Modular Divide-and-Conquer Parallelization of Nested Loops

Azadeh Farzan
University of Toronto
Canada

Victor Nicolet
University of Toronto
Canada

Abstract

We propose a methodology for automatic generation of divide-and-conquer parallel implementations of sequential nested loops. We focus on a class of loops that traverse read-only multidimensional collections (lists or arrays) and compute a function over these collections. Our approach is *modular*, in that, the inner loop nest is abstracted away to produce a simpler loop nest for parallelization. The *summarized* version of the loop nest is then parallelized. The main challenge addressed by this paper is that to perform the code transformations necessary in each step, the loop nest may have to be augmented (automatically) with extra computation to make possible the abstraction and/or the parallelization tasks. We present theoretical results to justify the correctness of our modular approach, and algorithmic solutions for automation. Experimental results demonstrate that our approach can parallelize highly non-trivial loop nests efficiently.

CCS Concepts • **Computing methodologies** → *Parallel programming languages*; • **Theory of computation** → *Program verification*; Parallel computing models;

Keywords Parallelization, Program Synthesis, Homomorphisms, Divide and Conquer

ACM Reference Format:
Azadeh Farzan and Victor Nicolet. 2019. Modular Divide-and-Conquer Parallelization of Nested Loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314612

1 Introduction

The advent of multicore computers and development of APIs like OpenMP [10], CUDA [32], and TBB [33] has increased the popularity of parallel programming for performance gains. Despite big advances in parallelizing compilers, *correct and efficient* parallel code is often hand-crafted through a time-consuming and error-prone process. These APIs implement commonly used *parallel programming skeletons* that ease the task of parallel programming. Instead of writing a parallel program from scratch, a programmer needs to only specify the key components of a particular skeleton. Divide-and-conquer parallelism is the most commonly used of such skeletons for which, the programmer has to specify a *split*, a *work*, and a *join* function. We propose a methodology to automatically generate these components from an input sequential code.

We focus on a class of divide-and-conquer parallel programs that operate on *multidimensional sequences* (e.g. multidimensional arrays, or in general any collection type with similar recursive structure) in which the divide (*split*) operator is assumed to be the inverse of the default sequence *concatenation* operator (i.e. divide s into s_1 and s_2 where $s = s_1 \bullet s_2$). Our input programs are *loop nests* that traverse the multidimensional data in accordance with their recursive structure. These programs are assumed not to modify their input and to have unbreakable data flow dependencies.

Consider a three dimensional $n \times m \times \ell$ array A (with both positive and negative elements) and the sums of the elements of subarrays $A[k..n-1, 0..m-1, 0..\ell-1]$ (for all $0 \leq k < n$). The code in Figure 1(a) discovers the maximum subarray sum over all such subarrays. Intuitively, considering the array as a 3D box with height n, it discovers the maximum sum of boxes of different heights, with the same width, length and bottom as the input box. Note that this optimal sequential implementation runs in *single pass* linear time over the input 3D array, at the cost of creating unbreakable loop

```
int max_bot_box_sum = 0;                                     (a)
for (i = 0; i < n; i++) {
   int plane_sum = 0;
   for (j = 0; j < m; j++) {
      for (k = 0; k < l; k ++)
         { plane_sum += A[i][j][k]; } }
   max_bot_box_sum =
     max(max_bot_box_sum + plane_sum, 0);
}
```

```
int max_bot_box_sum = 0;                                     (b)
int aux_sum = 0
for (i = 0; i < n; i++) {
   int plane_sum = 0;
   for (j = 0; j < m; j++) {
      for (k = 0; k < l; k ++) {
         {plane_sum += A[i][j][k];} }
   aux_sum = aux_sum + plane_sum;
   max_bot_box_sum =
     max(max_bot_box_sum + plane_sum, 0);
}
```

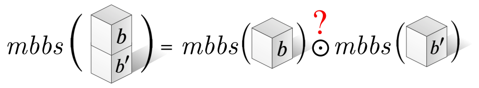
```
aux_sum = aux_sum_l + aux_sum_r;                             (c)
max_bot_box_sum = max(max_bot_box_sum_r,
            aux_sum_r + max_bot_box_sum_l);
```

Figure 1. Maximum bottom box sum (*mbbs*).

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314612

dependencies. A less efficient solution that enumerates all boxes is easier to parallelize.

It is easy to observe that the code is not (divide-and-conquer) parallelizable. Let us assume it is. Then there exists a binary function $\odot$ that can combine results of two instances of the code (*mbbs*) run on two adjacent boxes to produce the same results for the *concatenated* box.



Let $b = [5]$ (a $1 \times 1 \times 1$ box) and consider two choices for b', namely $[-3, 3]$ and $[0, 3]$ ($2 \times 1 \times 1$ boxes). Although $mbbs(b')$ is 3 in both cases, the join needs to produce two different answers for $mbbs(b \bullet b')$. This contradicts $\odot$ being a function.

Nonexistence of the join operator implies that *mbbs*, the function computed by the loop, is not a *homomorphism*. Now, consider the modified code illustrated in Figure 1(b). A new accumulator `aux_sum` is added (in orange), which maintains the sum of the elements in $A[0..i-1, 0..m-1, 0..\ell-1]$ at the i-th iteration of the outer loop. Note that $mbbs(b)$ is producing a pair of integers now, instead of a single integer. This extending of a function's signature is called *lifting*, in the standard sense of lifting a morphism in category theory, and is illustrated on the right. (I, O) denotes the input and output of the original sequential loop, and a *lifting* of the code additionally computes *auxiliary* information denoted by A. If the *lifted* function is a *homomorphism*, then a parallel join exists for it. Figure 1(c) illustrates the parallel join for the lifted maximum bottom box code.

$$\begin{tikzcd}
 & O \times A \arrow[d, "\text{projection}"] \\
I \arrow[ur, "\text{parallelizable loop}"] \arrow[r, "\text{sequential loop}"'] & O
\end{tikzcd}$$

1.1 Modular Parallelization

Figure 2(a) illustrates the flow of data in a generic nested loop (of arbitrary depth), where s_i denotes the *state* of the loop nest (e.g. a tuple of program variables). The *black* arrows correspond to the computation of one instance of the body of outermost loop, while the *blue* arrows correspond to the computation of one instance of the inner loop nest.

The goal is to parallelize, *divide-and-conquer* style, the outermost loop with the assumption that the dependencies are unbreakable. In [11], we proposed a semantic solution to this problem for **simple** (non-nested) loops by *lifting* their computations to homomorphisms. To generalize such a semantic solution to nested loops, one comes across the very hard problem of computing a *semantic summary* of the functionality of the inner loop nest, to be used in the analysis of the outer loop. Despite big strides in program analysis techniques [9, 20], this type of semantic summary computation remains limited to classes of loops whose invariants (summaries) are within decidable theories, and even then, mostly proof-driven rather than summarizing full functionality.

We propose a methodology that circumvents this problem through a modular solution. We divide the dependencies in

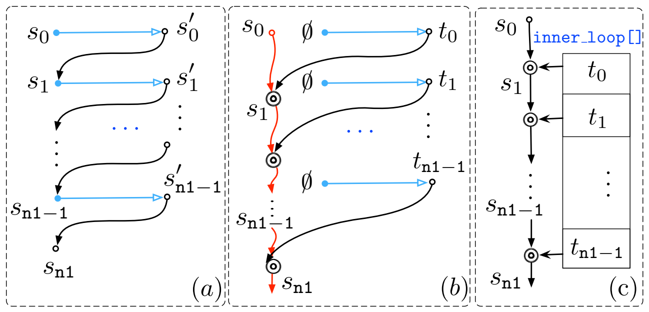


Figure 2. Dependencies in a general sequential nested loop (a) vs. a memoryless one (b), which is summarized in (c).

Figure 2(a) into two categories and resolve them separately. The black arrows force every instance of the inner loop nest to be executed only after the results of all previous instances are ready. Contrast this with the diagram in Figure 2(b), where each instance of the inner loop nest starts from a fixed (constant) initial state $\emptyset$, and therefore, all instances can be run in parallel. The sequential binary operator $\circledcirc$ merges the results of the inner loop nest (t_i) with the current state of the outermost loop (s_i) and makes the required adjustments (to get s_{i+1}). We call such a loop nest *memoryless*. The terminology is inspired by the fact that all the instances of the inner loop nest implement the same function (that starts from the same initial state $\emptyset$). If a general loop nest is transformed to a memoryless one through the introduction of new computation (i.e. $\circledcirc$), then this results in the removal of the black arrow dependencies. The inner loop nest can be executed by a *parallel map*. The outermost loop remains sequential. Observe that the loop in Figure 1(a) is memoryless.

Transforming a general loop to a memoryless one is not always straightforward. Due to lack of information in the loop state, no such binary function (operator) $\circledcirc$ may exist. In these cases, one needs to deduce additional information to be computed by the inner loop nest to facilitate the existence of $\circledcirc$, that is, the inner loop nest has to be *lifted*. Transforming a general loop to a memoryless one involves solving two subproblems: (i) producing an implementation for $\circledcirc$, and (ii) discovery of auxiliary computation when such an operator does not exist. Solving these two problems are two of our key contributions (Sections 7.2 and 5.3).

When the loop is memoryless, the inner loop nest can be abstracted away to get a *summarized* (potentially simpler) loop. As shown in Figure 2(c), the results of the computations of the inner loop nest are assumed to be stored in a (conceptual) array (called `inner_loop[]`), and therefore the loop nest is removed. The *summarized loop* fetches the results from `inner_loop[]` to perform its computation. Any *memoryless* loop can be *summarized* this way. For example, on the right, the summarized

```
int max_bot_box_sum = 0;
for (i = 0; i < n; i++) {
  max_bot_box_sum =
    max(max_bot_box_sum
    + inner_loop[i], 0);
}
```

version of the memoryless 3-nested loop of Figure 1(a) is illustrated.

The crucial observation is that the *summarized* loop is *efficiently parallelizable* if and only if the original one is (Theorems 4.7 and 5.3). Therefore, the problem of parallelizing the original loop is *soundly and completely* reducible to the problems of (i) producing the *summarized* loop, and (ii) parallelizing it. Summarization can substantially simplify the parallelization task. For example, the approach in [11] can parallelize the summarized loop above, precisely by adding the auxiliary computation illustrated in Figure 1(b), while it cannot be directly applied to the original loop in Figure 1(a). Summarization, however, does not always yield a simple loop like the one above, and therefore, the approach in [11] is not always applicable to a summarized loop.

To parallelize the summarized loop, two subproblems have to be solved: (a) Automatic *lifting* of *nested* loops to parallelizable code, and (b) automatic generation of the parallel join for *nested* loops. Problem (b) is easier to solve. In Section 7, we build on our technique from [11] to extend it to nested loops. The lifting problem is more complex. We solve it by reducing it to two well-known problems, namely *normalization* (in term rewriting systems) and *recursion discovery*. In section 8, we discuss the reduction and propose simple heuristics for both problems. Our modular parallelization methodology comprises theoretical results and algorithms for generating all required additional code. Figure 7 outlines the applications of the theorems and the contributed algorithmic modules, and therefore, serves as detailed summary of our technical contributions. Due to the undecidability of the problem, some of our algorithms are heuristics. We provide experimental results to demonstrate the effectiveness of these heuristics in fully automatically and efficiently producing divide-and-conquer parallelizations for some highly nontrivial nested loops. Beyond facilitating full automation, we believe that our methodology is also a systematic approach that can guide programmers in writing correct and efficient parallel code manually.

2 Motivating Examples

We use two difficult-to-parallelize examples to underline the challenges of parallelizing the class of nested loops targeted in this paper and outline the strengths of our methodology.

2.1 Balanced Parentheses

This example demonstrates that transforming a nested loop to a *memoryless* one can be complicated. A string is *balanced* if the total number of left and right brackets match, and any prefix of the string has at least as many left brackets as right ones. Assume that the input is a two-dimensional array containing a large bracketed math expression, one row per each line. A line l of input x is *level* if we have $x = x_1 \cdot l \cdot x_2$, where l and x_1 are both balanced. The code in Figure 3 counts the number of *level* lines of its input through a nontrivial algorithm. `offset` maintains the excess of left over right brackets seen so far. `bal` tracks if `offset` has always remained nonnegative. We encourage the reader to manually parallelize the outer loop to get a sense of the difficulty of this problem.

```
int offset, count_lines = 0; bool bal=true;
for(int i = 0; i < n; i++) {
  int line_offset = 0;
  for(int j = 0; j < length(a[i]); j++) {
    line_offset += a[i][j] == "(" ? +1 : 0;
    line_offset += a[i][j] == ")" ? -1 : 0;
    if (offset + line_offset < 0)
        { bal = false; }
  }
  offset += line_offset;
  if (bal && line_offset==0 && offset==0)
     { count_lines++; }
}
```

Figure 3. Balanced Parentheses.

The loop is not *memoryless*; unbreakable dependencies on `bal` and `offset` variables induce the black arrows from the diagram in Figure 2(a). One cannot remove the dependency of the update to `bal` on the value of `offset` without having the inner loop compute an extra value. Specifically, the minimum value of `line_offset`, during the execution of the inner loop, should be made available to the outer loop. If this does not cause `offset` to dip below 0, then `offset + line_offset` should have remained positive throughout the inner loop execution, and therefore the value of `bal` can be recovered. The code in Figure 4 illustrates the lifted code (modifications are highlighted). The loop in Figure 4 is `memoryless` and can be summarized as below.

```
int offset, count_lines = 0; bool bal = true;
for(int i = 0; i < n; i++) {
  offset += inner_loop[i].line_offset;
  bal &&= (offset + inner_loop[i].min_offset > 0);
  if (bal && inner_loop[i].line_offset == 0 && offset == 0)
     { count_lines++; }
}
```

In Sections 5 and 8, we discuss how the `min_offset` accumulator can be discovered automatically. Can the summarized loop (above) be parallelized? No! The reader can verify that a parallel join does not exist. Furthermore, the loop *cannot be efficiently lifted* (theoretically impossible); that is, the addition of more scalar accumulators will not transform it to a homomorphism. The transformation of the loop to a memoryless one parallelizes all instances of the inner loop

```
int offset, count_lines = 0; bool bal = true;
for(int i = 0; i < n; i++) {
  int line_offset, min_offset = 0; bool line_bal=true;
  for(int j = 0; j < length(a[i]); j++) {
    line_offset += a[i][j] == "(" ? +1 : 0;
    line_offset += a[i][j] == ")" ? -1 : 0;
    if (0 + line_offset < 0) {line_bal = false;}
    min_offset = min(min_offset, line_offset);
  }
  offset += line_offset;
  bal = bal && (offset + min_offset > 0);
  if (bal && line_offset == 0 && offset == 0)
     { count_lines++; }
}
```

Figure 4. Memoryless balanced parentheses.

```
int rec[];
for (i = 0; i < n; i++) {
   int row_sum = 0;
   for (j = 0; j < m; j++) {
      row_sum += A[i][j];
      rec[j] += row_sum;
      mtl_rec = max(mtl_rec, rec[j]);
   }
}
                                    (a)
```

```
int rec[];
int mtl_rec = 0;
for (i = 0; i < n; i++) {
  removed inner loop
  // the loop implementing ⊚
  for (j = 0; j < m; j++) {
    rec[j] = rec[j] + inner_loop[i][j];
    mtl_rec = max(mtl_rec, rec[j]);
  }
}
                                    (b)
```

```
int rec[], max_rec[];
int mtl_rec = 0;
for (i = 0; i < n; i++) {
  removed inner loop
  // the loop implementing ⊚
  for (j = 0; j < m; j++) {
    rec[j] = rec[j] + inner_loop[i][j];
    max_rec[j] = max(max_rec[j], rec[j]);
    mtl_rec = max(mtl_rec, rec[j]);
  }
}                                   (c)
```

Figure 5. Maximum top-left subarray sum (a), its summarized version (b), and the lifting to parallelizable code (c)

(implementable by a parallel *map*). But, the outer loop computation cannot be efficiently turned into a parallel *reduction*. Yet, the parallelization of the code through the discovery of the *map* alone yields a reasonable speedup (Section 9).

2.2 Maximum Top-Left Subarray Sum

This example demonstrates that parallelization of the outer loop may be nontrivial even after a successful summarization. Consider a two-dimensional array of integers (with both positive and negative) elements. Assume that the goal is to compute the maximum sum of the elements of a subarray $A[0..k, 0..\ell]$ for all $0 \leq k < n$ and $0 \leq \ell < m$, i.e. all subarrays that include the top-left corner $(0, 0)$. The code in Figure 5(a) is a clever single-pass implementation of this function. Note that the inner loop has a state (variable) `rec[]` that is the same size as the width of a row (m). In `rec[j]`, the loop maintains the sum of all elements in the subarray $A[0..i, 0..j]$. The loop is not *memoryless* due to the dependencies induced by both `rec[]` and `mtl_rec`. Again, we encourage the reader to think about how they would parallelize the code manually.

Figure 5(b) illustrates the memoryless and summarized variation of the code. The transformation is straightforward, but the summarized loop is still a 2-nested loop and not parallelizable (i.e. not a homomorphism); that is, the operator $\circledcirc$ from diagram in Figure 2(b) has to be implemented as a simple loop to correctly update variables `rec[]` and `mtl_rec`. The transformation underlines a subtle point, namely that, the relevant information from the input array is the *sum* values of the subarrays starting from the $(0, 0)$ and ending at (i, j), and not the values of `A[i][j]`'s. This *abstraction* is a key to the simplification of the *lifting* of the outer loop to a homomorphism for parallelization.

The code needs to be lifted as illustrated in Figure 5(c). A new variable `max_rec[]` has to be introduced that maintains the maximum value of `rec[j]` (for $0 \leq j < n$). Discovery of such variables, that is arrays of accumulators, is not required for parallelization of simple loops [11]. The time complexity budget for a parallel *join* operator of a simple loop is constant time, and therefore non-constant sized variables are pointless. For nested loops, however, as this example demonstrates, they may be essential. In Section 8, we propose a new algorithm for discovering liftings like this automatically.

Now, a parallel join operator can combine the value of `rec[]` from the top thread and `max_rec[]` from the bottom

```
int rec[] = rec_l[],  max_rec[] = max_rec_l[];
int mtl_rec = mtl_rec_l;
for (j = 0; j < m; j++) {
  rec[j] = rec[j] + rec_r[j];
  max_rec[j] = max(max_rec[j], rec_l[j] + max_rec_r[j]);
  mtl_rec = max(mtl_rec, max_rec[j]);
}
```

Figure 6. The parallel join for Figure 5(c).

thread to account for subarrays that intersect two adjacent array chunks, as illustrated in Figure 6. The two challenges underlined by this example are (i) the synthesis problem of a parallel join operator which is a looping computation, and (ii) the discovery of auxiliary information for lifting which is not constant-sized.

3 Notation and Background

Sequences and Functions. We assume a generic type Sc that refers to any scalar type used in typical programming languages, such as `int` and `bool` whenever the specific type is not important in the context. Scalars are assumed to be of *constant* size, and conversely, any constant-size representable data type is assumed to be scalar. Consequently, all operations on scalars are assumed to have constant time complexity. Type $\mathcal{S}$ defines the set of all *sequences* of elements of type Sc. For any sequence x, $x[i]$ (for $0 \leq i < |x|$) denotes the element of the sequence at index i, and $x[i..j]$ denotes the subsequence between indexes i and j (inclusive). The concatenation operator $\bullet : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ defined over sequences is associative. The sequence type stands in for *arrays*, *lists*, or any collection data type that admits a linear iterator and an *associative* composition operator.

Definition 3.1. A function $h : \mathcal{S} \to D$ is rightward iff there exists a binary operator $\oplus : D \times Sc \to D$ such that for all $x \in \mathcal{S}$ and $a \in Sc$, we have $h(x \bullet [a]) = h(x) \oplus a$.

Note that the notion of associativity for $\oplus$ is not well-defined, since it is not a binary operation defined over a set (i.e. the two arguments to the operator have different types). A leftward function is defined analogously using the recursive equation $h([a] \bullet x) = a \otimes h(x)$.

Homomorphisms are a well-studied class of mathematical functions. We are interested in a special class of homomorphisms, where the source structure is a set of sequences with the standard concatenation operator.

Definition 3.2. A function $h : \mathcal{S} \to D$ is $\odot$-homomorphic for binary operator $\odot : D \times D \to D$ iff for all sequences $x, y \in \mathcal{S}$ we have $h(x \bullet y) = h(x) \odot h(y)$.

Note that $\odot$ is necessarily associative since concatenation is associative (over sequences). Moreover, $h([])$ (where $[]$ is the empty sequence) is the unit of $\odot$, since $[]$ is the unit of concatenation. If $\odot$ has no unit, then $h([])$ is undefined. There is a formal connection between homomorphisms and divide-and-conquer style parallelism, when the divide operator is the inverse of concatenation:

Proposition 3.3. *(from [17]) A function f is a homomorphism if and only if it can be written as a composition of a map and a reduction.*

In the context of this paper, parallelization is formally the above transformation to a map and a reduction composition.

3.1 Program Model

Due to space restrictions, we forgo the formal definition of the input language since an informal exposition provides enough clarity. Our input sequential programs are written in a simple imperative language with basic constructs for branching and looping. The language includes scalar types `int` and `bool`, and a collection type `seq`. The syntax of the language is standard and is given in [12].

Model of a loop body We use a functional form to model and manipulate nested loops. We use a fairly standard translation (sketched in [12]) to transform the nested loops written in an imperative language to functional form using let-bindings, nested functions, and tuples.

State and Input Variables Let Var be the set of all variables that appear in the loop nest. We partition Var into two sets of variables: SVar denotes the set of *state variables* which are those that appear on the left-hand side of an assignment statement (anywhere, even outside the loop nest). IVar denotes the set of *input variables* and IVar = Var − SVar. Note that state variables may be subscripted array accesses.

4 Multidimensional Collections

Type $\mathcal{S}^n$ is inductively defined as the set of all n-dimensional *sequences* (for $n \geq 1$), with the base case of $\mathcal{S}^0 = Sc$ (set of scalars). We generalize the standard sequence concatenation operator $\bullet$ to a family of operators $\bullet : \mathcal{S}^n \times \mathcal{S}^n \to \mathcal{S}^n$ (for all $n \in \mathbb{N}^+$). For any $\sigma \in \mathcal{S}^{n-1}$, we have $[\sigma] \in \mathcal{S}^n$ which is an n-dimensional sequence with a single element σ.

4.1 Functions over Multidimensional Collections

In Section 3, we noted that loop nests are translated to functional form. We use this functional form as the formal representation for all of our theoretical results.

Definition 4.1. (Multidimensional Rightward) A function $f : \mathcal{S}^n \to D$ $(n > 1)$ is rightward iff there exists a family of rightward (or leftward) functions $\mathbb{G} : D \to (\mathcal{S}^{n-1} \to D)$ and an operator $\otimes : D \times D \to D$ such that for all $\sigma \in \mathcal{S}^n, \delta \in \mathcal{S}^{n-1}$, we have $f(\sigma \bullet [\delta]) = f(\sigma) \otimes \mathbb{G}(f(\sigma))(\delta)$.

The base case of $n = 1$ falls on the classic Definition 3.1. A rightward function's computation is illustrated in the diagram in Figure 2(a). Note that the value of $f(\sigma)$ (as the selector in the family of functions) serves as a type of *carry over state* and corresponds to the data flow represented by the black arrows in Figure 2(a). The family of functions can be viewed as only differing in their recursion base case.

When f corresponds to a loop nest, the family of rightward functions $\mathbb{G}$ represents all the instances of the inner loop nest (in isolation from the outermost loop) and the operator $\otimes$ represents the (loop free) computation performed in the body of the outer loop. The domain D corresponds to all valuations of the state variables (SVar) of the loop nest.

A special case of Definition 4.1 is when the family of functions collapses into exactly one function, which corresponds to *memoryless* loops as introduced in Section 1.1. We can formally define *memoryless functions* by removing the dependency on the context as follows:

Definition 4.2. (Memoryless) A function $f : \mathcal{S}^n \to D$ is (rightward) memoryless iff there exists a rightward (or leftward) function $g : \mathcal{S}^{n-1} \to D$ and a binary operator $\oplus : D \times D \to D$ such that for all $\sigma \in \mathcal{S}^n, \delta \in \mathcal{S}^{n-1}$ we have $f(\sigma \bullet [\delta]) = f(\sigma) \oplus g(\delta)$.

The key difference between the formulation in Definition 4.1, and that of Definition 4.2 is the computation performed over δ (i.e. function g) has no dependency on the partially computed value of $f(\sigma)$; hence the use of terminology *memoryless*. Figure 2(b) illustrates the computation of a memoryless function. As the example in Section 2.1 demonstrated, not all rightward functions are memoryless.

Proposition 4.3. *For every rightward memoryless function f (from Definition 4.2), we have $f(\sigma) = foldl(\oplus) \circ map(g)(\sigma)$.*

The proof of the above proposition is straightforward. It suggests that all instances of g (the inner loop nest) can be parallelized, through the *map*, even if their results have to be combined sequentially in the outermost loop with *foldl*.

4.2 Multidimensional Homomorphisms

Definition 3.2 applies to multidimensional rightward functions in a straightforward way. Function $h : \mathcal{S}^n \to D$ is $\odot$-homomorphic for the binary operator $\odot : D \times D \to D$ iff for all sequences $\sigma, \sigma' \in \mathcal{S}^n$, we have $h(\sigma \bullet \sigma') = h(\sigma) \odot h(\sigma')$. An interesting link exists between the structure of a multidimensional rightward function and its homomorphic properties, which is captured by the proposition below:

Proposition 4.4. *If a function $h : \mathcal{S}^n \to D$ is a homomorphism, then it is memoryless.*

The proof appears in [12]. The converse of Proposition 4.4 does not hold.

Example 4.5. Recall the maximum bottom box example from Section 1. The function corresponding to Figure 1 is memoryless, but as discussed, not a homomorphism.

For a memoryless function to be a homomorphism, an extra condition is required which is outlined below.

Proposition 4.6. *If a function $f : \mathcal{S}^n \to D$ is (rightward) memoryless and defined by function g and binary operator $\oplus$ (of Definition 4.2),* **and** *if the function $h : \mathcal{S}_D \to D$ defined as*

$$h([]) = f([])$$
$$\forall a \in D : h(x \bullet [a]) = h(x) \oplus a$$

is $\odot$-homomorphic for some binary operator $\odot : D \times D \to D$, then f is $\odot$-homomorphic. We refer to function h as the ***summarized*** *version of f.*

Function h corresponds to the concept of a *summarized* loop as introduced in Section 1.1. In fact, we can prove that the sufficient conditions in Proposition 4.6 are also necessary.

Theorem 4.7. *The following two statements are equivalent:*

1. *Multidimensional rightward function f is $\odot$-homomorphic for some binary operator $\odot : D \times D \to D$.*
2. *f is memoryless* ***and*** *function $h : \mathcal{S}_D \to D$, the summarized version of f (see Prop. 4.6) is $\odot$-homomorphic.*

Theorem 4.7 states the necessary and sufficient conditions for a recursive function to be parallelizable (see [12] for the proof). For one-dimensional sequences, the statement becomes trivial when the summarized version of the function and the function itself coincide.

The condition of memorylessness captures the essence of modularity of our approach. Instead of determining parallelizability of f through a direct discovery of a join ($\odot$) for f, Theorem 4.7 lets us check if f is memoryless first, and then discover a join for a simplified (summarized) version of f (i.e h). Recall the diagram in Figure 2(b). Memorylessness of f corresponds to the existence of the *map* part of a parallel computation of f. Parallelizability of h corresponds to the existence of the *reduction* part of a parallelization of f. The combination of the existence of both the map and the reduction is equivalent to f being homomorphic (according to Proposition 3.3). Theorem 4.7 makes this observation formal.

5 Manufacturing Homomorphisms

If a function is not a homomorphism, then the first step to parallelization is to *lift* it to a homomorphism.

Definition 5.1. (Lifting) Let $f : \mathcal{S}^n \to D$ be a rightward multidimensional function. $\hat{f}^{D'} : \mathcal{S}^n \to D \times D'$ is a lifting of f if and only if $\hat{f}^{D'}$ is rightward and $f = \pi_D \circ \hat{f}^{D'}$, where π_D is the standard projection down to D.

This definition is mostly consistent with the standard definition of lifting in category theory, other than the additional condition of rightward computability of the extension.

Two types of liftings of a non-homomorphic function f are of interest in this paper: (1) a lifting of a non-memoryless f to a memoryless function; we call this the *memoryless lift*, and (2) a lifting of a non-homomorphic f to a homomorphism; this is called a *homomorphism lift.*

5.1 Homomorphism Lift

Every non-homomorphic function can be made homomorphic by a rather trivial lifting. The observation, previously made in [19], is formalized below:

Proposition 5.2. *Given a rightward function $f : \mathcal{S}^n \to D$, the function $f \times \iota$ (function product) is a homomorphism where $\iota : \mathcal{S}^n \to \mathcal{S}^n$ is the identity function.*

Intuitively, the extension of the function remembers the entire input, and the join performs the original computation over the concatenated inputs from scratch, ignoring the partially computed results. Note that this trivial lifting does not really correspond to a parallelization of the function. Formally, it provides us with an associative *reduction* (hence the applicability of Proposition 3.3). Practically, it is analogous to a sequential computation. Proposition 5.2 is trivial but significant in that it states that a function can always be made homomorphic. It is then important to seek *an efficient lifting* of a non-homomorphic function to a homomorphism for the purpose of code parallelization. In Section 6.1, we formulate *efficient liftings.*

The key result of this section is the following: it is *sound and complete* to use the *summarized* loop for lifting instead of the original loop. Consider the diagram below:

$$\begin{tikzcd}
f : \mathcal{S}^n \to D \arrow[rr, "\text{summarize}"] \arrow[d, "\text{lift}"'] & & h : \mathcal{S}_D \to D \arrow[dd, "\text{lift}"] \\
\hat{f} : \mathcal{S}^n \to D \times D' \arrow[r, "\text{summarize}"] & h' : \mathcal{S}_{D \times D'} \to D \times D' \arrow[dr, "\text{domain restriction}"'] & \\
 & & \hat{h} : \mathcal{S}_D \to D \times D'
\end{tikzcd}$$

f is summarized and then lifted on the top, whereas it is first lifted and then summarized on the bottom path of the diagram. The path on the bottom is the default one; lift the function if it is not a homomorphism and then summarize it as the first step of its parallelization process. The path on the top is the shortcut that we propose which is better for automation. The idea is that the process of summarization maintains the relevant part of the computation for lifting and discards the rest for simplicity.

Note that $\hat{h}$ and h' do not have the same function signature; while they agree on their ranges, their domains are sequences of two different types. Therefore, this is not a clean commutative diagram. The key insight is that the two functions are identical up to a limitation of h' that forgets the extra information in its input sequences from D'; information that is provably redundant for the computation of h'.

The diagram commutes after this restriction is applied to h' to get to $\widehat{h}$.

Theorem 5.3. *Let $f : \mathcal{S}^n \to D$ be a (rightward) memoryless function, and summarized as $h : \mathcal{S}_D \to D$. There exists a homomorphic lifting $\widehat{h} : \mathcal{S}_D \to D \times D'$ of h if and only if there exists a homomorphic lifting $\widehat{f} : \mathcal{S}^n \to D \times D'$ of f. Moreover, $\widehat{h}$ coincides with a summarization of $\widehat{f}$.*

The theorem guarantees that the summarized loop is liftable to a homomorphism if and only if the original loop is, and that the auxiliary code synthesized for the summarized loop constitutes a lifting of the original loop. A more detailed statement of Theorem 5.3 (which corresponds to the diagram above) and its proof appear in [12].

5.2 Memoryless Lift

When a rightward function $f : \mathcal{S}^n \to D$ is not memoryless, a *lifting* may be required to add extra information to the signature of the function (state of the loop) so that functions g and $\circledcirc$ from Definition 4.2 exist. Every non-memoryless function can be made memoryless by a rather trivial lifting.

Proposition 5.4. *Given a rightward function $f : \mathcal{S}^n \to D$, the function $f \times \iota'$ (function product) is memoryless where $\iota' : \mathcal{S}^n \to \mathcal{S}^{n-1}$ is defined as $\forall \delta \in \mathcal{S}^{n-1}, \sigma \in \mathcal{S}^n : \iota'(\sigma \bullet [\delta]) = \delta$.*

In this trivial lifting, the extension to the function remembers the last line of the input $\sigma \bullet [\delta]$, that is δ, in a new component and the join effectively processes δ from scratch, ignoring the partially computed results by the inner loop computation. The construction of Proposition 5.4 is guaranteed to always maintain the time complexity of the code. It is essential, however, that the *cheapest* possible (non-trivial) lifting is used, to gain optimal parallelism. Recall the balanced bracket example from Section 2.1. The lifting (additions of `min_offset` and `line_bal` state variables) in that example is an instance of a non-trivial lifting. Proposition 5.4, in contrast, would suggest a simple admissible lifting which would not lead to as much parallelism.

5.3 Algorithmic Memoryless Lift

The algorithmic problems of lifting a function to a homomorphism or to a memoryless function are related. When a function is not memoryless, it means that there is not enough information for a *memoryless join* operator ($\circledcirc$) to exist in the style of the diagram in Figure 2(b). Where the *homomorphic lifting* algorithm asks what extra computation is required for the results of two instances of the entire loop nest to be joined together, an algorithm for *memoryless lifting* asks what extra computation is required for an instance of the loop nest to be joined with an instance of the *inner loop nest*. Considering that the two functions share the same signature, the problem is formally that of joining an inner loop nest to an arbitrary state $\vec{s}$, which is identical to the homomorphism lift of the inner loop nest. The following proposition, whose proof appears in [12] makes this observation precise.

Proposition 5.5. *A multidimensional rightwards function f defined through a family of functions $\mathbb{G}$ (as in Definition 4.1) can be lifted to a memoryless function if every member of $\mathbb{G}$ can be lifted to a $\circledcirc$-homomorphism for some $\circledcirc$.*

6 Algorithmic Parallelization

In Sections 4 and 5, we presented the theoretical foundations of our approach. Theorem 4.7 guarantees that it is *sound and complete* to parallelize the summarized loop in place of the original loop nest. Proposition 5.4 guarantees that any loop nest can be transformed into one that is summarizable. And, finally, Theorem 5.3 guarantees that a summarized loop can be *soundly and completely* lifted to a homomorphism in place of the original loop. In this section, we outline our algorithmic approach to parallelization.

6.1 Efficient Divide-and-Conquer Solution

Consider a loop nest L of depth n where the number of iterations of every loop is bounded by a parameter m. Assuming no function calls are made, the loop nest has a time complexity of $O(m^n)$. Since the translation to functional form preserves time complexity, this is also the time complexity of the function $h_L : \mathcal{S}^n \to D$ corresponding to the loop nest. For a parallel implementation of h_L based on a join operator $\odot$ to have reasonable speedups over constantly many processors, the (sequential) complexity of the implementation based on the join should not be higher than that of the original code. Constantly many processors cannot compensate for a variable increase in complexity.

Proposition 6.1. *Let $h_L \in O(m^n)$ be $\odot$-homomorphic. The sequential implementation of h_L based on $\odot$ is in $O(m^n)$ if $\odot \in O(m^{n-1})$.*

The proof appears in [12]. This observation leads to a formal definition of parallelizability.

Definition 6.2. (Parallelizability) A rightward (respectively leftward) function $h_L \in O(m^n)$ is efficiently parallelizable if and only if it is $\odot$-homomorphic and $\odot \in O(m^{n-1})$.

The deduced upper bound on $\odot$ is crucial to justify the algorithmic choices made in Sections 7, where the time complexity budget for *join* informs the choices of *syntax* for syntax-guided synthesis [1]. Similarly, there are time and space complexity budgets for an efficient *lifting*.

Corollary 6.3. *If a function $h_L \in O(m^n)$ is lifted to $\hat{h}_L^{D'} \in O(m^n)$, then any $d' \in D'$ has space complexity $O(m^{n-1})$.*

The proof follows directly from that of Proposition 6.1, which also imposes the time complexity of $O(m^{n-1})$ for computing d'. The time and space complexity bounds for d' inform the syntactic form of the auxiliary accumulators and

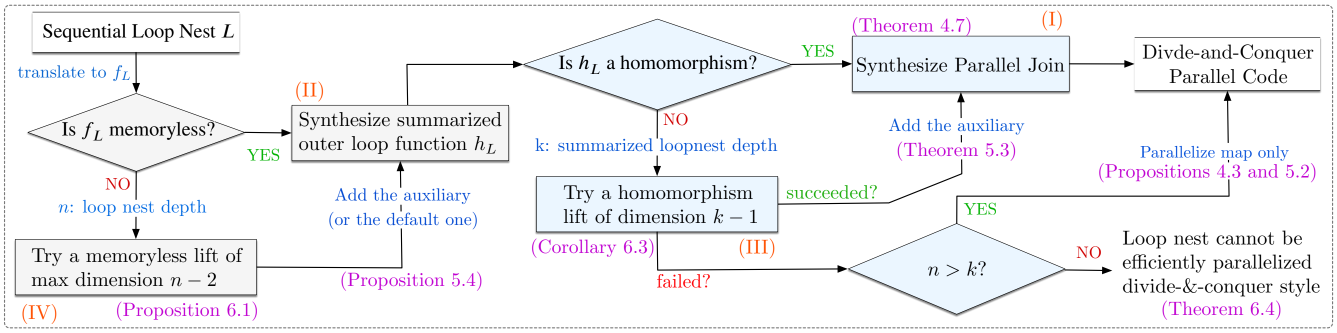

Sequential Loop Nest L
translate to f_L
Is f_L memoryless?
YES
NO
n: loop nest depth
Try a memoryless lift of max dimension $n-2$
(IV)
(Proposition 6.1)
(II)
Synthesize summarized outer loop function h_L
Add the auxiliary (or the default one)
(Proposition 5.4)
Is h_L a homomorphism?
NO
k: summarized loopnest depth
Try a homomorphism lift of dimension $k-1$
(Corollary 6.3)
(III)
failed?
succeeded?
YES
(Theorem 4.7)
(I)
Synthesize Parallel Join
Add the auxiliary
(Theorem 5.3)
Divde-and-Conquer Parallel Code
Parallelize map only
(Propositions 4.3 and 5.2)
YES
$n > k$?
NO
Loop nest cannot be efficiently parallelized divide-&-conquer style
(Theorem 6.4)

Figure 7. Parallelization Schema

the computation that produces them. In [12], we give a variation of the example from Section 2.2, and a proof that any lifting of that function to a homomorphism has a space complexity beyond the budget specified in Corollary 6.3. This information-theoretic proof is very involved, but makes the important point that an efficient lifting may not always exist, and consequently neither does a *complete* lifting algorithm.

Theorem 6.4. *An efficient lifting of a (multidimensional) rightward function may not always exist.*

6.2 Parallelization Schema

The diagram in Figure 7 illustrates the algorithmic steps in our methodology to parallelize an input sequential program. The light grey side performs the summarization of the loop, which corresponds to the discovery of a *map*. The light blue side parallelizes the summarized loop, which corresponds to the discovery of a *reduction*. The key algorithmic steps are the synthesis of the summarized outer loop and the parallel join (respectively labeled as (II) and (I) in Figure 7), which are solved using syntax-guided synthesis (Section 7), and the memoryless lift and the homomorphism lift (respectively labeled as (IV) and (III) in Figure 7) which are performed using a deductive-style algorithm (Section 8). Each step of the process is labeled with the theorem justifying it.

The test for a homomorphism in the schema is only nominal and implemented indirectly through the success or failure of join synthesis algorithm. It is important to note that Rice's theorem dictates that determining whether a computable function is a homomorphism in general is undecidable, and therefore there exists no decision procedure for this test.

If a function is not memoryless, then in (IV), an efficient *memoryless lift* is attempted; that is, the most efficient lift whose time complexity is no more than $O(m^{n-2})$. If this fails, we know we can always rely on the default admissible memoryless lift (which is incidentally of complexity $O(m^{n-1})$). If a homomorphism lift within the complexity budget (determined by k, the summarized loop depth) exists, then a classic divide-and-conquer parallel program is produced. Otherwise, we opt to parallelize the inner loop through the map and leave the outer loop's computation as sequential (as is the case for the example in Section 2.1). When summarization does not reduce the depth of the loop (i.e $n = k$), then there is no benefit in parallelizing the inner loop nest through a parallel *map* only; in this scenario, the parallelization fails.

7 Join Synthesis

In this section, we address the algorithmic problems of generation of the *parallel join* operator and the *summarized outer loop*, respectively steps (I) and (II) from the general schema of Figure 7. Although the two problems seem independent, the latter turns out to be a special instance of the former.

7.1 Syntax-guided Synthesis of Parallel Join

We employ syntax-guided synthesis (SyGuS)[1] to generate the parallel join. Given a *correctness specification* φ and *syntactic constraints* describing the syntactic space $\$$ of possible implementations for join, a syntax-guided synthesis solver finds a solution $x \in \$$ where $\varphi(x)$ holds. The correctness specification for the join operator $\odot$ is that h_L (the summarized loop function) forms a homomorphism with $\odot$ (i.e. Definition 3.2); i.e., $\varphi(\odot) \equiv \forall x, y \in S_D, h_L(x \bullet y) = h_L(x) \odot h_L(y)$.

The main challenge in SyGuS is to appropriately define the syntactic restrictions. On one hand, they need to be expressive enough to include an efficient $\odot$ that satisfies φ if one exists. On the other hand, the smaller the state space $\$$, the more tractable the search problem for its synthesis.

We use an insight to define $\$$ effectively. A function f^{-1} is a weak inverse of a function f iff $f \circ f^{-1} \circ f = f$, and f always has at least one weak inverse iff f is computable and its domain is countable. All the functions of interest in this paper have weak inverses of signature $D \rightarrow \mathcal{S}^n$. In the proof of the third homomorphism theorem in [17], it is observed that a *join* $\odot$ for a homomorphism h_L can be constructively defined based on h_L's weak inverse. That is, for all $d, d' \in D$ we have $d \odot d' = h_L(h_L^{-1}(d) \bullet h_L^{-1}(d'))$. This implies an $\odot$ with a similar syntactic structure to h_L exists. Moreover, Proposition 6.1 implies that for $\odot$ to remain within the complexity budget, $h_L^{-1}(d)$ and $h_L^{-1}(d')$ need to have *constant* length.

Example 7.1. Recall the maximum top-left subarray sum example from Figure 5(c). The summarized function h_{mtls}'s signature is the tuple of state variables ⟨`rec`,`max_rec`,`mtl_rec`⟩ and its weak inverse is a 2-row array with the same width as the original input. It is illustrated below.

`max_rec[0]`	$\cdots$	`max_rec[`$k-1$`]`
`rec[0]`$-$`max_rec[0]`	$\cdots$	`rec[`$k-1$`]`$-$`max_rec[`$k-1$`]`

A join constructed based on this weak inverse executes h_{mtls} on 4 rows; the concatenation of 2 sets of 2 rows coming from the left and the right threads.

In syntax-guided synthesis, $\mathbb{S}$ is defined by a program with *holes* (a sketch) and an *expression grammar* G specifying possible completions for the holes. Intuitively, the solver searches for substitutions from expressions in G for all holes in the sketch, such that the resulting program satisfies the correctness specification. The construction of the sketch we use is an extension of the one in [11]. For G, we use a standard expression grammar parameterized by the operators of the programs and the input variables of the join. The sketch is constructed by replacing every variable in the body of h_L by a hole. For example, the parallel join for the example in the introduction is synthesized using the following sketch:

```
aux_sum = ??LR + ??LR;
max_bot_box_sum = max(??LR + ??LR, ??R);
```

where ??$_{\mathsf{LR}}$ (resp. ??$_{\mathsf{R}}$) are holes to be filled with expressions over variables of the left and the right thread (resp. the right thread only). The join from Figure 1 is a solution to the above sketch, taken from [11], whose formal definition appears in [12]. In contrast to [11], the body of our loops and the join operator are not necessarily loop-free (e.g. the code from Figure 5(c) and the corresponding join). We extend the sketch construction by mapping the loops nested in the loop body to loop skeletons in the sketch.

For example, consider the sketch illustrated on the right. The crucial difference between a looped sketch like this one and those in [11] is that in a loop, variables may have to be referenced on the right-hand side of the assignments to effectively implement *recursion*. Therefore, the extended sketch allows for join variables to appear on the righthand side of the expressions (i.e. ?? stands for all variables in contrast to just left and right variables). The complete sketch admits bounded repetitions of the above loop (not illustrated), which then produces exactly the solution from Example 7.1 in 4 repetitions. But, one can piggy back on the first loop to update `mtl_rec` simultaneously with `max_rec[]` instead of having to wait for the next loop in the repetition. This leads to the discovery of an optimal join (i.e. the one in Section 2.2), compared to a less efficient join of Example 7.1. Note that both joins are valid solutions of the sketch.

```
int mtl_rec = ??;
int * rec = ??;
int * max_rec = ??;
for(j=0; j < n; j++){
  rec[j] = ?? + ??;
  max_rec[j] = max(??,??);
  mtl_rec = max(??,??);
}
```

Assuming that h_L^{-1} returns a constant-length (multidimensional) sequence, and h_L is a homomorphism, then a join is guaranteed to exist in the space described by our sketch. And, the synthesis procedure can *soundly* declare h_L not to be a a homomorphism when it cannot find a join.

7.2 Summarized Loop Synthesis

Assuming that the loop is memoryless, summarization of the loop boils down to the synthesis of the operator ⊚ from Figure 2(b). We argue why this problem is nearly identical to the synthesis of a homomorphic join.

Proposition 7.2. *A multidimensional rightward function f (as defined in Definition 4.1) is memoryless iff every member of the family of functions $\mathbb{G}$ is ⊚-homomorphic.*

The full proof can be found in [12]. Here, we sketch a high level argument to provide some intuition. Consider a multidimensional rightward function f as defined in Definition 4.1. Assume that there exists a binary operator ⊚ and an initial state $\emptyset_g$ that satisfy the following condition

$$\varphi(\circledcirc, \emptyset_g) \overset{def}{\equiv} \forall d \in D, \delta \in S^{n-1},\ \mathbb{G}(d)(\delta) = d \circledcirc \mathbb{G}(\emptyset_g)(\delta).$$

It is straightforward to see why the above characterization is equivalent to the one given in Definition 4.2 for a memoryless rightward function. Moreover, one can prove that condition φ is equivalent to stating that every member of the family of functions $\mathbb{G}$ is ⊚-homomorphic. Hence, the operator ⊚ can be synthesized, using the specification φ, as a homomorphism join operator of the function $\mathbb{G}$. This is the problem addressed in Section 7.1.

It is important to note that the complexity budget for a memoryless join operator ⊚ may differ from that of the parallel join operator ⊙. The budget for ⊙ is determined by the depth of the summarized loop nest as formulated in the definition of parallelizability (Definition 6.2). The budget for ⊚ is determined by the depth of the original loop nest. Note that in Figure 7, k is the depth of the summarized loop nest and n is the depth of the original loop nest.

8 Automatic Lifting

As argued in Section 5.3, a *memoryless lift* is a special instance of the homomorphism lift and both problems admit the same algorithmic solution. Here, we present an algorithm for discovering a *homomorphism lift*, which would respectively apply to modules (III) and (IV) in Figure 7.

8.1 Rewriting Oracle

Assume a memoryless function $f : \mathcal{S}^n \to D$ defined recursively as $f(\sigma) = \mathit{foldl}(\oplus) \circ \mathit{map}(g)(\sigma)$ is not a homomorphism. Let $h : \mathcal{S}_D \to D$ be the summarization of f, as defined in Proposition 4.6, that is:

$$h([]) = f([])$$
$$\forall a \in D : h(x \bullet [a]) = h(x) \oplus a$$

for $x \in \mathcal{S}_D$ and $a \in D$. Recall that according to Theorem 5.3, a lifting for h can be computed instead of a lifting for f.

Let $x, y \in \mathcal{S}_D$, $\vec{s} = h(x)$, and $y = [a_1, \ldots, a_k]$ (with $a_i \in D$). The sequential computation of $h(x \bullet y)$ can be written as:

$$h(x \bullet [a_1, \ldots, a_k]) = (\cdots(\vec{s} \oplus a_1) \oplus \cdots) \oplus a_k. \quad (1)$$

Lifting h to a homomorphism $\hat{h}^{D'}$ corresponds to extending the image of h to $D \times D'$ and lifting the initial state to $(\vec{s_0}, \vec{s_0'}) = \hat{h}^{D'}([])$. If $\hat{h}^{D'}$ is a homomorphism, then there exists a binary operator $\odot$ such that $((\vec{s}, \vec{s'}) = \hat{f}^{D'}(\sigma))$:

$$\hat{h}^{D'}(x \bullet y) = (\vec{s}, \vec{s'}) \odot \hat{h}^{D'}([a_1, \ldots, a_k])$$
$$= (\vec{s}, \vec{s'}) \odot \left(\cdots(((\vec{s_0}, \vec{s_0'})\hat{\oplus}a_1)\hat{\oplus}\cdots)\hat{\oplus}a_k\right). \quad (2)$$

First, the following proposition, adapted from Theorem 6.2 of [11], indicates that $\vec{s'}$ is not relevant to the discovery of the lifting $\hat{h}^{D'}$.

Proposition 8.1. *If there exists a $\odot$-homomorphic lifting $\hat{h}^{D'}$ of h, then there exists a $\circledast$-homomorphic lifting $\hat{h}^{D'}$ where for all $c, d \in D$ and $c', d' \in D'$, there exists functions θ and θ' such that*

$$(c, c') \circledast (d, d') = (\theta(c, d, d'), \theta'(c, d, c', d')).$$

The significance of Proposition 8.1 is that the value of the D component of the join result (i.e. $\theta(c, d, d')$) need not depend on the value of the D' component of its left input (i.e. c'). Interpreting this for equation 2, we conclude that the value of $\pi_D(h(x) \odot \hat{h}^{D'}([a_1, \ldots, a_k]))$, only depends on $h(x)$ and $\hat{h}^{D'}([a_1, \ldots, a_k])$. Therefore, one can imagine an algorithm that starts from an arbitrary state $\vec{s} = h(x)$ and tries to *guess* what $\hat{h}^{D'}([a_1, \ldots, a_k])$ should look like (as in what D' should be) so that such a join exists. Specifically, there is a function θ such that:

$$(\cdots((\vec{s} \oplus a_1) \oplus \cdots) \oplus a_k) = \quad (3)$$
$$\theta(\vec{s},\ h([a_1, \ldots, a_k]),\ \pi_{D'} \circ \hat{h}^{D'}([a_1, \ldots, a_k]))$$

and, the third component of θ is the auxiliary computation that needs to be discovered. Note that θ could stand for the computation of a loop nest, in contrast to the setting in [11] where it stood for an expression (i.e. loop-free code), which means the lifting algorithm in [11] is not applicable and a new lifting algorithm is required. Equation 3 is the key to our algorithmic solution. The left hand side corresponds to the sequential execution and the right hand side corresponds to a parallel one. Since the join does not have access to the input (i.e. $a_1, \ldots, a_k$), the value of $\pi_{D'} \circ \hat{h}^{D'}([a_1, \ldots, a_k])$ (i.e. the extra information in the signature of $\hat{h}^{D'}$) has to be computed by the worker threads and passed on to the join.

Example 8.2. Figure 8 illustrates the two sides of Equation 3 for the maximum top-left rectangle example of Section 2.2. Variables in red indicate the values of state variables from $\vec{s}$. Each α technically should include a field for `rec[]` and a field

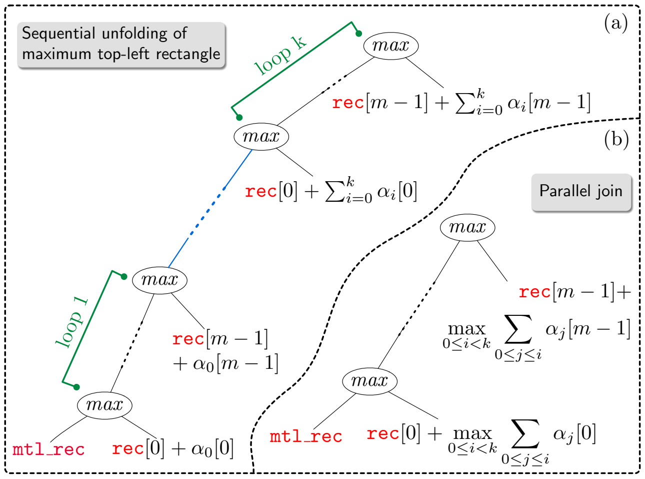

Sequential unfolding of maximum top-left rectangle
(a)
loop k
max
rec[m − 1] + ∑_{i=0}^k α_i[m − 1]
max
rec[0] + ∑_{i=0}^k α_i[0]
(b)
Parallel join
max
rec[m − 1]+ max_{0≤i<k} ∑_{0≤j≤i} α_j[m − 1]
loop 1
max
rec[m − 1] + α_0[m − 1]
max
max
mtl_rec
rec[0] + α_0[0]
mtl_rec
rec[0] + max_{0≤i<k} ∑_{0≤j≤i} α_j[0]

Figure 8. Sequential (a) vs parallel (b) computations.

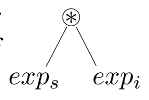
for `mtl_rec` of the saved states after summarization. But, we abuse notation and take $\alpha[i]$ to mean α.`rec`$[i]$ for brevity. Note that the sequential computation executes k instances of a loop that iterates m times to update the value of `mtl_rec` variable; one for each row of the original input. However, in the parallel join, there is budget only for one (or generally constantly many) of these loops. The two (expression) trees (a) and (b) correspond to equivalent computations. The tree (b) is more compact provided that the values of the terms $max_{0\le i<k} \sum_{0\le j\le i} \alpha_j[l]$ are available (i.e. computed before the join). These are exactly the auxiliary computations that are extrapolated, once the left tree (a) is rewritten to the equivalent right tree (b), and are stored in the `max_rec[]` variable in Figure 5(c).

8.2 The Algorithm

If one starts from an arbitrary unfolding of the sequential computation (i.e. the lefthand side of Equation 3 for an arbitrary k), and attempts to *rewrite* it to a form that adheres to the righthand side, then one can extrapolate a guess for $\pi_{D'} \circ \hat{h}^{D'}$ from $\pi_{D'} \circ \hat{h}^{D'}([a_1, \ldots, a_k])$. Let us assume an oracle *Normalize* performs the left-to-right hand side transformation, and another oracle *Discover-Recursion* discovers the implementation of the D' components of $\hat{h}^{D'}$. The *Normalize* oracle transforms an expression into another, while the goal of *Discover-Recursion* is to synthesize a function f' from the expression of its unfolding on an input sequence. We propose heuristic algorithms implementing these two oracles.

The algorithm for *Normalize* uses (generic) algebraic equalities, applies them step by step until it reaches the desired form. The key question is, how would the algorithm know that it has reached its target? To characterize this, we need to define a normal form for θ.

Normal form. An expression e_c over scalar variables is in *constant normal form* if it is of the form illustrated on the right where exp_s



is an expression containing only state variables, exp_i is an expression of input variables of $a_1, a_2, \ldots a_k$, and $\circledast$ stands for a constant-size expression skeleton consisting of operators and constants (i.e. no variables).

Definition 8.3. An expression e is in $\ovee$-*recursive normal form* if it is defined recursively using an operator $\ovee$ as $e = e_c \mid e_c \ovee e$, where e_c is in constant normal form (the base case) and $\ovee$ satisfies the same condition as $\circledast$.

For example, in Figure 8(b), each leaf of the tree is in constant normal form, and therefore, the entire tree is in *max*-recursive normal form.

The normal form does not have to be unique, and in the context of parallelization, one aims for the *simplest* normal form. Intuitively, the constant normal form corresponds to a constant time join. The expressions over input variables are precisely what need to be *additionally* computed and made available to the *parallel join*. However, the parallel join, in general, may not be constant time and the recursive nature of the definition addresses this. Note that the definition can be easily generalized to higher dimensions by replacing the constant normal form by another recursive normal form (over a distinct fresh operator, e.g. $\owedge$).

Normalization. Implementing an ideal rewriting oracle is impossible, since the problem of existence of a normal form is undecidable in the general case (equivalent to *the word problem*). There has been a lot of research in the area of rewrite systems notably for associative and commutative operators [28, 31] that can inspire several heuristics for normalization. We employ a cost-based search for the normal form as a heuristic to work around the undecidability. Our algorithm uses a set of standard algebraic equalities as rewrite rules $\mathcal{R}$ and searches for the *shortest* normal form. The rules in $\mathcal{R}$ are applied to an expression if they reduce its cost, and the rewriting process terminates when no rule can be applied.

Our algorithm operates in two phases. In the first phase, its goal is to find a constant normal form. If it succeeds, the task is done (e.g. this is the case for lifting the example from the introduction). If it fails, the second phase tries to rewrite the result of the first phase into a recursive normal form. Both phases perform a cost-based search and are distinguished by their cost functions. The cost function for the first phase is defined by the number of occurrences and the depth of the state variables (of h) in the expression tree. The cost function is identical to the one from [11], which is no coincidence since the approach in [11] only uses constant normal forms.

In the second phase, the algorithm makes a guess about $\ovee$, inspired by expression e which is the result of the first phase, and attempts to rewrite e to a $\ovee$-recursive normal form. Since phase one forces $\vec{s}$ (or $h(x)$) to the lowest possible depth, operators that appear near the root of expression e are good candidates for $\ovee$. The algorithm chooses the simplest $\ovee$ such that $e = e_c \ovee e'$, where e_c is in constant normal form and e' is an arbitrary expression. The normalization of the second phase is done using a cost function, which is parametric on $\ovee$. For a given $\ovee$, the function combines the sum of the sizes of expressions not in constant normal form and the count of expressions in constant normal form to produce a cost for the entire expression.

Definition 8.4. The cost function $\mathrm{Cost}_{\ovee} : \mathrm{Exp} \to int \times int$ relative to operator $\ovee$ returning a pair $(size, c_{\ovee})$ is defined by:

$$\mathrm{Cost}_{\ovee}(e) = \begin{cases} \mathrm{Cost}_{\ovee}(e_1) + \mathrm{Cost}_{\ovee}(e_2) & \text{if } e = e_1 \ovee e_2, \\ (0, 1) & \text{if } e \text{ is in constant normal form,} \\ (expsize(e), 0) & \text{otherwise.} \end{cases}$$

where $expsize$ returns the size of the expression tree.

Intuitively, the expression is in $\ovee$-recursive normal form when it has cost $(0, \_)$. Moreover, we are interested in the normal form with the smallest count of subexpressions in constant normal form. A rule is applied if it decreases $size$ or, if it increases $c_{\ovee}$ when $size$ cannot be decreased and $size > 0$.

In the example of Figure 8, the expression in (a) is initially in *max*-recursive normal form with cost $(0, km + 1)$. When the expression is rewritten in the first phase (using the cost function from [11]) with the aim of reducing the occurrences and depths of state variables, the cost goes down, but the normal form is lost. Since a constant normal form is not reached at the end of the first phase, the second phase is applied, using the cost function above, which yields the expression in Figure 8(b). This expression is in *max*-recursive normal form with cost $(0, m + 1)$.

If the process fails to find a normal form for $\ovee$, then another operator $\owedge$ is guessed and the process is repeated. Since the expressions are finite-sized, only a finite number of guesses are necessary and the process is guaranteed to terminate. This simple heuristic is a small part of the contributions of this paper, though it is promisingly effective as demonstrated in Section 9.

Recursion discovery The *normalize* heuristic distills the $\pi_D \circ \hat{h}^{D'}([a_1, \ldots, a_k])$ part from its input expression, which we know is required for a join operation to exist. It remains to discover the recursive (i.e. looped) computation that can be added to the original program that would produce this required information. More precisely, the goal of recursion discovery is to synthesize a function that computes the expression $u_k \equiv \pi_{D'} \circ \hat{h}^{D'}([a_1, \ldots, a_k])$ for any $k > 0$. Recursion discovery, based on input/output examples, has been previously studied [25]. Our specific instance of the problem is simpler and amenable to a simple heuristic solution.

Since u_k is recursively (rightward) computable, there is an operator $\boxplus$ such that $u_k = u_{k-1} \boxplus a_k$ for $k > 0$. If u_{k-1} and u_k are simple expressions, we can extrapolate a hint on what $\boxplus$ is by identifying u_{k-1} as a subexpression of u_k (that is precisely the subtree isomorphism [37]). In general, however, u_k, u_{k-1} and a_k can be collections of complex expressions; i.e. lists of expressions as is the case for the example of Section 2.2. The solution remains the same, except, we identify families of subtree isomorphisms. In our implementation, we simplify the problem further by looking for specific patterns of subtree isomorphisms corresponding to recursion schemes such as *zip*, *scans* or *folds*. For example, a *zip* operator translates to having each expression in u_{k-1} isomorphic to a subtree of one expression in u_k.

For example, the expression sequence $max_{0 \le i < k} \sum_{0 \le j \le i} \alpha_j[l]$ (for $0 \le l < m$) from Figure 8(b) can be computed in an auxiliary array `max_rec[]` using a *zip* operation and the state variable `rec[]` as follows:

$$\texttt{max\_rec} = zip\ (max)\ \texttt{rec}\ \texttt{max\_rec}.$$

More details on the recursion discovery algorithm can be found in [12].

Complexity. Our proposed lifting algorithm bounds an infinite search space through a cost function. In the worst case, the algorithm has to explore a full-size tree with the depth of the cost of the original expression (since the cost is strictly decreasing), which is linear on the size of the expression. The branching factor is bounded by the number of rewrite rules multiplied by the size of the input expression, which provides a very coarse upper bound on the number of applicable rules at each step. For an expression of size n and given m rewrite rules, the worst case complexity is $O((nm)^n)$. In practice, however, far fewer than mn rules are applicable at each point since only rules that decrease the expression cost are considered applicable, and the lifting algorithm is lightning fast, taking less than a second for all our benchmarks.

9 Experimental Evaluation

Implementation Our methodology is implemented as an addition to our existing tool ParSynt [13]. We employ a new incremental approach to synthesizing the join to mitigate the large search problem. For a function f with range D, the state of the loop is partitioned into incrementally larger substates $D_1 \subset D_2 \subset \cdots \subset D$ such that the computation in f of variables in D_i does not depend on variables in D_{i+1}. Once a join for the range restriction of f to D_i has been synthesized, we only need to synthesize the join components for the variables $D_{i+1} \setminus D_i$ to obtain the join for the range restriction to D_{i+1}. This remark drives our incremental approach to join synthesis. All synthesis times improve as a result of this strategy, but the complex ones improve more substantially; for example, for *maximum top-left subarray*, the synthesis time is reduced from over 1000 to 116.3 seconds.

9.1 Evaluation

To the best of our knowledge, there is no tool or technique that can parallelize the class of programs considered in this paper, and therefore, our evaluation of ParSynt is not comparative. The theoretical results presented in this paper justify many of the choices made in our proposed methodology. There are two key parts of the algorithmic modules, however, that require empirical evaluation:

1. The effectiveness of the heuristics proposed in Section 8 for lifting; the (necessarily) incomplete algorithm has no theoretical guarantees for success.
2. The efficiency of SyGuS-based solution for parallel join generation and loop summarization; it is impossible to theoretically predict join synthesis times.

Benchmarks. We use a diverse set of benchmarks, where some implement highly non-trivial single-pass algorithms over multidimensional arrays. Since a standard set of benchmarks for this problem space does not exist, we included any example we could find in the related work and parallel programming/algorithms text books, but only those that admit a divide-and-conquer solution according to Definition 6.2. Table 1 includes a complete list. It is important to note that the difficulty of an instance is not directly co-related with code size or the classic notions of dependence such as sparsity of dependencies. Rather, it depends on how complex the required added computation is. Benchmark codes are available online [13].

Performance of ParSynt. Table 1 presents the times spent in the two steps summarized loop and parallel join generation. We do not report the individual times for liftings, since they are negligible compared to the synthesis times; largest lifting time was 12ms (3 orders of magnitude less than smallest synthesis times). Table 1 reports how many auxiliary accumulators were discovered during the lifting. To get a sense of how significant the syntactic restrictions based on the weak inverse are, consider as an example that synthesizing a join for the benchmark `max top strip` without them would take 12.1 seconds. Moreover, using a straightforward syntax-guided synthesis scheme (instead of deductive style algorithm of Section 8), it took over 40 minutes to find the auxiliary for `mbbs` which is arguably the simplest instance that requires lifting.

Quality of the Synthesized Code. A divide-and-conquer parallelization, in the style of this paper, is a data parallel program with no inter-thread dependencies, and therefore, reasonable parallelization speedups are expected. We implemented our produced parallel solutions using Intel's Thread Building Blocks (TBB) library [33] as well as OpenMP, to measure the speedups over the sequential variations. TBB turned out to produce better performing parallel programs. Speedups for four instances (one from each category) at 16 threads are compared below.

	`max bot strip`	mbbs	mode	bp
OpenMP	11.0	8.6	11.0	7.8
TBB	12.7	10.7	11.5	8.9

	2D loop/input																		3D loop/input				2D loop, 1D inputs				
	`sum`	`sorted`	`vertical gradient`	`diagonal gradient`	`min-max`	`min-max col.`	`saddle point`	`max top strip`	`max bottom strip`	`max segment strip`	`max left strip`	**`mtls (Sec. 2.2)`**	**`max bot-left rect.`**	**`max top-right rect.`**	`bp (Sec. 2.1)`	`increasing ranges`	`pyramid ranges`	`overlapping ranges`	`max top box`	`mbbs (Sec. 1)`	`max segment box`	`max left box`	`mode`	`max-dist`	`balanced substr.`	`inter. ranges`	`LCS`
Summarization time	1.2	1.3	1.1	1.2	1.2	1.5	4.6	1.2	1.2	1.2	1.6	1.4	30.2	1.4	5.3	6.2	2.5	1.3	1.3	1.3	1.3	2.1	2.4	1.4	54.9	1.3	2.3
# Aux required	-	1	-	-	-	-	-	-	1	2	-	1	1	1	1*	1*	2	2	-	1	2	-	-	-	-	-	✗
Join synthesis time	2.3	1.1	1.1	1.1	2.5	2.3	5.4	6.1	11.8	64.1	11.2	116.3	216.2	313.5	8.1 †	10.5	2.6	3.3	2.5	7.1	52.3	11.2	22.7	4.0	11.5	1.5	✗

Table 1. Experimental results for performance of ParSynt . Times are in seconds. "–" indicates that lifting was not required. Hardware: laptop with 8G RAM and Intel dual core m3-6Y30. The starred numbers of the middle row correspond to auxiliaries for a *memoryless lift*, and otherwise for a *homomorphism lift*. †: time reported is spent by the solver to answer unsat, since the summarized loop is not parallelizable.

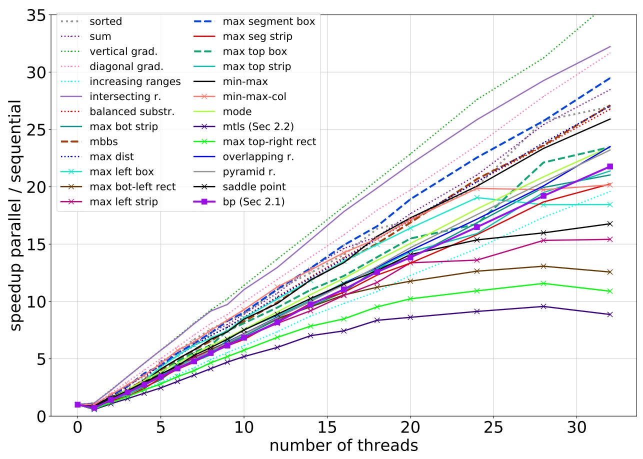


Figure 9. Speedups relative to the sequential implementation. *Hardware:* 8 eight-core Intel X6550 processors (64 cores total) and 256G of RAM running 64-bit Ubuntu

Figure 9 illustrates the TBB speedups for up to 32 cores. In the experiments, the size of the input arrays is about 2bn elements and the grain size is set at 50k. On average, the speedups are close to linear on the number of cores up to around 32 cores. The examples with smaller speedups over larger number of cores are those that have a more complex parallel join operator; in particular, those with looped joins. It is also known [8] that the overhead of TBB increases with more cores and becomes very significant at 32 cores.

Correctness. Rosette performs bounded verification for the solutions it generates. To have correctness over all inputs, we use the same scheme as [11] to produce correctness proofs verified through Dafny [27]. The majority of the programs were verified using the same proof generation scheme as [11]. However, the bold benchmarks in the table required additional *simple and generic* lemmas that lift standard algebraic identities over integers to those over sequences of integers; for example, $\vec{x} + max(\vec{y}, \vec{z}) = max(\vec{x} + \vec{y}, \vec{x} + \vec{z})$.

10 Discussion and Future Work

Input Programs. Theoretically, our approach admits any program that can be semantically translated to a nested system of recurrence equations. In this model, each loop is a system of recurrence equations nested in the corresponding system for its surrounding loop. This is in contrast to other widely used models like System of Affine Recurrence Equations (SARE) [39], designed to track dependencies, that represent the loop nest as a flat system of equations associated with an iteration domain. The only strict limitation for our input programs is that the input should not be modified by the program. More details on our input model can be found in [12].

Limitations. Not all loop nests admit an efficient divide-and-conquer parallel solution with a syntactic divide operator that is the inverse of concatenation; for example, *quick sort* is a divide-and-conquer solution with a non-trivial *divide* operation whereas *merge sort* is a divide-and-conquer solution with a divide that is inverse of concatenation. Synthesis of non-trivial divide operators is an interesting topic of research for future work.

The dynamic programming instances considered in Bellmania [23] also do not admit an efficient parallelization according to Definition 6.2. Bellmania uses *tiling* (a different *divide*) and a necessary scheduling of dependent tiles to correctly parallelize dynamic programming code. The produced code, however, is not fully data parallel in the manner resulting from manufacturing homomorphisms. A homomorphism can be executed in parallel without the need for scheduling.

LCS (longest common substring), a dynamic programming algorithm, can be rewritten to admit an efficient parallelization (according to Definition 6.2). We used ParSynt on the modified code, which failed to parallelize it. This is due to the fact that the auxiliary accumulators required for its lifting are *conditional*, and therefore fall beyond the reach of the heuristics of recursion discovery currently implemented in ParSynt . To sum up, within the class of programs that do admit efficient parallelizations (as defined in Definition 6.2), the limitations of ParSynt are mainly due to the heuristic natures of the implementations of the *normalization* and *recursion discovery* methods. For more complex computations, one can imagine that limitations of SyGuS solvers can play a role in the failure to discover a join that theoretically exists.

Predictability. Due to the semantic nature of our approach, one cannot predict parallelizability of a loop nest based on any of its syntactic properties. Since parallelizability is equivalent to the computation being semantically a homomorphism (or liftable to one), the only way to know if a loop nest is parallelizable is to try to parallelize it. In fact, a negative is quite hard to prove as the proof of Theorem 6.4 indicates.

Future work. The focus of this paper (and its predecessor [11]) has been on synthesizing homomorphisms which have simple fixed divide operations. An interesting direction for future research would be solutions for synthesizing non-trivial divide operations. Note that with the join and (potentially) the lifting being unknown, throwing in an unknown divide operation in the synthesis problem can make it substantially more difficult/interesting to solve.

11 Related Work

There is a vast body of literature on parallelizing code. We review only the closely related work to our approach here.

Homomorphisms for Parallelization Our work is most closely related to those that exploit homomorphisms for parallelization [18, 30], and builds up on our recent work [11], where sequence (list) homomorphisms are automatically synthesized to parallelize simple (non-nested) loops. This paper is a highly non-trivial generalization of the work in [11] to arbitrarily nested loops. Less recent attempts in using derivation of list homomorphisms for parallelization included methods based on the third homomorphism theorem [16, 18, 30], function composition [15], and quantifier elimination [29], as well as those based on recurrence equations [4]. These techniques are either not fully automatic, or rely on additional guidance from the programmer beyond the input sequential code.

Simple Loop Parallelization More recently in [36], symbolic execution is used to identify and break dependencies in loops that are hard to parallelize. This approach can be regarded as a dynamic counterpart to that of [11], and its scope is similarly limited to simple loops. In distributed computing, a related vein of research has been focused on automatic production of map/reduce programs, for example, by means of specific rewrite rules [35] or synthesis [38]. GraSSP [14] parallelizes a sequential implementation by analyzing data dependencies and its scope is functions over lists. The (constant sized) prefix information used in [14] is essentially a special case of the auxiliary accumulators in [11].

Program Synthesis for Parallel Code Generation In this paper, the specification for synthesis is the sequential input program, and no other information (such as input/output examples or a sketch) is required from the programmer. Synthesis techniques have been leveraged for parallel programs before, instances of which include synthesis of distributed map/reduce programs from input/output examples [38] and optimization and parallelization of stencils [24]. Aside from the use of synthesis, these problem areas and the solutions have little in common with the scope and approach in this paper. Bellmania [23] synthesizes divide-and-conquer variations of a class of dynamic programming algorithms *with programmer's guidance* and the notion of divide-and-conquer (with a fixed divide) in this paper differs from the one that Bellmania uses.

Parallelizing Compilers and Runtime Environments Automatic parallelization in compilers has been a prolific and highly effective field of research, with source-to-source compilers using highly sophisticated methods to parallelize generic code [2, 7, 21] or more specialized nested loops with polyhedral optimization [3, 40]. There is a body of work specific to reductions and parallel-prefix computations [5, 22, 26] that deals with dependencies that cannot be broken. In contrast to correct source-to-source transformation achieved through provably correct program transformation rules, the aim of this paper is to use search (in the style of synthesis), which facilitates the discovery of equivalent parallel implementations that are not reachable through a pre-established set of correct transformation rules. There is work in the literature on breaking static dependencies at runtime [34] based on the observation that actual runtime dependencies happen rarely in some sparse problems. The scope of applicability of our method is different and we consider these techniques to be complementary. In [6], a static two-phase solution is proposed that resolves dependencies in the first phase, and can proceed to perform independent parallel tasks in the second. We view the approach in this paper as complementary to these techniques.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design 2013 (FMCAD' 13)*. IEEE, 1–8.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-performance Computing. *ACM Comput. Surv.* 26, 4 (Dec. 1994), 345–420.

[3] Cedric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, 7–16.

[4] Yosi Ben-Asher and Gadi Haber. 2001. Parallel Solutions of Simple Indexed Recurrence Equations. *IEEE Trans. Parallel Distrib. Syst.* 12, 1 (Jan. 2001), 22–37.

[5] Guy E Blelloch. 1993. Prefix sums and their applications. In *Synthesis of Parallel Algorithms* (1st ed.). Morgan Kaufmann Publishers Inc.

[6] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proceedings of Symposium on Principles and Practice of Parallel Programming, PPOPP 2012*. 181–192.

[7] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. 1996. Parallel Programming with Polaris. *Computer* 29, 12 (Dec. 1996), 78–82.

[8] Gilberto Contreras and Margaret Martonosi. 2008. Characterizing and improving the performance of Intel Threading Building Blocks. In *4th International Symposium on Workload Characterization, 2008*. 57–66.

[9] Daniel Cordes, Heiko Falk, and Peter Marwedel. 2009. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO 2009*. IEEE, 136–146.

[10] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[11] Azadeh Farzan and Victor Nicolet. 2017. Synthesis of Divide and Conquer Parallelism for Loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. 540–555.

[12] Azadeh Farzan and Victor Nicolet. 2019. Modular Synthesis of Divide-and-Conquer Parallelism for Nested Loops (Extended Version). arXiv:cs.PL/1904.01031

[13] Azadeh Farzan and Victor Nicolet. 2019. Parsynt. http://www.cs.toronto.edu/~victorn/parsynt/index.html

[14] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. 2017. Gradual Synthesis for Static Parallelization of Single-pass Array-processing Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. 572–585.

[15] Allan L. Fisher and Anwar M. Ghuloum. 1994. Parallelizing Complex Scans and Reductions. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. 135–146.

[16] Alfons Geser and Sergei Gorlatch. 1997. Parallelizing Functional Programs by Generalization. In *Proceedings of the 6th International Joint Conference on Algebraic and Logic Programming (ALP '97-HOA '97)*. 46–60.

[17] Jeremy Gibbons. 1996. The Third Homomorphism Theorem. *J. Funct. Program.* 6, 4 (1996), 657–665.

[18] Sergei Gorlatch. 1996. Systematic Extraction and Implementation of Divide-and-Conquer Parallelism. In *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP '96)*. 274–288.

[19] Sergei Gorlatch. 1999. Extracting and Implementing List Homomorphisms in Parallel Program Development. *Sci. Comput. Program.* 33, 1 (Jan. 1999), 1–27.

[20] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. 2006. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*. IEEE, 57–66.

[21] Hwansoo Han and Chau-Wen Tseng. 2001. A comparison of parallelization techniques for irregular reductions. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*. 27.

[22] W Daniel Hillis and Guy L Steele Jr. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (1986), 1170–1183.

[23] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles E. Leiserson, and Rezaul Alam Chowdhury. 2016. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*. 145–164.

[24] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. 711–726.

[25] Emanuel Kitzelmann and Ute Schmid. 2006. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research* 7, Feb (2006), 429–454.

[26] Richard E Ladner and Michael J Fischer. 1980. Parallel prefix computation. *Journal of the ACM (JACM)* 27, 4 (1980), 831–838.

[27] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)*. Springer-Verlag, 348–370.

[28] Claude Marché and Xavier Urbain. 1998. Termination of associative-commutative rewriting by dependency pairs. In *International Conference on Rewriting Techniques and Applications*. Springer, 241–255.

[29] Akimasa Morihata and Kiminori Matsuzaki. 2010. Automatic Parallelization of Recursive Functions Using Quantifier Elimination. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*. 321–336.

[30] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2007. Automatic Inversion Generates Divide-and-conquer Parallel Programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 146–155.

[31] Paliath Narendran and Michael Rusinowitch. 1991. Any ground associative-commutative theory has a finite canonical system. In *International Conference on Rewriting Techniques and Applications*. Springer, 423–434.

[32] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. In *ACM SIGGRAPH 2008 classes*. ACM, 16.

[33] Chuck Pheatt. 2008. Intel® threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.

[34] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, 12–25.

[35] Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. 2014. Translating Imperative Code to MapReduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. 909–927.

[36] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing User-defined Aggregations Using Symbolic Execution. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. 153–167.

[37] Ron Shamir and Dekel Tsur. 1999. Faster subtree isomorphism. *Journal of Algorithms* 33, 2 (1999), 267–280.

[38] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, 326–340.

[39] YN Srikant and Priti Shankar. 2002. *The compiler design handbook: optimizations and machine code generation*. CRC Press.

[40] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. 2006. Polyhedral Code Generation in the Real World. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*. 185–201.