A Constraint-Based Approach to Automatic Data Partitioning for Distributed Memory Execution

Wonchan Lee
Stanford University
wonchan@cs.stanford.edu

Manolis Papadakis
Stanford University
mpapadak@cs.stanford.edu

Elliott Slaughter
SLAC National Accelerator Laboratory
eslaught@slac.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

ABSTRACT

Although data partitioning is required to enable parallelism on distributed memory systems, data partitions are not first class objects in most distributed programming models. As a result, automatic parallelizers and application writers encode a particular partitioning strategy in the parallelized program, leading to a program not easily configured or composed with other parallel programs.

We present a *constraint-based* approach to automatic data partitioning. By introducing abstractions for first-class data partitions, we express a space of correct partitioning strategies. Candidate partitions are characterized by *partitioning constraints*, which can be automatically inferred from data accesses in parallelizable loops. Constraints can be satisfied by synthesized partitioning code or user-provided partitions. We demonstrate that programs auto-parallelized in our approach are easily composed with manually parallelized parts and have scalability comparable to hand-optimized counterparts.

CCS CONCEPTS

• **Computing methodologies → Distributed programming languages**.

KEYWORDS

Auto-Parallelization; Data Partitioning; First-Class Data Partitions; Partitioning Constraints

ACM Reference Format:
Wonchan Lee, Manolis Papadakis, Elliott Slaughter, and Alex Aiken. 2019. A Constraint-Based Approach to Automatic Data Partitioning for Distributed Memory Execution. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356199

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356199

1 INTRODUCTION

Data partitioning is an essential step to exploit parallelism on distributed memory systems. For example, for a data parallel loop, parallelism can be realized by partitioning the data into *subregions* (subcollections of the original data) and running the set of loop iterations accessing a particular subregion. In general, for a given set of parallel tasks, only some partitions of the data are legal, because at a minimum the data accessed by a task must be included in that task's subregion arguments. Furthermore, as programs generally have multiple data access patterns in different loops, the possible legal partitions are those satisfying all the constraints of all loops, while the performant partitions are a subset of the legal partitions. In this view of program parallelization, the crux of parallelizing for distributed execution is identifying and satisfying these data partitioning constraints.

In this paper, we present a *constraint-based* approach to data partitioning. We first extract *partitioning constraints* under which data partitions will preserve program execution semantics. Partitioning constraints are inferred automatically from data accesses in programs and serve as a specification for implementations of data partitioning. Many different partitioning strategies may satisfy a system of partitioning constraints. To find implementations that match the specification, we employ a constraint solver that synthesizes partitioning code in DPL, the Dependent Partitioning Language [26], a domain-specific language for data partitioning (we give an overview of DPL below).

The constraint solver can also exploit externally provided invariants on partitions to discharge some or all partitioning constraints — this is the mechanism by which users or other systems (e.g., external libraries) can provide additional information to the automatic partitioning algorithm about the environment in which the parallelized code will execute. This feature is particularly appealing in scenarios where applying auto-parallelization to the whole program is infeasible or inefficient. For example, this approach naturally handles the common case where the code to be parallelized must accept input from another program component with a fixed data partitioning. Partitioning constraints serve as an interface conveying information about existing partitions from previously or manually parallelized parts to our automated data partitioning algorithm.

1.1 Overview

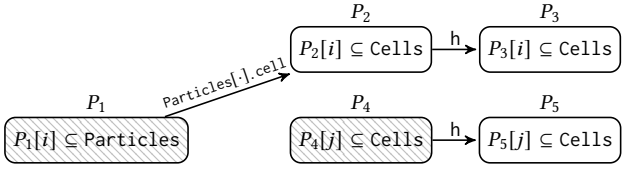
We illustrate our constraint-based approach using the program in Figure 1a, which showcases a common pattern of using indirect

```
1 for (p in Particles):
2   c = Particles[p].cell
3   Particles[p].pos += f(Cells[c].vel, Cells[h(c)].vel)
4 for (c in Cells):
5   Cells[c].vel += g(Cells[c].acc, Cells[h(c)].acc)
```

(a) Parallelizable loops

```
1 task T1(Particles, Cells1, Cells2):
2   for (p in Particles):
3     c = Particles[p].cell
4     Particles[p].pos += f(Cells1[c].vel, Cells2[h(c)].vel)
5 task T2(Cells3, Cells4):
6   for (c in Cells3):
7     Cells3[c].vel += g(Cells3[c].acc, Cells4[h(c)].acc)
8 parallel for (i in P1):
9   T1(P1[i], P2[i], P3[i])
10 parallel for (j in P4):
11   T2(P4[j], P5[j])
```

(b) Loops parallelized using tasks



(c) Partitioning constraints

Figure 1: Example

accesses to establish relationships between different physical entities. This program is excerpted from a larger program written in Regent [23], which is the language we use for all code examples and for our implementation. Regent provides a sophisticated set of data partitioning primitives, and so is a natural vehicle for our work.

The program in Figure 1a stores properties of particles and cells in *regions* `Particles` and `Cells`. A region is a collection of values. All elements of a region have the same type, and every element has a unique index. The values in a region may have *fields*, such as the `cell`, `vel`, and `acc` fields used in Figure 1a. Regent provides typical looping constructs over the elements of regions. The first loop iterates over `Particles` to update the position of each particle `p`. The index `c` of the cell where each particle resides is stored in `Particles[p].cell` (line 2). The change in each particle's position is then computed using the velocity of the cell at `c` and its neighbor `h(c)` (line 3). The second loop updates the velocity of each cell similarly (line 5).

The program in Figure 1b parallelizes the loops in Figure 1a using first-class data partitions; partitions are a primitive concept in Regent. Each *partition* P_i is an array of *subregions* (i.e., subsets of a region) and each **`parallel for`** loop launches *tasks* for subregions in the partition, each of which runs a subset of the original loop iterations. Subregions are names for subsets of collections and subregions can be recursively partitioned into subregions themselves. As is standard, tasks are designated functions that can be run asynchronously (in parallel). Tasks in Regent can take regions, partitions or scalars as arguments.

```
1 P1 = equal(Particles, N)
2 P2 = image(P1, Particles[·].cell, Cells)
3 P3 = image(P2, h, Cells)
4 P4 = equal(Cells, N)
5 P5 = image(P4, h, Cells)
```

(a) Program A

```
1 P2 = P4 = equal(Cells, N)
2 P1 = preimage(Particles, Particles[·].cell, P2)
3 P3 = P5 = image(P2, h, Cells)
```

(b) Program B

Figure 2: DPL programs solving the constraints in Figure 1c

The parallel program in Figure 1b preserves the semantics of the sequential program in Figure 1a provided the partitions $P_1, \ldots, P_5$ are *legal*, i.e., when they make the following indirect accesses safe:

- `Cells1[c].vel` at line 4;
- `Cells2[h(c)].vel` at line 4; and
- `Cells4[h(c)].acc` at line 7.

The space of legal partitions can be expressed by *partitioning constraints* that are inferred automatically from programs. Figure 1c shows the partitioning constraints that capture the conditions under which $P_1, \ldots, P_5$ in Figure 1b are legal. Each node in the graph corresponds to a partition. The node labeled with P_1 denotes a partition of `Particles`, whereas the others are (potentially different) partitions of `Cells`. Shaded nodes represent partitions that must be *complete*; a partition is complete when its subregions include all elements of the region. Nodes for partitions P_1 and P_4 are shaded because they must cover the iteration space of the loops at lines 1 and 4. Edges between nodes specify constraints on partitions. The edge from P_2 to P_3, labeled with the function `h`, requires that each subregion $P_3[j]$ contain the image of $P_2[j]$ under `h`, that is, $\forall (k, v) \in P_2[j].\ \exists v'.\ (\mathsf{h}(k), v') \in P_3[j]$. The edge from P_4 to P_5 describes the same constraint but on P_4 and P_5. The other edge between P_1 and P_2 is interpreted similarly:

$$\forall (k, v) \in P_1[i].\ \exists v'.\ \left(\texttt{Particles}[k]\texttt{.cell}, v'\right) \in P_2[i]$$

Figure 2 gives two partitioning strategies satisfying the constraints in Figure 1c, expressed as DPL programs that construct partitions using the high-level partitioning operators **`equal`**, **`image`**, and **`preimage`**. DPL is the partitioning sublanguage of Regent that computes partitions of regions at runtime. DPL has additional operators but these are the most commonly used. The main idea in DPL is that some partitioning operators, such as **`equal`**, create partitions of regions directly, while others compute a new partition as a function of an existing partition (thus the name *dependent* partitioning language). Sophisticated data partitions can be constructed by composing the small set of primitive DPL operators.

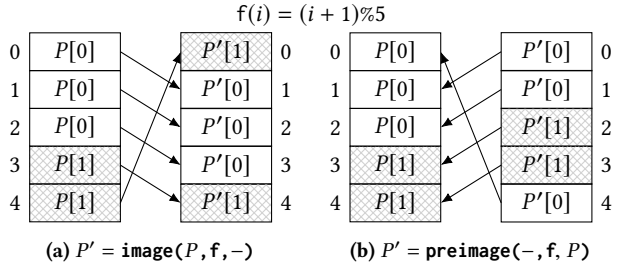
In Figure 2, program A derives P_2, P_3, and P_5 from **`equal`** partitions of P_1 and P_4; the **`equal`** operator creates a complete partition of a region with (approximately) equal size subregions. Partitions P_2, P_3 and P_5 use **`image`** partitions to satisfy the partitioning constraints. The **`image`** operator uses an existing partition and a function to define a compatible partition of a region. For example, if

$P_2 = \langle r_1, \ldots, r_n \rangle$, then the statement P_3 = **image**(P_2, h, Cells) creates $P_3 = \langle \mathsf{h}(r_1), \ldots, \mathsf{h}(r_n) \rangle$, where $\mathsf{h}(r_i) \subseteq$ Cells. Figure 3a gives a visual representation of the **image** operator: The region on the left is already partitioned into two subregions, indicated by the sets of light and dark elements. The image of function f mapping elements of the lefthand region to elements of the righthand region then defines two subregions of the righthand region.

Program B implements a different strategy, first creating an **equal** partition of Cells for both P_2 and P_4. Note that P_2 is assigned a complete partition even though the partitioning constraint does not require it to be complete; as long as P_2 contains the image of Particles[·].cell, it can have extra elements. To construct the partition P_1 from the already defined P_2, program B uses the **preimage** operator. As illustrated in Figure 3b, **preimage** takes an existing partition of the region on the righthand side and constructs a compatible partition using the preimage of the function; i.e., if the provided partition is $\langle r_1, \ldots, r_n \rangle$ and the function is h, then the computed partition is $\langle \mathsf{h}^{-1}(r_1), \ldots, \mathsf{h}^{-1}(r_n) \rangle$. Finally, P_3 and P_5 are computed using the image of P_2 under h.

Without any prior knowledge about Cells and Particles, it is not clear whether the partitioning strategy in Figure 2a or Figure 2b is better. Program A has an additional pair of partitions of Cells, which is not necessarily worse than program B if communication due to the extra partitions is justified; in a scenario where spatial distribution of the particles is significantly skewed, program B can suffer from load imbalance in the first loop in Figure 1b, whereas program A is immune to this issue because the subregions of P_1 have equal size. On the other hand, if the particles in each subregion of P_1 are spread throughout the domain, each subregion of P_2 can be as big as Cells, leading to excessive communication.

Given that we cannot identify an optimal DPL program that satisfies the constraints at compile-time, we use heuristics to guide the constraint resolution process. For example, when a given set of partitioning constraints admit multiple DPL programs, our constraint solver chooses the one with the fewest partitions (program B in this example). This approach does not always produce satisfactory solutions when important information about the execution context is missing. Programs also often have parts that are hard to auto-parallelize well, or may not be possible to auto-parallelize at all. Our approach can gracefully handle these situations by allowing programmers to provide additional constraints encoding knowledge of which strategies are best and/or existing partitions used outside the scope of auto-parallelization. For example, if the loops in Figure 1a were embedded in an outer loop where particles'



(a) P' = **image**(P, **f**, −) **(b)** P' = **preimage**(−, **f**, P)

Figure 3: Image and preimage operators

```
1  parallel for (i in pParticles):
2    for (p in pParticles[i]):
3      new_cell = locate(pParticles[i][p].pos)
4      if (pParticles[i][p].cell != new_cell)
5        pParticles[i][p].cell = new_cell
6        find j such that new_cell ∈ pCells[j]
7        if (i != j):
8          send pParticles[i][p] to pParticles[j]
9  assert(image(pParticles,Particles[·].cell,Cells) ⊆ pCells)
```

Figure 4: Example with a user-provided constraint

pointers to cells are updated every iteration, the DPL program in Figure 2b would need to repartition Particles every iteration to reflect the updates. If only a few particles change cells on each iteration, then repartitioning the entire Particles region is wasteful. A simple way to mitigate this inefficiency is to exchange particles manually; the pseudo-code in Figure 4 sends a particle to the right "owner" whenever the cell to which the particle moves belongs to a subregion different from the current one. The most important part in this pseudo code is the assertion at line 10 specifying the invariant on pParticles and pCells, i.e., that the subregion pCells[i] contains all the cells pointed to by the particles in pParticles[i]. The constraint solver uses this assertion to discharge all partitioning constraints in Figure 1c except those on P_3 and P_5, for which the solver emits the following DPL program using pCells to derive P_3 and P_5:

$$P_3 = P_5 = \textbf{image}(\texttt{pCells, h, Cells})$$

This example demonstrates the key benefit of constraint-based approaches that separate specification from implementation [3]; as long as the manual particle exchange code maintains the invariant, the entire program mixing parts that are parallelized by different means is correct. Furthermore, by providing constraints the user has a high-level but precise interface for informing the auto-parallelization process, and writing these interface constraints is much less work than parallelizing the entire code manually.

This paper makes the following contributions:

- We present the design of a static analysis that automatically infers partitioning constraints from programs.
- We design a constraint solver that synthesizes DPL code from partitioning constraints.
- We evaluate the implementation of our constraint-based approach using the Regent compiler [23]. For a set of Regent programs that are already hand-optimized for distributed memory execution [20, 24], their sequential counterparts auto-parallelized in our approach achieved comparable performance (within 5%).

In the following sections, we design the static analysis for constraint inference (Section 2) and the constraint solver (Section 3). We then describe key optimizations on DPL programs (Section 5), and we present experimental results (Section 6).

2 CONSTRAINT INFERENCE

Figure 5 shows the syntax of the partitioning constraint language. Ground terms are regions (using symbol R) and partitions (using symbol P); recall that a region is an indexed set of a fixed type

$$
\begin{array}{llll}
\text{Regions} & R & & \text{Partitions} \quad P \\
\text{Constraints} & C & ::= & \phi \mid E \subseteq E \mid C \wedge C \\
\text{Predicates} & \phi & ::= & \mathbf{PART}(E, R) \mid \mathbf{DISJ}(E) \mid \mathbf{COMP}(E, R) \\
\text{Expressions} & E & ::= & P \mid E \cup E \mid E \cap E \mid E - E \\
 & & \mid & \mathbf{image}(E, f, R) \mid \mathbf{preimage}(R, f, E) \mid \mathbf{equal}(R)
\end{array}
$$

Figure 5: Constraint language syntax

(possibly with fields) and a partition is an indexed set of subregions of a region. A partitioning constraint is a conjunction of subset constraints and predicates on partitions. A predicate $\mathbf{PART}(E, R)$ means that E is a partition of the region R; i.e., each subregion $E[i]$ must be a subset of the region R:

$$\mathbf{PART}(E, R) \triangleq \forall i. E[i] \subseteq R.$$

A predicate $\mathbf{DISJ}(E)$ requires E to be a disjoint partition and a $\mathbf{COMP}(E, R)$ requires E to be a complete partition of R:

$$\mathbf{DISJ}(E) \triangleq E[i] \cap E[j] = \varnothing \text{ when } i \neq j$$
$$\mathbf{COMP}(E, R) \triangleq \bigcup_i E[i] = R$$

A subset constraint $E_1 \subseteq E_2$ denotes that each subregion $E_2[i]$ contains the corresponding subregion $E_1[i]$:

$$E_1 \subseteq E_2 \triangleq \forall i.\ E_1[i] \subseteq E_2[i]$$

This relation implicitly requires that the set of indices of E_2 subsume that of E_1. The subset constraint is anti-symmetric, i.e.,

$$E_1 = E_2 \triangleq E_1 \subseteq E_2 \wedge E_2 \subseteq E_1.$$

Expressions are DPL operators that construct partitions; we use DPL operators in constraints to syntactically describe partitions of interest. The union, intersection, and difference operators on partitions are applied subregion-wise; for example, a union of two partitions results in a partition whose ith subregion is a union of the ith subregions of the operands:

$$(E_1 \diamond E_2)[i] \triangleq E_1[i] \diamond E_2[i] \qquad \text{where } \diamond \in \{\cup, \cap, -\}$$

The **image** operator creates a partition of a function's range from an existing partition of the function's domain; the expression $\mathbf{image}(E, f, R)$ is a partition of R derived from E using f as follows:

$$\mathbf{image}(E, f, R)[i] \triangleq \{(f(k), v') \in R \mid (k, v) \in E[i]\}.$$

(Note that the function f takes the indices of each subregion as arguments, not its values.) The **preimage** operator is an inverse of **image**, i.e., deriving a partition of a function's domain from an existing partition of the function's range; the expression $\mathbf{preimage}(R, f, E)$ is a partition of f's domain R derived from E as follows:

$$\mathbf{preimage}(R, f, E)[i] \triangleq \{(k, v') \in R \mid \big(f(k), v\big) \in E[i]\}.$$

Again, Figure 3 visualizes the **image** and **preimage** operators for an example function `f`. Lastly, the **equal** operator creates partitions without using any other partitions; the expression $\mathbf{equal}(R)$ creates a partition of R with approximately equal size subregions. (Integer arguments denoting the number of subregions are elided in partitioning constraints because they do not affect constraint solving.)

Algorithm 1 shows the constraint inference algorithm, which takes a loop and produces a system of partitioning constraints. The algorithm is concerned only with *parallelizable* loops. A loop is parallelizable when values defined in one loop iteration are never consumed by other iterations of the same loop. For brevity, we characterize parallelizable loops syntactically as follows.

Algorithm 1: Constraint inference algorithm

```
1  Procedure Infer(loop):
2      // Assume loop has the form for (i in R): body
3      // Assume body is normalized
4
5      // The function f_ID is the identity function, i.e., f_ID(x) = x.
6      // Note that image(P_R, f_ID, R) = P_R.
7      Env ← {i ↦ λr.image(P_R, f_ID, r)}                    (P_R is fresh)
8      C ← PART(P_R, R) ∧ COMP(P_R, R)
9      for each statement s ∈ body do
10         // Region accesses appear only in statements of these forms:
11         if s is y = S[x] or S[x] = y or S[x] += y:
12             E ← Env(x)(S)
13             C ← C ∧ PART(P, S) ∧ E ⊆ P                    (P is fresh)
14             if s is y = S[x]:
15                 Env ← Env ∪ {y ↦ λr.image(E, S[·], r)}
16             elseif s is S[x] += y and E ≠ P_R:
17                 C ← C ∧ DISJ(P_R)
18         elseif s is y = f(x):
19             Env ← Env ∪ {y ↦ λr.image(Env(x)(r), f, r)}
20         elseif s is y = x:
21             Env ← Env ∪ {y ↦ Env(x)}
```

- Region accesses are either *centered* or *uncentered*. A region access `R[e]` is centered when `e` is the loop variable (or an alias), and is uncentered otherwise.
- An uncentered access is *admissible* only when it has an index expression derived from another region access (e.g., `R[S[e]]`) or it has the form `R[f(i)]` where `i` is the loop variable.
- A parallelizable loop is an outermost loop of the form

  ```
  for (i in R): ...
  ```

  for some region `R` (the *iteration space* of the loop), which satisfies these conditions:
  - All write accesses to regions are centered. (A centered reduction is considered a centered read access followed by a centered write access.)
  - A region with an uncentered reduction (e.g., `R[S[e]] += ...`) does not have any other read access or a reduction with a different operator.
  - A region with an uncentered read (e.g., `... = R[S[e]]`) does not have any other write access.

This syntactic definition is sound but incomplete; i.e., there are loops that a more sophisticated analysis, such as polyhedral analysis [7], can prove parallelizable but our syntactic check cannot.

At the highest level, our method for constraint-based partitioning has three components:

- Initially a separate partition (represented by a unique partition variable) is assigned to every region access in a parallelizable loop. Another unique partition variable is associated with the loop index. For each of these variables, we generate constraints that guarantee the partition will have all the elements needed to execute correctly.
- We solve the constraints by rewriting them into an equivalent form where each remaining constraint corresponds to

Program	Constraints
`for (p in Particles):`	$\mathbf{PART}(P_1, \texttt{Particles}) \wedge$ $\mathbf{COMP}(P_1, \texttt{Particles}) \wedge$
`c = Particles[p].cell`	$\mathbf{PART}(P_2, \texttt{Particles}) \wedge P_1 \subseteq P_2 \wedge$
`Particles[p].pos += f(Cells[c].vel)`	$\mathbf{PART}(P_3, \texttt{Cells}) \wedge$ $\mathbf{image}(P_1, f_1, \texttt{Cells}) \subseteq P_3 \wedge$ $\mathbf{PART}(P_4, \texttt{Particles}) \wedge P_1 \subseteq P_4$ $(f_1 = \texttt{Particles}[\cdot].\texttt{cells})$

Figure 6: Example of constraint inference

Program	Constraints
`for (i in R):`	$\mathbf{PART}(P_1, \texttt{R}) \wedge \mathbf{COMP}(P_1, \texttt{R}) \wedge$
`S[g(i)] += R[i]`	$\mathbf{PART}(P_2, \texttt{S}) \wedge \mathbf{image}(P_1, \texttt{g}, \texttt{S}) \subseteq P_2 \wedge$ $\mathbf{DISJ}(P_1) \wedge \mathbf{PART}(P_3, \texttt{R}) \wedge P_1 \subseteq P_3$

Figure 7: Example with a disjointness predicate on the iteration space

a concrete dependent partitioning operation; the partitioning code can be read directly from the resolved form of the constraints.

- Allowing a separate partition for every region access admits the widest possible range of partitioning strategies, but can result in solutions with multiple equivalent partitions. We unify partition variables with isomorphic constraints to reduce the final number of partitions that need to be created.

The following example illustrates the constraint inference steps for the first loop in Figure 1a.

Example 1. *Algorithm 1 first conjoins the following predicates on a partition symbol P_1 for the iteration space* `Particles` *(line 8):*

$$\mathbf{PART}(P_1, \texttt{Particles}) \wedge \mathbf{COMP}(P_1, \texttt{Particles})$$

The algorithm also maintains an environment that maps each variable to a lambda function that returns an **image** *expression of a region argument. The initial environment at line 7 has a mapping of the loop variable* `p` *to a function $\lambda r.\mathbf{image}(P_1, f_{\mathsf{ID}}, r)$. For the region access* `Particles[p].cell`*, the algorithm introduces a partition symbol P_2 and generates the following constraints (lines 11-13):*

$$\mathbf{PART}(P_2, \texttt{Particles}) \wedge P_1 \subseteq P_2$$

Note that the expression $\mathbf{image}(P_1, f_{\mathsf{ID}}, \texttt{Particles})$ is simplified to P_1. Since the value of this region access is assigned to the variable `c`*, the algorithm updates the environment (lines 14-15), which then becomes the following:*

$$\{\texttt{p} \mapsto \lambda r.\mathbf{image}(P_1, f_{\mathsf{ID}}, r), \texttt{c} \mapsto \lambda s.\mathbf{image}(P_1, f_1, s)\},$$

where f_1 = `Particles[·].cell`. For the uncentered region access `Cells[c].vel`*, the algorithm infers the following constraints on a new partition symbol P_3 (lines 11-13), where the subset constraint has an* **image** *expression in the lower bound:*

$$\mathbf{PART}(P_3, \texttt{Cells}) \wedge \mathbf{image}(P_1, f_1, \texttt{Cells}) \subseteq P_3$$

Finally, the centered reduction `Particles[p].pos += ...` *is handled similarly to other centered accesses, resulting in the partitioning constraint in Figure 6.*

Note that the partitioning constraint in Figure 6 does not have a disjointness predicate on the partition of the iteration space. If the final solution uses a non-disjoint partition of the iteration space, there is redundant computation because some loop iterations are executed multiple times. This redundancy is useful in cases (as demonstrated by Zhou et al. [30]) when recomputing loop iterations on separate nodes is cheaper then the internode communication the redundant computation replaces. In Section 5, we discuss how we can optimize communication from uncentered reductions using an *aliased* (non-disjoint) partition of the iteration space.

However, we do need a disjoint partition of the iteration space when the loop has an uncentered reduction access (lines 16-17 in Algorithm 1). Figure 7 shows an example where an uncentered reduction on the region `S` imposes a disjointness constraint on the partition P_1 of the iteration space `R`. To see why disjointness is mandatory in this case, we need to understand how uncentered reductions are typically handled in distributed memory systems [6, 9, 21]. Unlike centered reductions, which are applied immediately, uncentered reductions require two steps. First, each distributed task allocates a temporary buffer to keep the reduction contribution from each iteration that it owns. Then, temporary buffers are merged, either eagerly or lazily, back to the partitions that the subsequent read accesses use. Because this merge step aggregates all contributions in temporary buffers, each contribution must be counted exactly once to preserve the original semantics. Therefore, the iteration space must be partitioned disjointly in this case.

Algorithm 1 runs in linear time in the size of the program and produces partitioning constraints sound with respect to the semantics of parallelizable loops. These partitioning constraints always have at least one trivial solution, obtained by replacing each subset constraint with an equality.

3 CONSTRAINT SOLVER

In this section, we describe a constraint solver that takes partitioning constraints as input and produces DPL programs as solutions. A *DPL statement* $P = E$ is expressible in the constraint language of Figure 5, and a *DPL program* is just a sequence of DPL statements.

Our constraint solver transforms the input partitioning constraint into a *resolved* form, the constraint conjoined with exactly one equality $P_i = E_i$ for each partition symbol P_i. Once the partitioning constraint is solved, the added equalities form one solution program. In the rest of this section, we explain the algorithm to resolve partitioning constraints and the heuristics to minimize the number of partitions constructed by the output program.

3.1 Resolution

Conceptually, a partitioning constraint C can be resolved by the following procedure:

(1) Synthesize expressions $E_1, \ldots, E_n$ for all partition symbols $P_1, \ldots, P_n$ in C.
(2) Check consistency of the strengthened constraint
$$C \wedge P_1 = E_1 \wedge \ldots \wedge P_n = E_n.$$
(3) If the consistency check fails, go to (1) and synthesize different expressions.

The consistency check in step (2) verifies that each predicate in the constraint is entailed by other predicates or known lemmas of DPL

L1 $\mathbf{PART}(\mathbf{equal}(R), R) \wedge \mathbf{DISJ}(\mathbf{equal}(R)) \wedge \mathbf{COMP}(\mathbf{equal}(R), R)$
L2 $\mathbf{PART}(\mathbf{image}(E, f, R), R)$ L3 $\mathbf{PART}(\mathbf{preimage}(R, f, E), R)$
L4 $\mathbf{PART}(P_1, R) \wedge \mathbf{PART}(P_2, R) \quad (\diamond \in \{\cup, \cap, -\})$
$\implies \mathbf{PART}(P_1 \diamond P_2, R)$
L5 $E_1 \subseteq E_2 \wedge \mathbf{COMP}(E_1, R) \wedge \mathbf{PART}(E_2, R) \implies \mathbf{COMP}(E_2, R)$
L6 $\mathbf{COMP}(E_1, R) \vee \mathbf{COMP}(E_2, R) \implies \mathbf{COMP}(E_1 \cup E_2, R)$
L7 $\mathbf{COMP}(E_1, R_1) \implies \mathbf{COMP}(\mathbf{preimage}(R_2, f, E_1), R_2)$
L8 $\mathbf{DISJ}(E_2) \wedge E_1 \subseteq E_2 \implies \mathbf{DISJ}(E_1)$
L9 $\mathbf{DISJ}(E_1) \vee \mathbf{DISJ}(E_2) \implies \mathbf{DISJ}(E_1 \cap E_2)$
L10 $\mathbf{DISJ}(E_1) \implies \mathbf{DISJ}(E_1 - E_2)$
L11 $\mathbf{DISJ}(E_1 \cup E_2) \implies \mathbf{DISJ}(E_1) \wedge \mathbf{DISJ}(E_2)$
L12 $\mathbf{DISJ}(E_1) \implies \mathbf{DISJ}(\mathbf{preimage}(R, f, E_1))$
L13 $E_1 \subseteq E_3 \wedge E_2 \subseteq E_3 \implies E_1 \cup E_2 \subseteq E_3$
L14 $E_1 \subseteq \mathbf{preimage}(R_1, f, E_2) \wedge \mathbf{PART}(E_2, R_2)$
$\implies \mathbf{image}(E_1, f, R_2) \subseteq E_2$

Figure 8: DPL lemmas for resolution

operators, shown in Figure 8. Any set of expressions that pass this check is a solution that satisfies the input constraint.

All lemmas in Figure 8 are direct consequences from definitions of the DPL operators and properties of sets. The first four lemmas enumerate all possible cases where partitions of a region R can be constructed. Lemmas L5-7 (resp. lemmas L8-12) state when the completeness (resp. disjointness) of a partition is propagated to others.

Algorithm 2 shows the constraint solving algorithm tailored to partitioning constraints inferred by Algorithm 1. This algorithm tries to minimize backtracking due to adding an equation that causes the constraint system to become inconsistent (have no solutions). The solver picks promising candidates using the following insights based on the lemmas in Figure 8:

(1) If a partition symbol P has subset constraints $E_1 \subseteq P, \ldots, E_k \subseteq P$ where each E_i is *closed*, i.e., contains no partition symbol, the union $E_1 \cup \cdots \cup E_k$ of these expressions is a good candidate for P (lemma L13).
(2) The only way to create a fresh disjoint partition is the **equal** operator (lemma L1) and only intersection, difference, and **preimage** operators preserve the disjointness of operands (lemmas L9, L10, and L12). Therefore, a partition symbol with a **DISJ** predicate must be created using only these operators. Likewise, complete partitions can be expressed only by **equal** (lemma L1), union (lemma L6), and **preimage** (lemma L7), or combinations of these operators.
(3) For a subset constraint $E_1 \subseteq E_2$, disjointness "flows" from right to left (lemma L8). When both sides of a subset predicate $E_1 \subseteq E_2$ must be disjoint, the solver resolves all symbols in the expression E_2 and then derives E_1.
(4) The **preimage** operator can produce partitions that satisfy subset constraints containing **image** (lemma L14). Combined with observation (2), these lemmas imply that the solver must use a **preimage** partition to discharge a subset constraint of the form $\mathbf{image}(E_1, \ldots) \subseteq E_2$ when both E_1 and E_2 must be disjoint.

Algorithm 2 can always solve partitioning constraints generated by Algorithm 1: Because Algorithm 1 introduces a fresh partition symbol for the RHS of each added subset constraint, the subset constraints never form a cycle. Thus, the solver can always find a

Algorithm 2: Constraint solving algorithm

```
1  Procedure Solve(C, S):
2      // C is a partitioning constraint to solve.
3      // S is a partial solution found so far.
4
5      // Each call to Solve() picks one remaining constraint, adds
6      // an equality to attempt to solve it, and calls Solve()
7      // recursively to solve the rest of the system.
8      for each P = E ∈ S do
9          // Replace P by E, eliminating P from the constraint C
10         C ← C[P ↦ E]
11     Remove all tautologies E ⊆ E from C
12     for each image(P, f, R) ⊆ E ∈ C for a closed E do
13         // Assume ∃R'.PART(P, R') ∈ C
14         S_next ← Solve(C, S ∧ P = preimage(R', f, E))
15         // If the system is not inconsistent (∅), return the solution
             if S_next ≠ ∅ : return S_next
16     for each P with subset constraints E_i ⊆ P for closed E_i s do
17         S_next ← Solve(C, S ∧ P = ⋃_i E_i))
18         if S_next ≠ ∅ : return S_next
19     // depth(P) ≜ k when E_1 ⊆ ··· ⊆ E_k ⊆ P
20     for d = max({depth(P_i) | P_i ∈ C}), 1 do
21         for each PART(P, R) ∧ DISJ(P) ∈ C s.t. depth(P) = k do
22             S_next ← Solve(C, S ∧ P = equal(R))
23             if S_next ≠ ∅ : return S_next
24     for each COMP(P, R) ∈ C do
25         S_next ← Solve(C, S ∧ P = equal(R))
26         if S_next ≠ ∅ : return S_next
27     // Lemmas in Figure 8 are used for this resolution
28     if ∀C_sub ∈ C.C − C_sub ⟹ C_sub : return S
29     else: return ∅
```

trivial solution that uses **equal** partitions for iteration spaces and has equalities strengthened from all subset constraints. However, this naïve solution is inefficient because it does not reuse partitions from one parallelizable loop in the others. To maximize the partition reuse in the solution, the constraint solver performs *unification* of partition symbols, which is the topic of the next subsection.

The following examples demonstrate how Algorithm 2 resolves partitioning constraints.

Example 2. *Suppose we have this partitioning constraint from Figure 7:*

$$\mathbf{PART}(P_1, \mathsf{R}) \wedge \mathbf{COMP}(P_1, \mathsf{R}) \wedge \mathbf{DISJ}(P_1) \wedge \mathbf{PART}(P_2, \mathsf{S}) \wedge \mathbf{image}(P_1, \mathsf{g}, \mathsf{S}) \subseteq P_2 \wedge \mathbf{PART}(P_3, \mathsf{R}) \wedge P_1 \subseteq P_3.$$

Because P_1 has a **DISJ** *predicate, the solver uses an* **equal** *partition for P_1 (line 22). After substituting P_1 with* **equal**(R)*, the original constraint simplifies to:*

$$\mathbf{PART}(P_2, \mathsf{S}) \wedge \mathbf{image}(\mathbf{equal}(\mathsf{R}), \mathsf{g}, \mathsf{S}) \subseteq P_2 \wedge \mathbf{PART}(P_3, \mathsf{R}) \wedge \mathbf{equal}(\mathsf{R}) \subseteq P_3.$$

Since P_2 and P_3 have closed expressions on the LHS of their subset constraints, the solver simply strengthens them into equalities (line 17) and produces the following solution (after performing common subexpression elimination):

$$P_1 = \mathbf{equal}(\mathsf{R}) \quad P_2 = \mathbf{image}(P_2, \mathsf{g}, \mathsf{S}) \quad P_3 = P_1$$

Algorithm 3: Constraint solver algorithm with unification

```
1  Procedure UnifyAndSolve(C_1 ∧ ... ∧ C_N):
2      // Each C_i is represented by a set of conjuncts
3      Sort C_1, ..., C_N in descending order of |C_i|
4      C ← C_1
5      for i = 2, N do
6          C' ← C_i
7          while C' ≠ ∅ do
8              G ← the next biggest common subgraph in C and C'
9              if G = ∅ :
10                 C ← C ∧ C'
11                 C' ← ∅
12             else:
13                 U ← P'_1 = P_1 ∧ ... ∧ P'_K = P_K induced by G
14                 if Solve(C ∧ C', U) ≠ ∅ :
15                     // Filter out unified terms
16                     C' ← C'[P'_1 ↦ P_1] ··· [P'_K ↦ P_K] − C
17     return Solve(C, ∅)
```

Example 3. *Suppose now we have an extra predicate* $\mathbf{DISJ}(P_2)$ *in the partitioning constraint as follows:*

$$\mathbf{PART}(P_1, \mathsf{R}) \wedge \mathbf{COMP}(P_1, \mathsf{R}) \wedge \mathbf{DISJ}(P_1) \wedge \mathbf{PART}(P_2, \mathsf{S}) \wedge \mathbf{image}(P_1, \mathsf{g}, \mathsf{S}) \subseteq P_2 \wedge \mathbf{DISJ}(P_2) \wedge \mathbf{PART}(P_3, \mathsf{R}) \wedge P_1 \subseteq P_3.$$

Then, the solver notices that P_2*, the RHS of the subset constraint* $\mathbf{image}(P_1, \mathsf{g}, \mathsf{S}) \subseteq P_2$*, must be disjoint, and creates an* **equal** *partition for* P_2 *(line 22) and a* **preimage** *partition for* P_1 *(line 14):*

$$P_2 = \mathbf{equal}(\mathsf{S}) \quad P_1 = \mathbf{preimage}(\mathsf{R}, \mathsf{g}, P_2).$$

The partition symbol P_3 *is resolved similarly to Example 2.*

3.2 Unification

A single unification step strengthens the original constraint by conjoining an equality between *unifiable* partition symbols. Partition symbols are unifiable only when they represent partitions of the same region. The constraint after unification can be further simplified by replacing one of the unified symbols with the other.

Example 4. *The partition symbols* P_1*,* P_2*, and* P_4 *in Figure 6 can be unified as follows:*

$$\mathbf{PART}(P_1, \texttt{Particles}) \wedge \mathbf{COMP}(P_1, \texttt{Particles}) \wedge \mathbf{PART}(P_3, \texttt{Cells}) \wedge \mathbf{image}(P_1, f_1, \texttt{Cells}) \subseteq P_3.$$

Because unification can introduce equalities inconsistent with the original constraint, the constraint after unification might not have any solution. For example, unification can make some subset constraints recursive as follows:

$$\mathbf{PART}(P_1, R) \wedge \mathbf{PART}(P_2, R) \wedge \mathbf{image}(P_1, f, R) \subseteq P_2 \Longleftarrow \mathbf{PART}(P_1, R) \wedge \mathbf{image}(P_1, f, R) \subseteq P_1 \wedge P_1 = P_2.$$

This recursive constraint can be satisfied only by constructing a fixpoint of the function f, which is not expressible in our constraint language. Therefore, the goal of unification is to find a maximal set of unifications that preserves consistency of the partitioning constraint.

Finding all viable unifications requires an exhaustive search in the general case. To make the search efficient, we focus on unifications that reduce the number of subset constraints; intuitively,

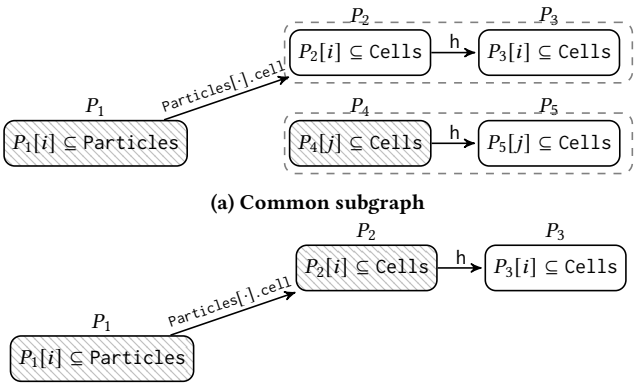


Figure 9: Unification as a common subgraph problem

if unification between partition symbols eliminates some subset constraints, the constraint after unification is no more difficult to resolve than the original one. Such unifications manifest as isomorphic subgraphs in a graph that represents a partitioning constraint. In this *constraint graph* each node corresponds to a partition symbol, an unlabeled edge from P_1 to P_2 represents the subset constraint $P_1 \subseteq P_2$, and an edge labeled with a function symbol f encodes the subset constraint $\mathbf{image}(P_1, f, R) \subseteq P_2$. (Other cases need not be expressed by this graph, because the inference algorithm only generates subset constraints of the two forms.) Isomorphic subgraphs in this graph correspond to partition symbols connected by the same subset constraints (after renaming symbols). Thus, unifying partition symbols in these isomorphic subgraphs also merges multiple subset constraints, one from each subgraph, into one.

Example 5. *Figure 9a shows a constraint graph for the following constraint (predicates are elided):*

$$\cdots \wedge \mathbf{image}(P_1, \texttt{Particles}[\cdot].\texttt{cells}, \texttt{Cells}) \subseteq P_2 \wedge \mathbf{image}(P_2, \mathsf{h}, \texttt{Cells}) \subseteq P_3 \wedge \mathbf{image}(P_4, \mathsf{h}, \texttt{Cells}) \subseteq P_5.$$

In Figure 9a, the subgraph of P_2 *and* P_3 *is isomorphic to that of* P_4 *and* P_5*. The solver unifies* P_2 *and* P_4 *and* P_3 *and* P_5*, with the result shown in Figure 9b.*

Algorithm 3 shows the constraint solver algorithm with unification. The algorithm uses Algorithm 2 to check if the system of constraints after unification is still solvable (line 13). Although finding the largest common subgraph in a constraint graph (line 7) is known to be NP-complete [14], in practice unification is not a significant cost as constraint graphs are small and we do not attempt to find the absolutely maximal common subgraph. Furthermore, the algorithm greedily tries to unify the first few largest subgraphs in a constraint graph (line 3), based on the observation that these subgraphs often contain other smaller subgraphs. In the average case, common subgraphs can be identified simply by constructing a product graph of constraint graphs. Assuming the unification succeeds in a constant number of trials, the asymptotic time complexity of this greedy algorithm is $O(NM^2)$, where N is the number of constraints to unify and M is the number of graph nodes.

```
1 for (i in Y):
2   range = Ranges[i]
3   for (k in range):
4     Y[i] += Mat[k].val * X[Mat[k].ind]
```

(a) SpMV code

```
1 P1 = equal(Y, N)
2 P2 = image(P1, fID, Ranges)
3 P3 = IMAGE(P2, Ranges[·], Mat)
4 P4 = image(P3, Mat[·].ind, X)
```

(b) Synthesized DPL code

Figure 10: SpMV example

3.3 External Constraints

As seen in Section 1, programmers often have invariants on existing partitions used in manually parallelized parts. The constraint solver can exploit these invariants by adding them to the partitioning constraint for a program and holding their partition symbols fixed (no expressions are synthesized for external constraints).

Example 6. *The program in Figure 4 specifies an invariant on partitions* pCells *and* pParticles*, which can be added to the partitioning constraint from Example 5 as follows:*

$$\cdots \wedge \textbf{image}(P_1, \texttt{Particles}[\cdot].\texttt{cells}, \texttt{Cells}) \subseteq P_2 \\ \wedge \textbf{image}(P_2, \texttt{h}, \texttt{Cells}) \subseteq P_3 \wedge \textbf{image}(P_4, \texttt{h}, \texttt{Cells}) \subseteq P_5 \\ \wedge \textbf{image}(\texttt{pParticles}, \texttt{Particles}[\cdot].\texttt{cells}, \texttt{Cells}) \subseteq \texttt{pCells}$$

The solver finds unifications between P_1 and pParticles*, P_2,* pCells*, and P_4, and P_3 and P_5, yielding the following constraint:*

$$\cdots \wedge \textbf{image}(\texttt{pParticles}, \texttt{Particles}[\cdot].\texttt{cells}, \texttt{Cells}) \subseteq \texttt{pCells} \\ \wedge \textbf{image}(\texttt{pCells}, \texttt{h}, \texttt{Cells}) \subseteq P_3.$$

Since the LHS of the subset constraint on P_3 is closed, the solver strengthens it to an equality and eventually produces this solution:

$$P_1 = \texttt{pParticles} \qquad P_2 = P_4 = \texttt{pCells} \\ P_3 = P_5 = \textbf{image}(\texttt{pCells}, \texttt{h}, \texttt{Cells}).$$

4 GENERALIZING IMAGE AND PREIMAGE

Some programs have loops where the iteration space is determined by values of a region, typically arising in sparse matrix algorithms. The SpMV code using Compressed Sparse Row (CSR) format in Figure 10a is one such example. In this code, the matrix is represented by the region Mat where the field val contiguously stores the non-zero values of the matrix and the field ind stores the column indices of those non-zeros. The inner loop at line 3 then iterates over columns of the ith row in the matrix using the value Ranges[i], a pair of lower and upper bounds of indices in Mat.

These loops with *data dependent* iteration spaces require partitioning operators that derive partitions using functions from indices to *sets* of indices. In Figure 10a, the region Ranges maps each iteration of the outer loop to a set of iterations of the inner loop, and thus partitions for regions accessed in this inner loop, such as Mat and X, must be constructed by collecting (and flattening) the image of this map. We can define such DPL operators **IMAGE** and **PREIMAGE** as follows:

$$\textbf{IMAGE}(E, F, R)[i] \triangleq \{(l, v') \in R \mid (k, v) \in E[i] \wedge l \in F(k)\} \\ \textbf{PREIMAGE}(R, F, E)[i] \triangleq \{(l, v') \in R \mid (k, v) \in E[i] \wedge k \in F(l)\}$$

The **image** and **preimage** operators in Section 2 are a special case of these operators; for example, with a lifting $f_\uparrow$ of a function f on indices, where $f_\uparrow(x) = \{f(x)\}$, we have that

$$\textbf{image}(E, f, R) = \textbf{IMAGE}(E, f_\uparrow, R).$$

Our framework can handle **IMAGE** and **PREIMAGE** just like **image** and **preimage** with the following minor modifications:

- Algorithm 1 now handles inner loops with data dependent iteration spaces (which are handled similarly to assignments).
- Lemmas L12 and L14 in Figure 8 do not hold for **IMAGE** and **PREIMAGE**.

The DPL code synthesized for the SpMV code is shown in Figure 10b.

Note that the partitioning strategy in Figure 10b can lead to suboptimal performance when the the number of non-zeros in each row is uneven, because the partition of the matrix is derived from an **equal** partition of Ranges. In this case the user can construct a balanced partition of Ranges using, for example, a graph partitioning heuristic, such as the one proposed by Ravishankar et al. [22], and provide it as an external constraint.

5 OPTIMIZATIONS

As described in Section 2, uncentered reductions on distributed memory systems are implemented using temporary buffers, because different tasks can make changes to the same element, and these changes must be reconciled to ensure the result is correct. Distributed runtime systems, such as Legion [6], require programs to specify which partitions need these buffers. However, using a reduction buffer of the size of the whole partition is often inefficient because the buffering is required only on the part that is accessed by multiple parallel processes. Furthermore, if the partition for uncentered reductions is disjoint, which means each location in the region is updated only by one process, no reduction buffer is necessary. In the rest of this section, we describe two optimizations in the solver to minimize the size of reduction buffers.

5.1 Relaxing Disjointness Requirements for Iteration Spaces

One strategy to synthesize a disjoint partition for uncentered reductions (thereby eliminating the reduction buffer) is to use an **equal** partition for these reductions and derive a **preimage** partition for the iteration space as in Example 3: The solver requires P_2 (the partition symbol for the uncentered reduction in Figure 7) to be a disjoint partition by introducing an extra predicate $\textbf{DISJ}(P_2)$, and the resolution algorithm produces a solution where P_2 is assigned to **equal**(S) and P_1 to a **preimage** partition derived from P_2. With this solution, the loop in Figure 7 need not request a reduction buffer to parallelize its uncentered reductions.

This strategy does not work when a loop has multiple uncentered reductions using different functions. If the solver uses an **equal** partition for these uncentered reductions, then the partition of the iteration space, which must be disjoint because of the uncentered reductions, must contain all preimages of those different functions and the solver cannot prove it to be disjoint using the resolution lemmas. The following example illustrates this issue with multiple uncentered reductions.

Program	Constraints
`for (i in R):`	$\mathbf{PART}(P_1, \mathtt{R}) \wedge \mathbf{COMP}(P_1, \mathtt{R}) \wedge \mathbf{DISJ}(P_1)$
`  S[f(i)] += R[i]`	$\wedge\ \mathbf{PART}(P_2, \mathtt{S}) \wedge \mathbf{image}(P_1, \mathtt{f}, \mathtt{S}) \subseteq P_2$
`  S[g(i)] += R[i]`	$\wedge\ \mathbf{PART}(P_3, \mathtt{S}) \wedge \mathbf{image}(P_1, \mathtt{g}, \mathtt{S}) \subseteq P_3$

(a) Original loop

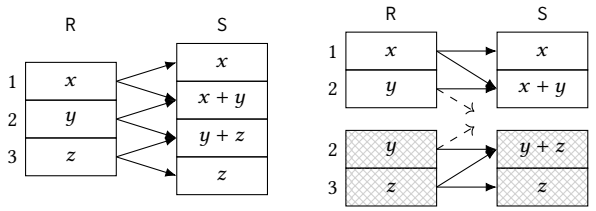
Program	Constraints
`for (i in R):`	
`  if (f(i) in S):`	$\mathbf{PART}(P_1, \mathtt{R}) \wedge \mathbf{COMP}(P_1, \mathtt{R})$ ~~$\wedge \mathbf{DISJ}(P_1)$~~
`    S[f(i)] += R[i]`	$\wedge\ \mathbf{PART}(P_2, \mathtt{S}) \wedge \mathbf{image}(P_1, \mathtt{f}, \mathtt{S}) \subseteq P_2$
`  if (g(i) in S):`	$\wedge\ \mathbf{PART}(P_3, \mathtt{S}) \wedge \mathbf{image}(P_1, \mathtt{g}, \mathtt{S}) \subseteq P_3$
`    S[g(i)] += R[i]`	

(b) Relaxed loop

```
parallel for (p in P1):
  for (i in P1[p]):
    if (f(i) in P2[p]): P2[p][f(i)] += P1[p][i]
    if (g(i) in P3[p]): P3[p][f(i)] += P1[p][i]
```

(c) Parallelized loop

Figure 11: Example with multiple uncentered reductions



(a) Original loop **(b) Parallelized loop**

Figure 12: Example execution of loops in Figure 11

Example 7. *Figure 11a shows the partitioning constraint for a loop with two uncentered reductions. Using an* **equal** *partition for both P_2 and P_3 would lead the solver to an assignment of P_1 to a union of preimages* **preimage**(R, f, ..) *and* **preimage**(R, g, ..)*, which cannot satisfy the predicate* **DISJ**(P_1).

The obvious alternative of assigning a disjoint partition to only one of the uncentered reductions would still require a reduction buffer for the other uncentered reduction. However, the disjointness requirement can be lifted completely by rewriting the loop into a *relaxed* form, shown in Figure 11b. This loop has a guard for each uncentered reduction. In a serial execution these guards are trivial (always true), but when regions used in these guards are replaced by partitions (shown in Figure 11c), the guards prevent contributions in the original loop from being applied multiple times. Therefore, the solver no longer needs a **DISJ** predicate on the iteration space partition and can use the union of preimages, which was not viable before the relaxation. Figure 12 shows how guard conditions work; even though some iteration space elements appear in more than one subregion of the iteration space partition, each iteration contributes to each reduction only once.

This relaxation is not always beneficial, because it introduces redundant computation and extra communication due to overlap among subregions of the iteration space partition, and is not always applicable. We heuristically relax loops only when all loops using the same region as the iteration space can be relaxed.

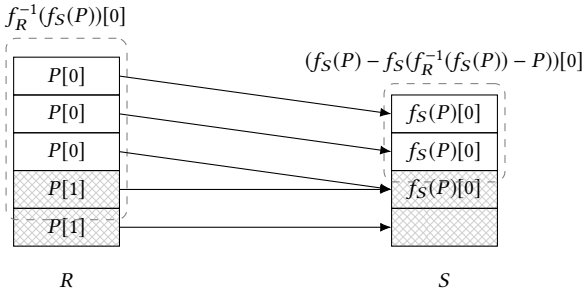


Figure 13: Private sub-partition theorem

5.2 Finding Private Sub-Partitions

In cases when the relaxation is not applied, the optimizer tries to subtract a *private sub-partition* from the reduction partition. A private sub-partition of a partition P is a disjoint partition P_p that satisfies $P_p \subseteq P$. Since the private sub-partition is disjoint, the program need not request a reduction buffer. However, the parallel loop must be modified to account for the fact that now the reduction partition is divided into two parts; if the original reduction partition P is divided into a private sub-partition P_p and the rest $P_s = P - P_p$, then the original parallel loop:

```
parallel for (j in P'):
  for (i in P'[j]):
    P[j][g(i)] += P'[j][i]
```

must be rewritten to:

```
parallel for (j in P'):
  for (i in P'[j]):
    if (g(i) in Pp[j]): Pp[j][g(i)] += P'[j][i]
    else:               Ps[j][g(i)] += P'[j][i]
```

Although there is no general construction of private sub-partitions for a partition, we can use the following theorem when the partition is derived by the **image** operator from another disjoint partition.

Theorem 5.1. *Let $f_R(P)$ and $f_R^{-1}(P)$ be defined as follows:*

$$f_R(P) \triangleq \mathbf{image}(P, f, R) \qquad f_R^{-1}(P) \triangleq \mathbf{preimage}(R, f, P)$$

For a disjoint partition P of a region R, the following expression constructs a private sub-partition of $f_S(P)$ for any f and S:

$$f_S(P) - f_S(f_R^{-1}(f_S(P)) - P).$$

Proof. (Sketch) Each image subregion $f_S(P)[i]$ contains all elements pointed to by those in $P[i]$. Then, the sub-expression $f_R^{-1}(f_S(P))$ extends each subregion $P[i]$ with the elements from the other subregions $P[j]$ $(j \neq i)$ that also point to the subregion $f_S(P)[i]$. Subtracting P from this expanded partition leaves each subregion with only the elements originally from other subregions. Therefore, its image (i.e. $f_S(f_R^{-1}(f_S(P)) - P)$) represents the shared part in the original image partition $f_S(P)$, and thus its complement is a private sub-partition. □

Figure 13 illustrates the private sub-partition construction in Theorem 5.1.

Once the solver identifies a private sub-partition from a partition, a reduction buffer needs to be allocated only for the shared part. This construction can be generalized to cases when the reduction partition consists of multiple image partitions, for which the solver simply takes an intersection of all private sub-partitions in individual image partitions.

6 EVALUATION

We have implemented our constraint-based approach in Regent, a high-level programming language for HPC applications with first-class support for data partitions [23]. Regent detects and enforces data dependencies between tasks by analyzing the relationships between region and partition arguments of tasks. Data movement between data partitions is automatically resolved by Legion [6], the runtime system for Regent. Regent also provides all DPL operators in Figure 5 [26]. The constraint inference algorithm and solver are implemented as an optimization pass in the Regent compiler.

As Regent is a task-based programming language, the inference algorithm examines parallelizable loops in tasks. When parallelizable loops are nested, the outermost loop is chosen as the target of parallelization. The final stage of auto-parallelization is a source-to-source transformation that converts the original program into a form that uses subregions in all region accesses, as illustrated in Figure 1b. All parallelizable loops are also amenable to CUDA code generation supported by the Regent compiler.

Guards introduced by the optimizations in Section 5 can be expensive to check when the subregions are sparse. To amortize the cost the compiler replaces them with a cache that remembers values of guard conditions. Alternatively, we could *split* the loop to statically disambiguate accesses to multiple regions, as Koelbel and Mehrotra [18] and Adve and Mellor-Crummey [2] distinguished accesses to local data from those to non-local data. We decided not to use this transformation because of the potential combinatorial explosion of cases in the output program.

We evaluate our implementation using the SpMV code in Figure 10 as well as four larger Regent programs: Stencil [27], MiniAero [15], Circuit [24], and PENNANT [13]. For the latter Regent programs, we measure weak scaling performance of auto-parallelized versions of these programs and compare them with hand-optimized counterparts. All programs have a "main" loop where they spend most of the execution time, and this main loop consists only of parallelizable loops. The hand-optimized versions have already been optimized for scalability in previous work [20, 24].

All experiments were performed on Piz Daint [1], a Cray X50 system; each compute node is equipped with one Intel Xeon E5-2690 CPU with 12 physical cores, one NVIDIA Tesla P100, and 64GB of system memory.

Table 1 presents a breakdown of compilation time for benchmark programs. The table also shows the size of each program in terms of the number of auto-parallelized loops and total compilation times of hand-optimized counterparts as a baseline. The constraint inference and solver algorithms and the rewriting to parallel code constitute less than 10 percent of the total compilation time, and the binary code generation is a dominant component. Note that the baseline does not strictly match the time for generating a binary from the auto-parallelized code, because the auto-parallelizer produces a program that is different from the hand-optimized counterpart.

Figure 14 summarizes the weak scaling performance of benchmark programs. Performance numbers in plots were measured once programs reached a steady state. All computation tasks running within the measurement window used only GPUs.

6.1 SpMV Microbenchmark

Figure 14a shows weak scaling performance of the SpMV code in Figure 10. In the experiments, we use a diagonal matrix where each row has a fixed number of non-zeros. With this balanced synthetic matrix the auto-parallelized SpMV code achieved 99% parallel efficiency on 256 nodes.

6.2 Stencil

Stencil is a 9-point stencil program for a 2D grid. The stencil consists of a center and eight neighbor points, two for each direction in 2D space. The uncentered access for each neighbor point corresponds to a distinct subset constraint, for which the constraint solver synthesizes an **`image`** partition of an affine function.

Figure 14b shows performance of the hand-optimized code and the auto-parallelized code. The auto-parallelized version achieves 93% parallel efficiency on 256 nodes, whereas the parallel efficiency of the hand-optimized version is 98%. In terms of absolute performance, the auto-parallelized version is slower than the hand-optimized version by 3% on average. The discrepancy is due to an optimization for communication manually applied to the hand-optimized version: The code maintains a copy of the halo part in its own region, which consolidates inter-node data movement for halo exchanges in each direction into a single transfer, while the eight partitions used by the auto-parallelized version require two data transfers per direction.

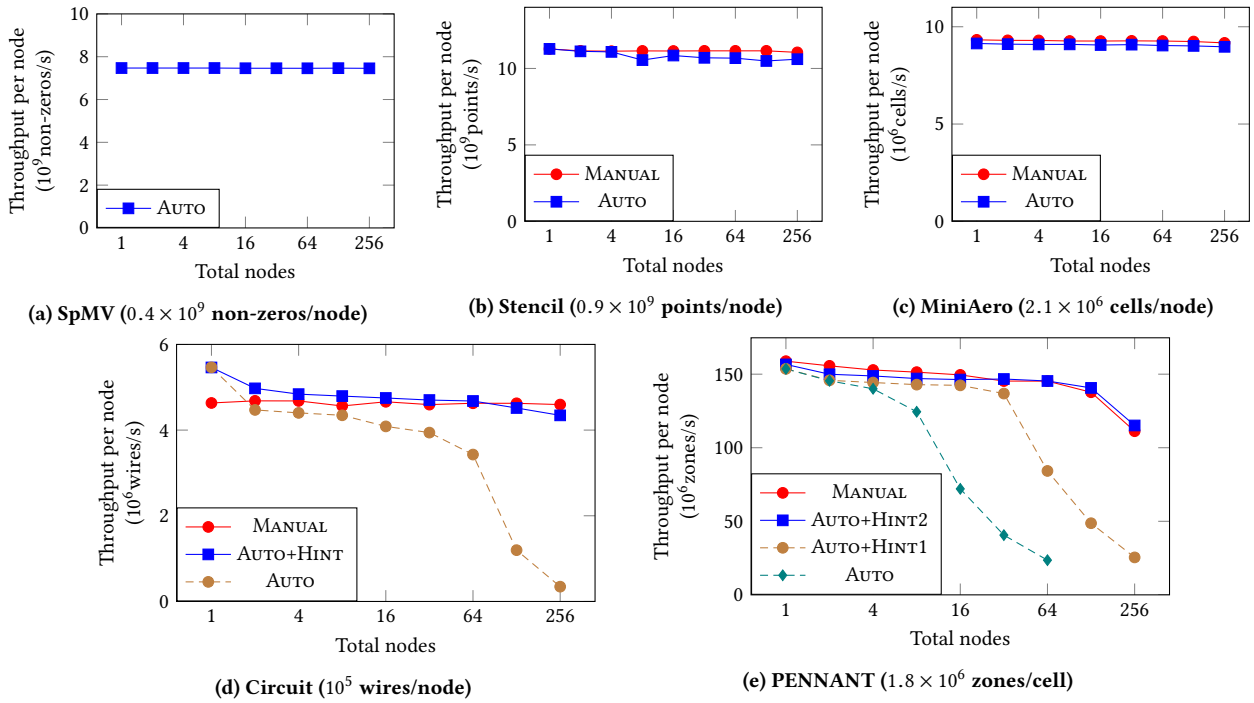
6.3 MiniAero

MiniAero is a proxy application that solves the Navier-Stokes equation for compressible flows. MiniAero uses a 3D hexahedron mesh with faces shared between neighboring hexahedron cells. The simulation calculates flux between cells pointed to by each face. All tasks in the simulation loop read face properties and update cell properties via uncentered reductions using pointers in each face, similar to Figure 11a; the optimizer applies the optimization in Section 5.1 to these reductions to eliminate reduction buffers completely.

Figure 14c shows performance of hand-optimized and auto-parallelized versions of MiniAero. Both achieve 98% parallel ef-

	SpMV	Stencil	Circuit	MiniAero	PENNANT
Constraint inference	1.7ms	5.0ms	28.4ms	58.5ms	110.7ms
Constraint solver	1.7ms	4.0ms	4.3ms	5.8ms	13.1ms
Code rewrite	49ms	0.3s	0.3s	1.6s	1.9s
Binary generation	2.3s	6.5s	7.2s	22.8s	31.4s
Total	2.4s	6.8s	7.5s	24.4s	33.4s
Num. parallel loops	1	2	3	26	37
Baseline	N/A	8.7s	8.3s	22.7s	27.6s

Table 1: Compilation time breakdown



(a) SpMV (0.4×10^9 non-zeros/node) **(b) Stencil (0.9×10^9 points/node)** **(c) MiniAero (2.1×10^6 cells/node)**

(d) Circuit (10^5 wires/node) **(e) PENNANT (1.8×10^6 zones/cell)**

Figure 14: Weak scaling performance

ficiency on 256 nodes, but the auto-parallelized version is 2% slower on average. This difference is explained by different mesh generators used in the two versions: The mesh generator in the hand-optimized code duplicates faces when they point to cells from two different subregions so that faces surrounding cells in each subregion can be contiguously indexed. On the other hand, because the auto-parallelized code uses a mesh generated for sequential execution, faces in each face subregion can be non-contiguously indexed, leading to a small performance degradation in CUDA kernels generated by the Regent compiler.

6.4 Circuit

Circuit simulates electric currents along wires in an unstructured circuit graph. Each wire has pointers to incoming and outgoing nodes, which are used for uncentered read accesses and reductions to the region of nodes. Circuit graphs are randomly generated in a way that circuit nodes form clusters; a maximum of 20% of wires connect nodes in two different clusters.

To showcase the ability to exploit external constraints, we use the existing parallel circuit graph generator that produces inputs to Circuit and auto-parallelize computation tasks with and without a user constraint describing the initial partition of nodes produced by the generator. Figure 14d compares these configurations (Auto+Hint and Auto) with the hand-optimized code (Manual). Without the user constraint, the auto-parallelized code uses an equal partition of circuit nodes, which makes the code match the hand-optimized one within 5% only up to eight nodes. The circuit generator is designed to simulate sparsely connected components, and thus assigns the first 1% of entries in the region of circuit nodes to those connected to nodes in other clusters. As a result, the equal partition of the region of nodes puts all these "shared" nodes in one subregion, making the task using this subregion a communication bottleneck.

To fix this performance issue, we give the solver an interface constraint describing the externally computed circuit partition. The parallel circuit generator uses two partitions of the region `rn` of circuit nodes, `pn_private` for private nodes and `pn_shared` for shared nodes, and the union of these partitions is a disjoint, complete partition of `rn`, as expressed by the following user constraint:

$$\textbf{DISJ}(\texttt{pn\_private} \cup \texttt{pn\_shared}) \wedge \textbf{COMP}(\texttt{pn\_private} \cup \texttt{pn\_shared}, \texttt{rn})$$

With this user constraint, the performance of the auto-parallelized code stays within 5% of the hand-optimized code on 256 nodes and shows better performance up to 64 nodes. The latter is due to the fact that the hand-optimized code always requests reduction buffers for the entire subset reserved for shared circuit nodes even when only a few nodes in this subset are shared, whereas the auto-parallelized code computes tight private sub-partitions to reduce the size of reduction buffers for uncentered reductions.

6.5 PENNANT

PENNANT is a proxy application for Lagrangian hydrodynamics on 2D meshes. Each polygonal *zone* in the mesh consists of triangular *sides*; each pair of sides share two *points*. Each side has five pointers used in uncentered accesses: two pointers to the previous and next neighbor sides in the same zone, one to the zone, and the last two to points at the vertices of the zone.

Similar to the random circuit generator in Circuit, PENNANT's mesh generator separates points shared by sides in different subregions from those owned by a single subregion of sides. Shared points reside in the initial entries in the region of points. Because of

this separation, performance of the auto-parallelized code without any user constraint (Auto in Figure 14e) keeps up with the hand-optimized one (Manual) only up to four nodes and then drops due to the communication bottleneck.

After adding an external constraint describing the partitioning of points, the auto-parallelized code matches the hand-optimized one within 6% up to 32 nodes (Auto+Hint1). The auto-parallelized code still struggles to scale beyond 64 nodes, but for a different reason: the partitions constructed by the synthesized DPL code exhibit sparsity patterns inefficiently handled by the underlying runtime system, even though they are equivalent to those used in the hand-optimized one in terms of induced inter-node communication. We circumvent this issue by providing additional constraints to guide the solver to synthesize simpler DPL code as follows:

- We reused the existing disjoint, complete partitions `rs_p` and `rz_p` of sides and zones, respectively. The parallel mesh generator guarantees that zones pointed to by the sides in the ith subregion of `rs_p` are all contained in the ith subregion of `rz_p` (i.e.,$\mathbf{image}(\texttt{rs\_p}, \texttt{rs}[\cdot].\texttt{mapsz}, \texttt{rz}) \subseteq \texttt{rz\_p}$).
- Additionally, each side s has all its neighbor sides accessed via `rs`$[s]$`.mapss3` and `rs`$[s]$`.mapss4` in the same subregion:
$$\mathbf{image}(\texttt{rs\_p}, \texttt{rs}[\cdot].\texttt{mapss3}, \texttt{rs}) \subseteq \texttt{rs\_p} \\ \wedge\ \mathbf{image}(\texttt{rs\_p}, \texttt{rs}[\cdot].\texttt{mapss4}, \texttt{rs}) \subseteq \texttt{rs\_p}$$
Although these constraints are recursive, the solver can still check the consistency as long as a satisfying partition (`rs_p`) is provided.
- Finally, the mesh generator creates a partition `rp_p_private` of private points, which can be used as a private sub-partition for uncentered reductions using `rs`$[\cdot]$`.mapsp1`:
$$\mathbf{preimage}(\texttt{rs}, \texttt{rs}[\cdot].\texttt{mapsp1}, \texttt{rp\_p\_private}) \subseteq \texttt{rs\_p}$$

With these additional user constraints there is no noticeable difference between the auto-parallelized and hand-optimized versions (Auto+Hint2). This example shows the constraint interfaces' ability to provide extra information to gracefully deal with cases where the auto-parallelizers' heuristics do not quite match reality. Writing the additional constraints is still much easier than parallelizing the code by hand, and preserves the option of parallelizing the code in a different way in a different context or after further improvements in the underlying runtime system.

7 RELATED WORK

Many program analysis problems can be reduced to constraint solving problems for which off-the-shelf solvers exist [4, 10, 12, 28]. The theory of first-class data partitions, however, is not a standard theory, nor is it obvious how to convert it into one, because the solutions of our constraints are functional programs with a special set of function primitives (the DPL operators). One of our contributions is the formalization of automatic parallelization as a space of possible data partitionings captured by a system of constraints, together with an algorithm for resolving those constraints.

High Performance Fortran (HPF) [17] and its predecessors [8, 16] have pioneered the idea of configurable auto-parallelization for distributed memory machines. These systems have a similar goal to our work; they provide control over data partitioning via *data distribution*, an annotation language for describing primary data partitions. The compiler then infers non-local data accesses in each rank and inserts communication and synchronization to preserve sequential semantics. However, data distributions give users limited control over data partitioning, making it challenging to compose programs and fix performance issues, because "distributions were not themselves data objects" [17]. The key discovery of our work is that first-class data partitions, which were then considered infeasible due to the runtime overhead, can elegantly solve issues with which the compiler-based systems like HPF would struggle.

Irregular accesses are a major challenge in distributed memory code generation. Techniques for handling irregular accesses in auto-parallelization, such as the Inspector/Executor (I/E) method [21, 22, 29], compilation techniques for OpenMP [5, 19], and the remapping operator in ZPL [11], commonly depend on meta-programmed partitioning code that tracks data partitions in some custom data structure. Although these techniques can be effective, the decisions and heuristics made in the meta-programmed partitioning code are opaque to programmers, and thus the code is hard to understand and compose. On the other hand, our work demonstrates that a programming language with native data partitions makes auto-parallelization of programs with irregular accesses transparent and composable.

The sparse polyhedral framework [25], which is used as a foundation for the I/E method, is similar in spirit to our approach; using sparse polyhedrons as a high-level abstraction, the framework facilitates composition of auto-generated inspector code. Sparse polyhedrons are also useful for applying compiler transformations to the inspector code. However, sparse polyhedrons are still internal to the compiler and thus cannot be used as an interface for mixing the inspector code with the manually parallelized code, whereas partitions in our approach are a user-facing interface for configuring the auto-parallelization process. We believe that the two approaches are complementary to each other.

Distributed code generation for affine programs has been well studied [2, 7, 17]. The optimizations for affine cases require no fundamental changes to our framework and can be easily incorporated in partitioning operator implementations specific to affine cases.

8 CONCLUSION

In this paper we have presented a constraint-based approach to data partitioning. Our approach captures conditions under which the program can be correctly parallelized as partitioning constraints, and synthesizes partitioning code that satisfies these constraints. Using partitioning constraints as a specification also allows us to compose auto-parallelized code with manually parallelized parts in one program. For a set of benchmark programs, auto-parallelized versions showed performance comparable to hand-optimized counterparts with much less programmer effort.

ACKNOWLEDGMENTS

This material is based upon work supported by the Department of Energy National Nuclear Security Administration under Award Number DE-NA0002373-1, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Finally, this work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID d80.

REFERENCES

[1] 2018. Piz Daint & Piz Dora - CSCS. http://www.cscs.ch/computers/piz_daint.

[2] Vikram S. Adve and John M. Mellor-Crummey. 1998. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*. 186–198.

[3] Alexander Aiken. 1999. Introduction to Set Constraint-Based Program Analysis. *Sci. Comput. Program.* 35, 2 (1999), 79–111.

[4] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*.

[5] Ayon Basumallik and Rudolf Eigenmann. 2006. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 119–128.

[6] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Supercomputing (SC)*.

[7] Uday Bondhugula. 2013. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures. In *Supercomputing (SC)*. ACM, 33.

[8] Barbara M. Chapman, Piyush Mehrotra, and Hans P. Zima. 1992. Programming in Vienna Fortran. *Scientific Programming* 1, 1 (1992), 31–50. https://doi.org/10.1155/1992/258136

[9] J. Davison de St.Germain, J. McCorquodale, S.G. Parker, and C.R. Johnson. 2000. Uintah: a massively parallel problem solving environment. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*. 33–41.

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*.

[11] Steven J. Deitz, Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. 2003. The Design and Implementation of a Parallel Array Operator for the Arbitrary Remapping of Data. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*.

[12] Niklas Eén and Niklas Sörensson. [n.d.]. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*.

[13] Charles R. Ferenbaugh. 2014. PENNANT: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience* (2014).

[14] M. R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

[15] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2009. *Improving Performance via Mini-applications*. Technical Report SAND2009-5574. Sandia National Laboratories.

[16] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. 1992. Compiling Fortran D for MIMD Distributed Memory Machines. *Commun. ACM* 35, 8 (1992), 66–80.

[17] Ken Kennedy, Charles Koelbel, and Hans Zima. 2007. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. ACM, 7–1.

[18] Charles Koelbel and Piyush Mehrotra. 1991. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Trans. Parallel Distrib. Syst.* 2, 4 (1991), 440–451.

[19] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff. 2012. A Hybrid Approach of OpenMP for Clusters *(PPoPP)*. ACM, 75–84.

[20] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. 2018. Dynamic Tracing: Memoization of Task Graphs for Dynamic Task-Based Runtimes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2018*.

[21] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. Distributed Memory Code Generation for Mixed Irregular/Regular Computations *(PPoPP)*. ACM, 65–75.

[22] Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2012. Code Generation for Parallel Execution of a Class of Irregular Loops on Distributed Memory Systems. In *Supercomputing (SC)*.

[23] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: a high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015*. 81:1–81:12.

[24] Elliott Slaughter, Wonchan Lee, Sean Treichler, Wen Zhang, Michael Bauer, Galen Shipman, Patrick McCormick, and Alex Aiken. 2017. Control Replication: Compiling Implicit Parallelism to Efficient SPMD with Logical Regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017*.

[25] Michelle Mills Strout, Mary W. Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934.

[26] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. 2016. Dependent partitioning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 344–358.

[27] Rob F. Van der Wijngaart and Timothy G. Mattson. 2014. The Parallel Research Kernels. In *HPEC*. 1–6.

[28] Sven Verdoolaege. 2010. Isl: An Integer Set Library for the Polyhedral Model. In *Proceedings of the Third International Congress Conference on Mathematical Software (ICMS'10)*.

[29] Reinhard von Hanxleden, Ken Kennedy, Charles Koelbel, Raja Das, and Joel H. Saltz. 1992. Compiler Analysis for Irregular Problems in Fortran D. In *Languages and Compilers for Parallel Computing, 5th International Workshop, New Haven, Connecticut, USA, August 3-5, 1992, Proceedings*. 97–111.

[30] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. 2012. Hierarchical Overlapped Tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*.