Distributed Memory Code Generation for Mixed Irregular/Regular Computations

Mahesh Ravishankar[1] Roshan Dathathri[1] Venmugil Elango[1]
Louis-Noël Pouchet[1] Atanas Rountev[1] P. Sadayappan[1] J. Ramanujam[2]

[1] The Ohio State University [2] Louisiana State University

{ravishan.3,dathathri.2,elango.4,pouchet.2,rountev.1,sadayappan.1}@osu.edu
ram@cct.lsu.edu

Abstract

Many applications feature a mix of irregular and regular computational structures. For example, codes using adaptive mesh refinement (AMR) typically use a collection of regular blocks, where the number of blocks and the relationship between blocks is irregular. The computational structure in such applications generally involves regular (affine) loop computations within some number of innermost loops, while outer loops exhibit irregularity due to data-dependent control flow and indirect array access patterns. Prior approaches to distributed memory parallelization do not handle such computations effectively. They either target loop nests that are completely affine using polyhedral frameworks, or treat all loops as irregular. Consequently, the generated distributed memory code contains artifacts that disrupt the regular nature of previously affine innermost loops of the computation. This hampers subsequent optimizations to improve on-node performance.

We propose a code generation framework that can effectively transform such applications for execution on distributed memory systems. Our approach generates distributed memory code which preserves program properties that enable subsequent polyhederal optimizations. Simultaneously, it addresses a major memory bottleneck of prior techniques that limits the scalability of the generated code. The effectiveness of the proposed framework is demonstrated on computations that are mixed regular/irregular, completely regular, and completely irregular.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Code generation

Keywords Distributed Memory, Inspector/Executor, Polyhedral Compilation, Irregular Computation

1. Introduction

Automatic parallelization of applications to target distributed memory systems remains a challenge for modern optimizing compilers. Recent developments in polyhedral compilation techniques [12, 17, 36] have addressed this problem in the context of affine computations. However, many scientific computing applications are outside the scope of such transformations due to data dependent control flow and array access patterns. Such computations are handled more effectively by the inspector/executor technique pioneered by Saltz et.al. [15, 46], and extended in [35]. Here, the compiler generates an inspector to analyze and partition the computation at run time. The generated executor uses the information collected by the inspector to execute the original computation in parallel. While effective for non-affine computations, applying these techniques directly to affine codes results in unnecessary inspector overheads.

```
1  #pragma parallel
2  for( k = 0; k < nf; k++ ){
3   fID = fine_boxes[k];
4   cID = ftoc[k];
5   for(i=start_y[fID]/2; i< end_y[fID]/2; i++)
6    for(j=start_x[fID]/2; j< end_x[fID]/2; j++){
7     int fy = 2*i - start_y[fID];
8     int fx = 2*j - start_x[fID];
9     int cy = i - start_y[cID];
10    int cx = j - start_x[cID];
11    phi[cID][cy][cx]=(phi[fID][fy][fx]+phi[fID][fy][fx+1]
       +phi[fID][fy+1][fx]+phi[fID][fy+1][fx+1])/4.0; } }
```

Listing 1. Example from AMR

More challenging are codes that have a mix of both affine and non-affine program regions, such as the example in Listing 1. This code is representative of Adaptive Mesh Refinement (AMR) computations in packages such as Chombo [14]. Here, the physical domain is divided into rectangular boxes which span the domain. Each box is further discretized using a structured grid, with values of physical quantities associated with each grid point. To capture sharp variation in these values, additional boxes that use a finer grid might be employed over those regions. These boxes (referred to as *fine boxes*) are co-located with boxes which use the coarser grid (referred to as *coarse boxes*). The outer loop in Listing 1 (loop `k`) iterates over all fine boxes and updates the values at grid points of a coarse box using values at grid points of a co-located fine box. Arrays such as `fine_boxes` and `ftoc` are used to index into data structures that contain data for individual boxes—e.g., arrays `start_x` and `phi`. Such arrays are known as *indirection arrays*. Using such arrays makes the computation within loop `k` *irregular* (or non-affine) since the control flow and data access pattern is known only at run time. Loops `i` and `j` are *regular* (or affine) since a standard stencil computation is used within them to update the value of `phi` at a grid point. Regular loops can be analyzed statically. Such a pattern of mixed irregular/regular computations is quite common in scientific applications.

Loop `k` in Listing 1 iterates over the set of fine boxes. The computation on each box can be performed independently. The loop is therefore parallel and its iterations can even be executed across

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA.
Copyright © 2015 ACM 978-1-4503-3205-7/15/02. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

```
1 loop_j=0;
2 for( k = ....)
3  for( i = ....){
4   offset = access_phi_0[loop_j] - lb[loop_j];
5   for( j = lb[loop_j] ; ...){
6    phi_l[offset+j] = phi_l[access_phi_1[body_j]]...
7    body_j++;    }
8   loop_j++;  }
```

Listing 2. Snippet of executor code from [35]

nodes of a distributed memory system. Due to data dependent loop bounds and array access patterns, automatic parallelization using purely static analysis cannot be employed here. Inspector/executor approaches developed in [35] can transform such codes, but they do not effectively handle parts of the code that are regular; for example, they do not recognize that for every iteration of loop `k`, `phi[cID][cy][cx]` accesses a rectangular patch of the array, or that the accesses on the right-hand side of the statement at line 11 represent a stencil operation. Without this information, expensive run-time analysis is needed to gather needed information for distributed memory parallel execution. For example, in [35], to access elements of a local array in a manner consistent with the original computation, traces of index expressions are generated by the inspector and used in the executor. Listing 2 shows a snippet of the generated executor code. `phi_l` is the local data array corresponding to `phi`. Arrays `access_phi_0` and `access_phi_1` store the traces (i.e., sequences of run-time values) for the index expressions used in `phi[cID][cy][cx]` and `phi[fID][fy][fx]`, respectively. Two problems are apparent even in this simplified example. First, generating, storing, and reading the large run-time traces for each array access expression is memory intensive and can significantly reduce the performance and scalability of the generate code. Further, the generated executor code does not maintain the structure of regular parts of the original code. This limits the composability of this technique with a wide variety of powerful optimizations for affine code regions. For example, state-of-the-art polyhedral compilers (e.g., PolyOpt [29]) can optimize the computation within loop `i` in Listing 1, while the corresponding loop in Listing 2 is outside the scope of such compilers.

The Sparse Polyhedral Framework [23, 43] aims to handle programs with a mix of affine and non-affine code regions by extending the polyhedral compilation framework. It uses *uninterpreted function symbols* (UFS) to model indirect accesses. While the framework is general enough to target a wide range of computations, it requires UFSs to be invertible to generate code. Such properties are hard to deduce statically, and typically are not true for common uses of indirection arrays.

We propose a framework that effectively models and transforms mixed irregular/regular computations for parallel execution on distributed memory systems. We focus on loops where the only dependence between iterations are due to associative and commutative reduction operations. In [35] such loops are refered to as *partitionable loops*; we follow the same nomenclature in this paper. The developed compiler algorithms analyze the regular parts of such loops statically. The generated inspector combines this information with that obtained at run time for the irregular parts of the loop to effectively partition its iterations. Completely affine and completely irregular computations are naturally expressed as special instances of the framework. The developed compiler framework is based on integer sets/maps/slices and has two important advantages:

1. By employing static models for regular-inner regions, it reduces the inspector overheads by eliminating the need to generate traces for affine array access expressions.
2. Unlike previous approaches, the executor code maintains the structure of the regular parts of the computation, making it amenable to further optimizations.

Compared to prior work, the proposed framework advances the state of the art. Unlike polyhedral techniques for distributed memory parallelization of affine codes [12, 17, 36], it handles a much broader class of computations. Importantly, this generality comes "for free." State-of-the-art polyhedral techniques can be easily incorporated into the developed framework with little or no overhead, as demonstrated experimentally. On the other hand, compared to modern inspector/executor techniques, exemplified by [35], our framework avoids a major bottleneck: the need to generate traces of access expressions for the regular-inner regions of the code. In the worst case, such traces exhaust the available memory and make it impossible to run the application. Our approach eliminates this bottleneck and improves the scalability of the code generated through inspector/executor techniques, as evident in our experiments. The result is a unified framework that can handle both irregular and regular program regions seamlessly.

In summary, the contributions of this work are as follows:

- A framework that can effectively model computations which exihibit an outer irregular/inner regular pattern;
- A code generation approach to improve the inspector/executor scalability for distributed memory parallelization by eliminating the use of large memory intensive traces;
- A transformation approach that preserves regular code regions in the generated code, enabling composability with polyhedral optimizations; and
- Experimental evaluation on mixed regular/irregular, completely regular, and completely irregular codes, demonstrating the effectiveness of the transformation framework.

2. Data Structures for Distributed Computations

Partitionable loops are annotated by the user as `#pragma parallel`, as shown in line 1 of Listing 1. Annotated loops cannot be nested within each other. All loops are assumed to have unit increments. Loop bounds and array index expressions are assumed to depend only on (1) iterator values, (2) values stored in read-only arrays, referred to as *indirection arrays*, and (3) scalars that not modified within the loop. Standard compiler techniques can be used to transform loops, like loop `k` in Listing 1, to satisfy these requirements.

We first define all the data structures needed to generate the final code executed in a distributed environment, namely,

- for a given processor, the set of iterations of the partitionable loop(s) it executes;
- for a given processor and the iterations it executes, the set of data elements accessed;
- for each data element, the processor owning that data.

Using classical inspector/executor approaches, several sets (typically in form of arrays) needed for the execution of the computation are populated using an inspector. On the other hand, when the computation can be fully analyzed and partitioned statically, use of an inspector is unnecessary since these sets can be derived at compile time. Instead of developing a framework that addresses one or the other based on the feature of the input program, we propose a unifying framework based on integer sets and maps that capture all the information needed to generate the distributed-memory programs. It captures in a single formalism, whether an inspector is needed to populate these sets or not, and even when an inspector is needed only for some parts of them.

2.1 Integer Sets and Maps

The two fundamental data structures used to represent computations and communications are *sets of integer points* and *maps between two sets of integer points*.

```
1 for (i = 0; i < N; ++i)
2   for (j = lb[i]; j < ub[i]; ++j)
3 S:   c[i] += A[j] + B[col[j]];
```

Listing 3. Sample SpMV Kernel

2.1.1 Integer Sets

A convenient way to represent the dynamic instances of a statement execution surrounded by `for` loops is to associate an integer point in a multi-dimensional space for each statement instance, such that its coordinates represent the value of the surrounding loop iterators for that instance. This is a classical concept in affine compilation, refered to as the *iteration domain* of a statement. A similar set of integer points is used to describe the set of data elements accessed. Here, the coordinates of each point captures the value used in the subscript function of an array reference.

Definition 1 (Integer Set)**.** *An integer set $\mathcal{S}$ is a set of integer points $\vec{x} \in \mathbb{Z}^d$. The dimensionality of the set is noted d, its cardinality is noted $\#S$. With $\vec{x} : (i_1, ..., i_d)$ we note:*

$$\mathcal{S} : \{(i_1, ..., i_d) \mid \textit{constraints on } i_1, ..., i_d\}$$

Standard operations on sets can be used, such as difference \, intersection $\cap$ and union $\cup$, as well as computing its image through a map/function. However, computing the result of these operations at compile time depends on the structural properties of the set. There exist numerous sub-categories of integer sets, each of which have different properties regarding the ability to compute them at compile time. When the constraints are a conjunction of affine inequalities involving only the variables i_i and parameters (constants whose value is not known at compile time), then the set is a polyhedron and all operations can be computed statically. A set defined using disjunctions of affine inequalities is a union of convex polyhedra. The intersection of a polyhedron and an integer affine lattice results in a $\mathcal{Z}$-polyhedron [18], which can also be computed statically using the Integer Set Library (ISL) [45]. For irregular computations, where the constraints involve affine inequalities of variables, parameters and functions whose value depends only on the value of their arguments, some operations may be computable at compile time with the Sparse Polyhedral Framework [23] using UFS, and an inspector. Finally, an inspector can be used to compute arbitrary sets as well.

2.1.2 Set Slicing

One key operation to reason about distributing a computation is *slicing*, i.e., taking a particular subset of a set. An integer set slice $\mathcal{S}_\mathcal{I}$ is a subset of $\mathcal{S}$ that is computed using another integer set $\mathcal{I}$.

Definition 2 (Integer Set Slice)**.** *Given an integer set $\mathcal{S}$, the integer set slice $\mathcal{S}_\mathcal{I}$ is $\mathcal{S}_\mathcal{I} = \mathcal{S} \cap \mathcal{I}$.*

Slicing can be used to extract polyhedral subsets from an arbitrary integer set. To achieve this we focus on a particular kind of slicing where the set $\mathcal{I}$ is a polyhedron made only of affine inequalities of the variables and parameters not occurring in the original computation but which we introduce for modeling purposes. For example, the iteration domain for the statement `S` in Listing 3, i.e., the set of dynamic instances of `S` can be written:

$$\mathcal{S} = \{(i,j) \mid 0 \le i < N \wedge lb[i] \le j < ub[i]\}$$

The expressions $lb[i]$ and $ub[i]$ are not parameters: they may take different values for different values of i. However, for a given i, these expressions are constants and can be viewed as parameters. Let us now define the slice $\mathcal{I}_1$, for $p_1 \in \mathbb{Z}$, as $\mathcal{I}_1 = \{(i,j) \mid i = p_1\}$. The set $\mathcal{I}_1$ is a set of two-dimensional integer points, with the first dimension set to a fixed but unknown value. The second dimension is unrestricted; this polyhedron is in fact a cone. The slice $\mathcal{S}_{\mathcal{I}_1}$ is defined as follows:

$$\mathcal{S}_{\mathcal{I}_1} = \mathcal{S} \cap \mathcal{I}_1 = \{(i,j) | 0 \le i < N \wedge lb[i] \le j < ub[i] \wedge i = p_1\} \tag{1}$$

This set now necessarily models a single point (e.g., a single loop iteration) along the first dimension. Two different iterations of the outer loop can be modeled by introducing another parameter $p_2 \in \mathbb{Z}$, with $p_1 \neq p_2$ for a slice $\mathcal{I}_2$ where $i = p_2$, to get:

$$\mathcal{S}_{\mathcal{I}_2} \quad = \quad \{(i,j) \mid 0 \le i < N \wedge lb[i] \le j < ub[i] \wedge i = p_2\}$$

Consequently, a set containing two arbitrary but different iterations of the outer loop is simply the union $\mathcal{S}_{\mathcal{I}_1} \cup \mathcal{S}_{\mathcal{I}_2}$ with $(p_1 > p_2) \vee (p_1 < p_2)$ as additional constraints. Two consecutive iterations can be modeled the same way, with $p_2 = p_1 + 1$.

In addition to the ease of modeling subsets of loop iterations (typically arising from the iterations of the partitionable loop(s) to be executed on a processor), these slices have a key property: *when fixing i to a unique value, the subset obtained is now a standard polyhedron*. This is because the expressions $lb[i]$ and $ub[i]$ are necessarily constants for a fixed value of i. We can now view them as parameters as well, say $lb[p_1]$ and $lb[p_2]$, and observe that a (union of) slice(s) containing a single iteration of the outer loop is now a (union of) classical polyhedra, which can be manipulated at compile time using tools like ISL. We do not require the use of uninterpreted functions, and enable polyhedral optimization on the computation to be executed on a particular processor, at the sole expense of extensively using unions of convex sets.

2.1.3 Integer Maps

The second data structure we use associates, or maps, integer points to other integer points. These are typical used to represent data elements accessed by a loop iteration.

Definition 3 (Integer Map)**.** *A map $\mathcal{M}$ defines a function from an integer set of dimension d to another integer set of dimension e, written as:*

$$\mathcal{M} : \{(i_1, ..., i_d) \to (o_1, ..., o_e) \mid \vec{o} = M(\vec{i})\}$$

where $\vec{i} : (i_1, ..., i_d)$ and $\vec{o} : (o_1, ..., o_e)$

Similar to integer sets, the tractability of the map depends on the form of the function M. If M can be represented as a matrix of integer coefficients (e.g., M is a multidimensional affine function) and is applied to a polyhedral set, then the output of the map (i.e., the set of points which are the image of the input set by the map) can be computed statically, as a union of integer sets (e.g., $\mathcal{Z}$-polyhedra). For example, consider the map representing the relationship between the iteration space and the data space for the reference $A[j]$, expressed as $\mathcal{D}_A : \{(i,j) \to (o_1) \mid o_1 = j\}$. To represent the set of data accessed by the entire computation for this reference, we note $\mathcal{D}_A(\mathcal{S}) : \{(o_1) \mid o_1 = j \wedge \vec{x} \in \mathcal{S}\}$. Here $\vec{x} \in \mathcal{S}$ is only a notation shortcut for the inequalities on i and j defining $\mathcal{S}$. For a particular slice $\mathcal{S}_{\mathcal{I}_1}$, this set is polyhedral. Therefore, the set of distinct data elements accessed by this reference can be computed at compile time. A code scanning exactly this set can then be generated, using polyhedral code generators like CLooG [1].

The map for the reference $B[col[j]]$ is $\mathcal{D}_B : \{(i,j) \to (o_1) \mid o_1 = col[j]\}$. $col[j]$ is not an affine function since its value for different j is unknown at compile time. Consequently, the data space $\mathcal{D}_B(\mathcal{S}) : \{(o_1) \mid o_1 = col[j] \wedge \vec{x} \in \mathcal{S}\}$ will require an inspector to be properly computed. Therefore the need for an inspector can be determined by building the data space for each reference and observing if the sets and maps of interest are not polyhedral sets or affine maps.

We conclude with the definition of the data space of an array which is simply the union of the data spaces touched by each reference to this array.

Definition 4 (Data space for an array)**.** *Given an array A and a collection of n references to it $\mathcal{D}_A^1, ..., \mathcal{D}_A^n$. To each reference k is associated an iteration set $\mathcal{S}_{\mathcal{D}_A^k}$. The set of distinct array elements accessed by the computation is $F_A : \bigcup_{i=1}^n \mathcal{D}_A^i(\mathcal{S}_{\mathcal{D}_A^i})$.*

2.2 Partitioning Computation and Data

We are now equipped to define all the sets needed to model a distribution of the computation. We rely extensively on set operations as well as slicing whenever appropriate to capture the partitioning of loop iterations and data communication.

2.2.1 Iteration Partitioning

The set K^q defines the set of iterations of a partitionable loop that are to be executed on a particular processor q. Depending on how the distribution scheme is determined (cyclic, block-cyclic, etc. or using run-time hypergraph partitioning) the set K^q of iterations of the partitioned loop(s) executed by q may be a consecutive subset of the loop iterations (thereby defined using affine inequalities) or an arbitrary, non-consecutive subset such as with hypergraph partitioning [42].

First, we construct a slice of the original iteration domain, $\mathcal{S}_{I_{\vec{p}}}$ by computing an intersection of the original iteration space $\mathcal{S}$ with a set $I_{\vec{p}}$. In set $I_{\vec{p}}$, iteration space dimensions whose iterators are used in the index expression of array accesses appearing in constraints of the various sets and map descriptions are fixed to a newly introduced parameter p (e.g., $i = p$ if the iterator i appears in a loop bound expression such as $lb[i]$). This allows all such expressions to be treated as parameters for a slice. All such dimensions will be refered to as the *irregular dimensions* of the computation. If the number of such dimensions is r, the set $I_{\vec{p}}$ is a cone of same dimensionality as $\mathcal{S}$, with the values of the r irregular dimensions each set to a newly introduced vector of parameter $\vec{p} \in \mathbb{Z}^r$. The set $\mathcal{C}$ contains all the different $\vec{p}_i$ needed to cover the full iteration space of the irregular dimensions with one distinct $\vec{p}_i$ per distinct iteration $\vec{i}$ of the irregular loop(s), with the property that $\vec{p}_i \neq \vec{p}_j$ for $\vec{i} \neq \vec{j}$. The original complete iteration space is therefore the union of all slices,

$$\mathcal{S} = \bigcup_{\vec{p} \in \mathcal{C}} \mathcal{S}_{I_{\vec{p}}} \quad (2)$$

In a parallel execution, each process executes a subset of the slices from the original computation, defined as $\vec{p} \in \mathcal{C}^q \subseteq \mathcal{C}$. The local iteration space $\mathcal{S}^q$ is simply the union of all slices executing on q,

$$\mathcal{S}^q = \bigcup_{\vec{p} \in \mathcal{C}^q} \mathcal{S}_{I_{\vec{p}}} \quad (3)$$

The set $\mathcal{C}^q$ is constructed from K^q by taking one distinct $\vec{p}_i \in \mathcal{C}$ per distinct element $k \in K^q$, and adding the constraint that $p_i^1 = k$ (p^1 is the value of the first dimension of $\vec{p}$). A key observation is that if the computation is affine, the set K^q can be computed statically. For example, if the outermost loop is block partitioned the set K^q can be expressed as $K^q = \{k : b_1 \leq k < b_1 + B\}$ where b_1 is a newly introduced parameter and B is the block size. The set $\mathcal{C}^q$ is reduced to a single parameter:

$$\mathcal{C}^q = \{\vec{p} : b_1 \leq p^1 < b_1 + B\}$$

Consequently the slice $I_{\vec{p}}$ will contain B iterations, and the above union can be fully computed statically.

2.2.2 Data Partitioning and Ghost Communication

In our execution model, the data is partitioned amongst the processes such that each process has all data needed to execute the set of iterations of the partitionable loop(s) mapped to it. If $\mathcal{D}_A$ is an integer map used to represent accesses to array A, for each slice of the iteration space, the elements of array A accessed by it can be computed as,

$$F_{A,I_{\vec{p}}} = \mathcal{D}_A(S_{I_{\vec{p}}}) \quad (4)$$

Eq (4) represents a slice of the local data space of array A. A union of these slices gives the local data space on a process.

$$F_A^q = \bigcup_{\vec{p} \in \mathcal{C}^q} \mathcal{D}_A(S_{I_{\vec{p}}}) \quad (5)$$

For affine computations, since $\mathcal{C}^q$ can be defined statically (as in Eq 2.2.1 when the loop is block-partitioned), F_A^q can also be computed statically.

In general, the same data element might be accessed by iterations mapped to two or more different processes. In such cases, one of the processes is assigned as the *owner* of the element and the location of the element on the other processes are treated as *ghosts*. The location at the owner contains the correct value of the data element, with ghost locations storing a snapshot of the value at the owner. Since the partitioned loops are parallel, these elements are either read from or are updated through commutative and associative operations. The loop itself can be executed in parallel without any communication as long as

- The ghost locations corresponding to elements that are read within the loop are updated with the value at the owner before the loop execution
- The ghost locations corresponding to elements that are updated within a loop are initialized to the identity of the update operator used (0 for '+', 1 for '*') before the loop execution. After the loop execution, the ghost locations contain partial contributions to the final value and are communicated to the owner process where values from all ghost locations are combined.

To setup the communication between processes, we define a set $\mathcal{O}_A^q$ which contains all the elements of array A that are owned by process q. This set could either be decided at compile time (using block or cyclic distribution of array elements), or could be computed based on run time analysis that uses the iteration-to-data affinity [35]. Since each array element has a unique owner, $\mathcal{O}_A^q \cap \mathcal{O}_A^{q'} = \phi$ if $q \neq q'$. Note that the choice of the set $\mathcal{O}_A^q$ does not change the communication volume as long as $\mathcal{O}_A^q \subseteq F_A^q \; \forall \; 0 \leq q < N$.

On a process q, the set of ghost locations for array A which are owned by process q' can be computed as follows:

$$G_A^{q,q'} = F_A^q \cap \mathcal{O}_A^{q'} \quad (6)$$

This gives the elements of array A on process q that are

- Received from process q' if A is read in the partitioned loop
- Sent to process q' if A is written in the partitioned loop.

To complete the setup for communication, we also need to compute the set of all ghost locations on process q' that are owned by process q. This can be computed as:

$$O_A^{q,q'} = F_A^{q'} \cap \mathcal{O}_A^q \quad (7)$$

$O_A^{q,q'}$ gives the elements of array A on process q that are

- Sent to process q' if A is read in the partitioned loop
- Received from process q' if A is written in the partitioned loop

Computing $O_A^{q,q'}$ requires computing the data space for iteration space slices mapped to process q' on process q. Since this process has to be repeated for all $q' \in \{[0, N-1] - q\}$, this requires enumerating all the iterations space slices in $\mathcal{C}$ on all the processes. To avoid this, since $G_A^{q,q'} = O_A^{q',q}$, each process computes only $G_A^{q,q'}$ and communicates this information to process q' for all $q' \in \{[0, N-1] - q\}$. Process q' uses this information to compute $O_A^{q',q}$.

Algorithm 1: GenerateInspector($\mathcal{A}$)

```
Input  : A : AST of the annotated parallel loop
Output : A_I : AST of the inspector
1  begin
2    A_I = φ ;
3    [N,R] = FindIrregularDimensions(A) ;
4    if N ≠ φ then
5      A_I = MakeCopy(A) ;
6      InsertCheckLocalIteration(A_I) ;
7      P = InsertTemporaryVariables(A_I,N,R) ;
8      I = ComputeAffineIterationSpace(A_I,N,R,P) ;
9      foreach a ∈ Arrays(A) do
10       D_a = ComputeAccessMap(A_I,N,R,P,a) ;
11       F_a = ComputeImage(D_a,I) ;
12       F_a^O = ProjectOutInnerDimensions(F_a) ;
13       if IsMultiDimensional(a) then
14         F_a^I = ParameterizeOuterDimension(F_a) ;
15         InsertCodeToComputeBounds(A_I,F_a^I,F_a^O) ;
16       InsertCodeForExactUnion(A_I,a,F_a^O) ;
17       foreach e ∈ ArrayIndexExpression(a,A_I) do
18         o = e.OuterDimension ;
19         if ¬IsOfDesiredForm(o) then
20           InsertCodeToCreateTrace(o) ;
21     RemoveRegularLoopsAndStatements(A_I,R) ;
22     GenerateGaurdsForIndirectionArrayAccesses(A_I) ;
23   return A_I ;
```

```
1 t1=start_y[fine_boxes[k]]; t2=start_x[fine_boxes[k]];
2 t3=end_y[fine_boxes[k]]; t4=end_x[fine_boxes[k]];
3 t5=start_y[ftoc[k]]; t6=start_x[ftoc[k]];
4 t7=fine_boxes[k]; t8=ftoc[k];
```

Listing 4. Temporary Variables for Listing 1

The above formulation assumes that each process communicates with all other processes. In reality for many scientific computing applications each process communicates with only a subset of processes, i.e. the $G_A^{q,q'}$ is non-zero for only a few values of q'.

3. Generation of Inspector/Executor Code

Once the iteration space has been partitioned by distributing its slices amongst processes (Section 2.2.1), to partition the data the data space for each array is computed using Eq (5). This data space represents the local array on each node and has to be computed by an inspector due to use of indirection arrays. For presentation purposes, each annotated loop is assumed to be perfectly nested. Imperfectly nested loops can be handled by considering each statement to be perfectly nested within its surrounding loops with different statements embedded within the same iteration space during code-generation.

3.1 Local Data Space of Arrays : Inspector Code

Algorithm 1 generates the inspector code. For a given annotated loop AST, function *FindIrregularDimensions* marks a loop as being irregular if the value of its iterator is used in index expressions of indirection array accesses. Since we target computations that are irregular outer loops and regular inner loops, loops surrounding an irregular loop are marked as irregular as well. In presence of one or more irregular loops, 1) the iteration space slices mapped to a process can be computed only at runtime, and 2) while the data space for a single slice can be computed statically using Eq (4), the union of these slices has to be evaluated at runtime by the inspector. When the annotated loop is affine, no inspector is needed since the iteration and data partitioning is computed statically.

The AST of the inspector code is constructed by first replicating the AST of the annotated loop. The loop body of the outermost loop is enclosed within a conditional that executes only those iterations that belong to set K^q (line 6). Since all indirect accesses are invariant with respect to the loops that constitute the regular portions of the computation, the value of all such expressions can be stored in temporary variables just before the outermost regular loop (line 7). All indirect accesses are be replaced with references to the corresponding temporary variable. Listing 4 shows the temporary variables used, and the expressions in Listing 1 they replace. The point in the inspector AST immediately after these statements enumerates elements of $\mathcal{C}^q$, and can be used to analyze the iteration space slices mapped to a process.

The iteration space slice representing the regular portion of the AST can be expressed using affine constraints involving the temporary variables added at line 7. This is computed at line 8 by *ComputeAffineIterationSpace*. For Listing 1, a slice would be,

$$I_P := \{(k,i,j) | k = p1 \wedge t1/2 \leq i < t3/2 \wedge t2/2 \leq j < t4/2\}$$

To compute the local data space for each array, the integer map representing accesses to it is built at line 10. For a single reference, such as `phi[cID][cx][cy]` in Listing 1, this map would be

$$\mathcal{D}^1_{phi} := \{(k,i,j) \rightarrow (l,a,b) | l = t8 \wedge a = i - t5 \wedge b = j - t6\}$$

Such a map is built for each access of the array. A union of these maps is computed statically and is applied to the iteration space slice, I_p at line 11 of Algorithm 1. The resulting set represents a slice of the data space for an array (Eq 4).

$$\mathcal{D}^1_{phi}(I_p) := \{(l,a,b) \mid l = t8 \ \wedge \ t1/2 - t5 \leq a < t3/2 - t5 \ \wedge \ t2/2 - t6 \leq b < t4/2 - t6\} \quad (8)$$

The union of data space slices can be computed at runtime by maintaining, for each array, a set of elements accessed on a process. For all 1D arrays, line 16 inserts code to add elements of the set computed at line 11 to this set, at runtime.

For large multi-dimensional arrays, computing an exact union of all elements accessed is very expensive. This cost can be reduced by recognizing that usually outer dimensions of such arrays are accessed using indirection arrays, while inner dimensions are accessed using affine expressions. For example, the access `phi[cID][cy][cx]` in Listing 1 results in the outer dimension being accessed using indirections, but for a given iteration of loop `k`, a rectangular patch of the inner dimensions of the array are accessed (see Eq (8)). Therefore, for a multi-dimensional array, the union of data space slices on a process is computed as follows,

- The exact union of the set of all the outermost indices of the array accessed by each slice is computed.
- For each index of the outer dimension, a bounding box approach is used to compute the union for all the inner dimension indices touched for an outer dimension index.

Since an exact union is computed only for the outer dimension indices, the cost of computing the union is drastically reduced.

To use this approach, the outer dimension of the data space slice computed at line 11 is parameterized by applying the following map at line 14.

$$PO = \{(o_1, o_2, \ldots, o_e) \rightarrow (o_2, \ldots, o_e) \mid o_1 = p_o\}$$

The resulting expression computes the set of indices of inner dimensions accessed for every outer dimension index, p_o, by a particular data slice. Applying PO to result of Eq 8 gives

$$PO((D^1_{phi}(I_p)) := \{(l,a,b) \mid l = t8 \ \wedge \ l = p_o \ \wedge \ t1/2 - t5 \leq a < t3/2 - t5 \ \wedge \ t2/2 - t6 \leq b < t4/2 - t6\}$$

Above, the slice of data on a process accesses only elements of a particular outer-dimension index of array `phi`. The expression

for the lexicographic minimum and maximum of above expression, parametrized using the variables added at line 7, is computed statically at line 15. The inspector code uses this expression to compute the lexicographic minimum and maximum across all data slices mapped to a process at runtime. The set of outer dimension indices of an array accessed on a process can be computed by projecting out the inner dimensions of the data space slice computed at line 11. By construction, this is a single point parametrized by the temporary variables added at Line 7. The union of these points is computed at runtime by the inspector. The code for this is generated at Line 16.

Once the data space of all arrays has been computed, the set of owned elements, $O_*^{q,q'}$ and ghost elements, $G_*^{q,q'}$ on each process needs to be computed as described in Section 2.2.2. Having computed the local data space, the inspector allocates an array of size equal to this space and populates it with values from the original array in lexicographic order.

Prefetching indirection array values. The developed approach is targeted towards applications where a single node does not have enough memory to replicate any of the data structures. For presentation purposes we assume all arrays (including indirection arrays) are initially block-partitioned across processes. The specific choice does not affect the techniques developed here. As a result, a process might not have all the indirection array elements necessary to compute loop bounds and array index expressions used in the inspector code. These values have to be prefetched on each process based on the iteration space slices being computed on it. This is done by modifying the inspector code to incorporate the approach developed earlier in [35]. A more detailed description is omitted due to space constraints.

3.2 Executing Iteration Space Slices : Executor Code

Algorithm 2 shows the steps involved in generating the executor code to execute the iteration space slices mapped to each process.

We first describe the modifications to the loops that are marked as regular. Similar to Algorithm 1, statements to assign all indirect access expressions to temporary variables are inserted just above the outermost regular loop. The regular loops within computations, which represent a slice of the iterations space, are expressed using affine inequalities involving these temporary variables, loop iterators and program parameters. Therefore, polyhedral code-generation tools like CLooG [1] can be used to generate the code for the regular portions of the executor code (line 6). The developed code-generation algorithm can seamlessly incorporate transformations, like those described in [13, 22], to optimize these code regions of the executor. Since the focus of this paper is not to explore the space of possible transformations but to enable such transformations in applications that fall under the irregular-outer regular-inner paradigm, no such transformations have been currently implemented in our framework.

For fully affine computations (lines 9-17) the loop nest generated at line 6 replaces the entire executor code. The loop bounds of the outermost loop are modified statically to execute only a portion of the iteration space on each process. The data space of all arrays on a process is computed statically as well. Sophisticated techniques developed within the polyhedral compiler framework [17, 36] can be incorporated within the formalism developed in Section 2. Our current implementation implements a block-partitioning scheme described in Section 2.2.1 for affine computations. All array access expressions are modified to refer to the corresponding local arrays. The array index expressions are replaced by the original expressions subtracted with the lexicographically smallest index of the array accessed on a process.

For an input AST with one or more irregular iterators, the function *ReplaceOuterLoopBounds*, modifies the bounds of the outer-most loop to iterate from 0 to $|K^q|$-1, where $|K^q|$ is com-

Algorithm 2: GenerateExecutor($\mathcal{A}$)

Input : $\mathcal{A}$: AST of the annotated parallel loop
Output: $\mathcal{A}_E$: AST of the executor

1 **begin**
2 $\quad\mathcal{A}_E$ = MakeCopy($\mathcal{A}$);
3 $\quad[N,A]$ = FindIrregularDimensions($\mathcal{A}$);
4 $\quad P$ = InsertTemporaryVariables($\mathcal{A}_E$,N,A);
5 $\quad I$ = ComputeAffineIterationSpace($\mathcal{A}_E$,N,A,P);
6 $\quad A_{new}$ = GenerateLoopNests(I);
7 $\quad$ReplaceWithLocalArrayReferences($\mathcal{A}_E$);
8 $\quad$**if** $N = \phi$ **then**
9 $\quad\quad$PartitionIterationSpace(A_{new});
10 $\quad\quad$ReplaceLoops($\mathcal{A}_E$,A,A_{new});
11 $\quad\quad$**foreach** $a \in$ *Arrays($\mathcal{A}$)* **do**
12 $\quad\quad\quad D_a$ = ComputeAccessMap($\mathcal{A}_E$,N,A,P,a);
13 $\quad\quad\quad F_a$ = ComputeImage(D_a,I);
14 $\quad\quad\quad L_{min}$ = ComputeLexMins(F_a);
15 $\quad\quad\quad$**foreach** $i \in$ *ArrayIndexExpression(a,$\mathcal{A}_E$)* **do**
16 $\quad\quad\quad\quad i_{new}$ = NewSubtractExpression(i,L_{min});
17 $\quad\quad\quad\quad$ReplaceExpression(i,i_{new});
18 $\quad$**else**
19 $\quad\quad$ReplaceLoops($\mathcal{A}_E$,A,A_{new});
20 $\quad\quad$ReplaceOuterLoopBounds($\mathcal{A}_E$);
21 $\quad\quad$**foreach** $a \in$ *Arrays($\mathcal{A}$)* **do**
22 $\quad\quad\quad$**foreach** $e \in$ *ArrayIndexExpression(a,$\mathcal{A}_E$)* **do**
23 $\quad\quad\quad\quad o$ = e.OuterDimension;
24 $\quad\quad\quad\quad$**if** $\neg$*IsOfDesiredForm(o)* **then**
25 $\quad\quad\quad\quad\quad$InsertCodeToReadTrace($o$);
26 $\quad\quad\quad\quad$**if** *IsMultiDimensional(a)* **then**
27 $\quad\quad\quad\quad\quad l_{min}$ = GetLexminValueExpression(a,o,e.InnerExpressions);
28 $\quad\quad\quad\quad\quad i_{new}$ = NewSubtractExpression(e.InnerExpressions,l_{min});
29 $\quad\quad\quad\quad\quad$ReplaceExpression($e$.InnerExpression,$i_{new}$);
30 $\quad$**return** $\mathcal{A}_E$;

puted by the inspector. Since the original value of the iterator is needed within the loop body, the inspector creates a temporary array, `local_k`, to store the elements of K^q in increasing order. All references to the outer-most loop iterator, `k`, are replaced with `local_k[k]`. Section 4.3 describes cases where this array can be eliminated. The loop bound expressions of inner loops are left as is so that their iterators assume the same values as the original computation.

For array access expressions of the following form,

$$\langle \text{MapExpr} \rangle ::= \langle \text{Iterator} \rangle | \langle \text{Array} \rangle [\langle \text{MapExpr} \rangle$$

Section 4.3 describes the corresponding index expression to be used in the executor code. For expressions used to access inner dimensions of multidimensional arrays, the expression used in the executor is obtained by subtracting the original expression with the lexicographic minimum of the inner dimensions accessed for a particular index of the outermost dimension on that process, computed at line 15 of Algorithm 1.

Once the executor code has been generated, communication calls necessary to exchange the values of ghost locations are inserted before and after the executor AST.

4. Modifying Array Index Expressions

The index expressions in the executor code generated in Section 3.2 have to be modified to access elements of local arrays in a manner consistent with the original computation. In [35] the executor read a trace of the original index expression, modified to access corresponding locations in the local array. Sections 5.1 and 5.3 show that the size of these traces adversely affect the scalability and performance of the generated code. In the presented approach, this issue

is avoided by creating local indirection arrays that mimic the behavior of the original indirection arrays on each process.

It is helpful to recognize that indirection arrays represent an encoding of a map from one set of *entities* in the computation to another. For example, in Listing 1, the array `ftoc` represents a map between boxes at a finer level of refinement and colocated boxes which are refined at a coarser level. The outer loop `k` iterates over the fine boxes and uses the array `ftoc` to locate the corresponding coarse box. In unstructured grid computations, indirection arrays represent a map from a face to the two adjacent cells. Further, the same map, or indirection array, is used to access multiple arrays, all of which store data associated with a particular entity. For example, in Listing 1 the array `fine_boxes` is used to access elements of `start_y`, `end_y`, `start_x` and `end_x`, all of which store information associated with a particular box. Multiple levels of indirection represent a composition of such maps.

In the partitioned computation, each process accesses data related to a subset of the different entities from the original computation. For example, partitioning the computation in Listing 1 results in each process accessing information associated with a subset of boxes used in the original computation. As a result, the arrays `start_y`, `end_y`, `start_x` and `end_x` have similar data space on a process. A local indirection array, say `local_fine_boxes`, could be used to access the local versions of all these arrays in the executor code. These local indirection arrays encode the same mapping as the original code, but in terms of a local numbering of entities on a process.

$\langle\text{MapExpr}\rangle$ represent a common form of array index expressions used in scientific computing applications in which indirection arrays are used as maps between entities. Here we present an approach where $\langle\text{Arrays}\rangle$s in such expressions can be replaced with local indirection arrays in the executor code. The creation of such arrays can be viewed as a two-step process: (1) create a local buffer that contains all the elements of the original indirection array accessed on a process. This is already done by the inspector code generated in Section 3.1; (2) Change the values stored in the local indirection arrays to point to the corresponding local positions of elements of the target array accessed. The latter presents a challenge when multiple arrays are accessed using the same indirection array, since the change is valid only if all target arrays have the same local data spaces. To address this issue, we first introduce *access graphs* that help in capturing the levels of indirection and later describe how they are used to achieve the second aim listed above.

4.1 Access Graphs

Information about arrays accessed using indirections, and the levels of indirection used in the original code is represented using a Directed Acyclic Graph (DAG), called an *access graph*. Nodes in this graph represent arrays or iterators of loops marked as irregular. For all expressions of the form $\langle\text{Array}\rangle[\langle\text{Iterator}\rangle]$, where $\langle\text{Iterator}\rangle$ corresponds to a loop marked as irregular, an edge is added from the node representing the iterator to the node representing the array. For array access expressions of the form $\langle\text{Array}\rangle_1[\langle\text{Array}\rangle_2[\langle\text{MapExpr}\rangle]]$, an edge is added from the node representing $\langle\text{Array}\rangle_2$ to the node representing $\langle\text{Array}\rangle_1$.

Expressions of the form $\langle\text{Array}\rangle[\langle\text{MapExpr}\rangle]$s might appear in loop bounds and inner dimensions of multi-dimensional arrays (`start_x`, `start_y`, etc. in Listing 1). $\langle\text{Array}\rangle$s used here are also indirection arrays, but their values are used in the executor as is. To capture such uses of an indirection array, an edge is added from the node representing the $\langle\text{Array}\rangle$ to a special node, *Global*. Figure 1 shows the access graph built for the code in Listing 1. Since maps in scientific computing are rarely used in a cyclic manner, the resultant graph is indeed a DAG.

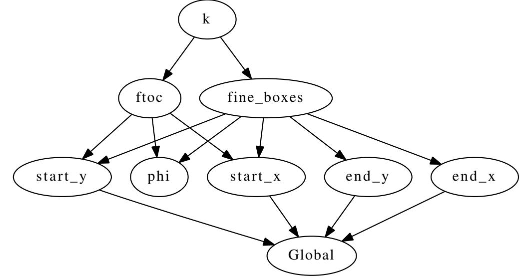

Figure 1. Access Graph for Listing 1

All arrays used as indirection arrays in the computations have one or more immediate successors in the access graph. Nodes that have multiple immediate successors represent indirection arrays used to access multiple arrays. Values of the local indirection array can be modified to access the corresponding local arrays for all its immediate successors if the data space computed by Eq (5) for all of them is the same. This can be enforced by setting the local data space of all the arrays represented by the immediate successor nodes to be equal to the union of all the individual data spaces. Since the arrays represented by the immediate successors typically store information related to a particular entity, these arrays would have similar local data spaces to begin with.

A trivial solution would be to add all the array nodes to one set. This would lead to a significant over-estimation of the data space on each process. To avoid this trivial solution, sets can be created such that no node and its immediate predecessor belong to the same group. Finally, a node which is an immediate predecessor to the *global* node represents an indirection array whose values are not modified in the executor. Therefore, such indirection arrays cannot be used to access other arrays in the executor. The grouping is done such that no array node is added to the same set as the global node.

In summary, the grouping of array nodes has to satisfy the following three conditions,

1. All immediate successors of an array node must belong to the same set.
2. No node should be in the same set as the global node.
3. An array node and its immediate successors must not belong to the same set.

4.2 Grouping Nodes of the Access Graph

Algorithm 3 presents a scheme to group array nodes into sets while satisfying the above requirements, adding new nodes whenever necessary. Each node is assumed to have two fields, (1)*group*, to denote the set which the node belongs to, and (2) *Type* which can either be *ArrayNode*, *LoopNode* or *GlobalNode*. Initially, the field *group* for all nodes is set to *Unknown*.

The access graph is traversed in reverse topological order. For every node encountered, all of its immediate successors have already been assigned to groups. The algorithm tries to add all its immediate successors to the same group as its first immediate successor by calling the function *ChangeGroupNum* for each of them. This function, described in Algorithm 4, takes as input the node v whose group number has to be changed and the group number, g, to change to. If the node v is the *GlobalNode* the function returns *False* (condition 2). If the node has already been assigned to a group, then at least one of its immediate predecessors has already been visited by Algorithm 3. For all such previously visited predecessors, the function returns *False* if any of them are also assigned to the same group, g (condition 3). Otherwise, the node v can be assigned to the group if all the immediate successor of the previously visited immediate predecessors of v, are assigned to the same group

Algorithm 3: GroupNodes(G)

```
    Input  : G = (V, E) : An access graph with group numbers Unknown
    Output: G = (V, E) : Graph with nodes added into groups
1   begin
2     ngroups = 0 ; done = False ;
3     while ¬ done do
4       done = True ;
5       foreach v in ReverseTopologicalOrder(G) do
6         group = Unknown ; Unchanged_set = φ ;
7         foreach s in v.successors do
8           if group = Unknown then
9             group = s.group ;
10          else if group ≠ s.group then
11            processed_set = v ;
12            if ChangeGroupNum(s,group,processed_set) then
13              Unchanged_set = Unchanged_set ∪ s ;
14        if v.Type = ArrayNode then
15          if Unchanged_set ≠ φ then
16            r = Copy(v) ; V = V ∪ r ;
17            r.predecessors = v.predecessors ;
18            r.successors = Unchanged_set ;
19            v.successors = v.successors - Unchanged_set ;
20            done = False; Break ;
21          v.group = ngroups ; ngroups = ngroups + 1 ;
22    return G ;
```

Algorithm 4: ChangeGroupNum(v,group,processed_set)

```
    Input  : v : Node in the graph
             g : Group number to be added to
             processed_set : Predecessors to be ignored
    Output: True if the node was added to the group, False otherwise
1   begin
2     if v.Type = GlobalNode then
3       return False ;
4     if v.group = Unknown then
5       v.group = g ;
6       return True ;
7     foreach p in (v.predecessors - processed_set) do
8       if p.Type = ArrayNode ∧ p.group ≠ Unknown then
9         if p.group ≠ g then
10          processed_set.insert(p) ;
11          foreach s in p.successors do
12            if ¬ ChangeGroupNum(s,group,processed_set) then
13              return False;
14        else
15          return False ;
16    v.group = g ;
17    return True ;
```

(condition 1). This is checked recursively. The set, *processed_set*, ensures that there is no infinite mutual recursion.

If the call to *ChangeGroupNum* at line 12 of Algorithm 3 returns *False*, it implies that current successor, s, conflicts with the previous successors and cannot be added to the same group. It is removed as an immediate successor of node v being analysed, and added as an immediate successor to a copy of node v to remove this conflict. This modification implies that two copies of the indirection array represented by node v is needed to satisfy all the grouping constraints. This new graph can now be traversed for grouping the nodes. When no conflicts are found all array nodes have been grouped appropriately and the algorithm terminates. In the worst-case this algorithm will create a graph where all nodes have only one successor. Such a scenario would not be common in the kind of applications targeted in this paper.

The values in the local indirection arrays are modified to point to the corresponding locations in the local data space of the arrays represented by its immediate successor. The values are not modified when the immediate successor is the node *Global*.

For the graph in Figure 1, arrays `ftoc` and `fine_boxes` belong to the same group. The arrays `start_x`, `start_y`, `end_y`, `end_x`, and `phi` belong to another group. Since the arrays of the second group are enforced to have same local data space on each process, a local indirection array `local_fine_boxes` can be used to access all of these arrays in the executor. This local indirection array mimics the behavior of the array `fine_boxes` in Listing 1.

4.3 Modifying Index Expressions in the Executor Code

Array index expressions in the executor code generated in Section 3.2 are modified as follows. For access expressions the form $\langle\text{Array}\rangle_1\langle\text{Array}\rangle_2[\langle\text{MapExpr}\rangle]$, the $\langle\text{Array}\rangle_2$ is replaced to refer to the the copy of local indirection array represented by the immediate predecessor of the node that represents $\langle\text{Array}\rangle_1$.

Since iterators of inner loops have same values as the original computation, array index expressions of the form $\langle\text{Iterator}\rangle$ which use inner loop iterators have to be manipulated to point to local elements of the array. The method used in [35] to recreate unit-stride accesses can be adapted for this purpose. Since loops in the input codes have unit increments, for every invocation of the inner loop, the index expression $\langle\text{Iterator}\rangle$ evaluates to a contiguous sequence of values. A local array is created to record the first element of this sequence for every invocation of the loop. This sequence can be renumbered to point to the corresponding location in the local target array. The rest of the elements can be accessed by adding the value of the loop iterator subtracted by the lower bound of the current loop invocation. The size and values in the arrays to be used can be computed by an inspector. The same sequence can be used to access all arrays which are immediate successors of the node corresponding to this iterator in the access graph, and belong to the same group.

For array access expressions of the form $\langle\text{Array}\rangle[\langle\text{Iterator}\rangle]$ in the original code, where the iterator is from the partitioned loop; the executor code generation replaced such expressions with a new expression of the form $\langle\text{Array}\rangle_1[\langle\text{Array}\rangle_2[\langle\text{Iterator}\rangle]]$. $\langle\text{Array}\rangle_2$ represents a temporary array that contained the original values of the iterator mapped to a process (array `local_k` in Section 3.2). In the access graph, the node p corresponding to this iterator would have only one immediate successor l, the node that represents $\langle\text{Array}\rangle_2$. If a successor m of this node, representing $\langle\text{Array}\rangle_1$, has no other predecessors then it can be concluded that every iteration of the outer loop accesses only one element of $\langle\text{Array}\rangle_1$. Since the outer loop iterations are executed in order of their original index, and array elements are laid out in order of their original index too, these array elements can be accessed using the iterator value itself. The node m is removed as a successor from node l and added as an immediate successor to node p. The array expression in the executor is changed back to be $\langle\text{Array}\rangle[\langle\text{Iterator}\rangle]$, where $\langle\text{Array}\rangle$ refers to the corresponding local array, and $\langle\text{Iterator}\rangle$ is the outer loop iterator. The temporary array added could be eliminated if the node l has no successors after this modification. The above holds true for nodes representing `ftoc` and `fine_boxes` in Figure 1. Therefore, in the executor code, `ftoc[k]` and `fine_boxes[k]` are replaced by `local_ftoc[k]` and `local_fine_boxes[k]` respectively. A sample of the generated inspector code and executor code is available elsewhere [2].

Finally, for array index expressions not of the form $\langle\text{MapExpr}\rangle$, as a fallback the inspector code generated by Algorithm 1 is modi-

Nprocs	2	4	8	16	32	64	128	256
Executor(s)	895	461	244	125	63.8	32.4	16.5	8.03
Inspector(s)	2.87	2.78	1.10	0.18	0.11	0.06	0.05	3.32
Total(s)	898	464	244	125	63.9	32.5	16.6	11.3

Table 1. Mixed Irregular/Regular benchmark : Poisson equation solver. Sequential Time = 1055s

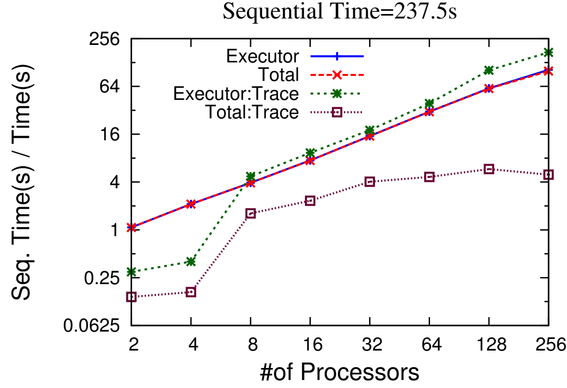

Figure 2. Comparison with [35]

fied to generate a trace of this index expression. The executor code generated by Algorithm 2 is modified to read from this trace.

5. Case Studies

We evaluated the performance of the generated distributed memory code for computations that are (1) Mixed Irregular/Regular, (2) completely regular, and (3) completely irregular. The code generator was implemented as a source-to-source transformation within ROSE [5] for C-codes. MVAPICH2-1.9 was used for communication between processes. The communication costs were reduced by using ARMCI/GA-5.2 [26] for one-sided communication during ghost updates. Intel C Compiler 13.1.2 was used to compile the generated distributed memory code. Experiments were run on a cluster of quad-core Intel Xeon-E5630 at clock speed of 2.53GHz, connected using Infiniband.

All applications used for evaluation contain a sequence of parallel loops enclosed within one or more sequential time or convergence loops. The control flow and data access pattern for the parallel loops remains unchanged for every invocation within these outer sequential loops. Therefore, the inspector code generated for all the annotated parallel loops could be hoisted out of surrounding loops to amortize its overhead.

5.1 Mixed Irregular/Regular Computation

This benchmark solves the Poisson Equation over a rectangular domain. An example of a parallel loop from this benchmark was shown in Listing 1. It contains 4 parallel loops enclosed within an outer sequential time loop. The inspector code generated for the parallel loops could be hoisted outside of this loop. The problem size used for evaluation had 1024 coarse boxes and 2048 fine boxes. Each coarse box used a grid of size 128×128, while each fine box used a grid of size 126×252. Table 1 shows the running time for the executor and the total running time of the transformed code for 1000 timesteps. The generated executor shows good scaling with the inspector cost adding only a slight overhead. Since the inspector is parallel, the cost of the inspector reduces up to 128 processes.The vectorization report generated by ICC shows that loops that were vectorized previously are vectorized in the generated executor code too, indicating that the transformation scheme did not introduce artifacts that affect subsequent compiler optimizations. Consequently, the on-node performance of the executor could be further enhanced by using compile time transformations like [22, 41] that rely on the regular nature of target loops.

The code generated by the inspector/executor approach developed in [3, 35], fails to execute since the size of the traces generated exhausts the memory on a process. To get around this, the problem size used was reduced to $1/4^{th}$ of the above. While this code is now able to execute, the size of traces generated is still quite large. The execution times are shown in Figure 2. The lines labeled *Executor* and *Executor:trace* show the execution time of the executor generated by the proposed approach and the trace-bsaed approach of [35], respectively. The lines labeled *Total* and *Total:trace* show the total execution time of the generated inspector and executor code from the two approaches. All times are reported as speedup relative to the execution time of the original sequential code. For 2 and 4 processes, the executor code from the trace approach is 4 times slower than the executor code from the current approach. Due to high inspector overheads, the total execution time is 8 times slower. As the number of processes increase the size of the traces generated per process reduces and fit in some level of cache, resulting in reduced stress on the bandwidth to main memory. This improves the executor time of the trace based approach to be slightly better than the executor code generated using the present approach. This is because the former flattens indirect access, reducing the number of memory locations accessed before getting to the actual data. Since the present approach maintains the indirect access pattern of the original code for the irregular dimensions, the number of memory accesses is higher. The flattening though comes at a high inspector overhead effectively nullifying the benefit gained by parallelizing the computation. With the approach proposed in this paper there is virtually no inspector overhead. These results demonstrate that while the trace-based approach might be able to effectively handle small benchmarks with smaller footprints, the approach presented here is more scalable and can be used to effectively parallelize real-world applications.

5.2 Affine Computations

Affine computations are at one extreme of the range of applications modeled here. We evaluated the performance of the generated code when the input computation is completely affine. For bandwidth bound codes like FDTD and 2D Jacobi stencil [28], time tiling is an effective approach to increase the arithmetic intensity by increasing data reuse across iterations of the outer time loop. Tiling for concurrent start [8], generates time tiled code where the loop that iterates over tiles for a particular time tile are parallel, while the loop that iterates over time tiles is sequential. This approach to time tiling eliminates the load imbalance created by traditional schemes that use wavefront parallelism across tiles. This scheme has been implemented within Pluto [4] and the generated code, targeting a single-core CPU, was used as the input to the transformation scheme described in this paper. Being fully affine, the parallel loop was block partitioned across processes.

Starting from the same input code, Pluto itself can be used to generate distributed memory code. Table 2 compares the performance of the code generated by the present approach with the distributed memory code generated by Pluto [12, 17] itself. All arrays used were of size 8192×8192. The original untiled code executed 1000 timesteps and a tile size of 32 was used for each loop. For these examples, the communication pattern used by both these codes are similar resulting in comparable performance with linear scaling up to 256 processes. The representation used by the framework developed in this paper is similar to that used in polyhderal

FDTD: Sequential Time = 486.9s								
Nprocs	2	4	8	16	32	64	128	256
Executor(s)	248.1	124.6	63.0	31.7	16.4	8.6	4.7	2.8
Pluto(s)	253.1	128.3	64.5	32.6	16.6	8.8	4.9	2.9
Jacobi2D : Sequential Time = 750.5s								
Nprocs	2	4	8	16	32	64	128	256
Executor(s)	382.8	191.2	96.2	48.2	24.8	13.1	7.2	4.4
Pluto(s)	362.4	182.1	92.4	47.0	24.2	13.1	7.5	4.8

Table 2. Affine Computations: Time-tiled FDTD, Jacobi2D

Nprocs	2	4	8	16	32	64	128	256
Executor(s)	1466	733	372	188	115	55.1	30.5	15.6
Inspector(s)	2.90	1.93	1.32	0.84	0.67	0.61	0.74	0.95
Total(s)	1469	735	374	188	116	55.7	31.3	16.5

Table 3. 3D BTE Solver, Sequential Time = 2833.4s

compilers like Pluto. Consequently, sophisticated techniques developed for affine computations can be easily incorporated.

5.3 Irregular Computations

Finally, we evaluate the performance of the generated parallel code for computations that contain no affine parts. This application uses the Finite Volume Method to discretize and solve the Boltzmann Transport Equation (BTE) for phonons [25] used to model heat conduction in semiconductor materials over a 3D unstructured grid of tetrahedral cells. The computation proceeds by iterating over bands of phonon frequencies and discretized directions of the physical domain. A system of linear equations for the entire physical domain is solved for each band-direction pair. This is followed by an integration phase that combines data for a particular band. The loops that perform these computations are parallel and can be targeted for distributed memory execution. For transient problems, these steps are performed repeatedly within a time loop. Since this application is written in Fortran, the transformations described in Algorithms 1 and 2 were implemented manually. This application represents the other extreme of the range of computations modeled by the framework developed here.

For evaluation, we used an input grid of 2491 cells with 40 frequency bands and 40 discretized directions. Deep loop nests used in this computation result in the inspector generated by the trace-based approach exhausting the processor's local memory while building the trace of index expressions. The inspector code generated by the approach presented here avoids this by using the technique developed in Section 4, which created local indirection arrays on each process to mimic the behavior of indirection arrays of the original computation. Table 3 shows the execution times of the parallelized code for 10 timesteps. A linear scaling is achieved up to 256 processes with minimal inspector overheads. These results demonstrate that the techniques developed in this paper further enhance the scalability of inspector/executor approaches while targeting purely irregular applications as well by reducing the reliance on traces to recreate index expressions.

6. Related Work

The inspector/executor approach was pioneered by Saltz et al. They developed runtime infrastructure for distributed memory parallelization of irregular applications [11, 30, 39, 40]. These were augmented with compiler approaches that automatically generated parallel code [6, 15, 16, 46]. These approaches relied on the *Execute On Home* directives of HPF, that restricted the scope of applications that could be targeted. Ravishankar et al. [35] built on this to target a wider range of applications. Both of these approaches can be labeled as trace-based approaches, with the latter reducing the size of the trace by recognizing contiguous accesses within the input code. Section 5 clearly demonstrates that the framework developed here is more scalable than such trace-based approaches. The Sparse Polyhedral Framework [23, 43] also provides a unified framework to express affine and irregular parts of the code by representing indirection array access using *uninterpreted function symbols* (UFS). Still, transformations and code generation process within this framework requires asserting properties of these UFS (e.g., invertibility) at compile time, which is usually not possible. Indeed, we believe SPF can be made more effective by reasoning about slices of iteration domains that can be represented using affine inequalities and can be readily incorporated into it.

Basumallik et al. [9, 10] developed an OpenMP to MPI translator that executed OpenMP parallel loops on distributed memory architecture. Their approach relied on replication of data structures accessed using indirections. Such an approach is infeasible when these data structures are too large to fit in a single processor's memory. Additionally, the communication volume required to satisfy dependences was equal to the size of these replicated data structures. The approach developed here generates distributed memory code that doesn't replicate any data, with communication required only for ghost locations.

The Inspector/Executor approach was used by Rauchwerger and Padua [31] to analyze if irregular loops can be parallelized through array privatization and recognition of associative and commutative reduction operations. The same framework was extended in [33, 34] to capture dependences between iterations of a partially parallel loop at runtime and generate a schedule to execute these in wavefront fashion. August et al. [19, 27] developed compiler algorithms that could group instructions to form producer-consumer relationship between these groups, which are then executed in a pipelined fashion. Their recent work [20] could exploit cross-invocation parallelism in loops. Zhuang et al. [47] used speculation to execute independent iterations of irregular loops in parallel with dependences tracked at runtime. Parasol [32, 37, 38] generated run-time checks, evaluated in increasing order of overheads incurred, that determine if a loop is parallel or not. Both these approaches target shared memory systems. Liu et al. [24] used an inspector to deduce an optimal execution strategy for computational mechanics code.

Inspector/Executor approaches have also been used to generate code to target NVIDIA GPUs. Huo et al. [21] reordered computation to reduce synchronization costs for irregular reductions. This approach was targeted towards mesh-based applications and could handle a single-level of indirection. Anantpur et al. [7] used an inspector to group iterations into phases that could be executed concurrently on a GPU. Venkat et al. [44] developed extensions to the sparse polyhedral framework to develop optimizations aimed at GPUs.

7. Conclusion

In this paper we have developed a framework that effectively models computations that have a mix of affine and non-affine program regions. Purely affine and completely irregular codes form two extremes of the range of computations modeled within this framework. The use of affine techniques to handle the regular program regions removes artifacts from the generated executor that hamper subsequent compiler optimizations. It also addresses a major bottleneck in terms of memory usage of previous inspector/executor techniques for distributed memory parallelization of irregular computations. This improves the scalability of generated code. The formalism developed in this paper can be used to effectively integrate future developments in distributed memory code generation for both affine and irregular computations.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation through grants 0811457, 0904549, 0926127, 0926687, 1059417, 1321147 and 1404995, by the U.S. Department of Energy through grant DE-SC0008844, and by Louisiana State University.

References

[1] Cloog: Chunky loop generator. www.cloog.org/.

[2] Example of the generated inspector/executor code. http://tinyurl.com/m2mun6y.

[3] Inspector/executor compiler. http://tinyurl.com/kwhp453.

[4] Pluto - an automatic parallelizer and locality optimizer for multicores. pluto-compiler.sourceforge.net.

[5] ROSE compiler infrastructure. www.rosecompiler.org.

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

[7] J. Anantpur and R. Govindarajan. Runtime dependence computation and execution of loops on heterogeneous systems. In *CGO*, 2013.

[8] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *SC*, 2012.

[9] A. Basumallik and R. Eigenmann. Towards automatic translation of OpenMP to MPI. In *ICS*, 2005.

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

[11] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 1991.

[12] U. Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *SC*, 2013.

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

[14] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen. Chombo software package for AMR applications-design document, 2000.

[15] R. Das, J. Saltz, and R. von Hanxleden. Slicing analysis and indirect accesses to indirect arrays. *LCPC*, 1994.

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

[17] R. Dathathri, C. Reddy, T. Ramashekar, and U. Bondhugula. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *PACT*, 2013.

[18] G. Gupta and S. V. Rajopadhye. The Z-polyhedral model. In *PPOPP*, 2007.

[19] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled software pipelining creates parallelization opportunities. In *CGO*, 2010.

[20] J. Huang, T. B. Jablin, S. R. Beard, N. P. Johnson, and D. I. August. Automatically exploiting cross-invocation parallelism using runtime information. In *CGO*, 2013.

[21] X. Huo, V. Ravi, W. Ma, and G. Agrawal. An execution strategy and optimized runtime support for parallelizing irregular reductions on modern gpus. In *ICS*, 2011.

[22] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *PLDI*, 2013.

[23] A. LaMielle and M. Strout. Enabling code gen. with sparse polyhedral framework. Technical report, Colorado State University, 2010.

[24] C. Liu, M. H. Jamal, M. Kulkarni, A. Prakash, and V. Pai. Exploiting domain knowledge to optimize parallel computational mechanics codes. In *ICS*, 2013.

[25] A. Mittal and S. Mazumder. Hybrid discrete ordinatesspherical harmonics solution to the boltzmann transport equation for phonons for non-equilibrium heat conduction. *Journal of Computational Physics*, 2011.

[26] J. Neiplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The ARMCI approach. *Int. J. High Performance Computing Applications*, 2006.

[27] G. Ottoni, R. Rangan, A. Stoler, and D. August. Automatic thread extraction with decoupled software pipelining. In *MICRO*, 2005.

[28] Polybench. Polybench/c: the polyhedral benchmark suite. http://tinyurl.com/m7ztgex.

[29] PolyOpt. A polyhedral optimizer for the ROSE compiler. www.cs.ucla.edu/∼pouchet/software/polyopt/.

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

[31] L. Rauchwerger and D. Padua. The privatizing doall test : A run-time technique for doall loop identification and array privaization. In *ICS*, 1994.

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

[33] L. Rauchwerger, N. Amato, and D. Padua. Run-time methods for parallelizing partially parallel loops. In *ICS*, 1995.

[34] L. Rauchwerger, N. Amato, and D. Padua. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming*, 1995.

[35] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *SC*, 2012.

[36] C. Reddy and U. Bondhugula. Effective automatic computation placement and data allocation for parallelization of regular programs. In *ICS*, 2104.

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

[38] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: Static & dynamic memory reference analysis. In *ICS*, 2002.

[39] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *J. Parallel Distrib. Comput.*, 1990.

[40] S. Sharma, R. Ponnusamy, B. Moon, Y. shin Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *SC*, 1994.

[41] K. Stock, M. Kong, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *PLDI*, 2014.

[42] M. M. Strout and P. D. Hovland. Metrics and models for reordering transformations. In *MSP*, 2004.

[43] M. M. Strout, G. George, and C. Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *LCPC*, September 2012.

[44] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. . URL http://doi.acm.org/10.1145/2544137.2544141.

[45] S. Verdoolaege. ISL: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*, pages 299–302. Springer, 2010.

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

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