is needed either before executing the iterations of the partitionable loop to initialize the ghosts within arrays read in the loop body, or after the loop execution to combine the values from all the ghost locations for data elements of arrays updated within the loop. Therefore, the amount of communication required is directly related to the number of ghost locations needed for the distributed memory execution. Since a data element need not have any ghosts if all the iterations that access this element are mapped to the same process, an accurate representation of the iteration-to-data affinity would allow the inspector to partition the computation while minimizing the number of ghosts. Similar to Strout and Hovland [2004] and Catalyurek and Aykanat [2009], we use a hypergraph to represent this affinity.

A hypergraph $\mathcal{H}$ is defined by a set of vertices, $\mathcal{V}$ and a set of edges, $\mathcal{N}$ and is a generalization of a graph. An edge of a hypergraph, refered to as a *hyperedge* or a *net*, connects two or more vertices. The connected vertices are refered to as the *pins* of the net. The iteration-to-data affinity is captured by having vertices for the iterations of the partitionable loops, and nets for individual array elements. The vertices that represent the iterations that access a data element form the pins of the corresponding net. While Strout and Hovland [2004] used the hypergraph abstraction to reorder iterations to achieve better temporal locality, here we use a disjoint partitioning of the vertices of the hypergraph to compute the group of iterations of the partitionable loops to be executed on a process. The partitioning is subjected to constraints that enforce the two listed requirements. We use the PaToH hypergraph partitioner [Catalyurek and Aykanat 2009], for this constrained partitioning. Following is a description of these constraints adapted from their work.

3.1. Minimizing Communication

Since the communication cost depends on the number of bytes to be exchanged between the processes, each net $n \in \mathcal{N}$ is associated with a weight c_j, whose value is the same as the size in bytes of the data-type represented by the net (4 for integer and float types, 8 for doubles). For each partition P, the set of nets that have pins in them can be divided into two disjoint subsets, a set of *internal* nets, I that have pins only in P, and a set of *external* nets, E that have pins in other partitions also. Each external net, $j \in E$ represents a data element that is accessed by more than one process. One of the partitions is chosen as the owner of a corresponding data element, and the other

partitions have corresponding ghost locations. Therefore, the total number of ghost locations needed depends not only on the number of external nets, but also on the number of partitions λ_j in which each external net has a pin, resulting in $(\lambda_j - 1)$ ghost locations. The volume of communication can therefore be minimized by minimizing the metric, Π for each partition, defined as

$$\Pi = \sum_{j \in E} c_j(\lambda_j - 1). \tag{1}$$

3.2. Achieving Load Balancing

To achieve load balance across processes, the iterations of every partitionable loop have to be evenly distributed amongst them. This is done by using a *multi-constrained partitioning scheme*. 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 the k-th partitionable loop has the k-th element as 1, with all other elements being 0. The number of iterations of each partitionable loop mapped to a partition P can be computed by the vector sum of the weights of the vertices belonging to the partition. This is known as the weight of the partition, W_P and is computed as follows.

$$W_P = \sum_{i \in P} \vec{w}_i. \tag{2}$$

The number of iterations for each partitionable loop that would be executed on a process in a perfectly load-balanced execution is given by W_{avg}, computed as

$$W_{avg} = W_{\mathcal{V}}/P, \tag{3}$$

where, $W_{\mathcal{V}}$ is the sum of weights for all vertices in $\mathcal{V}$ and P is the number of processes used. Restricting the weight of every partition to be within a tolerance ϵ of W_{avg}, ensures that iterations of the partitionable loop are distributed equally amongst the partitions. Therefore the constraints on the partitioning can be described by the vector equation

$$W_P \leq W_{avg}(1 + \epsilon). \tag{4}$$

Partitioning the hypergraph under these constraints while minimizing Π for each partition achieves a load-balanced partitioning of the original computation.

This model assumes that all iterations of a partitionable loop have similar execution times. It is possible to relax this assumption and have the inspector use heuristics (such as total number of inner-loop iterations executed) to model the execution time of an iteration. Using this value for the nonzero element of the weight vector would better capture the requirements of load-balancing. For the tested benchmarks, we did not find that this assumption significantly affected the performance of the generated distributed memory code.

4. IDENTIFYING PARTITIONABLE LOOPS

This section describes the structure and automatic detection of partitionable loops in a given computation. The analysis assumes that the input code is consistent with the grammar described in Figure 3. Computations can consist of loops, conditional statements, and assignment statements. Each loop must have a unique iterator, which is referenced only in the loop body and is modified only by the loop increment expression (with unit increment). An assignment could use the standard assignment operator, or an update operator of the form `op=`, where `op` is a commutative and associative binary operator. Loop bounds, conditional expressions, right-hand sides of assignments,

```
⟨Start⟩ ::= ⟨ElementList⟩
⟨ElementList⟩ ::= ⟨Element⟩ ; ⟨ElementList⟩ | ⟨Element⟩
⟨Element⟩ ::= ⟨Assignment⟩ | ⟨Loop⟩ | ⟨If⟩
⟨Loop⟩ ::= for (⟨Iterator⟩=⟨Expr⟩ ; ⟨Iterator⟩<⟨Expr⟩ ; ⟨Iterator⟩++) {⟨ElementList⟩}
⟨If⟩ ::= if (⟨Expr⟩) {⟨ElementList⟩} else {⟨ElementList⟩}
⟨Assignment⟩ ::= ⟨LHSExpr⟩⟨AssignOp⟩⟨Expr⟩
⟨Expr⟩ ::= Side-effect-free expression of ⟨BasicExpr⟩
⟨LHSExpr⟩ ::= ⟨Scalar⟩ | ⟨Array⟩[⟨Expr⟩]
⟨BasicExpr⟩ ::= ⟨Scalar⟩ | ⟨Iterator⟩ | ⟨Array⟩[⟨Expr⟩]
⟨AssignOp⟩ ::= = | += | *= | ...
```

Fig. 3. Grammar for valid inputs to the static analysis for partitionable loop detection.

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. It is also assumed that no two arrays have overlapped regions of memory in the original computation. Asserting this at compile time is outside the scope of the present work.

Algorithm 1 takes as input an Abstract Syntax Tree (AST) corresponding to a ⟨Loop⟩ from this grammar, and determines whether the loop is partitionable. Section 4.1 discusses the analysis to detect indirection arrays and scalars; this analysis corresponds to lines 2–29 in the algorithm. Section 4.2, corresponding to line 43 in the algorithm, checks the parallelism of the loop. The overall approach for identifying a sequence of maximal partitionable loops by applying Algorithm 1 several times is described in Section 4.3.

4.1. Indirection Arrays and Indirection Scalars

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

To identify such variables, all loop bounds, conditionals, and array index expressions are analyzed. All array variables appearing in them (retrieved by the function *GetAllArrays*) are added to the set of indirection array variables. All scalars appearing in such expressions (retrieved by the function *GetAllScalars* 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 that assigns to this scalar is analyzed. All arrays appearing on this right-hand side are added to the set of indirection arrays. Since index expressions appearing on the right-hand side have been processed earlier, they are ignored at this stage (by passing the flag *IGNORE_INDEX_EXPR* to *GetAllArrayVariables*). All scalars that appear on the right-hand side that are not already in the set of indirection scalars, are added to that set and are also added to the worklist. Again, all scalars appearing within index expressions are ignored. Once all statements that assign to the scalar have been analyzed, the scalar 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,

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 = ∅ ; I_A = ∅ ; I = A.Iterator ; is_partitionable = true ;
3    foreach s ∈ GetLoopStmts(A.Body) do
4      I = I ∪ s.Iterator ;
5    foreach s ∈ GetLoopStmts(A.Body) do
6      l = s.LowerBound ; u = s.UpperBound ;
7      I_A = I_A ∪ GetAllArrayVariables(l) ∪ GetAllArrayVariables(u) ;
8      I_S = I_S ∪ ( GetAllScalars(l) ∪ GetAllScalars(u) − I ) ;
9    foreach s ∈ GetIfStmts(A.Body) do
10     c = s.Cond ;
11     I_A = I_A ∪ GetAllArrayVariables(c) ;
12     I_S = I_S ∪ ( GetAllScalars(c) − I ) ;
13   foreach s ∈ GetAssignStmts(A.Body) do
14     foreach e ∈ GetArrayRefExprs(s) do
15       c = e.IndexExpr ;
16       I_A = I_A ∪ GetAllArrayVariables(c) ;
17       I_S = I_S ∪ ( GetAllScalars(c) − I );
18   worklist = I_S ;
19   while ¬IsEmpty(worklist) do
20     v = RemoveElement(worklist) ;
21     foreach s ∈ GetAssignStmts(A.Body) do
22       if IsScalar(s.LHS) ∧ s.LHS = v then
23         c = s.RHS ;
24         I_A = I_A ∪ GetAllArrayVariables(c) ;
25         S = GetAllScalars(c) − I ;
26         foreach u ∈ S do
27           if u ∉ I_S then
28             I_S = I_S ∪ {u} ; AddElement(worklist,u) ;
29   foreach s ∈ GetAssignStmts(A.Body) do
30     if ¬IsScalar(s.LHS) ∧ s.LHS.Array ∈ I_A then
31       is_partitionable = false ;
32       break ;
33     if ¬IsScalar(s.LHS) ∨ s.LHS ∉ I_S then
34       S = GetAllScalars(s.RHS,IGNORE_INDEX_EXPR);
35       if S ∩ I_S ≠ φ then
36         is_partitionable = false ;
37         break ;
38       S = GetAllArrayVariables(s.RHS,IGNORE_INDEX_EXPR) ;
39       if S ∩ I_A ≠ φ then
40         is_partitionable = false ;
41         break ;
42   if is_partitionable then
43     is_partitionable = CheckForParallelism(A.Body,I_S,I_A,I) ;
44     return [is_partitionable,I_A,I_S,I] ;
45   else
46     return [false,∅,∅,∅] ;
```

⟨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. Grammar for candidate loops (does not enforce parallelism).

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 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 directly or indirectly affect values written to data arrays/scalars. Section 4.4 outlines how to handle cases where this property does not hold.

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

This 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 that 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 article.

4.2. Parallelism of the Target Loop

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

— all indirection arrays are read-only within the loop, and
— 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 classified as ⟨DArray⟩ can appear either only on the right-hand side of ⟨DAssignment⟩s or only on the left-hand side, but not both, and
— 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, then the inter-iteration dependence caused by updates to this scalar can be handled by the scheme described in Section 5.6. 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 variables using associative and commutative update operators. Section 5.6 provides a description of how these dependences are handled in a parallel execution of the loop.

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

4.3. Finding Sequences of Maximal Partitionable Loops

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

Suppose that the input program AST is an ⟨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. 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 cannot 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 one 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 nonpartitionable 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.

4.4. Possible Generalizations

One of the restrictions described in Section 4.1 was that an assignment statement that 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 the article would still be applicable.

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

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 in the following text.

5.1. Phase I: Hypergraph Generation

5.1.1. Runtime 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

```
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      shadow_xindex = false;
8      vi = AddVertex(id_i_loop);
9      if(is_known(id_ia_array,i) && is_known(id_ia_array,i+1))
10       for( j = get_elem(id_ia_array,i) ; j < get_elem(id_ia_array,i+1) ; j++ ) {
11         if( is_known(id_col_array,j) ){
12           xindex = get_elem(id_col_array,j);
13           shadow_xindex = true;
14         }
15         else
16           shadow_xindex = false;
17         AddPin(id_y_array,i,vi,true);
18         AddPin(id_A_array,j,vi,true);
19         if( shadow_xindex )
20           AddPin(id_x_array,xindex,vi,false);
21       }
22     else
23       shadow_xindex = false;
24   }
25 }while( DoneGraphGen() );
```

Listing 3. Phase I of the inspector.

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 has been analyzed, all outstanding requests are serviced. On reanalyzing 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 associated shadow scalar is set to false. Statements that use the values of these indirection scalars 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 the statement at line 20 of Listing 3, it is guarded to check 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 3, where P is the number of processes. Each process is assigned a unique partition representing the iterations to be executed on it. The iterations are renumbered such that they form a contiguous set on each process, while maintaining the relative ordering of the iterations mapped to that process.

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 : ⟨IScalar⟩s defined in *ss*

```
1  begin
2    A_H = ∅ ; I = ∅ ;
3    foreach s ∈ 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 = ∅ ;
12       foreach d ∈ GetDataArrayRefExprs(s) do
13         b = ConvertArraysToFunctions(d.IndexExpr) ;
14         s_H = NewIfStmt() ; s_H.Cond = GenGuards(d.IndexExpr) ;
15         t = GenAddPinFn(H,d.Array,b,IsUnitStride(d.IndexExpr)) ;
16         if s_H.Cond ≠ ∅ 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(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(H,s.Then) ;
32       [s_H.Else,I_2] = CodeGenHyperGraph(H,s.Else) ;
33       l_H = NewIfStmt() ; l_H.Cond = GenGuards(s.Cond) ;
34       I_L = I_1 ∪ I_2 ; l_H.Then = s_H ;
35     foreach v ∈ I_L do
36       l_H.Else.Append(SetShadowScalar(v,false)) ;
37     A_H.Append(l_H) ; I = I ∪ I_L ;
38   return [A_H,I] ;
```

5.1.2. Code Generation at Compile Time. The inspector code that achieves the described functionality (e.g., the code in Listing 3) is automatically generated by the compiler. The compiler algorithm for generating 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

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.ShadowScalar,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 = ∅ ;
10     foreach d ∈ e.Children do
11       c = NewAndCond(c,GenGuards(d)) ;
12   return c ;
```

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 condition 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, 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 ⟨BasicIExpr⟩, 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 are 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 11–16 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 that indicates whether the current expression increments by one for two consecutive values of a surrounding loop iterator, *i.e.*, the expression is stride-1 or unit-stride. 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 17–20 of Listing 3 are the statements generated for the ⟨DStatement⟩ at line 9 of Listing 1.

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

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

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

5.2. Initializing Local Data Arrays

Initially all the data arrays were assumed to be block-partitioned. After partitioning the hypergraph, Phase I of the inspector repartitions the data such that all elements touched by the iterations mapped to a process are in its local memory. 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 it. All other processes have a ghost location for that element. The local copy of the array consists of elements that a process owns and 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 accessed array. 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. Runtime 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

```
1  do{
2    body_i = 0 ; loop_j = 0 ; body_j = 0; body_k = 0;
3    for( k = 0 ; k < n ; k++ )
4      if( home(id_k_loop,k) == myid )
5        body_k++;
6    for( i = 0 ; i < n ; i++ )
7      if( home(id_i_loop,i) == myid ){
8        shadow_xindex = false;
9        if( is_known(id_ia_array,i) && is_known(id_ia_array,i+1)){
10         for( j = get_elem(id_ia_array,i) ; j < get_elem(id_ia_array,i+1) ; j++ ){
11           if( is_known(id_col_array,j) ){
12             xindex = get_elem(id_col_array,j);
13             shadow_xindex = true;
14           }
15           else
16             shadow_xindex = false;
17           body_j++;
18         }
19         loop_j++;
20       }
21       else
22         shadow_xindex = false;
23       body_i++;
24     }
25   }while( DoneCounters() );
```

Listing 4. Phase II of the inspector.

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, its size is the number of times the `if` statement is executed.

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, and `if_*` counts 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 that of 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

```
1  body_i = 0 ; loop_j = 0 ; body_j = 0; body_k = 0;
2  for( k = 0 ; k < n ; k++ )
3    if( home(id_k_loop,k) == myid )
4      body_k++;
5  for( i = 0 ; i < n ; i++ )
6    if( home(id_i_loop,i) == myid ){
7      lb_j[loop_j] = get_elem(id_ia_array,i);
8      ub_j[loop_j] = get_elem(id_ia_array,i+1);
9      for( j = lb_j[loop_j] ; j < ub_j[loop_j] ; j++){
10       xindex = get_elem(id_col_array,j);
11       if( j == lb_j[loop_j] )
12         access_A[loop_j] = j;
13       access_x[body_j] = xindex;
14       body_j++;
15     }
16     loop_j++;
17     body_i++;
18   }
```

Listing 5. Phase III of the inspector.

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) the statements that add pins and vertices to the hypergraph are removed.

5.4. Phase III: Initializing Local Access Arrays

5.4.1. Runtime 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 partitionable loops from the original program.

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 to be used to store the value of the index expression is retrieved by function *GetAccessArray*. If the index expression used is a 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

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

```
1  begin
2    A_P = ∅ ;
3    foreach s ∈ ss in order of appearance do
4      if IsIStatement(s) then
5        b = ConvertArraysToFunctions(s.RHS) ;
6        l_P = NewAssignmentStmt(s.LHS,b) ;
7      else if IsDStatement(s) then
8        l_P = ∅ ;
9        foreach d ∈ GetDataArrayRefExprs(s) do
10         a = GetAccessArray(L,d.Array,d.IndexExpr) ;
11         c = GetCounterVariable(C,d.IndexExpr) ;
12         b = ConvertArraysToFunctions(d.IndexExpr) ;
13         e = NewArrayRefExpr(a,c) ; s_P = NewAssignmentStmt(e,b) ;
14         if IsUnitStride(d.IndexExpr) then
15           l = GetLoop(c) ; s_P = GenIfFirstIter(c,s_P) ;
16         l_P.Append(s_P) ;
17     else if IsLoop(s) then
18       l = s.LowerBound ; u = s.UpperBound ;
19       b_l = ConvertArraysToFunctions(l) ; b_u = ConvertArraysToFunctions(u) ;
20       a_l = GetLowerBoundArray(L,s) ; a_u = GetUpperBoundArray(L,s) ;
21       c_l = GetLoopCounterVariable(C,s) ;
22       e_l = NewArrayRefExpr(a_l,c) ; e_u = NewArrayRefExpr(a_u,c) ;
23       l_P = NewAssignmentStmt(e_l,b_l) ;
24       l_P.Append(NewAssignmentStmt(e_u,b_u)) ;
25       s_P = NewLoopStmt(s.Iterator,e_l,e_u) ;
26       s_P.Body = CodeToInitializeArrays(s.Body,L,C) ;
27       c_b = GetBodyCounterVariable(C,s) ; s_p.Body.Append(NewIncrementStmt(c_b)) ;
28       l_P.Append(s_P) ;
29       l_P.Append(NewIncrementStmt(c_l)) ;
30     else if IsIf(s) then
31       c_P = ConvertArraysToFunctions(s.Cond) ;
32       a = GetConditionalArray(L,s) ;
33       c = GetConditionalCounterVariable(C,s) ;
34       e = NewArrayRefExpr(a,c) ; l_P = NewAssignmentStmt(e,c_P) ;
35       s_P = NewIfStmt() ; s_P.Cond = e ;
36       s_P.Then = CodeToInitializeArrays(s.Then,L,C) ;
37       c_b = GetThenCounterVariable(C,s) ; s_P.Then.Append(NewIncrementStmt(c_b)) ;
38       s_P.Else = CodeToInitializeArrays(s.Else,L,C) ;
39       c_b = GetElseCounterVariable(C,s) ; s_P.Else.Append(NewIncrementStmt(c_b)) ;
40       l_P.Append(s_P) ;
41       l_P.Append(NewIncrementStmt(c)) ;
42     A_P.Append(l_P) ;
43   return A_P ;
```

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, `access_A` 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 assigned array locations. The body of the loop is generated by recursively processing 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.

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 modifying the original code using Algorithm 5. The AST of the original loop is traversed. ⟨IStatement⟩s are removed, since the control-flow and array-access patterns are explicitly represented through access arrays. For ⟨DStatement⟩s, all accesses to data arrays are replaced with accesses to the corresponding local data arrays. The index expressions used to access these arrays are also modified as necessary.

For expressions that are unit stride with respect to an inner loop, the index expression is the sum of the loop iterator and the value stored in an offset variable. This offset variable is initialized to the value stored in the access array associated with the index expression, subtracted with the lower bound of the loop (variable offset_A in Listing 2). 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, the bounds/conditionals are modified to read from arrays that were populated in Phase III of the inspector.

5.6. Communication Between Processes

Once the executor code has been generated for all partitionable loops, calls to perform the communication of ghost values between the processes are added to executor AST. The execution model assumes that before the execution of a partitionable loop, the value of a data element is known only at the owner process. Communication is required to initialize all ghost elements for data arrays read within the loop, before its execution. Calls to the runtime to perform this communication are inserted before the partitionable loop in the executor.

For arrays whose elements are updated within the partitionable loop (using commutative and associative operators), the ghost elements of these arrays have to be initialized to the identity element of the update operator, before the loop executions. At the end of the loop execution, the ghost elements contain partial contributions to the

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

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

data element. These partial contributions from all ghost locations are communicated to the owner process, where they are combined. The calls to the runtime to initialize the ghost elements are inserted before the partitionable loop in the executor. The calls to combine the partial contributions are inserted after the loop. As a result, after the communication is complete, the owner process has the correct value for the data element.

For computations where elements of an array are assigned to (instead of updated through an operator like += or *=), there might be output dependences for a particular data element. To resolve this dependence, the inspector tracks the last iteration of the partitionable loop that assigned to a data element. During the combination phase, the owner process selects the value received from the process that executed that iteration.

For arrays that are accessed at unit-stride from the partitionable loop in the original code (for example, `y[i]` in Listing 1), and when this unit-stride is maintained in the executor code as well (Section 2.7), it can be asserted statically that unique array elements are accessed by the different iterations of the partitionable loop, and that these iterations are executed on the process that owns the data element. Therefore no communication is inserted due to such accesses.

The communication pattern for both the read-updates and write-updates 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].

To handle updates to ⟨DScalar⟩s within a partitionable loop, the generated executor code initializes the value of the scalar to the identity of the update operator used on all processes except process 0. After the loop execution, an MPI_Allreduce is used to combine the values from all processes and get the resulting value to every process.

6. OVERALL CODE GENERATION APPROACH

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

The first step is to generate the code to initialize the hypergraph, using function *InitHyperGraph*. Following this, the Phase I code for all loops in the sequence is generated and appended to the inspector AST. A new loop is created, with the bounds from the original loop modified to be block-partitioned. A call to function `AddVertex` is generated by function *GenAddVertex* and added to the new loop body. The rest of the loop body is generated by traversing the AST of the original partitionable loop using Algorithm 2. The statement to initialize the shadow scalars associated with the indirection scalars used in the loop body is added to the generated loop AST by the function *InitShadowScalars*. 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 is 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.

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

InOut : $\mathcal{P}$: Sequence of partitionable loop ASTs
Output: $\mathcal{A}_I$: Code for the inspector

```
1  begin
2    H = InitHyperGraph() ; A_I = ∅ ; A_H = ∅ ;
3    foreach p ∈ 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(H,p.Iterator) ;
7      [L_1,S_1] = CodeGenHyperGraph(H,p.Body) ;
8      F_1 = InitShadowScalars(S_1) ; P_1.Body.Append(F_1) ;
9      P_1.Body.Append(L_1) ; A_H.Append(P_1) ;
10   A_H = NewDoWhile("DoneGraphGen()",A_H); A_I.Append(A_H) ;
11   A_I.Append(CodeToPartitionIterations(H)) ;
12   D = CodeToAllocateLocalData(H) ; A_I.Append(D) ;
13   C = DeclareCounterVariables(P) ; A_I.Append(C) ;
14   A_C = ∅ ;
15   foreach p ∈ P in order do
16     P_2 = NewLoopStmt(p.Iterator,p.LowerBound,p.UpperBound) ;
17     L_2 = NewIfStmt() ; L_2.Cond = GenerateIsHome() ;
18     [L_2.Then,S_2] = CodeToGetAccessArraySizes(p.Body,C) ;
19     F_2 = InitShadowScalars(S_2) ; P_2.Body.Append(F_2) ;
20     P_2.Body = L_2 ; A_C.Append(P_2) ;
21   A_C = NewDoWhile("DoneCounters()",A_C) ; A_I.Append(A_C) ;
22   I = CodeToAllocateAccessArrays(C) ; A_I.Append(I ) ;
23   A_P = ∅ ;
24   foreach p ∈ P in order do
25     P_3 = NewLoopStmt(p.Iterator,p.LowerBound,p.UpperBound) ;
26     L_3 = NewIfStmt(); L_3.Cond = GenerateIsHome() ;
27     L_3.Then = CodeToInitializeArrays(p.Body,I,C) ;
28     P_3.Body = L_3; A_P.Append(P_3) ;
29   A_P.Append(CodeToRenumberAccessArrays(C,D,I)) ;
30   A_I.Append(A_P) ;
31   foreach p ∈ P in order do
32     GenerateExecutorCode(D,I,C,p.Body) ;
33     InsertCommunicationCode(D,p) ;
34   return A_I ;
```

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].

This code-generation scheme may result in some statements that are dead-code, *e.g.* the assignment to `xindex` at Line 12 of Listing 4. Since the proposed algorithm is a source-to-source transformation, we rely upon the host compiler to identify and remove such statements.

7. OPTIMIZATIONS FOR AFFINE CODES

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

```
1  /* Original affine computation */
2  for( i = 0 ; i < n ; i++ )
3    for( j = 0 ; j < n ; j++ )
4      y[i] += A[i][j] * x[j];
5
6  /* Transformed parallel computation */
7  // ... Update values of ghost locations for Al and xl
8  for( i = 0 ; i < nl ; i++ )
9    for( j = 0 ; j < n ; j++ )
10     yl[i] += Al[i][j] * xl[j];
11 // ... Update values of owners for yl
```

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

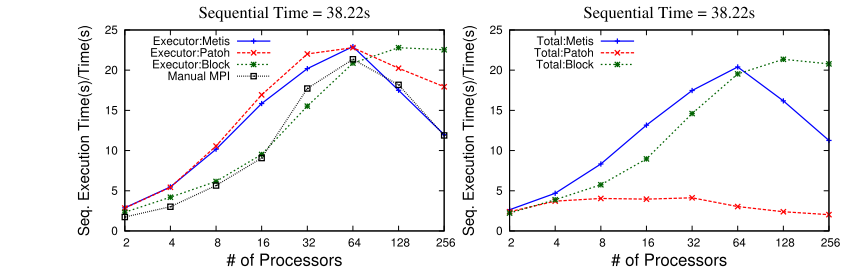
unnecessary overhead. For such loops, inspector code is unnecessary since control-flow and array-access patterns can be characterized statically. For example, if loop `i` in Listing 1 were of the form shown in Listing 6, where matrix `A` is dense and hence not stored in the CSR format, the control-flow and data accesses 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 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 nonlinear 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 the number of iterations executed on a process as a parameter. That is, if N is the number of iterations of a partitionable loop, for P processes $NP = N/P$ can be introduced as a new parameter to model the loop iterations of a process 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. Since all array access expressions are affine, the expression to access local arrays in the executor can be computed statically, eliminating the need to use arrays to store the value of this expression. Loop bounds of inner loops also use affine expressions in the executor instead of storing the loop bounds in arrays. Since the hypergraph partitioning is not used, and no arrays are needed to recreate the control-flow and data access patterns, the only role of the inspector would be to allocate local arrays on each process and populating them with appropriate values from the original array.

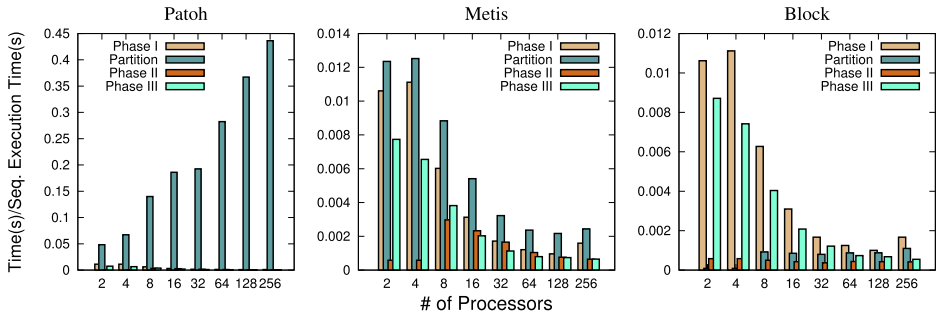
8. EVALUATION

For our experimental evaluation we used a cluster with Intel Xeon E5630 processors with four 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 Intel ICC 12.1 at the -O3 optimization level.

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



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



(c) Inspector overhead breakdown

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

partitioning can be completely avoided. The hypergraph still has to be built since the partitioning of data is done based on the iteration-to-data affinity it captures.

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.[1] Further, the performance of the generated code for each application was compared against manual MPI implementations either developed by the authors or by domain experts who developed the original application.

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 every iteration of that outer loop. All reported execution times are averaged over 10 runs. The speedup 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 5(a) 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

[1]www.rosecompiler.org

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

8.1.2. CG Kernel. The conjugate gradient (CG) method to solve a 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 nonzero elements, *tmt_sym* has more (726713) rows and fewer (5080961) non-zero elements. The structure of the latter is such that the nonzero elements fall along diagonals of the matrix.

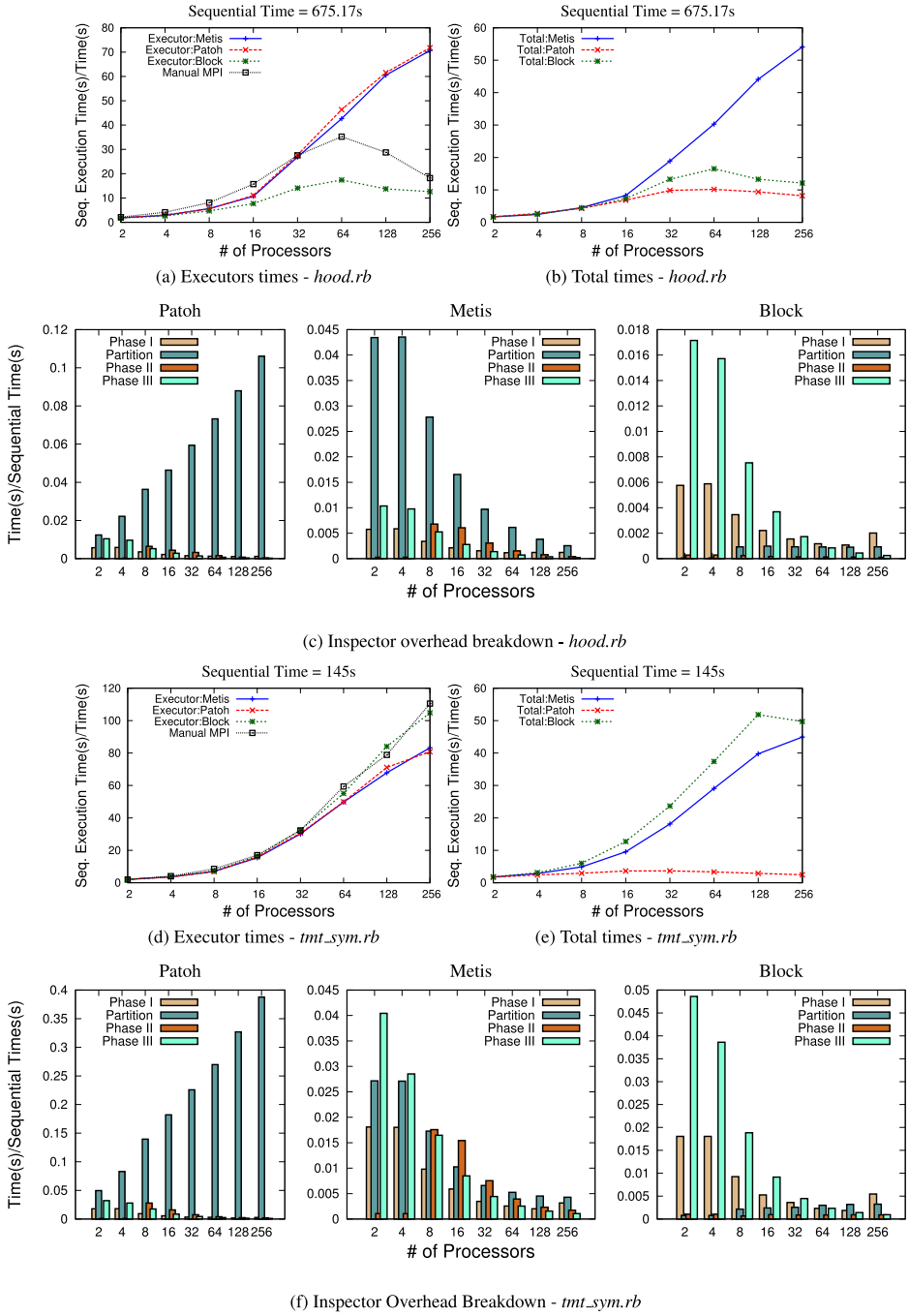
Figures 6(a) and 6(d) show that the executor code achieves good scaling overall with better than ideal scaling between 8 and 32 processes, due to the partitions becoming small enough to fit in caches. Using block-partitioning, gives good performance with *tmt_sym* but not for *hood*. Due to the structure of the latter, block partitioning results in a larger number of ghost 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 6(b) and 6(e). 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 nonzero 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 6(a) shows that 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 6(d)) upto 128 processes.

The breakdown of the inspector overheads for each of the matrices with different partitioning schemes are shown in Figures 6(c) and 6(f). 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, fewer ghosts results in less computation. At the same time, the block partitioning scheme has less overhead for Phase II of the inspector since the iterations of the partitionable loop analyzed 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. The Finite-Volume Method is used for discretizing the RTE with the 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 7(a) compares the executor times for the three schemes with a manual MPI implementation, which uses a partitioning of the underlying physical grid to partition the computation. Since the partitioning scheme used by the auto-generated code,



(a) Executors times - *hood.rb*

(b) Total times - *hood.rb*

(c) Inspector overhead breakdown - *hood.rb*

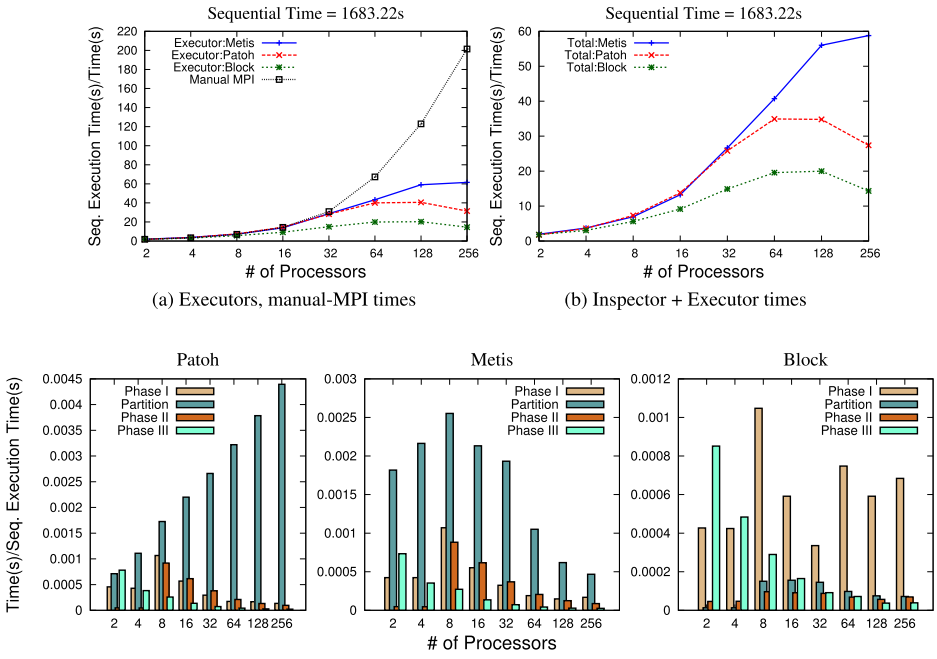
(d) Executor times - *tmt_sym.rb*

(e) Total times - *tmt_sym.rb*

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

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

groups together iterations that touch the same data elements, the generated partitions are similar to those used by the manual MPI implementation. As a result, the executor code while using PaToH or ParMetis achieves performance comparable to the manual



(a) Executors, manual-MPI times

(b) Inspector + Executor times

(c) Inspector overhead breakdown

Fig. 7. P3-RTE on unstructured mesh.

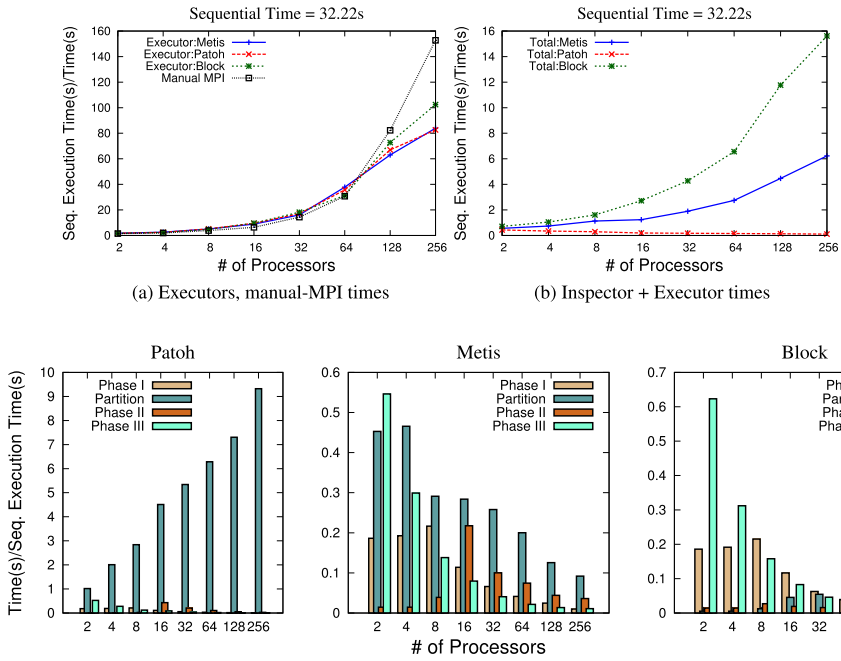
MPI implementation up to 32 processes. Since the block-partitioning scheme does not do this, the corresponding executor code achieves poor performance.

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 7(b) 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[2] developed by Sandia National Laboratories. It uses an implicit finite-element method over an unstructured 3D mesh. A problem size of 100 points along each axis was used for the evaluation. The suite also provides a manual MPI implementation of the computation, which was used for comparison.

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. Past that, the communication costs in the manual implementation are reduced by overlap of communication and computation. This aspect can be

[2]https://software.sandia.gov/mantevo



(a) Executors, manual-MPI times

(b) Inspector + Executor times

(c) Inspector overhead breakdown

Fig. 8. miniFE-1.1 with 100x100x100 grid.

incorporated into the current code-generation scheme by static analysis of the def-use relationship of arrays between the different partitionable loops.

Figures 8(b) and 8(c) show the impact of the inspector overheads on the total speedup achieved and a breakdown 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 miniapplication does not capture the behavior 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. Ocean, Land, and Atmosphere Modeling (OLAM)

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) [Walko and Avissar 2008] used for climate simulations of the entire planet, written in Fortran 90. It employs finite-volume methods to solve for physical quantities such as pressure, temperature, and wind velocity over a 3D unstructured grid consisting of 3D prisms covering the surface of the earth. Physical quantities are associated with centers of prisms and prism edges. The input grid contained 155520 prisms.

Since our current compiler implementation targets C codes and does not do any interprocedural 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 article. We focused on the atmospheric model simulation

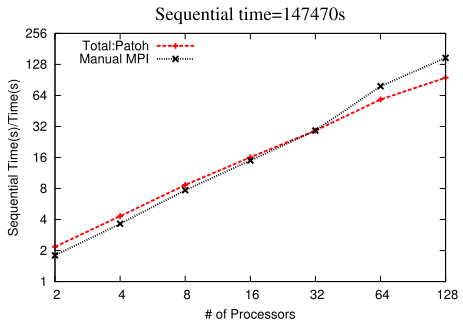


Fig. 9. OLAM atmospheric model.

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 time-loop, the overhead of the inspector is almost negligible, even when using the sequential hypergraph partitioner for generating the partitions. Therefore, the results shown in this section were obtained using PaToH.

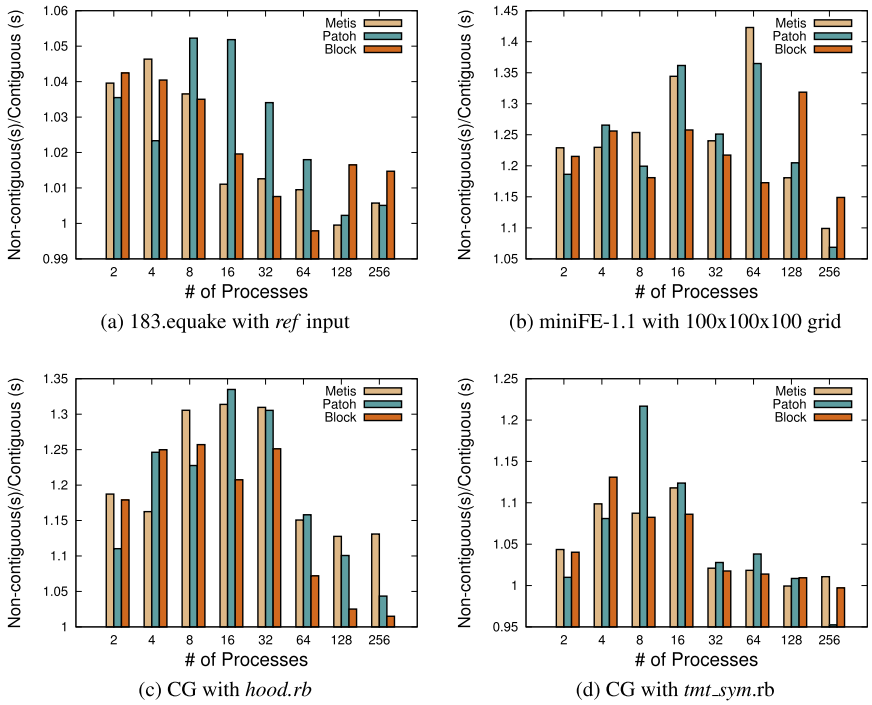
We compared the performance of our implementation with a reference MPI implementation developed by domain experts. The latter uses an efficient domain decomposition scheme to partition the computation across the MPI processes. Figure 9 shows that up to 32 processes, the performance of the code generated 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 article takes special care to preserve the contiguous accesses present in the original code within the generated executor code. The advantage of this is the following.

— At most one read from an indirection array is needed for every set of contiguous accesses to data arrays, reducing the total memory operations performed 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 runtime of the executor with and without this optimization applied to inner-loops alone for all the benchmarks where this optimization was applied. The benefit of the optimization is especially important in benchmarks that have a high footprint, like the CG kernel with *hood.rb* and miniFE-1.1 (Figures 10(c) and 10(b)), resulting in a 25% reduction in running times. 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. For *183.equake*, which has a small footprint to begin with, the benefit is within 5%.



(a) 183.equake with *ref* input

(b) miniFE-1.1 with 100x100x100 grid

(c) CG with *hood.rb*

(d) CG with *tmt_sym*.rb

Fig. 10. Impact of exploiting contiguity.

The results demonstrate the benefits of the proposed transformation on real-world applications. Significant engineering effort would be required to develop a compiler that could automatically handle such codes. Standard compiler techniques like interprocedural analysis and alias analysis would be needed to find partitionable loops. The code generation scheme would also have to be suitably modified to ensure that the local arrays are accessed correctly across different calling contexts.

9. RELATED WORK

Saltz and co-workers [Das et al. 1993, 1995; Saltz et al. 1990, 1991] proposed the inspector-executor (I/E) approach for distributed-memory code generation for scientific applications with irregular access patterns. The PARTI/CHAOS libraries [Berryman et al. 1991; Ponnusamy et al. 1993] 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 [Agrawal et al. 1995; von Hanxleden et al. 1993]. 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 not applicable to codes such as 183.equake and P3-RTE, where multiple elements of an array are updated within an inner loop.

Lain et al. [1995, 1996] 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 runtime reordering transformations [Ding and Kennedy 1999; Han and Tseng 2006; Mitchell et al. 1999]. Strout et al. [2002, 2003] proposed a framework for code generation that combined runtime 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 [2005, 2006] presented techniques for translating OpenMP programs with irregular accesses into code for distributed-memory machines. The translation scheme employed there would result in some arrays being replicated on all processes. Furthermore, in order to satisfy cross-iteration dependences, these shared data-structures have to be communicated across all processes, resulting in significant communication volume. While the cost of this communication is reduced by identifying monotonic access patterns and employing communication-computation overlap, such an execution model would result in larger footprints and high communication volume of the generated distributed memory code when compared to the scheme described in this article.

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 [Bondhugula et al. 2008; Feautrier 1992; Griebl 2004; Irigoin and Triolet 1988; Lim and Lam 1997; Ramanujam and Sadayappan 1992]. For loops not amenable to static analysis, speculative techniques have been used for runtime parallelization [Leung and Zahorjan 1993; Oancea and Rauchwerger 2012; Rauchwerger and Padua 1995; Rus et al. 2002; Yu and Rauchwerger 2000]. Zhuang et al. [2009] inspect runtime dependences to check if contiguous sets of loop iterations are dependent. Recently, dynamic inspection techniques have been developed to compute dependences between iterations of a loop, allowing for a grouping of independent iterations [Anantpur and Govindarajan 2013] to be executed in parallel and to allow for cross invocation parallelism of loops [Huang et al. 2013]. All of these approaches target shared-memory parallel systems or GPGPUs, and develop lightweight fine-grained synchronization schemes to respect cross-iteration dependences. Adapting these to target distributed-memory architectures in a scalable manner is a challenging problem.

In contrast to prior work, we develop an effective automatic parallelization approach for an extended class of affine computations with some forms of indirect array accesses. The approach described here has none of the data structures replicated, with communication required only for ghost elements. The resultant code is similar to manually developed distributed memory implementations from domain experts, especially in the context of scientific computing. 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 for reducing cache misses as well as enabling SIMD

optimizations in later compiler passes. Finally, the developed approach ensures that a sequence of parallel loops is partitioned such that iterations that touch the same data-elements are executed on the same process.

10. CONCLUSION

In this article 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, generates a parallel inspector that analyzes the iterations of the loop at runtime to partition both the iterations and the data. The iteration-to-data affinity is captured at runtime and used to reduce the communication volume. The inspector also generates auxiliary data structures that are used by the executor for efficient parallel execution. Our proposed optimizations for exploiting contiguity of accesses reduce the memory overhead significantly and also enable further compiler optimizations.

The effectiveness of the approach is demonstrated on several benchmarks and a complete climate simulation application, OLAM. 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.

ACKNOWLEDGMENTS

We thank Prof. Umit Catalyurek (BMI, OSU) for making available the PaToH software and for his help with formulating the partitioning constraints. We thank Prof. Gagan Agrawal (CSE, OSU) and the 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.

REFERENCES

Agrawal, G., Saltz, J., and Das, R. 1995. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.*

Anantpur, J. and Govindarajan, R. 2013. Runtime dependence computation and execution of loops on heterogeneous systems. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization.*

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, 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. *Comput. Meth. Appl. Mech. Eng. 152*, 85–102.

Baskaran, M. M., Ramanujam, J., and Sadayappan, P. 2010. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*, 264–263.

Basumallik, A. and Eigenmann, R. 2005. Towards automatic translation of openmp to mpi. In *Proceedings of the International Conference on Supercomputing*.

Basumallik, A. and Eigenmann, R. 2006. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the International Conference on Principles and Paradigms of Parallel Programming*.

Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., and Bastoul, C. 2010. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, 283–303.

Berryman, H., Saltz, J., and Scroggs, J. 1991. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Pract. Exper. 3*, 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 *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

Bondhugula, U., Hartono, A., Ramanujan, J., and Sadayappan, P. 2008. A practical automatic polyhedral program optimization system. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*.

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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

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 92*, 42.

Ding, C. and Kennedy, K. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*.

Eswar, K., Sadayappan, P., and Huang, C.-H. 1993. Compile-time characterization recurrent patterns in irregular computations. *Proceedings of the International Conference on Parallel Processing*.

Feautrier, P. 1992. Some efficient solutions to the affine scheduling problem - Part I: One-dimensional time. *Int. J. Para. Prog. 21*, 16, 389–420.

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 *Proceedings of the International Conference on Language and Compilers for Parallel Computing*.

Han, H. and Tseng, C.-W. 2006. Exploiting locality for irregular scientific codes. *IEEE Trans. Parallel Distrib. Syst.*

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.

Huang, J., Jablin, T., Beard, S., Johnson, N., and August, D. 2013. Automatically exploiting cross-invocation parallelism using runtime information. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*.

Irigoin, F. and Triolet, R. 1988. Supernode partitioning. In *Proceedings of the ACM Symposium on Principles of Programming Languages*.

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 *Proceedings of the International Symposium on Parallel Processing*.

Lamielle, A. and Strout, M. 2010. Enabling code generation 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 *Proceedings of the Symposium on Principles and Practice of Parallel Programming*.

Lim, A. W. and Lam, M. S. 1997. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the ACM Symposium on Principles of Programming Languages*.

Mitchell, N., Carter, L., and Ferrante, J. 1999. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

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 Perform. Comput. Appl.*

Oancea, C. E. and Rauchwerger, L. 2012. Logical inference techniques for loop parallelization. In *Proceedings of the Conference on Programming Language Design and Implementation*.

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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

Ramanujam, J. and Sadayappan, P. 1992. Tiling multidimensional iteration spaces for multicomputers. *J. Parallel Distrib. Comput. 16*, 2, 108–120.

Rauchwerger, L. and Padua, D. 1995. The LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization. In *Proceedings of the Conference on Programming Language Design and Implementation*.

Ravishankar, M., Mazumder, S., and Kumar, A. 2010. Finite-volume formulation and solution of the p3 equations of radiative transfer on unstructured meshes. *J. Heat Transfer 132*, 2.

Rus, S., Pennings, M., and Rauchwerger, L. 2002. Sensitivity analysis for automatic parallelization on multi-cores. In *Proceedings of the International Conference on Supercomputing*.

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*, 4.

Saltz, J. H., Berryman, H., and Wu, J. 1991. Multiprocessors and run-time compilation. *Concurrency and Computation: Pract. Exper. 3*, 6.

Schloegel, K., Karypis, G., and Kumar, V. 2002. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Pract. Exper. 14*.

Strout, M. M., Carter, L., and Ferrante, J. 2003. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the Conference on Programming Language Design and Implementation*.

Strout, M. M., Carter, L., Ferrante, J., Freeman, J., and Kreaseck, B. 2002. Combining performance aspects of irregular Gauss-Seidel via sparse tiling. In *Proceedings of the International Conference on Language and Compilers for Parallel Computing*.

Strout, M. M., George, G., and Olschanowsky, C. 2012. Set and relation manipulation for the sparse polyhedral framework. In *Proceedings of the International Conference on Languages and Compilers for Parallel Computing*.

Strout, M. M. and Hovland, P. D. 2004. Metrics and models for reordering transformations. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance*.

Von Hanxleden, R., Kennedy, K., Koelbel, C., Das, R., and Saltz, J. 1993. Compiler analysis for irregular problems in Fortran D. *Proceedings of the International Conference on Language and Compilers for Parallel Computing*.

Walko, R. L. and Avissar, R. 2008. The Ocean-Land-Atmosphere Model (OLAM). Part I: Shallow-Water Tests. *Monthly Weather Rev. 136*, 4033–4044.

Yu, H. and Rauchwerger, L. 2000. Techniques for reducing the overhead of run-time parallelization. In *Proceedings of the Conference on Compiler Construction*.

Zhuang, X., Eichenberger, A. E., Luo, Y., and O'Brien, K. 2009. Exploiting parallelism with dependence-aware scheduling. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*.

Received June 2013; revised December 2013; accepted March 2014