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  
eslaughter@slac.stanford.edu

Alex Aiken  
Stanford University  
aiken@cs.stanford.edu

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
for (p in Particles):
    c = Particles[p].cell
    Particles[p].pos += f(Cells[c].vel, Cells[h(c)].vel)
for (c in Cells):
    Cells[c].vel += g(Cells[c].acc, Cells[h(c)].acc)

(a) Parallelizable loops

1 task T1(Particles, Cells1, Cells2):
   for (p in Particles):
      c = Particles[p].cell
      Particles[p].pos += f(Cells[c].vel, Cells[h(c)].vel)
   for (c in Cells):
      Cells[c].vel += g(Cells[c].acc, Cells[h(c)].acc)
2 parallel for (i in P):
   T1(P[i][j], P[i][j])
3 parallel for (j in P):
   T2(P[j][j], P[j][j])

(b) Loops parallelized using tasks

P2[i] \subseteq Cells\quad P1 \subseteq Cells
P3[j] \subseteq Cells\quad P4 \subseteq Cells

\begin{figure}
\centering
\begin{tikzpicture}[auto, node distance=1.5cm, thick, main node/.style={circle,fill=blue!20,draw,minimum size=5mm},
                    fnode/.style={circle,fill=red!20,draw,minimum size=5mm},
                    hnode/.style={circle,fill=green!20,draw,minimum size=5mm},
                    i node/.style={circle,fill=blue!20,draw,minimum size=5mm},
                    j node/.style={circle,fill=red!20,draw,minimum size=5mm},
                    k node/.style={circle,fill=green!20,draw,minimum size=5mm},
                    edge/.style={->,> = latex'}]
    \node [main node] (P2) {P2[i] \subseteq Cells};
    \node [fnode] (h) [right of=P2] {};\node [i node] (i1) [above of=h, yshift=-1cm] {};\node [i node] (i2) [below of=h, yshift=1cm] {};
    \node [j node] (j1) [right of=i1] {};\node [j node] (j2) [right of=i2] {};
    \node [main node] (P1) [left of=P2] {P1 \subseteq Cells};
    \node [hnode] (h) [below of=P1] {};\node [i node] (i1) [above of=h, yshift=-1cm] {};\node [i node] (i2) [below of=h, yshift=1cm] {};
    \node [j node] (j1) [right of=i1] {};\node [j node] (j2) [right of=i2] {};
    \node [main node] (P3) [right of=P2] {P3[j] \subseteq Cells};
    \node [fnode] (h) [right of=P3] {};\node [i node] (i1) [above of=h, yshift=-1cm] {};\node [i node] (i2) [below of=h, yshift=1cm] {};
    \node [j node] (j1) [right of=i1] {};\node [j node] (j2) [right of=i2] {};
    \node [main node] (P4) [left of=P3] {P4[j] \subseteq Cells};
    \node [hnode] (h) [below of=P4] {};\node [i node] (i1) [above of=h, yshift=-1cm] {};\node [i node] (i2) [below of=h, yshift=1cm] {};
    \node [j node] (j1) [right of=i1] {};\node [j node] (j2) [right of=i2] {};
    \node [main node] (P5) [below of=P3] {P5 \subseteq Cells};
    \node [fnode] (h) [right of=P5] {};\node [i node] (i1) [above of=h, yshift=-1cm] {};\node [i node] (i2) [below of=h, yshift=1cm] {};
    \node [j node] (j1) [right of=i1] {};\node [j node] (j2) [right of=i2] {};
    \draw [->] (P1) -- (i1) node [midway,above] {\text{Particles}} node [midway,below] {\text{Partitions}};
    \draw [->] (P2) -- (i2) node [midway,above] {\text{Particles}} node [midway,below] {\text{Partitions}};
    \draw [->] (P3) -- (j1) node [midway,above] {\text{Cells}} node [midway,below] {\text{Cells}};
    \draw [->] (P4) -- (j2) node [midway,above] {\text{Cells}} node [midway,below] {\text{Cells}};
    \end{tikzpicture}
\caption{Partitioning constraints}
\end{figure}

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 P1 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.

The parallel program in Figure 1b preserves the semantics of the sequential program in Figure 1a provided the partitions P1, . . . , P5 are legal, i.e., when they make the following indirect accesses safe:

- Cells[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 P1, . . . , P5 in Figure 1b are legal. Each node in the graph corresponds to a partition. The node labeled with P1 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 P1 and P4 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 P2 to P3, labeled with the function h, requires that each subregion P3[j] contain the image of P2[j] under h, that is, \( \forall (k, v) \in P2[j]. \exists v'. (h(k), v') \in P3[j] \). The edge from P4 to P5 describes the same constraint but on P4 and P5. The other edge between P1 and P2 is interpreted similarly:

\( \forall (k, v) \in P1[i]. \exists v'. (Particles[k].cell, v') \in P2[i] \)

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 P2, P3, and P5 from equal partitions of P1 and P4; the equal operator creates a complete partition of a region with (approximately) equal size subregions. Partitions P2, P3 and P5 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 $P_3 = \text{image}(P_2, h, \text{Cells})$ creates $P_3 = \langle h(r_1), \ldots, h(r_n) \rangle$, where $h(r_i) \subseteq \text{Cells}$. Figure 3a gives a visual representation of the $\text{image}$ 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 $P_3$ and $P_4$. Note that $P_3$ is assigned a complete partition even though the partitioning constraint does not require it to be complete; as long as $P_3$ contains the image of $\text{Particles}[:].\text{cell}$, it can have extra elements. To construct the partition $P_3$ from the already defined $P_2$, program B uses the $\text{preimage}$ operator. As illustrated in Figure 3b, $\text{preimage}$ takes an existing partition of the region on the righthand side and constructs a compatible partition using the $\text{preimage}$ 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 $\langle h^{-1}(r_1), \ldots, h^{-1}(r_n) \rangle$. Finally, $P_3$ and $P_5$ are computed using the image of $P_2$ under $h$.

Without any prior knowledge about Cells and $\text{Particles}$, 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' $f(i) = (i + 1) \% 5$,

![Figure 3: Image and preimage operators](image)

![Figure 4: Example with a user-provided constraint](image)
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 \( \text{PART}(E, R) \) means that \( E \) is a partition of the region \( R \); i.e., each subregion \( E[i] \) must be a subset of the region \( R \):

\[
\text{PART}(E, R) \equiv \forall i. E[i] \subseteq R.
\]

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

\[
\text{DISJ}(E) \equiv E[i] \cap E[j] = \emptyset \quad \text{when} \ i \neq j
\]

\[
\text{COMP}(E, R) \equiv \bigcup_i E[i] = R
\]

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

\[
E_1 \subseteq E_2 \equiv \forall i. E_1[i] \subseteq E_2[i]
\]

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.,

\[
E_1 \subseteq E_2 \equiv E_2 \subseteq E_1.
\]

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 \( i \)th subregion is a union of the \( i \)th subregions of the operands:

\[
(E_1 \cup E_2)[i] \equiv E_1[i] \cup E_2[i], \quad \text{where} \ \diamond \in \{\cup, \cap, -\}
\]

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

\[
\text{image}(E, f, R)i \equiv \{ (f(k), k') \in R \mid (k, v) \in E[i] \}.
\]

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

\[
\text{preimage}(R, f, E)i \equiv \{ (k, v) \in R \mid (f(k), v) \in E[i] \}.
\]

Again, Figure 3 visualizes the \text{image} and \text{preimage} operators for an example function \( f \). Lastly, the \text{equal} operator creates partitions without using any other partitions; the expression \( \text{equal}(R) \) 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. \( \text{Procedure Infer}(\text{loop}) \):
2. \( \quad // \text{Assume loop has the form} \ (i \ \text{in} \ R) : \text{body} \)
3. \( \quad // \text{Assume body is normalized} \)
4. \( \quad // \text{The function} f_{ID} \text{is the identity function, i.e.,} f_{ID}(x) = x. \)
5. \( \quad // \text{Note that} \ \text{image}(P_{i}, f_{ID}, R) = P_{i}. \)
6. \( \quad \text{Env} \leftarrow \{ i \mapsto \lambda r. \text{image}(P_{i}, f_{ID}, r) \} \) \quad (P_{i} \text{ is fresh})
7. \( \quad C \leftarrow \text{PART}(P_{i}, R) \land \text{COMP}(P_{i}, R) \)
8. \( \text{for each statement} \ s \in \text{body} \text{do} \)
9. \( \quad // \text{Region accesses appear only in statements of these forms:} \)
10. \( \quad \text{if} \ s \ \text{is} \ y = S[x] \text{ or} S[x] = y \text{ or} S[x] \Rightarrow y : \)
11. \( \quad \quad E \leftarrow \text{Env}(S) \)
12. \( \quad \quad C \leftarrow C \land \text{PART}(P, S) \land E \subseteq P \) \quad (P is fresh)
13. \( \quad \quad \text{Env} \leftarrow \text{Env} \cup \{ y \mapsto \lambda r. \text{image}(E, S[:], r) \} \)
14. \( \quad \quad \text{else} \ s \ \text{is} \ S[x] \Rightarrow y \text{ and} E \neq P_{k} : \)
15. \( \quad \quad \quad C \leftarrow C \land \text{DISJ}(P_{k}) \)
16. \( \quad \quad \text{else} \ s \ \text{is} \ y = f(x) : \)
17. \( \quad \quad \quad \text{Env} \leftarrow \text{Env} \cup \{ y \mapsto \lambda r. \text{image}(\text{Env}(x)(r), f, r) \} \)
18. \( \quad \quad \text{else} \ s \ \text{is} \ y = x : \)
19. \( \quad \quad \quad \text{Env} \leftarrow \text{Env} \cup \{ y \mapsto \text{Env}(x) \} \)

- Region accesses are either \text{centered} or \text{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 \text{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

\[
\text{for} \ (i \ \text{in} \ R) : \ldots
\]

for some region \( R \) (the \text{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 different operator.
- A region with an uncentered read (e.g., \( \ldots = R[S[e]] \)) 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
A Constraint-Based Approach to Automatic Data Partitioning

We unify partition variables with isomorphic constraints to reduce the final number of partitions that need to be created. This 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 \( \text{Particles} \) (line 8):

\[
\text{PART}(P_1, \text{Particles}) \land \text{COMP}(P_1, \text{Particles})
\]

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 \( \lambda \text{image}(P_1, f_{ID}, r) \). For the region access \( \text{Particles}[p].\text{cell} \), the algorithm introduces a partition symbol \( P_2 \) and generates the following constraints (lines 11-13):

\[
\text{PART}(P_2, \text{Particles}) \land P_1 \subseteq P_2
\]

Note that the expression \( \text{image}(P_1, f_{ID}, \text{Particles}) \) 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:

\[
(p \mapsto \lambda \text{image}(P_1, f_{ID}, r), c \mapsto \lambda s.\text{image}(P_1, f_i, s)),
\]

where \( f_i = \text{Particles}[\cdot].\text{cell} \). For the uncentered region access \( \text{Cells}[c].\text{vel} \), 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:

\[
\text{PART}(P_3, \text{Cells}) \land \text{image}(P_1, f_i, \text{Cells}) \subseteq P_3
\]

Finally, the centered reduction \( \text{Particles}[p].\text{pos} += \ldots \) 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 than 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**

\[
C \land P_1 = E_1 \land \ldots \land P_n = E_n.
\]

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.
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 \( \text{image}(E_1, \ldots) \subseteq E_2 \) 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 trivial solution that uses equal partitions for iteration spaces and has equalities strengthened from all subset constraints. However, this naive 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:

\[
\text{PART}(P_1, R) \land \text{COMP}(P_1, R) \land \text{DISJ}(P_1) \land \text{PART}(P_2, S) \\
\land \text{image}(P_1, g, S) \subseteq P_2 \land \text{PART}(P_3, R) \land P_1 \subseteq P_3.
\]

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

\[
\text{PART}(P_2, S) \land \text{image}(\text{equal}(R), g, S) \subseteq P_2 \\
\land \text{PART}(P_3, R) \land \text{equal}(R) \subseteq P_3.
\]

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):

\[
P_1 = \text{equal}(R) \quad P_2 = \text{image}(P_2, g, S) \quad P_3 = P_1
\]
Algorithm 3: Constraint solver algorithm with unification

```plaintext
1 Procedure UnifyandSolve(C1 ∧ . . . ∧ CN):
2    // Each Cj is represented by a set of conjuncts
3    Sort C1, . . . , CN in descending order of |Cj|
4    C ← C1
5    for i = 2, N do
6        C* ← Ci
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′ K = Pk induced by G
14                    if Solve(C ∧ C′, U) # ∅:
15                        // Filter out unified terms
16                        C′ ← C′[P′ 1 → P1] · · · [P′ K → PK] - C
17            return Solve(C, ∅)
```

**Example 3.** Suppose now we have an extra predicate DISJ(P2) in the partitioning constraint as follows:

\[
\text{PART}(P1, R) \land \text{COMP}(P1, R) \land \text{DISJ}(P1) \land \text{PART}(P2, S) \land \text{image}(P1, g, S) \subseteq P2 \land \text{DISJ}(P2) \land \text{PART}(P2, R) \land P1 \subseteq P3.
\]

Then, the solver notices that P2, the RHS of the subset constraint \(\text{image}(P1, g, S) \subseteq P2\), must be disjoint, and creates an equal partition for P2 (line 22) and a preimage partition for P1 (line 14):

\[
P2 = \text{equal}(S) \quad P1 = \text{preimage}(R, g, P2).
\]

The partition symbol P3 is resolved similarly to Example 2.

### 3.2 Unification

A single unification step strengthens the original constraint by conjointing 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 P1, P2, and P4 in Figure 6 can be unified as follows:

\[
\text{PART}(P4, \text{Particles}) \land \text{COMP}(P4, \text{Particles}) \land \text{PART}(P4, \text{Cells}) \land \text{image}(P4, f, \text{Cells}) \subseteq P4.
\]

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:

\[
\text{PART}(P1, R) \land \text{PART}(P2, R) \land \text{image}(P1, f, R) \subseteq P2 \iff \text{PART}(P1, R) \land \text{image}(P1, f, R) \subseteq P1 \land P1 = P2.
\]

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, 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 P1 to P2 represents the subset constraint P1 ⊆ P2, and an edge labeled with a function symbol f encodes the subset constraint \(\text{image}(P1, f, R) \subseteq P2\). (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.

**Figure 9:** Unification as a common subgraph problem

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

\[
\cdots \land \text{image}(P1, \text{Particles}[\cdot].\text{Cells}, \text{Cells}) \subseteq P2 \land \text{image}(P3, h, \text{Cells}) \subseteq P3 \land \text{image}(P4, h, \text{Cells}) \subseteq P5.
\]

In Figure 9a, the subgraph of P2 and P3 is isomorphic to that of P4 and P5. The solver unifies P2 and P4 and P3 and P5, 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.
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:

\[
\begin{align*}
\forall (i) \in Y: & \quad \text{range} \ = \ \text{Ranges}[i] \\
\forall (k) \in \text{range}: & \quad Y[i] \ += \ \text{Mat}[k].\text{val} \ast X[\text{Mat}[k].\text{ind}]
\end{align*}
\]

(a) SpMV code

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>for (i in Y)</td>
</tr>
<tr>
<td>2</td>
<td>range = Ranges[i]</td>
</tr>
<tr>
<td>3</td>
<td>for (k in range)</td>
</tr>
<tr>
<td>4</td>
<td>Y[i] += Mat[k].val * X[Mat[k].ind]</td>
</tr>
</tbody>
</table>

(b) Synthesized DPL code

Figure 10: SpMV example

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 1st 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:

\[
\begin{align*}
\text{IMAGE}(E, E', R)[i] & \triangleq \{l, v' \in R | (k, v) \in E[i] \land l \in E'(k)\} \\
\text{PREIMAGE}(R, E, E')[i] & \triangleq \{l, v' \in R | (k, v) \in E[i] \land k \in E'(I)\}
\end{align*}
\]

The IMAGE and PREIMAGE operators in Section 2 are a special case of these operators; for example, with a lifting \(f_1\) of a function \(f\) on indices, where \(f_1(x) = \{f(x)\}\), we have that

\[
\text{IMAGE}(E, f, R) = \text{IMAGE}(E, f_1, R).
\]

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 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 \(\text{DISJ}(P_2)\), and the resolution algorithm produces a solution where \(P_2\) is assigned to \(\text{DISJ}(S)\) and \(P_2\) 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.
A Constraint-Based Approach to Automatic Data Partitioning

Program Constraints

for (i in R):
  PART(P_i, R) ∧ COMP(P_i, R) ∧ DISJ(P_i)
S[f(i)] += R[i] ∧ PART(P_f, S) ∧ image(P_f, f, S) ⊆ P_f
S[g(i)] += R[i] ∧ PART(P_g, S) ∧ image(P_g, g, S) ⊆ P_g

(a) Original loop

Program Constraints

for (i in R):
  if (f(i) in S):
    PART(P_f, R) ∧ COMP(P_f, R) ∧ DISJ(P_f)
S[f(i)] += R[i]
  if (g(i) in S):
    PART(P_g, S) ∧ image(P_g, g, S) ⊆ P_g
S[g(i)] += R[i]

(b) Relaxed loop

parallel for (p in P_i):
  for (i in P_i[p]):
    if (f(i) in P_i[p][f(i)]):
      P_i[p][f(i)] += P_i[p][i]
    if (g(i) in P_i[p][g(i)]):
      P_i[p][g(i)] += P_i[p][i]

(c) Parallelized loop

Figure 11: Example with multiple uncentered reductions

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_f and P_g would lead the solver to an assignment of P_i to a union of preimages preimage(R, f, ..) and preimage(R, g, ..), which cannot satisfy the predicate DISJ(P_i).

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.

5.2 Finding Private Sub-Partitions

In cases when the relaxation is not applied, the optimizer tries to extract 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 ⊆ 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 P_p[j]): P_p[j][g(i)] += P'[j][i]
    else:
      P_s[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) ≜ image(P, f, R)

f_R^{-1}(P) ≜ preimage(R, f, P)

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

f_S(P) - f_S(f_R^{-1}(f_R(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_R(P)) extends each subregion P[i] with the elements from the other subregions P[j] (j ≠ 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_R(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.
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 parallelizations 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 efficiency.
The circuit generator is designed in a way that circuit nodes form clusters; a maximum of 20% of wires connect nodes in two different 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 $\text{rn}$ of circuit nodes, $\text{pn}_{\text{private}}$ for private nodes and $\text{pn}_{\text{shared}}$ for shared nodes, and the union of these partitions is a disjoint, complete partition of $\text{rn}$, as expressed by the following user constraint: $\text{DISJ}(\text{pn}_{\text{private}} \cup \text{pn}_{\text{shared}}) \land \text{COMP}(\text{pn}_{\text{private}} \cup \text{pn}_{\text{shared}}, \text{rn})$

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 \( r_{s, p} \) and \( r_{2, p} \) of sides and zones, respectively. The parallel mesh generator guarantees that zones pointed to by the sides in the ith subregion of \( r_{s, p} \) are all contained in the ith subregion of \( r_{2, p} \) (i.e., \( \text{image}(r_{s, p}, r_{s}) \cdot \text{mapsz}, r_{2}) \subseteq r_{2, p} \)).
- Additionally, each side \( s \) has all its neighbor sides accessed via \( r_{s[s]} \cdot \text{mapss1} \) and \( r_{s[s]} \cdot \text{mapss4} \) in the same subregion:
  \[
  \text{image}(r_{s, p}, r_{s[s]} \cdot \text{mapss3}, r_{s}) \subseteq r_{s, p} \\
  \land \text{image}(r_{s, p}, r_{s[s]} \cdot \text{mapss4}, r_{s}) \subseteq r_{s, p}
  \]

Although these constraints are recursive, the solver can still check the consistency as long as a satisfying partition \( (r_{s, p}) \) is provided.
- Finally, the mesh generator creates a partition \( r_{p, p, \text{private}} \) of private points, which can be used as a private sub-partition for uncentered reductions using \( r_{s[s]} \cdot \text{mapss1} \):

\[
\text{image}(r_{s, p}, r_{s[s]} \cdot \text{mapss1}, r_{p, p, \text{private}}) \subseteq r_{s, p}
\]

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 solving 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 (UE) 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 UE 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 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.