Distributed-Memory Code Generation for Irregular-Outer Regular-Inner Loop Computations

OSU Technical Report Number : OSU-CISRC-4/14-TR09. Date : April 23, 2014

Mahesh Ravishankar*, Roshan Dathathri*, Venmugil Elango*,Louis-Noel Pouchet†,
J. Ramanujam‡, Atanas Rountev*, P. Sadayappan*
*The Ohio State University, †University of California, Los Angeles, ‡Louisiana State University
Email: *{ravishankar.3,dathathri.2,elango.4,rountev.1,sadayappan.1}@osu.edu,
†pouchet@cs.ucla.edu,‡jxr@ece.lsu.edu

I. Introduction

Automatic parallelization of applications to target distributed memory systems remains a challenge for modern optimizing compilers. Recent developments in polyhedral compilation techniques [1]–[3] 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. [4], [5]. 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. Our recent work [6] built on this approach, while broadening the scope of targeted applications. While effective for non-affine computations, applying these techniques to affine codes results in unnecessary inspector overheads.

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) packages such as Chombo [7]. 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`. This makes the computation within loop `k` *irregular*. At the same time, loops `i` and `j` are *regular* since a standard stencil computation is used within them to update the value of `phi` at a grid point. Such a pattern of irregular-outer regular-inner is quite common in scientific applications.

```
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;
12   }
13 }
```

Listing 1: Example from AMR

```
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 [6]

Loop `k` in Listing 1 is parallel and its iterations can be executed across nodes of a distributed memory machine. Due to data dependent loop bounds and array access patterns, automatic parallelization using purely static analysis cannot be employed here. On the other hand, inspector/executor approaches developed previously do not effectively reason about parts of the code that are regular. For example, techniques developed in [6] 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, such approaches rely on expensive run-time analysis to gather information needed to execute the computation in parallel. For example, in [6], 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 from [6]. `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 can become expensive and can significantly reduce the performance of the inspector/executor approach. Further, the generated executor code does not maintain the structure of regular parts of the original code. This dramatically 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 [8]) can optimize the computation within loop `i` in Listing 1, while the corresponding loop in Listing 2 is outside the scope of such compilers. As a result, many opportunities for improving on-node performance are lost.

We propose a framework that effectively models irregular-outer regular-inner computations. This framework allows the code generator to reason statically about computation and data access patterns within regular parts of the code. The generated inspector combines this information with that obtained at run time for the irregular parts of the code to effectively partition the computation. Completely affine and completely irregular computations are naturally expressed as special instances of the framework. This general reasoning scheme is based on integer sets/maps/slices and has two important advantages. First, by employing static models for inner-regular regions, it reduces the inspector overheads by eliminating the need to generate traces for affine array access expressions. Further, the executor code maintains the structure of the regular parts of the computation, making it amenable to further optimizations. Compared to existing work, the proposed framework significantly advances the state of the art. Unlike polyhedral techniques for distributed memory parallelization of affine codes [1]–[3], it handles a much broader class of computations. Importantly, this generality comes “for free”: as we demonstrate experimentally, for purely affine codes our approach performs on-par with these state-of-the-art polyhedral techniques. Compared to modern inspector-executor techniques, exemplified by [6], our framework avoids a major bottleneck: the need to generate traces of access expressions for the inner-regular regions of the code. The benefits of this feature are evident in our experiments. In addition, existing approaches obscure the regular nature of these code regions. In contrast, we specifically aim to preserve this regular structure in order to enable further polyhedral optimizations of on-node code. In summary, the contributions of this work are as follows:

- A general framework that can effectively model irregular-outer regular-inner computations.

- Code generation to improve the inspector/executor scalability for distributed memory parallelization.
- Preservation of regular code regions in the generated code, to enable composability with existing polyhedral optimizations for improved on-node performance.
- Experimental evaluation on irregular-outer/regular-inner, completely regular, and completely irregular codes, demonstrating the advantages of our approach over existing techniques.

A. Compile time input from the user

Loops that are parallel are annotated with `#pragma parallel`, as shown in line 1 of Listing 1. Annotated loops cannot be nested within each other. It is assumed that loop bounds and array index expressions depend only on (1) iterator values, (2) values stored in read-only arrays, referred to as *indirection arrays*, and (3) scalars whose values are not modified within the loop. Loop `k` in Listing 1 can be modified to fit the last requirement by replacing all uses of scalars `fy`, `fx`, `cy` and `cx` with the expression used to evaluate them. Dependence between the iterations of an annotated loop can only be through associative and commutative reduction operations. Further, all loops are assumed to have unit increments.

II. Data Structures for Distributed Computations

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 the form 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 these sets, whether an inspector is needed to populate them or not, and even when an inspector is needed only for some parts of them.

A. 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) Integer Sets: A convenient way to represent the dynamic executions of a statement surrounded by `for` loops is to associate an integer point in a multi-dimensional space for each instance of the statement, 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 $\setminus$, 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 exists 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 conjunction of affine inequalities involving only the variables i_i and parameters (constants

```
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

whose value is not known at compile time), then the set is a polyhedron and all operations can be computed statically. When the set is defined using disjunctions of affine inequalities then it is a union of convex polyhedra. When the set is the intersection of a polyhedron and an integer affine lattice then it is a $\mathcal{Z}$-polyhedron [9], which can also be computed statically using the Integer Set Library [10]. When it involves 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 [11] using uninterpreted functions, or using an inspector. When the set is arbitrary, an inspector is needed.

2) Set Slicing: One key operation to reason about distributing a computation is *slicing*, that is, taking a particular subset of a set. Intuitively, 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 purpose. For example, the iteration domain for the statement `S` in Listing 3, i.e., the set of dynamic instances of `S` can be written:

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

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:

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

This set now necessarily models a single point (e.g., a single loop iteration) along the first dimension. To model two different iterations of the outer loop, one can simply introduce 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:

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

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 of subsets of loop iterations (typically arising from the iterations of the partitionable loop(s) to be executed on a processor), a key property on the computability of these subsets has arisen: *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, noted $lb[p_1]$ and $lb[p_2]$ for instance, 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 for instance 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 using extensively unions of convex sets.

3) Integer Maps: The second data structure used associates, or maps, integer points to other integer points. A typical use is 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:*

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

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

In a manner similar to integer sets, the tractability of $\mathcal{M}$ depends on the form of the function M. If M is a matrix of integer coefficients (e.g., an affine function), and is applied to a polyhedral set, then the output of the map (that is, the set of points which are the image of the input set by the map) can be computed statically, as a union of polyhedral sets. For example, consider the map representing the relationship between the iteration space and the data space for the reference $A[j]$, expressed as $\mathcal{D}_A : \{(i, j) \rightarrow (o_1) \mid o_1 = j\}$. 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 for instance the CLooG polyhedral code generator [12].

The map for the reference $B[col[j]]$ is $\mathcal{D}_B : \{(i, j) \rightarrow (o_1) \mid o_1 = col[j]\}$. Here, since $col[j]$ is not an affine function, its value for different j is not known 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. We remark that in our formalism, we will express the data spaces of each reference to model the data that needs to be communicated, and determine the need of an inspector by simply observing if the sets and maps of interest are polyhedral sets and affine maps. If not, then we use an inspector to compute them.

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 $F_A : \bigcup_{i=1}^{n} \mathcal{D}_A^i(\mathcal{S}_{\mathcal{D}_A^i})$.*

B. 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.

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 [13].

First, we construct a slice of the original iteration domain, $\mathcal{S}_{I_j}$ by computing an intersection of the original iteration space $\mathcal{S}$ with a set I_p. In set I_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. The outermost dimension is always set as irregular. If the number of such dimensions is r, the set I_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 i of the irregular loop(s), with the property that $\vec{p}_i \neq \vec{p}_j$ for $i \neq j$. The original complete iteration space is therefore the union of all slices,

$$\mathcal{S} = \bigcup_{\vec{p} \in \mathcal{C}} \mathcal{S}_{I_{\vec{p}}} \tag{2}$$

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

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

The set $\mathcal{C}^q$ is constructed from K^q by taking one distinct 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 and the outermost loop is block partitioned then the set K^q is known at compile-time, and the set $\mathcal{C}^q$ is reduced to a single parameter $\mathcal{C}^q = \{\vec{p} : b_1 \leq p^1 < b_1 + B\}$ where b_1 is a newly introduced parameter and B is the block size. Consequently the slice $I_{\vec{p}}$ will contain B iterations, and the above union can be fully computed statically.

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,

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

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.

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

For affine computations, when the outer loop is block-partitioned, $\mathcal{C}^q$ can be defined as $\mathcal{C}^q = \{p_1 : b_1 \leq p^1 < b_1 + B\}$. Consequently, F_A^q can also be computed statically.

In general, the same data element might be accessed by iteration mapped to two 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 the all 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 [6]. 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:

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

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

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

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

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

- Sent to process q' if A is read within the partitioned loop
- Received from process q' if A is written within 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 $O_A^{q',q}$.

III. Syntactic Structure of Target Loops

⟨*Start*⟩ ::= ⟨*Loop*⟩

⟨*Loop*⟩ ::= '`for`' '`(`' ⟨*Iterator*⟩ '`=`' ⟨*IExpr*⟩ '`;`' ⟨*Iterator*⟩ '`<`' ⟨*IExpr*⟩ '`;`' ⟨*Iterator*⟩ '`++`' '`)`' ⟨*ElementList*⟩

⟨*Elementlist*⟩ ::= ⟨*Element*⟩ | '`{`' ⟨*ElementList*⟩ ⟨*Element*⟩ '`}`'

⟨*Element*⟩ ::= ⟨*IAssignment*⟩ | ⟨*DAssignment*⟩ | ⟨*Loop*⟩ | ⟨*If*⟩

⟨*If*⟩ ::= '`if`' '`(`' ⟨*IExpr*⟩ '`)`' ⟨*ElementList*⟩ '`else`' ⟨*ElementList*⟩

⟨*IAssignment*⟩ ::= ⟨*IScalar*⟩ = ⟨*IExpr*⟩ '`;`'

⟨*DAssignment*⟩ ::= ⟨*DScalar*⟩ ⟨*AssignOp*⟩ ⟨*DExpr*⟩ '`;`'

⟨*IExpr*⟩ ::= Affine expressions of ⟨*BasicIExpr*⟩

⟨*DExpr*⟩ ::= Side-effect-free expressions of ⟨*BasicDExpr*⟩

⟨*BasicIExpr*⟩ ::= ⟨*IScalar*⟩ | ⟨*Iterator*⟩ | ⟨*IArray*⟩ '`[`' ⟨*IExpr*⟩ '`]`'

⟨*BasicDExpr*⟩ ::= ⟨*DScalar*⟩ | ⟨*DArray*⟩ '`[`' ⟨*IExpr*⟩ '`]`'

⟨*AssignOp*⟩ ::= '`=`' | '`+=`' | '`*=`' | ...

Fig. 1: Syntactic structure of annotated loops

Here we specify the syntactic and semantic properties of annotated parallel loops. The code within the annotated loop should be defined by the nonterminal ⟨Loop⟩. This grammar is the same as that targeted by [6]. ⟨IArray⟩s correspond to the read-only indirection arrays used in the computation, and are used only to determine the control flow and array access patterns. The ⟨DArray⟩s correspond to arrays that hold meaningful information used or updated by the code. ⟨IScalar⟩s are scalars whose values are used in loop bounds conditionals and array access expressions. All loops have unit increments. All loop bounds are assumed to be invariant with respect to the body of the loop.

⟨IAssignment⟩ statements are commonly used as place holders for expressions used in index expression and loop-bounds, for examples, statements that assign to `fID`, `cID`, `fy`, `fx`, `cy` and `cx` in Listing 1. To make subsequent compiler analysis simpler, a pre-processing step removes as many ⟨IAssignment⟩ statements as possible. All ⟨IExpr⟩s within the annotated loop are analyzed. For every ⟨IScalar⟩ used in them, the set of reaching definitions of this scalar is obtained. All reaching definitions are necessarily from ⟨IAssignment⟩s. If there is only a single reaching definition, then the current reference is replaced with the expression on the right-hand side of that ⟨IAssignment⟩. Once all ⟨IExpr⟩s have been analyzed, dead-code elimination can be used to remove all ⟨IAssignment⟩s whose definitions are not used. This process is

```
1 for( k = 0; k < nf; k++ ){
2   for(i=start_y[fine_boxes[k]]/2; i< end_y[fine_boxes[k]]/2; i++)
3     for(j=start_x[fine_boxes[k]]/2; j< end_x[fine_boxes[k]]/2; j++){
4       phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]] =
5         ( phi[fine_boxes[k]][2*i-start_y[fine_boxes[k]]][2*j-start_x[fine_boxes[k]]]
            + phi[fine_boxes[k]][2*i-start_y[fine_boxes[k]]+1][2*j-start_x[fine_boxes[k]]]
            + phi[fine_boxes[k]][2*i-start_y[fine_boxes[k]]][2*j-start_x[fine_boxes[k]]+1]
            + phi[fine_boxes[k]][2*i-start_y[fine_boxes[k]]+1][2*j-start_x[fine_boxes[k]]+1]
            ) / 4.0;
6   }
7 }
```

Listing 4: After pre-processing Listing 1

continued till no more ⟨IAssignment⟩s are removed. Listing 4 shows the result of this pre-precessing step on Listing 1.

Following the above pre-processing, loops within the annotated parallel loop are marked as being *regular* or *irregular*. The computation within regular loops are model using sets and maps where constraints used in their definitions are affine. A loop is marked irregular if

- its loop body contains an ⟨IAssignment⟩ statement.
- its loop body contains an ⟨If⟩ statement.
- its iterator is used in ⟨IArray⟩[⟨IExpr⟩].
- any loop nested within it is marked as irregular.

By construction, computation within regular loops can be expressed using affine constraints since all expressions of the form ⟨IArray⟩[⟨IExpr⟩] can be treated as parameters for a given value of the iterators of the outer irregular loops. All ⟨IScalar⟩s within loops that are regular can also be treated as parameters.

IV. Generation of Inspector and Executor Code

Once the iteration space has been partitioned by distributing its slices amongst processes (Section II-B1), to partition the data, the data space for each array using Eq (5) has to be computed. This data space represents the local array on each node and is computed by an inspector. 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.

A. Local Data Space of Arrays : Inspector Code

Algorithm 1 describes the generation of the inspector code. For a given annotated loop AST, function *FindIrregularDimensions* marks a loop as being irregular if its value is used in index expressions of indirection array accesses. Listing 5 shows the inspector code generated for Listing 1 using Algorithm 1. Since we target computations that are irregular-outer regular-inner, if a particular loop is irregular, all its surrounding loops are also marked as irregular. The annotated loop is marked as irregular, unless the computation within the loop is affine and its partitioning is decided statically. No inspector is needed in such cases. Presence of one or more irregular loops, the iteration space slices mapped to a process are known only at runtime. While the data space for a single slice can be computed statically using Eq (4), the union of these slices is evaluated at runtime by the inspector.

The AST of the inspector code is constructed by first replicating the AST of the annotated loop. At line 6 of Algorithm 1, the loop body of the outermost loop is enclosed within a conditional that executes only those iteration that belong to set K^q. All indirection array accesses are invariant with respect to the loops that constitute the regular portions of the computation. Therefore, the value of all such expressions can be stored in temporary variables just before the outermost loop that corresponds to a regular dimension

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

```
   Input  : A : AST of the annotated parallel loop
   Output: A_I : AST of the inspector
1  begin
2    [N,R] = FindIrregularDimensions(A) ;
3    A_I = φ ;
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      forall the 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       forall the e ∈ ArrayIndexExpression(a,A_I) do
18         o = e.OuterDimension ;
19         if ¬IsOfDesiredForm(o) then
20           InsertCodeToCreateTrace(o) ;
21         else if IsNonAffineIterator(o) then
22           InsertCodeToTraceContiguousAccess(o) ;
23     RemoveRegularLoopsAndStatements(A_I,R) ;
24     GenerateGaurdsForIndirectionArrayAccesses(A_I) ;
25   return A_I ;
```

of the iteration space. Line 7 inserts such statements into the inspector AST and replaces indirection array accesses with the corresponding temporary variable. Lines 4-11 of Listing 5 shows these assignment statements. The point in the inspector AST immediately after these statements enumerates all elements of $\vec{p} \in \mathcal{C}^q$, and can therefore analyze all iteration space slices mapped to a process.

The iteration space slice representing the regular portion of the AST can now be expressed using affine constraints and is computed at line 8. For Listing 1, a slice would be,

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

For each array, the integer map representing accesses to it is built at line 10. For a particular 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 result represents a slice of the data space for an array (Eq 4). For example,

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

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.

For large multi-dimensional arrays, computing an exact union of all elements accessed is very expensive. This cost can be reduced by recognizing that accesses to such arrays also follow the irregular-outer regular-inner pattern. For example, the access `phi[cID][cy][cx]` in Listing 1 results in the outer dimension being

```
1  for( k = 0 ; k < nf ; k++ )
2    if( get_home(loop_k) == myid) {
3
4      t1 = get_elem(id_start_y,get_elem(id_fine_boxes,k)); //t1=start_y[fine_boxes[k]];
5      t2 = get_elem(id_start_x,get_elem(id_fine_boxes,k)); //t2=start_x[fine_boxes[k]];
6      t3 = get_elem(id_end_y,get_elem(id_fine_boxes,k)); //t3=end_y[fine_boxes[k]];
7      t4 = get_elem(id_end_x,get_elem(id_fine_boxes,k)); //t4=end_x[fine_boxes[k]];
8      t5 = get_elem(id_start_y,get_elem(id_ftoc,k)); //t5=start_y[ftoc[k]];
9      t6 = get_elem(id_start_x,get_elem(id_ftoc,k)); //t6=start_x[ftoc[k]];
10     t7 = get_elem(id_fine_boxes,k); //t7=fine_boxes[k];
11     t8 = get_elem(id_ftoc,k); //t8=ftoc[k];
12
13     ///Record access to outer-most dimension for phi[cID][cy][cx]
14     p_outer = t8;
15     update_access(id_phi,p_outer);
16     ///Record Lexmin/Lexmax for the inner dimensions
17     lexmin_dim1 = -t5 + t1;
18     lexmin_dim2 = -t6 + t2;
19     lexmax_dim1 = -t5 + t3 - 1;
20     lexmax_dim2 = -t6 + t2;
21     update_lexmin(id_phi,p_outer,lexmin_dim1,lexmin_dim2);
22     update_lexmax(id_phi,p_outer,lexmax,dim1,lexmax_dim2);
23     ....
24   }
```

Listing 5: Inspector Code Snipper for Listing 1

accessed using indirections, but for a given iteration of loop `k`, a rectangular patch of the inner dimensions of the array is accessed (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 outer most indices of arrays 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 union is drastically reduced.

To use this approach, the inner dimensions of the data space slice computed at line 11 is modified to parameterize the outer dimension (line 14). This can be done applying the following map to the data space slice

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

This expression now computes the set of indices of inner dimensions accessed for every outer dimension index, p_o of the array. The lexicographic minimum and maximum of the resulting expression is computed statically at line 15. The inspector employs the bounding box approach for the inner dimensions by computing the minimum of all lexicographic minimums and maximum of all lexicographic maximums for every outer dimension index of the array accessed on a process. Lines 17-20 shows the inspector code that computes the lexicographic minimum and maximum for each inner dimension for the access `phi[cID][cy][cx]` in Listing 1. This is then used to compute the bounding box at lines 21-22 of the inspector code in Listing 5. The set of these outer dimension indices can be computed by projecting out the inner dimensions of the data space slice computed at line 11. Line 16 inserts code to compute the exact union of such sets for all data space slices on a process. Lines 14-15 correspond to the code generated to record the outer-dimension index accessed using `phi[cID][cy][cx]`. The code for other array access expressions is now shown in Listing 8 for brevity.

If an index expression used to access 1D arrays, or the outer most dimension of multi-dimensional arrays, are of the form,

$$\langle \text{MapExp} \rangle ::= \langle \text{Array} \rangle \text{`['} \langle \text{MapExpr} \rangle \text{`]'} \mid \langle \text{Iterator} \rangle \tag{9}$$

Section V describes an approach that allows the generated code to recreate the accesses in a manner consistent with the original computation, without having to generate a trace of such index expressions. If not of this form, the inspector code is modified to generate a trace of this index expression similar to the approach described in [6]. Note that the inspector will never have to trace values of inner dimension index expressions of multi-dimensional arrays, significantly reducing the size of traces generated. Since the regular parts of the computation have been modeled statically they can be removed from the inspector (line 23). 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.

1) Prefetching Indirection Array Values not in the Local Memory: Since the developed approach is targeted towards applications where a single node does not have enough memory to replicate any of the data structures, all arrays (including indirection arrays) are assumed to be initially block-partitioned across processes. 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.

For every indirection array, the inspector maintains a list of elements whose values are known on that process. This list initially contains the block-partitioned portion of the array in the processor's local memory. Statements that use indirection array values are enclosed within conditional statements which evaluates to true if that value is known on the process. The false branch of the conditional flags this element as unknown and skips the execution of the statement. Nested conditional statements are generated if the expression used to index into the indirection array itself uses values stored in other indirection arrays. All indirection array references are replaced with a function call that retrieves the value of the accessed element. Loop statements whose bounds depend on values in indirection arrays are handled similarly. The conditional statement generated around these loops skips the loop execution if the value of any indirection array element used is not known on a process. After executing all iterations of the outer loop, if any elements were flagged as unknown, their values are fetched from the processes that initially stored it. This also implies that the inspector analysis was incomplete. Therefore, the inspector code has to be re-executed until no array element is flagged as unknown. A more detailed description of the algorithm can be found in [6]. The example in Listing 5 does not show this for brevity.

Once the data space of all arrays has been computed, the sets $O_*^{q,q'}$ and $G_*^{q,q'}$ on each process needs to be computed for all of them, as described in Section II-B2.

B. Executing the Iteration Space Slices : The executor code

```
1  for( k = 0 ; k < nf ; k++ ){
2    t1 = start_y_l[fine_boxes_l[k]];
3    t2 = start_x_l[fine_boxes_l[k]];
4    t3 = end_y_l[fine_boxes_l[k]];
5    t4 = end_x_l[fine_boxes_l[k]];
6    t5 = start_y_l[ftoc_l[k]];
7    t6 = start_x_l[ftoc_l[k]];
8    t7 = fine_boxes_l[k];
9    t8 = ftoc_l[k];
10
11   lexmin_dim1 = lexmin_phi_dim1[t8];
12   lexmin_dim2 = lexmin_phi_dim2[t8];
13   for( i = t1/2 ; i < t3/2 ; i++ )
14     for( j = t2/2 ; j < t4/2 ; j++ )
15       phi[t8][i-t5-lexmin_dim1][j-t6-lexmin_dim2] =
16         ( phi[t7][i-t1-lexmin_dim1][j-t2-lexmin_dim2]
17           + phi[t7][i+1-t1-lexmin_dim1][j-t2-lexmin_dim2]
18           + phi[t7][i-t1-lexmin_dim1][j+1-t2-lexmin_dim2]
19           + phi[t7][i+1-t1-lexmin_dim1][j+1-t2-lexmin_dim2]) / 4.0;
20 }
```

Listing 6: Executor Code for Listing 1

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

```
   Input  : A : AST of the annotated parallel loop
   Output : A_E : AST of the executor
1  begin
2    [N,A] = FindIrregularDimensions(A);
3    A_E = MakeCopy(A);
4    P = InsertTemporaryVariables(A_E,N,A);
5    I = ComputeAffineIterationSpace(A_E,N,A,P);
6    A_new = GenerateLoopNests(I);
7    ReplaceWithLocalArrayReferences(A_E);
8    if N = φ then
9      PartitionIterationSpace(A_new); ReplaceLoops(A_E,A,A_new);
10     forall the a ∈ Arrays(A) do
11       D_a = ComputeAccessMap(A_E,N,A,P,a);
12       F_a = ComputeImage(D_a,I); L_min = ComputeLexMins(F_a);
13       forall the i ∈ ArrayIndexExpression(a,A_E) do
14         i_new = NewSubtractExpression(i,L_min);
15         ReplaceExpression(i,i_new);
16   else
17     ReplaceLoops(A_E,A,A_new);
18     ReplaceOuterLoopBounds(A_E);
19     forall the a ∈ Arrays(A) do
20       forall the e ∈ ArrayIndexExpression(a,A_E) do
21         o = e.OuterDimension;
22         if ¬IsOfDesiredForm(o) then
23           InsertCodeToReadTrace(o);
24         else if IsNonAffineIterator(o) then
25           ModifyContiguousAccess(o);
26         if IsMultiDimensional(a) then
27           i = e.InnerExpressions;
28           l_min = GetLexminValueExpression(a,o,i);
29           i_new = NewSubtractExpression(i,l_min);
30           ReplaceExpression(i,i_new);
31   return A_E;
```

Algorithm 2 shows the steps involved in generating the executor code. Listing 6 shows the generated executor code when Listing 1 is used as input. The generated code executes the iteration space slices mapped to each process in the same order as the original computation. Similar to Algorithm 1, the executor AST is initialized to be a copy of the original AST. Since the iteration space slices are described using affine inequalities, polyhedral code-generation tools like CLooG [12] can be used to generate the code for a particular slice (line 6). This allows the manipulation of the computation represented by each slice using transformations previously developed within the polyhedral model like [14], [15]. While currently no such transformations have been implemented in our framework, this approach allows the easy integration of these approaches. The CLooG generated code is used to replace the loop-nest corresponding to the regular parts of the input AST. Lines 13-19 shows the regular parts of the executor code. It is evident that its structure is similar to the regular parts of the original code in Listing 1.

For fully affine computations, since 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. Since the data space of all arrays on a process can also be computed statically, this is used to allocate the local arrays. All array access expressions are modified to refer to these local arrays. In Listing 6, `phi_l` is the local array corresponding to array `phi` in Listing 1. A similar naming convention is used for all arrays used. The index expressions are replaced by the original expressions subtracted with the lexicographically smallest index of the array accessed on a process. Lines 9-15 correspond to the

code-generation scheme for purely affine computations with static outer loop partitioning.

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 computed by the inspector. Since the original value of the iterator is needed within the loop body, the inspector creates a temporary array, `local_k`, that stores the elements of K^q in increasing order. All references to the outer-most loop iterator, `k`, are replaced with `local_k[k]`. Section V-C describes cases where this temporary 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 form ⟨MapExpr⟩, Section V-C describes the corresponding index expression to be used in the executor code. For array access expressions that are not of this form, the index expression is modified to read from the trace generated by the inspector [6]. For inner dimensions of expressions used to access multidimensional arrays, the expression in the executor is obtained by subtracting the lexicographic minimum of the inner dimensions accessed for the given index of the outermost dimension on that process. In Listing 6, the array `lexmin_phi_dim1` and `lexmin_phi_dim2` store the lexicographic minimum index for each outer dimension index accessed on a process for the array `phi`.

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.

V. Generating Array Index expressions for the Executor Code

The index expressions in the executor code generated at the end of Section IV-B have to be modified to access elements of local arrays in a manner consistent with the original computation. In our previous work [6], this was done by first using an inspector to generate a trace of the different index expressions used. The executor then read this trace to access the elements of the local arrays. As mentioned in Section I, this approach would resulted in large traces for computations with deep loop-nests, like the one used for evaluation purposes in Section VI-C1. Apart from increasing the inspector overheads, using this approach exhausted the memory on a node while storing the traces of index expressions used in that application.

The need for generating such traces can be avoided by creating local indirection arrays that mimic the behaviour 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 levels of refinement to colocated boxes at a coarser level of refinement. The outer loop `k` iterates over the fine boxes in the computation and uses the array `ftoc` to locate the corresponding coarse box. Similarly, in unstructured grid computations, indirection arrays represent a map from a face to the two cells adjacent to it, and so on. Further, the same map could be used to access multiple arrays, all of which contain data associated with a particular entity. In Listing 2, the array `fine_boxes` is used to access elements of `start_y`, `end_y`, `start_x` and `end_x`, all of which contain information associated with a particular box of the computation. Multiple levels of indirection represent a composition of such maps.

Partitioning the computation, results in data related to a subset of the different entities of the original computation being accessed on each process. For example, partitioning the computation in Listing 1, results in each process accessing information associated with a subset of all boxes used in the computation. As a result, the arrays `start_y`, `end_y`, `start_x` and `end_x` have similar data space on a process and their elements are also laid out in the local memory in a similar way. A local indirection array, say `fine_boxes_l`, could be created to access the local versions of all these arrays in the executor code. In effect, these local indirection arrays encode the same mapping as the original code, but in terms of a local numbering of entities on a process.

Use of indirection arrays as maps result in array index expressions being of form described by the non-terminal ⟨MapExpr⟩ defined in Equation (9). ⟨Array⟩s used in ⟨MapExpr⟩ in the original code can be replaced with local indirection arrays in the executor code. The creation of such local indirection arrays

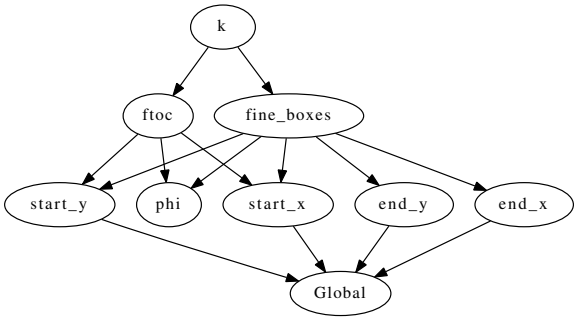

Fig. 2: Indirection Array Access Graph

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 done by the inspector code generated in Section IV-A; (2) Change the values stored in the local indirection arrays to point to the corresponding local positions of elements of the target array accessed. This process is described in Section V-A.

A. Access Graphs

Information about arrays that are accessed by an indirection array, and the different levels of indirection used in the original code is represented using a Directed Acyclic Graph (DAG), called an *access graph*. Nodes in this graph either represent arrays or non-affine iterators. For every array access expressions of the form $\langle$Array$\rangle$[$\langle$Iterator$\rangle$], an edge is added from the node representing the iterator, and the node representing the array. By construction, $\langle$Iterator$\rangle$s used in this context are non-affine iterators. 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 $\langle$Array$\rangle$[$\langle$MapExpr$\rangle$]s might also appear in loop bounds and inner dimensions of multi-dimensional arrays. $\langle$Array$\rangle$s used here are also indirection arrays, but their values are used as is in the executor. To capture such a uses of an indirection array an edge is added from the node representing the $\langle$Array$\rangle$ to a special node, *global*. This node can have has no successors. Figure 2 represent the access graph built for the computation in Listing 1. In cases where the graph has cycles, edges are removed from the graph till no cycle exists. The expressions corresponding to the removed edges are recreated in the executor using the trace approach developed in [6]. However, we havent come across computations where such a scenario exists.

All indirection arrays of 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 if the data space computed by Equation (5) for all of them are the same. This can be enforced by setting the local data space of all these arrays to be equal to the union of all the individual data spaces. As explained previously, in many scientific computing applications, the arrays accessed using the same indirection array store information regarding a particular entitiy of the computation, resulting in similar data spaces for these arrays on a process. The modified data space would at most represent only a slight over-estimation of the data space for individual arrays. The decision about which arrays need to have identical data spaces can be made by grouping nodes that represent arrays in the access graph into disjoint sets, such that immediate successors of an array node belong to the same set.

A trivial solution would be to add all array nodes to the same set. This would lead to a significant over-estimation of the data space on each process. To avoid this trivial solution, the 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 arrays whose values are not modified in the executor. Since 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.

Algorithm 3: GroupNodes(G)

```
   Input  : G = (V, E) : An access graph with group number of all nodes set to −1
   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 ;
```

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

1) All immediate successors of an array node have to 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.

In general, it might not be possible to create such groups for all possible access graphs, but modifications can be made to an input graph such that these properties can be enforced. For nodes in the graph that have multiple immediate successors, one of which is the global node, a new node is added with the same predecessors. The global node is added as an immediate successor of the new node, and is removed as a successor of the original node. This operation represents duplication of the corresponding indirection array in the executor, with the values of one copy modified to access the elements of local arrays, while values of the other copy remains unchanged. A similar strategy is used when all immediate successors of a node cannot be added to the same set. If successors of a node belong to two separate groups, values of one copy of the indirection array can be modified to access elements of one group and while values of the other copy modified to access elements of the other group.

B. 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 to change to.

Algorithm 4: ChangeGroupNum(v,group,processed_set)

```
Input  : v : Node in the graph
         group : 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 = group ;
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 ≠ group 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 = group ;
17   return True ;
```

In function *ChangeGroupNum*, if the input node v, is the *global node* the function returns *False* to signify it couldnt be added to the group. 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 *Unknown* if any of them are also assigned to the same group. Otherwise, the node v can be assigned to the group if all the immediate successor of the previously visited immediate predecessors are assigned to the same group. This is checked recursively.

If the call to *ChangeGroupNum* at line 12 of Algorithm 3 returns *False*, it implies that current successor, s, cannot be added to the same group as the previous successors. It is removed as an immediate successor of node v being analysed, and added as an immediate successor to a copy of the node. 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.

C. Modifying index expressions in the executor code

Once the grouping of nodes is complete, the index expressions in the executor code generated in Section IV-B are modified as follows. For ⟨MapExpr⟩s used in index expressions which are of the form ⟨Array⟩[⟨MapExpr⟩], the ⟨Array⟩ is replaced to refer to one of the copies of the corresponding local indirection array, as decided by the grouping algorithm. The values in the local indirection arrays are modified to point to corresponding locations of the local target array. ⟨MapExpr⟩s of the same form occuring in loop bound expressions and inner index expressions of multidimensional arrays are replaced with local array references without changing its values.

Since iterators of inner loops assume the same value as the original computation, array index expressions of the form ⟨Iterator⟩ which use inner loop iterators have to be manipulated to point to local elements of the array. The method used in [6] to recreate unit-stride accesses can be adapted for this purpose. Since loops in the targeted codes have unit increments, for every invocation of the inner loops, the index expression ⟨Iterator⟩ evaluates to a contiguous sequence of values. A local array is created to record the

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 I: Irregular-Outer Regular-Inner benchmark : Poisson equation solver. Sequential Time = 1055s

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 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 [6]. The same sequence can be used to access all arrays which are immediate successors of the node corresponding to the iterator in the access graph, and belong to the same group.

Finally, for array access expressions of the form $\langle$Array$\rangle$[$\langle$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 $\langle$Array$\rangle_1$[$\langle$Array$\rangle_2$[$\langle$Iterator$\rangle$]]. $\langle$Array$\rangle_2$ represents a temporary array that contained the original values of the iterator mapped to a process (K^q of Equation (3)). In the access graph, the node corresponding to the iterator, say p, would have only one immediate successor, say l, before the grouping algorithm (Algorithm 3) is employed. If, any successor of this node, say m, has no other predecessors then it can be concluded that every iteration of the outer loop accesses only one element of the array represented by node m. Since the 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 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 $\langle$Array$\rangle$[$\langle$Iterator$\rangle$], where $\langle$Array$\rangle$ refers to the corresponding local array. The temporary array added could be eliminated if the node l has no successors after this modification.

VI. Case Studies

We evaluated the performance of the generated distributed memory code for computations that are (1) Irregular-outer Regular-Inner, (2) Completely Regular, and (3) Completely Irregular. The code generator was implemented as a source-to-source transformation within ROSE [16] to target C-codes. MVAPICH2-1.9 was used for communication between processes. To reduce the cost of communication, one-sided communication APIs provided by ARMCI/GA-5.2 [17] were used for the 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-E5360 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.

A. Irregular-Outer Regular-Inner Computations

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 I 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

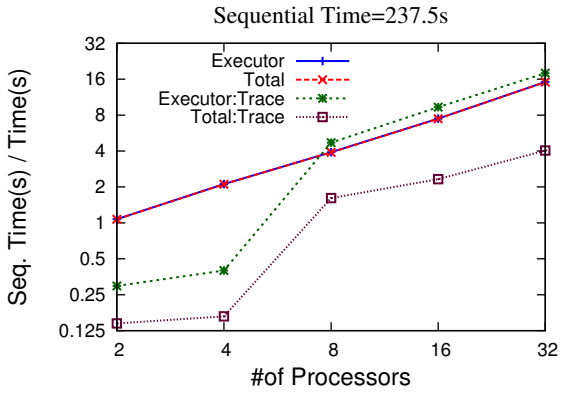

Fig. 3: Comparison with [6]

parallel, the cost of the inspector reduces upto 128 processes. There is an abrupt increase in the inspector cost at 256 processes due to synchronization overhead while prefetching values of indirection arrays. 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. The on-node performance of the executor could be further enhanced by using transformations [18] that maximize reuse of data for stencil computations used in the regular parts of this code.

The code generated by the previous inspector/executor approach developed by us [6] 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 now this code is able to execute, the size of traces generated are still quite large. 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 traces generated per process becomes smaller and fit in some level of cache, resulting in lesser stress on the bandwidth to main memory. This improves the executor time of the trace approach, but the inspector costs still remain high. Consequently, the execution time is on average 3 times slower than the approach presented in this paper. These results show that the code generated by the current approach handles irregular-outer regular-inner more efficiently and is more scalable than previous approaches that are tailored for irregular computations.

B. 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 [19], 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 [20], 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 [21] and the generated code was used as the input to the transformation scheme described in this paper. Being fully affine, the parallel loop was block partitioned across processes. 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.

Table II compares the performance of the generated executor with the distributed memory code generated by Pluto [1], [2]. The communication pattern used by both these codes are similar resulting in comparable performance with linear scaling upto 256 processes. These results highlight that the framework developed in this paper is as adept at handling affine computations as state-of-the-art techniques for distributed

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 II: Affine Computations: Time-tiled FDTD and 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 III: Irregular Application : 3D BTE Solver, Sequential Time = 2833.4s

memory parallelization developed specifically for affine codes. Further, the Pluto generated code used for comparison replicates the data on all the processes. This is a fundamental limitation since it cannot handle problem sizes which do not fit in a single processors memory. Our approach doesn't suffer from this since all data is partitioned across nodes. This limitation in Pluto has been addressed recently [3].

C. Irregular Computations

Finally, we evaluate the performance of the generated parallel code for computations that contain no affine parts.

1) Boltzmann Transport Equation for Phonons: This application uses the Finite Volume Method to discretize and solve the Boltzmann Transport Equation (BTE) for phonons [22] 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 V, which created local indirection arrays on each process to mimic the behavior of indirection arrays of the original computation. Table III shows the execution times of the parallelized code for 10 timesteps. A linear scaling is achieved upto 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.

2) 183.equake: This is a benchmark from the SPEC2000 benchmark suite which simulates seismic wave propogation over a 3D physical domain discretized using tetrahedal elements. The *ref* input size provided by the benchmark suite was used. Figure 4 shows the performance of the generated executor code and the total time of execution including the inspector overheads. The performance obtained from using the trace based approach in [6] is also shown for comparison. The executor code generated from

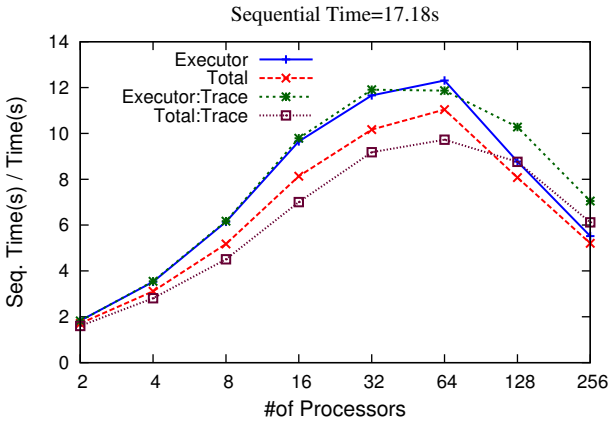

Fig. 4: 183.equake from SPEC 2000

both approaches have similar performance. Due to the relatively small problem size, the size of the traces of index expressions is too small to adversely affect the execution times. Additionally, these traces are accessed in a streaming fashion, allowing hardware prefetchers to optimize these accesses. On the other hand, there is a significant reduction in the inspector overheads from the approach proposed in this paper since there is no need to generate the trace of index expressions. Consequently, the overall execution time of the generated code is improved.

VII. Related Work

The inspector/executor approach was pioneered by Saltz et.al. They developed runtime infrastructure for distributed memory parallelization of irregular applications [23]–[26]. These were augmented with compiler approaches that automatically generated parallel code [4], [5], [27], [28]. These approaches relied on the *Execute On Home* directives of HPF, that restricted the scope of applications that could be targeted. Our previous work [6] 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 VI clearly demonstrates that the framework developed here is more scalable than such trace-based approaches. The Sparse Polyhedral Framework [11], [29] 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. [30], [31] 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 processors 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.

Inspector/Executor approaches have also been used in domain specific contexts. Liu et.al. [32] 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. [33] 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. [34] used an inspector to group into phases that could be executed dependence free on a GPU. August et.al. [35], [36] developed compiler algorithms that could group instructions to form producer-consumer relationship

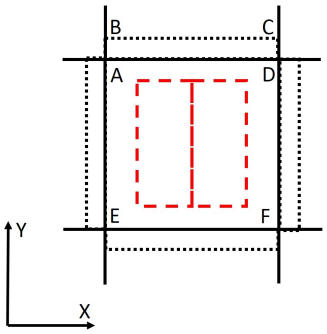

Fig. 5: Coarse and Fine Boxes used in AMR

between these groups, which are then executed in a pipelined fashion. Their recent work [37] could exploit cross-invocation parallelism in loops. Zhuang et.al. [38] used speculation to execute independent iterations of irregular loops in parallel with dependences tracked at run time. Parasol [39]–[41] 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. Domain Specific Languages (DSLs) like Listz [42], OP2 [43] and the Galois [44], provide abstractions that allow application developers to expose the parallelism of a computation. The compiler is then able to generate parallel code automatically. In this paper we develop a compiler approach that targets applications written in languages like C and do not need to be ported to such frameworks.

VIII. Conclusion

In this paper we have developed a framework that effectively models an important class of applications, namely *irregular-outer, regular inner*. The generated distributed memory code is efficient and more scalable than previous approaches that treat the entire computation as being irregular. Purely affine and completely irregular codes form two extremes of the range of computations modeled within this framework. The performance of the generated distributed memory code for both these classes are on-par, or better, than previous solutions developed to target only affine or only irregular codes. Finally, due to effective handling of regular parts of the original code the generated executor code does not introduce artifacts that would hamper subsequent compiler optimizations for those program regions.

Appendix

Example

A. Original Code

Figure 5 represents the abstraction used in AMR applications. The solid line represents a box which uses a coarse grid, while the dashed lines represents boxes that use a finer grid and are colocated in physical space with boxes using a coarse grid. In the example code shown in Listing 7, the fine grid resolution is twice the coarse grid resolution along both directions, i.e., each coarse grid-point is associated with 4 fine grid-points. The resolution of the coarse grid is assumed to be same as the resolution of integer points of the underlying physical domain. The total number of boxes (fine and coarse) used in the computation is `nboxes`, with each box having an index between 0 and `nboxes-1`, both included. The arrays `fine_boxes` is of size equal to the number of fine boxes used in the computation and stores their indices. Similarly, array `coarse_boxes` is of size equal to the number of coarse boxes used in the computation and stores their indices.

The array `start_x` and `start_y` is of size equal to the number of boxes used in the computations, both coarse and fine. They store the integer co-ordinates of the lexicographically smallest point of each

box, i.e., the co-ordinates of point E for the box shown in Figure 5. The array `end_x`, `end_y` are of the same size as `start_x`, `start_y` and store the co-ordinates of the lexicographically largest point of a box, i.e., point D. The values of ϕ and ϕ_{old} at grid points within a box are stored in arrays `phi` and `phi_old` respectively. These are 3D arrays, such that `phi[k][..][..]` holds the values of ϕ for the `k`-th box.

In the first annotated parallel loop at line 3, the values of `phi` at the coarse grid-points are updated using stencil operations with values of `phi_old` at the coarse grid-points used as input. The dotted line represent regions in surrounding boxes that are accessed during this operation. The array `phi_c_t` is used as a scratch pad to store values of `phi_old` from current and neighboring boxes used for the stencil operation. For each box, the information about patches of neighboring boxes accessed is maintained by storing the integer co-ordinates of the lexicographic minimum and maximum values of the patch in the arrays, `patch_start_y`, `patch_start_x`, `patch_end_y` and `patch_end_x`. For example, region within the north-neighboring box accessed is captured by storing the co-ordinates of point A in `patch_start_x` and `patch_start_y`, and the co-ordinates of point C are stored in `patch_end_x` and `patch_end_y`. The array `neighb` holds the index of the neighboring coarse boxes that contains this patch. Loops `i` and `j` at lines 10 and 11, iterate through the points within the patches. The position in `phi_old[neighb[p]][..][..]` of the required value from the neighboring box is obtained by subtracting the lexicographic minimum point of that box, i.e., point A, from the value of the loop iterators.

The second annotated loop at Line 20, updates values at fine-grid points with the values at coarse-grid points. Each fine box is assumed to be colocated in physical space with only one coarse box. The array `ftoc` stores the index of this coarse box for every fine box. The loops at line 21 and 22 iterate over the integer points bounded by the fine box. Since there are 4 fine grid-points for every coarse grid-points, statements from line 23 to 26 updates the value for each of them. In general, the value at a fine grid-point is updated using a stencil operation involving values at the coarse grid-point, with a different co-efficients used for each statement from line 23 to 26. Here, a simple copy is used. The points in the array `phi_old[fine_boxes[k]][..][..]` to be updated is indexed by subtracting the lexicographic minimum point of the fine box from the iterator values and multiplying it by the resolution of the finer grid along that dimension, i.e. 2 for both x and y. The position of grid points within `phi[ftoc[k]][..][..]` used on the right-hand side of these statements are obtained by subtracting from the iterator values, the lexicographic minimum of the coarse box.

B. Iteration Space Description

The first step in the code-generation scheme is to replace indirection array accesses within regular parts of the code with temporary variables. For each statement in Listing 7, the expressions replaced, the iteration space and data access map for arrays `phi_old`, `phi` and `rho` are described below.

1) Statement at Line 6: Indirection array accesses expressions replaced by temporary variables for the statement :

- `p1 := coarse_boxes[k]`
- `p2 := size_y[coarse_boxes[k]]`
- `p3 := size_x[coarse_boxes[k]]`

Non Affine Iterators : `k`

Iteration Space

$$I_1 := \{(k, i, j) \mid k = kc \ \wedge \ 0 \leq i < p2 \ \wedge \ 0 \leq j < p3\} \tag{10}$$

Access to array `phi_old`

$$D_1 := \{(k, i, j) \rightarrow (l, a, b) \mid l = p1 \ \wedge \ a = i \ \wedge \ b = j\} \tag{11}$$

```
1  #pragma parallel
2  #pragma temporary_array phi_c_t
3  for( k = 0; k < nc; k++ ){ ///Non-affine Iterator
4    for( i = 0; i < size_y[coarse_boxes[k]]; i++ ) ///Affine Iterator
5      for( j = 0; j < size_x[coarse_boxes[k]]; j++ ) ///Affine Iterator
6        phi_c_t[i+1][j+1] = phi_old[coarse_boxes[k]][i][j];
7
8    ///Non-affine Iterator
9    for( p = patches[coarse_boxes[k]]; p < patches[coarse_boxes[k]+1]; p++)
10     for( i = patch_start_y[p]; i < patch_end_y[p]; i++ ) ///Affine Iterator
11       for( j = patch_start_x[p]; j < patch_end_x[p]; j++ ) ///Affine Iterator
12         phi_c_t[i+1-start_y[coarse_boxes[k]]][j+1-start_x[coarse_boxes[k]]]
             = phi_old[neighb[p]][i - start_y[neighb[p]]][j - start_x[neighb[p]]];
13
14   for( i = 0; i < size_y[coarse_boxes[k]]; i++ ) ///Affine Iterator
15     for( j = 0; j < size_x[coarse_boxes[k]]; j++ ) ///Affine Iterator
16       phi[coarse_boxes[k]][i][j] += ( phi_c_t[i][j+1] + phi_c_t[i+1][j] + phi_c_t[i+1][j+2]
           + phi_c_t[i+2][j+1] - 4 * phi_c_t[i+1][j+1] - rho[coarse_boxes[k]][i][j] ) / 8.0;
17 }
18
19 #pragma parallel
20 for( k = 0; k < nf; k++) { ///Non-affine Iterator
21   for( i = start_y[fine_boxes[k]]; i < end_y[fine_boxes[k]]; i++ ) ///Affine Iterator
22     for( j = start_x[fine_boxes[k]]; j < end_x[fine_boxes[k]]; j++ ){ ///Affine Iterator
23      phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2][(j-start_x[fine_boxes[k]])*2]
          = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
24      phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2+1][(j-start_x[fine_boxes[k]])*2]
          = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
25      phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2][(j-start_x[fine_boxes[k]])*2+1]
          = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
26      phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2+1][(j-start_x[fine_boxes[k]])*2+1]
          = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
27     }
28 }
```

Listing 7: Original loop-nests of AMR

2) Statement at Line 12: Indirection array accesses expressions replaced by temporary variables for the statement :

- `p4 := patches[coarse_boxes[k]]`
- `p5 := patches[coarse_boxes[k]+1]`
- `p6 := patch_start_y[p]`
- `p7 := patch_end_y[p]`
- `p8 := patch_start_x[p]`
- `p9 := patch_end_x[p]`
- `p10 := start_y[coarse_boxes[k]]`
- `p11 := start_x[coarse_boxes[k]]`
- `p12 := start_y[neighb[p]]`
- `p13 := start_x[neighb[p]]`
- `p14 := neigh[p]`

Non Affine Iterators : `k,p`

Iteration Space

$$I_2 := \{(k,p,i,j) \mid k = kc \ \wedge \ p = pc \ \wedge \ p6 \leq i < p7 \ \wedge \ p8 \leq j < p9\} \tag{12}$$

Access to array `phi_old`

$$D_2 := \{(k,p,i,j) \rightarrow (l,a,b) \mid l = p14 \ \wedge \ a = i - p12 \ \wedge \ b = j - p13\} \tag{13}$$

3) Statement at Line 16: Indirection array accesses expressions replaced by temporary variables for the statement :

- `p1 := coarse_boxes[k]`
- `p2 := size_y[coarse_boxes[k]]`
- `p3 := size_x[coarse_boxes[k]]`

Non Affine Iterators : `k`

Iteration Space

$$I_3 := \{(k,i,j) \mid k = kc \ \wedge \ 0 \leq i < p2 \ \wedge \ 0 \leq j < p3\} \tag{14}$$

Access to array `phi`

$$D_3 := \{(k,i,j) \rightarrow (l,a,b) \mid l = p1 \ \wedge \ a = i \ \wedge \ b = j\} \tag{15}$$

Access to array `rho`

$$D_4 := \{(k,i,j) \rightarrow (l,a,b) \mid l = p1 \ \wedge \ a = i \ \wedge \ b = j\} \tag{16}$$

4) Statement at Line 23-26: Indirection array accesses expressions replaced by temporary variables for the statement :

- `p15 := fine_boxes[k]`
- `p16 := ftoc[k]`
- `p17 := start_y[fine_boxes[k]]`
- `p18 := start_x[fine_boxes[k]]`
- `p19 := end_y[fine_boxes[k]]`
- `p20 := end_x[fine_boxes[k]]`
- `p21 := start_y[ftoc[k]]`
- `p22 := start_x[ftoc[k]]`

Non Affine Iterators : `k`

Iteration Space

$$I_4 := \{(k,i,j) \mid k = kf \ \wedge \ p17 \leq i < p19 \ \wedge \ p18 \leq j < p20\} \tag{17}$$

Access to array `phi_old`

$$\begin{aligned} D_5 :=& \{(k,i,j) \rightarrow (l,a,b) \mid l = p15 \ \wedge \ a = 2*(i-p17) \ \wedge \ b = 2*(j-p18)\} \\ & \cup \{(k,i,j) \rightarrow (l,a,b) \mid l = p15 \ \wedge \ a = 2*(i-p17)+1 \ \wedge \ b = 2*(j-p18)\} \\ & \cup \{(k,i,j) \rightarrow (l,a,b) \mid l = p15 \ \wedge \ a = 2*(i-p17) \ \wedge \ b = 2*(j-p18)+1\} \\ & \cup \{(k,i,j) \rightarrow (l,a,b) \mid l = p15 \ \wedge \ a = 2*(i-p17)+1 \ \wedge \ b = 2*(j-p18)+1\} \end{aligned} \tag{18}$$

Access to array `phi`

$$D_6 := \{(k,i,j) \rightarrow (l,a,b) \mid l = p16 \ \wedge \ a = i - p21 \ \wedge \ b = j - p22\} \tag{19}$$

C. Inspector Code to compute the footprint

The inspector code generated using Algorithm 1 is shown in Listing 8.

```
1  for (k = 0; k < nc ; k++ )
2    if( get_home(loop_0,k) == myid ){
3      p1 = get_elem(array_coarse_box,k);
4      p2 = get_elem(array_size_y,get_elem(array_coarse_boxes,k));
5      p3 = get_elem(array_size_x,get_elem(array_coarse_boxes,k));
6
7      if( p2 >= 1 && p3 >= 1 ){
8        ///Record access to phi_old[coarse_boxes[k]]
9        p_outer = p1;
10       update_access(array_phi_old,p_outer);
```

```
11        ///Record access lexmin/max for all the inner dimension
12        ///lexmin(P_outer(D1(I1))
13        lexmin_dim1 = 0; lexmin_dim2 = 0;
14        update_lexmin(array_phi_old,p_outer,lexmin_dim1,lexmin_dim2);
15        ///lexmax(P_outer(D1(I1))
16        lexmax_dim1 = p2-1 ; lexmax_dim2 = p3-1;
17        update_lexmax(array_phi_old,p_outer,lexmax_dim1,lexmax_dim2);
18      }
19
20      for( p = get_elem(array_patches,get_elem(array_coarse_boxes,k)) ;
                p < get_elem(array_patches,get_elem(array_coarse_boxes,k)+1) ; p++ ){
21        p4 = get_elem(array_patches,get_elem(array_coarse_boxes,k));
22        p5 = get_elem(array_patches,get_elem(array_coarse_boxes,k)+1);
23        p6 = get_elem(array_patch_start_y,p);
24        p7 = get_elem(array_patch_end_y,p);
25        p8 = get_elem(array_patch_start_x,p);
26        p9 = get_elem(array_patch_end_x,p);
27        p10 = get_elem(array_start_y,get_elem(array_coarse_boxes,k));
28        p11 = get_elem(array_start_x,get_elem(array_coarse_boxes,k));
29        p12 = get_elem(array_start_y,get_elem(array_neighb,p));
30        p13 = get_elem(array_start_x,get_elem(array_neighb,p));
31        p14 = get_elem(array_neighb,p);
32
33        if( p7 >= p6 + 1 && p9 >= p8 + 1 ){
34          ///Record access to phi_old[neighb[p]]
35          p_outer = p14;
36          update_access(array_phi_old,p_outer);
37          ///Record access lexmin/max for all the inner dimension
38          ///lexmin(P_outer(D2(I2))
39          lexmin_dim1 = p6-p12; lexmin_dim2 = p8-p13;
40          update_lexmin(array_phi_old,p_outer,lexmin_dim1,lexmin_dim2);
41          ///lexmax(P_outer(D2(I2))
42          lexmax_dim1 = p7-p12-1; lexmax_dim2 = p9-p13-1;
43          update_lexmin(array_phi_old,p_outer,lexmax_dim1,lexmax_dim2);
44        }
45
46      if( p2 >= 1 && p3 >= 1 ){
47        ///Record access to phi[coarse_boxes[k]]
48        p_outer = p1;
49        update_access(array_phi,p_outer);
50        ///Record access lexmin/max for all the inner dimension
51        ///lexmin(P_outer(D3(I3))
52        lexmin_dim1 = 0 ; lexmin_dim2 = 0;
53        update_lexmin(array_phi,p_outer,lexmin_dim1,lexmin_dim2);
54        ///lexmax(P_outer(D3(I3))
55        lexmax_dim1 = p2-1 ; lexmax_dim2 = p3-1;
56        update_lexmax(array_phi,p_outer,lexmax_dim1,lexmax_dim2);
57
58        ///Record access to rho[coarse_boxes[k]]
59        p_outer = p1;
60        update_access(array_rho,p_outer);
61        ///Record access lexmin/max for all the inner dimension
62        ///lexmin(P_outer(D4(I3))
63        lexmin_dim1 = 0 ; lexmin_dim2 = 0;
64        update_lexmin(array_rho,p_outer,lexmin_dim1,lexmin_dim2);
65        ///lexmax(P_outer(D4(I3))
66        lexmax_dim1 = p2-1 ; lexmax_dim2 = p3-1;
67        update_lexmax(array_rho,p_outer,lexmax_dim1,lexmax_dim2);
68      }
69    }
70
71  for (k = 0; k < nf ; k++ )
72    if( get_home(loop_1,k) == myid ){
73
74      p15 = get_elem(array_fine_boxes,k);
```

```
    p16 = get_elem(array_ftoc,k);
    p17 = get_elem(array_start_y,get_elem(array_fine_boxes,k));
    p18 = get_elem(array_start_x,get_elem(array_fine_boxes,k));
    p19 = get_elem(array_end_y,get_elem(array_fine_boxes,k));
    p20 = get_elem(array_end_x,get_elem(array_fine_boxes,k));
    p21 = get_elem(array_start_y,get_elem(array_ftoc,k));
    p22 = get_elem(array_start_x,get_elem(array_ftoc,k));

    if( p19 >= p17 + 1 && p20 >= p18 + 1 ){
      ///Record access to phi_old
      p_outer = p15;
      ///Record access lexmin/max for all the inner dimension
      ///lexmin(P_outer(D5(I4))
      lexmin_dim1 = 0; lexmin_dim2 = 0;
      update_lexmin(array_phi_old,lexmin_dim1,lexmin_dim2);
      ///lexmax(P_outer(D5(I4))
      lexmax_dim1 = 2 * p19 - 2 * p17 - 1 ; lexmax_dim2 = 2 * p20 - 2 * p18 - 1;
      update_lexmax(array_phi_old,lexmax_dim1,lexmax_dim2);

      ///Record access to phi
      p_outer = p16;
      ///Record access lexmin/max for all the inner dimension
      ///lexmin(P_outer(D6(I4))
      lexmin_dim1 = p17 - p21; lexmin_dim2 = p18 - p22;
      update_lexmin(array_phi,lexmin_dim1,lexmin_dim2);
      ///lexmax(P_outer(D6(I4))
      lexmax_dim1 = p19 - p21 - 1 ; lexmax_dim2 = p20 - p21 - 1;
      update_lexmax(array_phi,lexmax_dim1,lexmax_dim2);
    }
  }
```

Listing 8: Inspector Code to compute footprint

D. Executor Code

The executor code generated using Algorithm 2 is shown in Listing 9.

```
///Update ghosts in array phi_old with values at owners
communicate_reads(loop_0);
///Initialize ghosts in array phi to 0.0;
init_write_ghosts(loop_0);
loop_0=0;
for (k = 0; k < nlocal_2; ++k) {
  p1 = coarse_boxes_l[k];
  p2 = size_y_l[coarse_boxes_l[k]];
  p3 = size_x_l[coarse_boxes_l[k]];
  for (i = 0; i < p2; ++i)
    for (j = 0; j < p3; ++j)
      phi_c_t[i+1][j+1] =
        phi_old_l[p1][i-lexmin_phi_old_dim1[p1]][j-lexmin_phi_old_dim2[p1]];

  access_offset = access_phi_old[loop_0] - patches_l[coarse_boxes_l[k]];
  for (p = patches_l[coarse_boxes_l[k]]; p < ub[k]; ++p) {
    p4 = patches_l[coarse_boxes_l[k]];
    p5 = ub[k];
    p6 = patch_start_y_l[p+access_offset];
    p7 = patch_end_y_l[p+access_offset];
    p8 = patch_start_x_l[p+access_offset];
    p9 = patch_end_x_l[p+access_offset];
    p10 = start_y_l[coarse_boxes_l[k]];
    p11 = start_x_l[coarse_boxes_l[k]];
    p12 = start_y_l[neighb_l[p+access_offset]];
    p13 = start_x_l[neighb_l[p+access_offset]];
    p14 = neighb_l[p+access_offset];
    for (i = p6 ; i < p7; ++i)
```

```
28        for (j = p8 ; j < p9; ++j)
29         phi_c_t[i+1-p10][j+1-p11] =
            phi_old_l[p14][i-p12-lexmin_phi_old_dim1[p14]][j-p13-lexmin_phi_old_dim1[p14]];
30    }
31    loop_0++;
32
33    for (i = 0; i < p2; ++i)
34      for (j = 0; j < p3; ++j)
35        phi_l[p1][i-lexmin_phi_dim1[p1]][j-lexmin_phi_dim2[p1]] +=
            (phi_c_t[i][j+1] + phi_c_t[i+1][j] + phi_c_t[i+1][j+2] + phi_c_t[i+2][j+1]
            - 4 * phi_c_t[i+1][j+1]
             - rho_l[p1][i-lexmin_rho_dim1[p1]][j-lexmin_rho_dim2[p1]] ) / 8.0;
36  }
37  ///Transfer partial contributions in ghosts of array phi to the owners
38  communicate_writes(loop_0);
39
40  ///Update ghosts in array phi with values at owners
41  communicate_reads(loop_1);
42  ///Initialize ghosts in array phi_old to 0;
43  init_write_ghosts(loop_1);
44  for (k = 0; k < nlocal_3; ++k) {
45   p15 = fine_boxes_l[k];
46   p16 = ftoc_l[k];
47   p17 = start_y_l[fine_boxes_l[k]];
48   p18 = start_x_l[fine_boxes_l[k]];
49   p19 = end_y_l[fine_boxes_l[k]];
50   p20 = end_x_l[fine_boxes_l[k]];
51   p21 = start_y_l[ftoc_l[k]];
52   p22 = start_x_l[ftoc_l[k]];
53   for (i = p17 ; i < p19 ; i++ )
54    for (j = p18 ; j < p20 ; j++ ){
55     phi_old_l[p15][2*(i-p17)-lexmin_phi_old_dim1[p15]][2*(j-p18)-lexmin_phi_old_dim2[p15]]
        = phi_l[p16][i - p21 - lexmin_phi_dim1[p16]][j - p22 - lexmin_phi_dim2[p16]];
56     phi_old_l[p15][2*(i-p17)+1-lexmin_phi_old_dim1[p15]][2*(j-p18)-lexmin_phi_old_dim2[p15]]
        = phi_l[p16][i - p21 - lexmin_phi_dim1[p16]][j - p22 - lexmin_phi_dim2[p16]];
57     phi_old_l[p15][2*(i-p17)-lexmin_phi_old_dim1[p15]][2*(j-p18)+1-lexmin_phi_old_dim2[p15]]
        = phi_l[p16][i - p21 - lexmin_phi_dim1[p16]][j - p22 - lexmin_phi_dim2[p16]];
58   phi_old_l[p15][2*(i-p17)+1-lexmin_phi_old_dim1[p15]][2*(j-p18)+1-lexmin_phi_old_dim2[p15]]
        = phi_l[p16][i - p21 - lexmin_phi_dim1[p16]][j - p22 - lexmin_phi_dim2[p16]];
59    }
60   }
61  ///Transfer partial contributions in ghosts of array phi_old to the owners
62  communicate_writes(loop_1);
```

Listing 9: Executor Code

REFERENCES

[1] U. Bondhugula, "Compiling affine loop nests for distributed-memory parallel architectures," in *SC*, 2013.

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

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

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

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

[6] 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.

[7] 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.

[8] "Polyopt/c a polyhedral optimizer for the ROSE compiler," `http://www.cs.ucla.edu/~pouchet/software/polyopt/`.

[9] G. Gupta and S. V. Rajopadhye, "The z-polyhedral model," in *PPOPP*, 2007.

[10] S. Verdoolaege, "ISL: An integer set library for the polyhedral model," in *Lecture Notes in Computer Science,*, 2010.

[11] A. LaMielle and M. Strout, "Enabling code gen. with sparse polyhedral framework," Colorado State University, Tech. Rep., 2010.

[12] "Cloog : Chunky loop generator," `http://www.cloog.org/`.

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

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

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

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

[17] 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.

[18] 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.

[19] "Polybench/c: the polyhedral benchmark suite," `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`.

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

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

[22] 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.

[23] 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.

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

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

[26] 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.

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

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

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

[30] A. Basumallik and R. Eigenmann, "Towards automatic translation of OpenMP to MPI," in *ICS*, 2005.

[31] ——, "Optimizing irregular shared-memory applications for distributed-memory systems," in *PPoPP*, 2006.

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

[33] 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.

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

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

[36] 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.

[37] 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.

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

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

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

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

[42] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: A domain specific language for building portable mesh-based pde solvers," in *SC*, 2011.

[43] M. B. Giles, G.R.Mudalige, Z.Sharif, G.Markall, and P. Kelly, "Performance analysis and optimization of the op2 framework on many-core architectures," *The Computer Journal*, 2010.

[44] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew, "Optimistic parallelism benefits from data partitioning," in *ASPLOS*, 2008.