Modular Synthesis of Divide-and-Conquer Parallelism for Nested Loops (Extended Version)

Azadeh Farzan
Department of Computer Science
University of Toronto
Toronto, Canada
azadeh@cs.toronto.edu

Victor Nicolet
Department of Computer Science
University of Toronto
Toronto, Canada
victorn@cs.toronto.edu

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. Then, the *summarized* version of the loop nest is 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.

Keywords Divide and Conquer Parallelism, Program Synthesis, Homomorphisms

1 Introduction

The advent of multicore computers and development of APIs like OpenMP [11], CUDA [33], and TBB [35] 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.

This is the extended version of PLDI 2019 paper by the same authors which includes the proofs of theorems and additional details.

PLDI'19, June 22–28, 2019, Phoenix, AZ, USA
2019.

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 to have fundamentally unbreakable data flow dependencies.

Consider the code in Figure 1(a), that implements a sequential solution to the problem of computing, for a three dimensional $n \times m \times \ell$ array A (with both positive and negative elements), the sum of the elements of a subarray $A[k..n-1, 0..m-1, 0..\ell-1]$ (for all $0 \leq k < n$) which has the maximum sum compared to all other such subarrays. Intuitively, considering the array as a 3D box with height n, the goal is to discover the maximum sum of boxes of different heights, with the same width, length and bottom as the input box.

(a)
```
int max_bot_box_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]; } }
   max_bot_box_sum =
     max(max_bot_box_sum + plane_sum, 0);
}
```

(b)
```
int max_bot_box_sum = 0;
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);
}
```

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

Figure 1. Maximum bottom box sum.

Note that this optimal sequential implementation runs in *single pass* linear time over the input 3D array, at the cost of creating unbreakable loop dependencies. A less efficient solution that would enumerate all boxes would have been easier to parallelize.

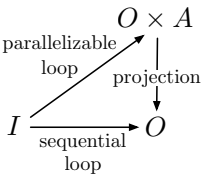
It is easy to observe that the code is not (divide-and-conquer) parallelizable. Let us assume it is. There then exists a binary function $\odot$ that can combine results of two instances

of the code (*mbbs*) run on two adjacent boxes to produce the same results for the *concatenated* box.

$$mbbs\left(\begin{array}{c}b\\b'\end{array}\right) = mbbs\left(b\right) \overset{?}{\odot} mbbs\left(b'\right)$$

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 $mbbs(b \bullet b')$.

Nonexistence of the join operator indicates 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.



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 [12] 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 [10, 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(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

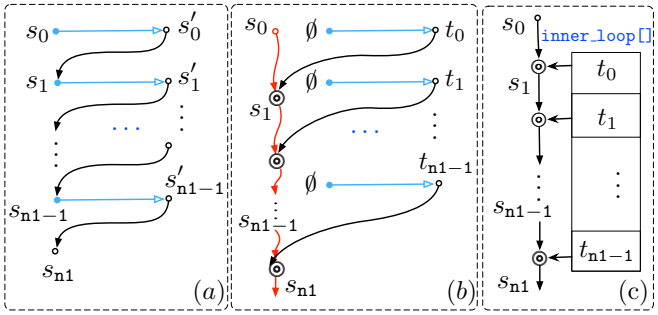


Figure 2. Dependencies in a general sequential nested loop (a) vs. a memoryless one (b), which is summarized in (c).

(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, the 3-nested loop of Figure 1(a) is summarized to a single loop (illustrated on the right).

```
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);
}
```

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 [12] can parallelize the summarized loop above while it is not applicable to the original loop in Figure 1(a).

Summarization, however, does not always yield a non-nested loop like the one above, and therefore, the approach in [12] cannot always parallelize 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 [12] 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 11 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.

```
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.

```
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.

We encourage the reader to manually parallelize the outer loop to get a sense of the difficulty of this problem.

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 (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 10).

```
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)

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 ◎ from 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 where each cell `max_rec[j]` 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 [12]. 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.

```
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 Figure5(c).

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

This section introduces the notation used in the remainder of the paper. While the formal work is based on studying functions on sequences, the description of the algorithm requires to define our inputs programs and a model for loop bodies which can be translated to a functional form.

3.1 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 $\bullet : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ is defined over sequences in the standard way, and 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 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.

3.2 Model of a loop body

Our input programs are imperative whereas the representation of the loop nests for the theoretical results in this paper and for algorithmic units is functional. The input program is translated to nested systems of equations, which can easily be converted to a recursive functional form. Here, we quickly outline the steps of this transformation and define the program models at each stage.

Input programs Figure 7 presents the syntax of the input sequential programs. We assume an imperative language with basic constructs for branching and looping. Variables are of scalar types `int` or `bool` and we can build nested sequences from these types.

For readability in our paper, we use simple iterators and integer indexes (instead of the generic $i \in \mathcal{I}$). In principle, any collection with an iterator and a split function that implements the inverse of concatenation works. There has been a lot of research on iteration spaces and iterators (e.g. [44] in the context of translation validation and [25] in the context of partitioning) that formalize complex traversals by abstract iterators.

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.

$$
\begin{array}{lr}
v \in \mathsf{LhVar} ::= v'[e] & v' \in \mathsf{LhVar}, e \in \mathsf{Exp} \\
\quad | \; x & x \in \mathsf{Var} \\
\\
e \in \mathsf{Exp} ::= e \circ e' & e, e' \in \mathsf{Exp} \\
\quad | \; e \lesseqgtr e' & e, e' \in \mathsf{Exp} \\
\quad | \; be \wedge be' | \neg be & be, be' \in \mathsf{Exp} \\
\quad | \; v & v \in \mathsf{LhVar} \\
\quad | \; \mathsf{if}\ be\ \mathsf{then}\ e\ \mathsf{else}\ e' & \\
\quad | \; k & k \in \mathbb{Z}, \mathbb{Q}, \mathbb{R} \\
\quad | \; \mathsf{true} \mid \mathsf{false} & \\
\mathsf{Program} ::= c; c' & c, c' \in \mathsf{Program} \\
\quad | v := e & v \in \mathsf{LhVar}, e \in \mathsf{Exp} \\
\quad | \mathtt{if}\ (e)\ \{c_\top\}\ \mathtt{else}\ \{c_\bot\} & be \in \mathsf{Exp}, c_\top, c_\bot \in \mathsf{Program} \\
\quad | \mathtt{for}\ (i \in \mathcal{I})\ \{\ c\ \} & i \in \mathsf{Iterator}
\end{array}
$$

Figure 7. Program Syntax . The binary $\circ$ operator represents any arithmetic operation $(+, -, *, /)$, $\lesseqgtr$ operator represents any comparator $(<, \leq, >, \geq, =, \neq)$. $\mathcal{I}$ is an iteration domain, and $\wedge$ operator represents any boolean operation $(\wedge, \vee)$.

$$
E = \begin{pmatrix}
v_1 = Exp_1(\mathsf{SVar}, \mathsf{IVar}) \\
\cdots \\
(s_{i_1}, s_{i_2}, \ldots, s_{i_p}) = \mathtt{for}\ (j \in \mathcal{J})\ \{\ E'\ \} \\
\cdots \\
v_q = Exp_q(\mathsf{SVar}, \mathsf{IVar})
\end{pmatrix}
$$

Figure 8. Nested systems of equations: E' is nested in E.

Nested systems of equations A loop body is modelled by a system of ordered recurrence equations, where each equation is either a *simple* equation or a *loop* equation. Given state variables $\mathsf{SVar} = \{s_1, \ldots, s_q\}$ and input variables IVar, a simple equation is of the form $v_i = Exp_i(\mathsf{SVar}, \mathsf{IVar})$ where $v_i \in \mathsf{LhVar}$ and the right hand side is a constant-time computable expression of the input program (see Figure 7). A loop equation is of the form of the middle line of Figure 8, where $\{s_{i_1}, s_{i_2}, \ldots, s_{i_p}\}$ are all the variables modified by the loop body, $j \in \mathcal{J}$ is an arbitrary iterator, and E' is the body of the nested loop.

The body of any loop in our input language can be translated to the system of recurrence equations defined above.

Conversion to a system of equations Converting the body of a loop nest to a system of ordered recurrence equations (of the type outlined by Figure 8) is a process that involves a transformation of the loop and conditional statements, and a mapping of simple assignments ($v := e$) to equations.

For a conditional statement $\mathtt{if}\ (e)\ \{c_\top\}\ \mathtt{else}\ \{c_\bot\}$ where e is an expression of the input program and $c_\top$ and $c_\bot$ are two programs, we apply the conversion procedure recursively to each of the programs, and obtain two systems of ordered recurrence equations $E_\top$ and $E_\bot$. For each variable v_i that appears either in $E_\top$ or $E_\bot$ on the left hand side of an equation,

we add an equation of the form $v_i = e\ ?\ Exp_\top\ :\ Exp_\bot$ in the current system, where the expressions $Exp_\top$ and $Exp_\bot$ are the expressions on the right hand side of $v_i = \ldots$ in $E_\top$ and $E_\bot$ respectively. If the equation assigning v_i is not present in one of the branches, the expression on the right hand side is just the variable itself (the branch does not modify it).

For a loop $\texttt{for}\ (i \in \mathcal{I})\ \{\ c\ \}$ where c is a program, we apply the conversion procedure to c and obtain a system E'. In the parent system, we add the equation $(s_{i_1}, s_{i_2}, \ldots, s_{i_q}) = \texttt{for}\ (i \in \mathcal{I})\ \{\ E'\ \}$ where $(s_{i_1}, s_{i_2}, \ldots, s_{i_q})$ are the state variables modified by the body c. If only one cell of a collection is assigned in the loop body c, we consider that the whole collection has been modified.

Conversion to functional form Given a loop body in the form of a system of ordered recurrence equations, one can produce a function (implemented in a simple functional language with let-bindings) by replacing each equation by a binding and creating a recursive function for each of the inner loops. We choose to represent arrays by lists, and an assignment to a cell in the system of equations is translated by binding a list where the corresponding element has been modified.

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.*

Proof. Since h is $\odot$-homomorphic, for all sequences $\sigma, \sigma' \in \mathcal{S}^n$ we have:

$$h(\sigma \bullet \sigma') = h(\sigma) \odot h(\sigma')$$

and therefore, more specifically, for all $\sigma \in \mathcal{S}^n$ and $\delta \in \mathcal{S}^{n-1}$ we have:

$$h(\sigma \bullet [\delta]) = h(\sigma) \odot h([\delta])$$

. Now, let $g : \mathcal{S}^{n-1} \to D'$ be defined so that $g(\delta) = h([\delta])$ and let $\oplus = \odot$ in Definition 4.2; we can conclude that h is memoryless. □

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.*

Proof. $\Rightarrow$: By Proposition 4.4, we can conclude that f is memoryless. Let $x, y \in \mathcal{S}_D$, $y = y_1 \bullet \ldots \bullet y_k$, $y_i = g(\delta_i)$ for some $\delta_i \in \mathcal{S}^{n-1}$, $x = x_1 \bullet \ldots \bullet x_m$, and $x_i = g(\gamma_i)$ for some $\gamma_i \in \mathcal{S}^{n-1}$.

$$\begin{aligned}
h(x \bullet y) &= h(x \bullet [y_1 \ldots y_k]) \\
&= h(x \bullet [y_1 \ldots y_{k-1}]) \oplus y_k \\
&= (h(x \bullet [y_1 \ldots y_{k-2}]) \oplus y_{k-1}) \oplus y_k \\
&= \ldots \\
&= (\ldots (h(x) \oplus y_1) \oplus \ldots) \oplus y_k \\
&= \ldots \\
&= (\ldots (h([]) \oplus x_1) \oplus \ldots) \oplus y_k \\
&= (\ldots (f([]) \oplus g(\gamma_1)) \oplus \ldots) \oplus g(\delta_k) \\
&= \ldots \\
&= f(\gamma_1 \bullet \ldots \bullet \gamma_m \bullet \delta_1 \bullet \ldots \delta_k) \\
&= f(\gamma_1 \bullet \ldots \bullet \gamma_m) \odot f(\delta_1 \bullet \ldots \delta_k) \\
&= \ldots \\
&= h(x) \odot h(y)
\end{aligned}$$

$\Rightarrow$: Let $\sigma = \gamma_1 \bullet \ldots \bullet \gamma_m$, $\sigma' = \delta_1 \bullet \ldots \delta_k$, $y = y_1 \bullet \ldots \bullet y_k$, $y_i = g(\delta_i)$, $x = x_1 \bullet \ldots \bullet x_m$, and $x_i = g(\gamma_i)$.

$$\begin{aligned}
f(\sigma \bullet \sigma') &= f(\gamma_1 \bullet \ldots \bullet \gamma_m \bullet delta_1 \bullet \ldots delta_k) \\
&= \ldots \\
&= (\ldots (f([]) \oplus g(\gamma_1)) \oplus \ldots) \oplus g(\delta_k) \\
&= (\ldots (h([]) \oplus x_1) \oplus \ldots) \oplus y_k \\
&= \ldots \\
&= h(x \bullet y) \\
&= h(x) \odot h(y) \\
&= \ldots \\
&= f(\gamma_1 \bullet \ldots \bullet \gamma_m) \odot f(\delta_1 \bullet \ldots \delta_k) \\
&= f(\sigma) \odot f(\sigma')
\end{aligned}$$

□

Theorem 4.7 states the necessary and sufficient conditions for a recursive function to be parallelizable. 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 a parallel computation of f. Parallelizability of h corresponds to the existence of the *reduction* part of a parallelizaiton 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 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 to the function remembers the entire input, and the join performs the original computation over the concatenated inputs from scratch, ignoring the partially computed results.

Proof. It is straightforward to see that $f \times \iota$ is a $\circledcirc$-homomorphic with the join operator $\circledcirc : (D \times \mathcal{S}^n) \times (D \times \mathcal{S}^n) \to D \times \mathcal{S}^n$ which is defined as

$$\forall a, b \in D, \sigma, \sigma' \in \mathcal{S}^n : \ (a, \sigma) \circledcirc (b, \sigma') = (f(\sigma \bullet \sigma'), \sigma \bullet \sigma')$$

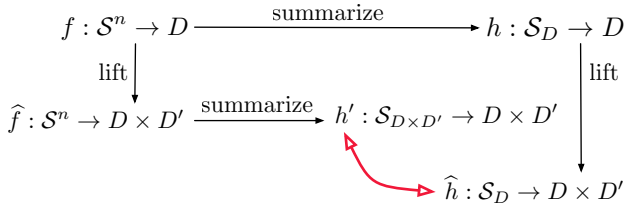
since

$$\begin{aligned} f \times \iota(\sigma \bullet \sigma') &= (f(\sigma \bullet \sigma'), \sigma \bullet \sigma') \\ &= (f(\sigma), \sigma) \circledcirc (f(\sigma'), \sigma') \quad = f \times \iota(\sigma) \circledcirc f \times \iota(\sigma') \end{aligned}$$

□

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*.

Here, we state a result which parallels Theorem 4.7, provides the theoretical guarantee that it is *sound and complete* to use the *summarized* loop for lifting instead of the original. Consider the diagram below:



f is summarized and then lifted on the top, whereas it is first lifted and then summarized on the bottom part of the diagram. Note that $\widehat{h}$ and h' do not have the same function signature; they agree on their ranges, but 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 $\widehat{h}$.

The main ingredients of a lift, that is what the extra information D' is and how it should be computed, are both discoverable through a lifting of the simple function h in place of f.

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 $\widehat{f}$.*

Additionally, the theorem guarantees that auxiliary code synthesized for the summarized loop constitutes a lifting of the original loop.

In order to give a proof of Theorem 5.3, we state and prove each path in the diagram as a separate proposition, which correspond to the *if* and the *only if* directions of Theorem 5.3.

Proposition 5.4. *Let $f : \mathcal{S}^n \to D$ be a rightward) memoryless function defined by helper function g (from Definition 4.2), and let $h : \mathcal{S}_D \to D$ be its summarized version defined through $\oplus$. If h can be lifted to a $\odot$-homomorphic function $h' : \mathcal{S}_D \to D \times D'$, then there exists a lifting $f' : \mathcal{S}^n \to D \times D'$ of f that is $\odot$-homomorphic, and h is equivalent to the summary of f' up to the projection of its input sequence down to domain D.*

Proof. Assume there exists a lifting of h called h' that is $\odot$-homomorphic. Then, for all sequences $x, y \in \mathcal{S}_D$ we have:

$$h'(x \bullet y) = h'(x) \odot h'(y)$$

and therefore, more specifically, for all $x \in \mathcal{S}_D$ and $d \in D$ we have:

$$h'(x \bullet [d]) = h'(x) \odot h'([d])$$

.

Now, let $g' : \mathcal{S}^{n-1} \to D \times D'$ be defined so that $g'(\delta) = h'([g(\delta)])$. Define f' as:

$$\begin{aligned} f'([]) &= h'([]) \\ f'(\sigma \bullet [\delta]) &= f'(\sigma) \odot g'(\delta) \end{aligned}$$

which is by definition *memoryless*. For all $\delta \in \mathcal{S}^{n-1}$, we have:

$$\begin{aligned} \pi_D \circ g'(\delta) &= \pi_D \circ h' \circ g(\delta) \\ &= h \circ g(\delta) \\ &= f([\delta]) \\ &= g(\delta) \end{aligned}$$

based on the assumption that $h([])$ is defined and therefore has to be the unit of $\odot$. It is easy to show (by induction

and definition) that for all $\delta_1, \ldots, \delta_m \in \mathcal{S}^n - 1$ where $\sigma = [\delta_1] \bullet \ldots \bullet [\delta_m]$, we have:

$$f'(\sigma) = h'([g(\delta_1)] \bullet \ldots \bullet [g(\delta_m)]) \tag{1}$$

Let $\sigma, \sigma' \in \mathcal{S}^n$ and $\sigma' = [\delta_1] \bullet \ldots \bullet [\delta_m]$ where $\delta_i \in \mathcal{S}^{n-1}$. We have:

$$\begin{aligned} f'(\sigma \bullet \sigma') &= (\cdots (f'(\sigma) \odot g'(\delta_1)) \odot \cdots) \odot g'(\delta_m) \\ &= f'(\sigma) \odot (g'(\delta_1) \odot \cdots \odot g'(\delta_m)) \\ &= f'(\sigma) \odot f'(\sigma') \end{aligned}$$

by associativity of $\odot$. Therefore, f' is also $\odot$ homomorphic. f' is a lifting of f since:

$$\begin{aligned} \pi_D \circ f'([]) &= \pi_D \circ h'([]) = h([]) = f([]) \\ \pi_D \circ f'(\sigma) &= \pi_D \circ h'([g(\delta_1)] \bullet \ldots \bullet [g(\delta_m)]) \\ &= h([g(\delta_1)] \bullet \ldots \bullet [g(\delta_m)]) \\ &= f(\sigma) \end{aligned}$$

It remains to show that h' is a summary of f' up to projection. Let $\bar{h} : \mathcal{S}_{D \times D'} \to D \times D'$ be the summary of f' defined through $\odot$ that is

$$\begin{aligned} \bar{h}([]) &= f'([]) \\ \forall y \in \mathcal{S}_{D \times D'}, b \in D \times D' :\ \bar{h}(y \bullet [b]) &= \bar{h}(y) \odot b \end{aligned}$$

Observe that h' and $\bar{h}$ have the same range, but the sequences in the domain of $\bar{h}$ have strictly more information in each element of the sequence than those in h'. The claim that we want to prove is that

$$h' = \bar{h} \circ \bar{\pi}_D$$

where $\bar{\pi}_D$ is the natural extension of the projection function from elements to sequence of elements.

We have $h'([]) = f'([]) = \bar{h}([])$, by definition. This serves as our induction base case. Let $y \in \mathcal{S}_{D \times D'}$, and assume that $h' \circ \bar{\pi}_D(y) = \bar{h}(y)$. Let $a \in D \times D'$ and $a = g'(\delta)$ for some $\delta \in \mathcal{S}^{n-1}$:

$$\begin{aligned} h' \circ \bar{\pi}_D(y \bullet [a]) &= h'(\bar{\pi}_D(y) \bullet \pi_D(a)) \\ &= h' \circ \bar{\pi}_D(y) \odot h'(\pi_D(a)) \\ &= h' \circ \bar{\pi}_D(y) \odot h'(\pi_D(g'(\delta))) \\ &= h' \circ \bar{\pi}_D(y) \odot h'(g(\delta)) \\ &= h' \circ \bar{\pi}_D(y) \odot g'(\delta) \\ &= \bar{h}(y) \odot a \\ &= \bar{h}(y \bullet [a]) \end{aligned}$$

□

Proposition 5.5. *Let $f : \mathcal{S}^n \to D$ be a (rightward) memoryless function defined by helper function g (from Definition 4.2), and let $h : \mathcal{S}_D \to D$ be its summarized version. If f can be lifted to a homomorphism $f' : \mathcal{S}_D \to D \times D'$, then there exists a lifting $h' : \mathcal{S}^n \to D \times D'$ of h that is a homomorphism, and h' is equivalent to the summary of f' up to projection of its input sequence down to domain D.*

Proof. Assume there exists a lifting f' of f which is $\odot$ homomorphic. Note that:

$$f'(\sigma \bullet \delta) = f'(\sigma) \odot f'([\delta])$$

Let $g' : \mathcal{S}^{n-1} \to D \times D'$ be defined as $g'(\delta) = f'([\delta])$.

Define h' as:

$$\begin{aligned} h'([]) &= f'([]) \\ \forall x \in \mathcal{S}_D, a \in D(\text{ s.t. } a = g(\delta)) : h'(x \bullet a) &= h'(x) \odot g'(\delta) \end{aligned}$$

Let us argue that h' is $\odot$-homomorphic and a lifting of h. The former is immediately implied by associativity of $\odot$. For the latter, we need to show that $\pi_D \circ h' = h$ (rightward computability of h' is implied by the computability of $\odot$). Observe that:

$$\pi_D \circ h'([]) = \pi_D \circ f'([]) = f([]) = h([])$$

Let $x \in \mathcal{S}_D$ and $x = d_1 \bullet \ldots d_m$ where $d_i = g(\delta_i)$. Then:

$$\begin{aligned} \pi_D \circ h'(x) &= \pi_D(h'(x)) \\ &= \pi_D(f'([\delta_1] \ldots\ldots \bullet [\delta_m])) \\ &= f([\delta_1] \bullet \ldots \bullet [\delta_m]) && (f' \text{ is a lifting of } f) \\ &= h([g(\delta_1)] \bullet \ldots \bullet [g(\delta_m)]) && (\text{by equation 1}) \\ &= h(x) \end{aligned}$$

Finally, it remains to show that h' is a summarized version of f' up to projection. Let $\bar{h} : \mathcal{S}_{D \times D'} \to D \times D'$ be the summary of f' defined through $\odot$ that is

$$\begin{aligned} \bar{h}([]) &= f'([]) \\ \forall y \in \mathcal{S}_{D \times D'}, b \in D \times D' :\ \bar{h}(y \bullet [b]) &= \bar{h}(y) \odot b \end{aligned}$$

The claim that we want to prove is that

$$h' = \bar{h} \circ \bar{\pi}_D$$

where $\bar{\pi}_D$ is the natural extension of the projection function from elements to sequence of elements. The argument is identical to the one made at the end of the proof of Proposition 5.4 to prove the same claim.

□

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 5.6. *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 $\forall \delta \in \mathcal{S}^{n-1}, \sigma \in \mathcal{S}^n :\ \iota'(\sigma \bullet [\delta]) = \delta$.*

Proof. Since f is rightward, there exists a binary operator $\otimes$ and family of functions $\mathbb{G}$ such that for all $\sigma \in \mathcal{S}^n$ and $\delta \in \mathcal{S}^{n-1}$:

$$f(\sigma \bullet \delta) = f(\sigma) \otimes \mathbb{G}(f(\sigma))(\delta)$$

Note that the signature of the lifted function $f \times \iota'$ is $\mathcal{S}^n \to D \times \mathcal{S}^{n-1}$. Let $\oplus : (D \times \mathcal{S}^{n-1}) \times (D \times \mathcal{S}^{n-1}) \to D \times \mathcal{S}^{n-1}$ be defined as:

$$\forall a, b \in D, \delta, \delta' \in \mathcal{S}^{n-1} : (a, \delta) \oplus (b, \delta') = (a \otimes \mathbb{G}(a)(\delta'), \delta')$$

Let $\iota'' : \mathcal{S}^{n-1} \to \mathcal{S}^n$ be defined as $\forall \delta \in \mathcal{S}^{n-1} : \iota''(\delta) = [\delta]$ and let $g : \mathcal{S}^{n-1} \to D \times \mathcal{S}^{n-1}$ to be function that on all inputs δ returns $(\emptyset, \delta)$ for some constant value $\emptyset \in D$.

It is straightforward to see that $f \times \iota'$ is memoryless with the loop join operator $\oplus$ and helper function g since:

$$\begin{aligned} f \times \iota'(\sigma \bullet [\delta]) &= (f(\sigma \bullet [\delta]), \iota'(\sigma \bullet [\delta])) \\ &= (f(\sigma) \otimes \mathbb{G}(f(\sigma))(\delta), \delta) \\ &= (f(\sigma), \iota'(\sigma)) \oplus (\emptyset, \delta) \\ &= f \times \iota'(\sigma) \oplus g(\delta) \end{aligned}$$

Therefore, by definition 4.2, we can conclude that $f \times \iota'$ is memoryless. □

Complexity preservation of the trivial memoryless lift. It is easy to intuitively see why a trivial lift like the above does not increase the time complexity of computation of f. To argue for this, it is easier to think about the loops (instead of functions). Imagine the original function corresponds to the loop:

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < m; j++) {
    ...
  }
}
```

which has complexity $O(nm)$. Then the lifted one would correspond to the loop:

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < m; j++) {
    ...
  }
  ...
  for(int j = 0; j < m; j++) {
    ...
  }
}
```

where the second copy of the inner loop effectively redoes the computation of the inner loop. This still has the complexity $O(nm)$ albeit with larger constants.

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.

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 5.6, in contrast, would suggest a simple admissible lifting which would not lead to as much parallelism.

5.3 Algorithmic Memoryless Lift

Algorithmically, the 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 the same problem as the homomorphism lift of the inner loop nest. The following proposition makes this observation precise.

Proposition 5.7. *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 $\circledcirc$.*

Proof. Consider a multidimensional rightward function $f : \mathcal{S}^n \to D$ that is not memoryless, defined by a family of functions $\mathbb{G} : D \to (\mathcal{S}^{n-1} \to D)$ as in Definition 4.1. The function is effectively defined using the recursive equation $f(\sigma \bullet [\delta]) = f(\sigma) \otimes \mathbb{G}(f(\sigma))(d)$.

Let us show that lifting $\mathbb{G}$ to a family of homomorphisms (Definition 7.4) is sufficient to lift f to a memoryless function. For any $d \in D$, $\mathbb{G}(d)$ is defined by:

$$\begin{aligned} \mathbb{G}(d)([]) &= d \\ \mathbb{G}(d)(\delta \bullet [\gamma]) &= \mathbb{G}(d)(\delta) \ominus \gamma \end{aligned}$$

Imagine that we lift $g_\emptyset = \mathbb{G}(\emptyset)$ for some $\emptyset \in D$ to a homomorphism. We will have a $\hat{g_\emptyset}^{D'}$ such that there exists a $\boxdot$ operator that satisfies for all $\delta, \delta' \in S^{n-1}$:

$$\hat{g_\emptyset}^{D'}(\delta \bullet \delta') = \hat{g_\emptyset}^{D'}(\delta) \boxdot \hat{g_\emptyset}^{D'}(\delta')$$

We define the lifting of $\mathbb{G}(d)$ by using the homomorphic lifting of $g_\emptyset$:

$$\hat{\mathbb{G}(d)}^{D'}(\delta) = d' \boxdot \hat{g_\emptyset}^{D'}(\delta)$$

where $\pi_D(d') = d$ and $\pi_{D'}(d') = \pi_{D'}(\hat{g_\emptyset}^{D'}([]))$.

$\hat{\mathbb{G}(d)}^{D'}$ is naturally a homomorphism since $\hat{g_\emptyset}^{D'}$ is one. We can verify that it is a lifting of $\mathbb{G}(d)$ by projecting to D. We use w the weak inverse of $\hat{g_\emptyset}^{D'}$ defined by $\forall d' \in$

$D', \hat{g_\emptyset}^{D'}(\delta) \circ w\ (d') = d'$.

$$\begin{aligned}
\pi_D \circ \hat{\mathbb{G}(d)}^{D'}(\delta \bullet [a]) &= \pi_D(d' \boxdot \hat{g_\emptyset}^{D'}(\delta)) \\
&= \pi_D(\hat{g_\emptyset}^{D'}(w(d') \bullet \delta)) \\
&= g_\emptyset(w(d') \bullet \delta) \\
&= (\ldots(g_\emptyset(w(d'))) \ominus \delta_1 \ldots \ominus \delta_n) \\
&= (\ldots(d \ominus \delta_1)\ldots \ominus \delta_n) \\
&= \mathbb{G}(d)(\delta)
\end{aligned}$$

Since $\hat{\mathbb{G}(d)}^{D'}$ is a lifting of $\mathbb{G}(d)$ we can use its projection on D to redefine f:

$$f(\sigma \bullet [\delta]) = \vec{s} \otimes (\pi_D \circ (\hat{\mathbb{G}(d)}^{D'}([]) \boxdot \hat{g_\emptyset}^{D'}(\delta')))$$

f can be lifted to a memoryless function, explicitly by defining a lifted operator $\hat{\otimes}^{D'} : D' \times D' \to D$ such that for any $\vec{t}, \vec{t'} \in D'$, with $\vec{s} = \pi_D \circ \vec{t}$ and $\vec{s'} = \pi_D \circ \vec{t'}$

$$\pi_D \circ (\vec{t}\ \hat{\otimes}^{D'}\ \vec{t'}) = \vec{s} \otimes (\pi_D \circ (\hat{\mathbb{G}(d)}^{D'}([]) \boxdot \vec{t'}))$$
$$\pi_{D'} \circ (\vec{t}\ \hat{\otimes}^{D'}\ \vec{t'}) = \pi_{D'} \circ \vec{t'}$$

The lifted function $\hat{f}^{D'}$ is defined by:

$$\hat{f}^{D'}(\sigma \bullet [\delta]) = \hat{f}^{D'}(\sigma)\ \hat{\otimes}^{D'}\ \hat{g_\emptyset}^{D'}(\delta)$$

which matches the definition of a memoryless function. Remark that it is a valid lifting of f since $\pi_D \circ \hat{f}^{D'} = f$ by construction of the lifted operator. □

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 5.6 guarantees that any loop nest can be transformed into one that is summarizable. 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 $\odot \in O(m^{n-1})$.*

Proof. It is straightforward to see that for a rightward function $f : \mathcal{S}^n \to D$ with time complexity $O(m^n)$ defined through the recursive equation $f(\sigma \bullet [\delta]) = f(\sigma) \otimes \mathbb{G}(f(\sigma, d))(\delta)$ we have:

- Every $g \in \mathbb{G}$ is a (leftward or) rightward function of complexity $O(m^{n-1})$.
- Any $d \in D$ is strictly of space complexity $O(m^{n-1})$.[1]
- $\otimes$ is computable in time $O(m^{n-1})$.

Since if any of the upper bounds are violated, then one can show that the time complexity of f would surpass $O(m^n)$. Now, if h_L is a homomorphism and we want the parallel computation based on the homomorphism's *join* operator $\odot$ to have the same complexity as h_L. $\odot$ replaces $\otimes$, and we can conclude $\odot \in O(m^{n-1})$.

□

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 the computation that produces them. In Section 6.2, we provide 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.

6.2 Incompleteness

Consider a two-dimensional array of integers (with both

[1] This is under the assumption that the data is fully read. So, this excludes, for example, operations on lists performed through reference manipulation without reading the entire list content.

$$A = \left(\begin{array}{ccc} f([1:1]) & -L & -L \\ -L - f([1:1]) & L + f([2:2]) & 0 \\ 0 & -L - f([2:2]) & L + f([3:3]) \\ \hline L + f([1:2])/2 & L + f([1:2])/2 & -L - f([3:3]) \\ -L - f([1:2])/2 & f([2:3])/2 - f([1:2])/2 & L + f([2:3])/2 \\ \hline L + f([1:3])/3 & f([1:3])/3 - f([2:3])/2 & f([1:3])/3 - f([2:3])/2 \end{array}\right).$$

Figure 9. Definition of Matrix A

positive and negative) elements. Assume that the goal is to compute the maximum sum of the elements of a subarray $A[0..\ell, j..k]$ for all $0 \le \ell < n$ and $0 \le j \le k < n$, i.e. all subarrays that start from the top row of the original array, but can include a subset of its rows and columns. The code in Figure 10 is a clever single-pass implementation of this function. We will prove that this code, although very similar syntactically to the one in Figure 5, does not admit an efficient divide-and-conquer parallelization.

```
int col[];
int max_trec = 0;
for (i = 0; i < n; i++) {
  int max_rrec = 0;
  for (j = 0; j < m; j++) {
    col[j] += A[i][j];
    max_rrec = max(0, max_rrec + col[j]);
    max_trec = max(max_trec, max_rrec);
  }
}
```

Figure 10. Maximum top subarray sum.

Let $\mathbb{R}^+$ be the set of positive real numbers. For integers $i \le j$, let $[i:j]$ denote the set $\{i, \ldots, j\}$, and for each integer n and $r \le n$ define

$$\mathbb{I}_n^r := \{[i:j] : 1 \le i \le j \le n \mid j < i + r\}.$$

We write $\mathbb{I}_n$ for $\mathbb{I}_n^n$. Note that $|\mathbb{I}_n^r| = O(nr)$ and in particular $|\mathbb{I}_n| = O(n^2)$.

We call a function $f : \mathbb{I}_n^r \to \mathbb{R}^+$ *graded* if for any $J, J' \in \mathbb{I}_n^r$ with $|J| < |J'|$:

$$f(J)/|J| > f(J')/|J'|.$$

Lemma 6.4. *Given an arbitrary function $f : \mathbb{I}_n^r \to \mathbb{R}^+$ and $X \ge max_{J,J' \in \mathbb{I}_n^r} |f(J) - f(J')|$, the function $\tilde{f}$ defined as*

$$\tilde{f}(J) = |J|(n - |J|)X + |J|f(J), \tag{2}$$

is graded.

Proof. If $|J| < |J'|$:

$$\frac{\tilde{f}(J)}{|J|} - \frac{\tilde{f}(J')}{|J'|} = (|J'| - |J|)X + f(J) - f(J') \ge X + f(J) - f(J') > 0.$$

□

Given a matrix $A = [a_{ij}]$ with n columns, the *column-interval maximum prefix sum* of A is a real-valued function μ_A defined over $\mathbb{I}_n$, as

$$\mu_A(J) := max_{k \ge 1} \sum_{i=1}^{k} \sum_{j \in J} a_{ij},$$

and the *maximum top subarray sum* of A is

$$\mu(A) := max_{J \in \mathbb{I}_n} \mu_A(J).$$

Lemma 6.5. *For any graded function $f : \mathbb{I}_n^r \to \mathbb{R}^+$, there is an n-column matrix A with $O(nr)$ rows for which $\mu_A = f$.*

Proof. Choose $L > max_{J \in \mathbb{I}_n^r} f(J)$. We construct the matrix A by stacking r *groups* of rows, where the ith group is a set of $n - i + 1$ rows, each of which corresponding to one of the distinct sets $J \in \mathbb{I}_n^r$ with $|J| = i$, thus the total number of rows adding up to $\sum_{i=1}^{r}(n - i + 1) = O(nr)$.

Let k_J denote the row that corresponds to the set $J \in \mathbb{I}_n^r$. The entries in row k_J are determined as follows: for any column j, let $s_j = \sum_{i=1}^{k_J - 1} a_{ij}$. Then we set

$$a_{ij} = \begin{cases} f(J)/|J| - s_j & j \in J \\ -L - s_j & \text{otherwise.} \end{cases}$$

For example, for $n = 3$ and f defined on $\mathbb{I}_n$ the matrix A would be as illustrated in Figure 9.

The matrix for $\mathbb{I}_n^r$ would simply keep the first r groups of the one for $\mathbb{I}_n$.

We claim that for any $J \in \mathbb{I}_n^r$, $\mu_A(J) = f(J)$. For any row number k define for a column j,

$$s_j(k) := \sum_{i=1}^{k} a_{ij},$$

and for an $J \in \mathbb{I}_n$ define

$$s_J(k) := \sum_{j \in J} s_j(k).$$

It can be readily confirmed from our construction that for any $J \in \mathbb{I}_n^r$, and any column j:

$$s_j(k_J) = \begin{cases} f(J)/|J| & j \in J \\ -L & \text{otherwise.} \end{cases}$$

In particular, we have $s_J(k_J) = f(J)$, immediately implying that $\mu_A(J) \ge f(J)$. To prove $\mu_A(J) = f(J)$, we show next that for any $k \ne k_J$, $s_J(k) < f(J)$. By our construction, if $k = k_{J'} \ne k_J$, for some $J' \in \mathbb{I}_n^r$ then

$$s_J(k) = \sum_{j \in J' \cap J} f(J')/|J'| - \sum_{j \in J \setminus J'} L = |J' \cap J| f(J')/|J'| - L|J \setminus J'|. \tag{3}$$

If $k < k_J$, then

$$s_k(J) = |J' \cap J| f(J')/|J'| - L|J \setminus J'| \le f(J') - L|J \setminus J'| < 0,$$

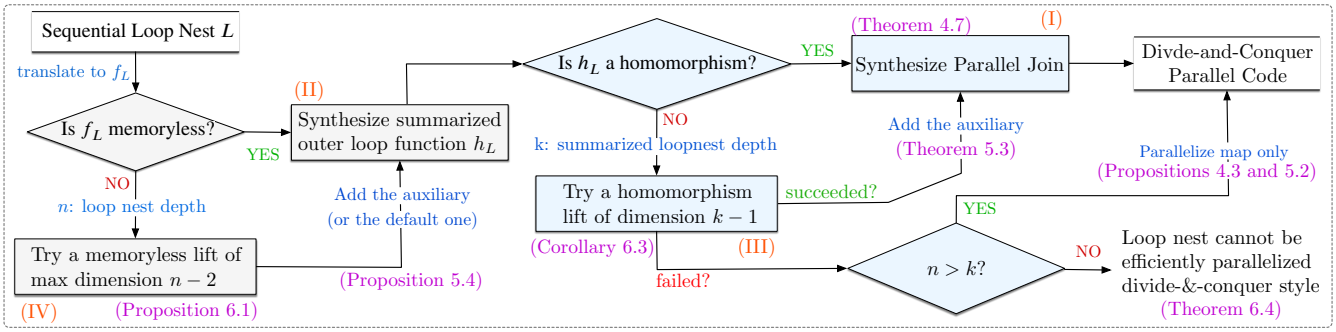


Figure 11. Parallelization Schema

where the last inequality follows because $J' \neq J$ and $f(J') < L$. In fact, the above inequality holds true by the same reasoning, even if $k > k_J$ but $J \setminus J' \neq \emptyset$. To complete the proof, suppose $k > k_J$ and $J \subset J'$, then

$$s_k(J) = |J| f(J')/|J'| < f(J),$$

where this final inequality follows from the fact that f is graded. □

The construction in the proof of the above Lemma can be regarded as an *encoding*: any function $f : \mathbb{I}_n^r \to \mathbb{R}^+$, can be encoded by a tuple (A, X), where A is the constructed $O(nr) \times n$ matrix and X is strict upper bound on the absolute value of the difference between values of f from the proof of the Lemma 6.4 needed for turning f into the graded $\tilde{f}$. The decoding for input $J \in \mathbb{I}_n^r$ is done by as:

$$f(J) = \mu_A(J)/|J| - (n - |J|)X.$$

Let us now restrict the range of values of f to the set of positive integers $\{1, \ldots, 2^k\}$ representable by $O(k)$ bits. Thus $X = 2^k + 1$ would be a valid upper bound to form values $\tilde{f}$ which would then be $O(k)$ bit integers themselves. Observe also that by construction $\tilde{f}(J)$ is always divisible by $|J|$. Following the arithmetics in Lemma 6.5, it can be verified that the entries of matrix A will all be $O(k)$-bit integers. We can thus state the following.

Lemma 6.6. *Let C be an algorithm that given an $m \times n$ matrix A, with $O(k)$-bit integer entries and with $m \leq n(n+1)/2$, produces a data structure $C(A)$ that can be used (independently of A) to evaluate $\mu_A(J)$ for every $J \in \mathbb{I}_n$, then $C(A)$ needs $\Omega(km)$ bits of space.*

Proof. Let $r = O(m/n)$. From the above discussion, we can encode any mapping from $\mathbb{I}_n^r$ to k-bit integers using an $m \times n$ matrix A of $O(k)$-bit integers (plus $O(k)$ bits to represent X from Lemma 6.4). Since there are $2^{O(knr)}$ such mappings, $C(A)$ must use at least the logarithm of that many bits to be able to distinguish different functions from each other, meaning it must have size $\Omega(knr) = \Omega(km)$. □

Theorem 6.7. *For any divide and conquer algorithm for computing $\mu(\cdot)$ of n-column matrices of $O(k)$-bit integers, the output of a sub-problem of $r \leq n(n+1)/2$ rows has size $\Omega(kr)$. In particular, solutions to subproblems of size $O(n^2)$ require $\Omega(kn^2)$ bits.*

Proof. Let A_0 and A_1 be two consecutive subproblems with A_1 consisting of r rows. Let A_{01} represent the concatenation of A_0 and A_1. We show that setting a single row of A_0 adversarially is enough to force the join of A_0 and A_1 to compute $\mu_J(A_1)$ for any $J \in \mathbb{I}_n^r$. Lemma 6.6 then implies that the output computed for A_1 must have size $\Omega(kr)$. Let $[i : j] \in \mathbb{I}_n^r$ and let $L > max_{J \in \mathbb{I}_n^r} \mu_{A_1}(J)$. We set the entries one row of A_0 to L for columns in $[i : j]$ and $-L$ for the remaining columns. All other rows of A_0 are set to zero. Since $\mu_{A_1}(J) \ll L$ for all J. $\mu(A)$ has to use as many L entries in our set row and no $-L$ entries to be maximized. Therefore, $\mu(A_{01}) = |J|L + \mu_{A_1}(J)$. □

Therefore, we can conclude with the following theorem:

Theorem 6.8. *An efficient lifting of a (multidimensional) rightward function may not always exist.*

6.3 Parallelization Schema

The diagram in Figure 11 illustrates the algorithmic steps in our methodology to parallelize an input sequential program. The light grey section performs the summarization of the loop, which corresponds to the discovery of a *map*. The light blue section 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 11), which are solved using syntax-guided synthesis (Section 7), and the memoryless lift and the homomorphism lift (boxes respectively labeled as (IV) and (III) in Figure 11) 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 practically through the success or failure of join synthesis algorithm. It is important to note that

Rice's theorem dictates that deciding 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*; i.e. the parallelization has failed.

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 11. 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 $\mathbb{S}$ of possible implementations for join, a syntax-guided synthesis solver finds a solution $x \in \mathbb{S}$ 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 $\mathbb{S}$, the more tractable the search problem for its synthesis.

We use an insight to define $\mathbb{S}$ 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 $D \to \mathcal{S}^n$. 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`,`row_mrec`, `mtl_rec`⟩ and its weak inverse is a 2-row array with the same width as the original input. It is illustrated below.

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.

`max_rec[0]`	⋯	`max_rec[`$k-1$`]`
`rec[0]`$-$`max_rec[0]`	⋯	`rec[`$k-1$`]`$-$`max_rec[`$k-1$`]`

In syntax-guided synthesis, $\mathbb{S}$ is defined by a *sketch* (a program with unknowns) and an *expression grammar* G specifying possible completions for the holes (unknowns). 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 [12]. It needs to be extended, since this paper introduces a technique to synthesize superscalar joins, and in [12] only constant-time computable joins are considered.

Sketch To obtain the sketch, a compilation function C (presented in Figure 12) is applied to the system of recurrence equations representing the body of the summarized loop. The result is a system of equations where the right-hand side of the equations become expressions with holes. We define C on expressions first, and then extend it to a systems of equations.

In Figure 12(a), we recall the compilation function C of [12]. It transforms expressions of the input loop body into expressions of the sketch by replacing variables with holes. Recall that the join takes as input the computation results of two threads: we will refer to them as the *left* thread and the *right* thread. In order to reduce the size of the state space of solutions, we identify two types of holes: (1) right holes $??_R$, which can be completed by expressions using only variables from the right thread and (2) left holes $??_{LR}$, which can completed using variables from both threads. The compilation function C is defined recursively on expressions e of the input language. *op* is an operator, x is a (possibly subscripted) variable, and c is a constant.

Since the join will use recursion (or, equivalently iteration with accumulation), we need to allow the use of recursion variables in the join. We extend the compilation function with C_{rec} and add a third type of hole: recursive hole $??_{Rec}$ which can completed using variables from both threads and local variables defined in the join. C_{rec} is defined in Figure 12(b): it coincides with C on constants and expressions, and replaces state variables with recursive holes $??_{Rec}$ instead of left-right holes $??_{LR}$.

Finally, the compilation function is defined over a system of recurrence equations E in Figure 12(c). $C(E)$ is the result of applying the compilation functions to the expressions on

$$
\begin{array}{rcl}
C(k) & = & ??_R \\
C(x) & = & \begin{cases} ??_R & \text{if } x \in \text{IVar} \\ ??_{LR} & \text{if } x \in \text{SVar} \end{cases} \\
C(op(e_1, .., e_m)) & = & op(C(e_1), \ldots, C(e_m))
\end{array}
$$

(a) Sketch compilation for scalar joins from [12].

$$
C_{rec}(x) = \begin{cases} ??_R & \text{if } x \in \text{IVar} \\ ??_{\text{Rec}} & \text{if } x \in \text{SVar} \end{cases}
$$

(b) Extension of the compilation function with C_{rec}.

$$
C(E) = \left(\begin{array}{c} s_1 = C(Exp_1(\text{IVar}, \text{SVar})) \\ \vdots \\ (s_{i_1}, s_{i_2}, \ldots, s_{i_p}) = \text{for } (i \in \mathcal{I}) \ \{ \ C_{rec}(E_{loop}) \ \} \\ \vdots \\ s_q = C_{rec}(Exp_q(\text{IVar}, \text{SVar})) \end{array} \right)
$$

(c) Compilation of the sketch body from the system of equations E.

Figure 12. Sketch compilation function C

the right-hand side of the equations in the loop body. C is applied to the right-hand side of simple equations appearing before all loop equations. C_{rec} is used for the body of a loop equation ($C_{rec}(E_{loop})$) and all the equations after. $C_{rec}(E)$ is defined similarly in a recursive manner, but only C_{rec} is applied to the expressions of the right hand side of each equation in E. Remark that for local variables to be used in the loop of the sketch, the local variables will first need to be initialized, and we will add equations $s_i = ??_{LR}$ for any variable s_i that can be used in a loop and that has not been initialized before that loop. To include the solutions described in Example 7.1, we allow bounded repetitions of the sketch. To produce the exact join of this example, four would have been necessary. But, practically, in the vast majority of the cases one repetition of the sketch is sufficient. Additionally, we extend the state space of solutions represented by the sketch to include potential summarized solutions: any equation sketch $s_i = C_{rec}(Exp)$ appearing after a loop is copied in the body of the preceding loop. The construction still ensures that the solution based on the weak inverse is in the space of possible solutions.

For example, consider the sketch illustrated on the right, written in the syntax of the input language for simplicity. The crucial difference between a looped sketch like this one and those in [12] 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). A 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. In the implementation we first search for the simplest join by initially allowing only one repetition.

```
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. The synthesis procedure can *soundly* declare h_L not to be a a homomorphism when it cannot find a join.

Expression grammar The grammar of expressions used to complete the holes $??_R$, $??_{LR}$ and $??_{Rec}$ during the syntax-guided synthesis of the join operator is presented in Figure 13. This grammar is parameterized by (1) the operators that can be used in the expression and (2) the set of expressions (*nVars* and *bVars*) that can be used in the leaves of the expression tree and (3) the maximal expression height allowed κ. Sets of operators of different types ($\oplus$, $\ominus$ and $\oslash$) are given in the figure and can be extended if the input program uses additional operators. In practice, the set of available operators is gradually increased until a solution is found, starting with the set of operators that appear in the input program. The set of variables available in the expression depends on the hole type ($??_R$, $??_{LR}$ or $??_{Rec}$), as discussed previously. Finally, the parameter κ is gradually increased until a solution is found; in practice, we observed that $\kappa \leq 2$.

For example, in the sketch presented above, most holes only need to be replaced by a single variable and only one hole needs to be replaced by an expression of height one (`rec_l[j] + max_rec_r[j]`) to get the solution presented in Figure 6.

7.2 Summarized Loop Synthesis

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

Proposition 7.2. *A multidimensional rightwards function is memoryless iff we have $\circledcirc$ and $\emptyset_g$ that satisfy the specification* $\varphi(\circledcirc, \emptyset_g) \overset{def}{\equiv} \forall d \in D, \delta \in S^{n-1}, \ \mathbb{G}(d)(\delta) = d \circledcirc \mathbb{G}(\emptyset_g)(\delta)$.

It is straightforward to see why the above characterization is equivalent to the one given in Definition 4.2. One can also show that φ is identical to the definition of a homomorphism.

Proposition 7.3. *φ holds for a family of functions $\mathbb{G}$ iff every member of the family is $\circledcirc$-homomorphic.*

$$
\begin{array}{lll}
ne_0 & ::= x \mid x[ej] \mid c\ x \in nVars,\ c \text{ a numeric constant} & \\
ne_{\kappa>0} & ::= ne_{\kappa-1} \oplus ne_{\kappa-1} \mid -ne_{\kappa-1} & \\
 & \mid \texttt{if}\ (be_{\kappa-1})\ ne_{\kappa-1}\ \texttt{else}\ ne_{\kappa-1} & \\
be_0 & ::= b \mid b[ej] \mid\mid true \mid false & b \in bVars \\
be_{\kappa>0} & ::= be_{\kappa-1} \owedge be_{\kappa-1} \mid \neg be_{\kappa-1} & \\
 & \mid ne_{\kappa-1} \ocomparison ne_{\kappa-1} & \\
 & \mid \texttt{if}\ (be_{\kappa-1})\ be_{\kappa-1}\ \texttt{else}\ be_{\kappa-1} & \\
ej & := j \oplus c \quad j \text{ an iterator, } c \text{ integer constant} & \\
\hline
\oplus & := +, -, min, max, \times, \div & \text{binary numeric} \\
\ocomparison & := >, >=, <, <=, = & \text{comparisons} \\
\owedge & := \wedge, \vee & \text{binary boolean}
\end{array}
$$

Figure 13. Grammar of expressions used for $??_R$, $??_{LR}$ and $??_{Rec}$ holes. ne_κ and be_κ correspond to expressions of depth up to and equal to κ. *nVars* and *bVars* stand for numeric and booleans variables.

Proof. Consider a family of rightwards (or leftwards) functions $\mathbb{G} : D \to (\mathcal{S}^{n-1} \to D)$ defined by $\forall d \in D, \delta \in \mathcal{S}^{n-1}$, $\mathbb{G}(d)(\delta) = foldl(\oplus)\ \delta\ d$ with some operator $\oplus : D \times \mathcal{S}^{m-2} \to D$.

Let us first define what it means for a function in $\mathbb{G}$ to be homomorphic, since their signature is slightly different from the functions in Definition 3.2. Then, we remark in Proposition 7.5 that for every g in $\mathbb{G}$ to be $\circledcirc$-homomorphic, that is for the *family of functions* to be homomorphic (defined below in Definition 7.4), we only need to prove that the function $\mathbb{G}(\emptyset_g) : \mathcal{S}^{n-1} \to D$ is $\circledcirc$-homomorphic, for some $\emptyset_g$ in D. This leads us to our conclusion, equating φ to the specification of a family of homomorphisms.

Definition 7.4. A family of rightwards (or leftwards) functions $\mathbb{G} : D \to (\mathcal{S}^{n-1} \to D)$ is a family of $\circledcirc$-homomorphisms for binary operator $\circledcirc : D \times D \to D$ with identity element $\emptyset \in D$ iff for all sequences $\delta, \delta' \in \mathcal{S}^{n-1}$ and $\forall d \in D$ we have $\mathbb{G}(d)(\delta \bullet \delta') = \mathbb{G}(d)(\delta) \circledcirc \mathbb{G}(\emptyset)(\delta')$.

Remark the asymmetry in the definition: the right hand operand of the $\circledcirc$ operator is independent from d. This is necessary, as illustrated by the following example. Take the family of sum functions initialized with an arbitrary integer: we have $D = int$ and $\mathbb{G}(d)(\delta) = d + sum(\delta)$ where *sum* returns the sum of all the elements of δ. Then, for every integer d:

$$\mathbb{G}(d)(\delta \bullet \delta') = d + sum(\delta) + 0 + sum(\delta') = \mathbb{G}(d)(\delta) + \mathbb{G}(0)(\delta').$$

Using d on both side of + would have yielded the wrong answer. The asymmetry allows for every member of the family to have the same homomorphic join operator, in this case +.

To prove that a family of rightwards functions is homomorphic, there is no need to prove that for every d, the function $\mathbb{G}(d)$ is homomorphic. It suffices to prove it for $\mathbb{G}(\emptyset)$, as the following proposition states.

Proposition 7.5. *A family of rightwards (or leftwards) functions $\mathbb{G}$ is a family of homomorphisms iff there is an element $\emptyset \in D$ such that $\mathbb{G}(\emptyset)$ is $\circledcirc$-homomorphic for some $\circledcirc : D \times D \to D$.*

Remark that if $\mathbb{G}$ is a family of homomorphisms, then in particular $\mathbb{G}(\emptyset)$ is a homomorphism.

Now, assume that we have an element $\emptyset$ and operator $\circledcirc$ such that $\mathbb{G}(\emptyset)$ is $\circledcirc$-homomorphic. We are only interested in computable functions that have a countable domain, and therefore have weak inverses. We denote the weak inverse of $\mathbb{G}(\emptyset)$ by $\mathbb{G}(\emptyset)^{-1}$: we have $\forall d \in D$, $\mathbb{G}(\emptyset)(\mathbb{G}(\emptyset)^{-1}(d) = d$. Let $\delta = [\delta_0, \ldots, \delta_n]$ a sequence of length n, we can develop the function application as follows (where $\gamma = \mathbb{G}(\emptyset)^{-1}(d) = [\gamma_0, \ldots, \gamma_{n'}]$):

$$
\begin{aligned}
\mathbb{G}(d)(\delta) &= (\ldots(d \oplus \delta_0) \oplus \ldots \oplus \delta_n) \\
&= (\ldots(\mathbb{G}(\emptyset)\left(\mathbb{G}(\emptyset)^{-1}(d)\right) \oplus \delta_0) \oplus \ldots \oplus \delta_n) \\
&= (\ldots((\ldots(\emptyset \oplus \gamma_0)\ldots \oplus \gamma_{n'}) \oplus \delta_0) \oplus \ldots \oplus \delta_n) \\
&= \mathbb{G}(\emptyset)(\gamma \bullet \delta) = \mathbb{G}(\emptyset)(\mathbb{G}(\emptyset)^{-1}(d) \bullet \delta)
\end{aligned}
$$

Let δ and δ' two sequences. We use the previous result, and the fact that $\mathbb{G}(\emptyset)$ is homomorphic:

$$
\begin{aligned}
\mathbb{G}(d)(\delta \bullet \delta') &= \mathbb{G}(\emptyset)(\mathbb{G}(\emptyset)^{-1}(d) \bullet \delta \bullet \delta') \\
&= \mathbb{G}(\emptyset)(\mathbb{G}(\emptyset)^{-1}(d) \bullet \delta) \circledcirc \mathbb{G}(\emptyset)(\delta') \\
&= \mathbb{G}(d)(\delta) \circledcirc \mathbb{G}(\emptyset)(\delta')
\end{aligned}
$$

Therefore, $\mathbb{G}$ is a family of homomorphisms.

Proposition 7.5 justifies the Definition 7.4 by proving that the latter matches exactly the definition of a homomorphism in Definition 3.2.

Let us come back to the original problem. Recall the correctness specification used in the summarized loop synthesis:

$$\varphi(\circledcirc, \emptyset) = \forall d \in D, \forall \delta \in \mathcal{S}^{n-1}, \mathbb{G}(d)(\delta) = d \circledcirc \mathbb{G}(\emptyset)(\delta)$$

We want to prove that φ holds for the family of function $\mathbb{G}$ iff every g in $\mathbb{G}$ is a homomorphism.

If $\mathbb{G}$ is a family of homomorphisms, then φ is satisfied: it is the homomorphism definition with $\delta = []$ and $\delta = \delta'$.

If φ is satisfied, we have $\circledcirc$ and $\emptyset$ such that $\forall d \in D, \forall \delta \in \mathcal{S}^{n-1}$:

$$\mathbb{G}(\emptyset)(\mathbb{G}(\emptyset)^{-1}(d) \bullet \delta) = \mathbb{G}(\emptyset)(\mathbb{G}(\emptyset)^{-1}(d)) \circledcirc \mathbb{G}(\emptyset)(\delta)$$

which shows that $\mathbb{G}(\emptyset)$ is $\circledcirc$-homomorphic, and, by Proposition 7.5, every g in $\mathbb{G}$ is $\circledcirc$-homomorphic. □

Therefore, the operator $\circledcirc$ can be synthesized as a homomorphism join operator of the functions in family $\mathbb{G}$, which is the problem that we have already addressed in the previous subsection. The only point of difference is that the complexity budget set for a memoryless join operator $\circledcirc$ and the parallel join operator $\odot$ (previously discussed) are different. The budget for $\odot$ is determined by the depth of the summarized loop, whereas the budget for $\circledcirc$ is determined

by the original loop nest's depth, as indicated in Figure 11 respectively by k and n.

There are two modifications to the sketch compilation function of the join synthesis. First, instead of replacing state variables $x \in \text{SVar}$ by holes, we simply put the variable x_L from the left thread, since the right operand of $\circledcirc$ is the result of applying g to only one element of $\mathcal{S}^{n-1}$ and looking for a join that iterates on its inputs more than once makes no sense. Then, the sketch compiled from the body of g is wrapped in a loop iterating over the size of an element of D instead of a constant. That is, if D contains scalar elements, the sketch is constant-time.

If the loop is memoryless, this synthesis procedure always succeeds, even if it has to fall back on returning the trivial answer (see Proposition 5.6). But, as discussed in section 5, the goal is to find the simplest join. This is achieved by two mechanisms. First, the sketch complexity is at most the complexity of the data. For example, a linear join for scalar variables will only be necessary if a linear variable has been added through lifting. Second, the expressions completing the holes are the simplest possible, because the search for a solution increases the depth κ until a solution is found.

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 11.

8.1 Rewriting Oracle

Assume a memoryless function $f : \mathcal{S}^n \to D$ defined recursively as $f(\sigma) = foldl(\oplus) \circ map(g)(\sigma)$ 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:

$$h(x \bullet [a_1, \ldots, a_k]) = (\cdots(\vec{s} \oplus a_1) \oplus \cdots) \oplus a_k. \tag{4}$$

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 $(\vec{s}, \vec{s'}) = \hat{f}^{D'}(\sigma)$):

$$\begin{aligned}\hat{h}^{D'}(x \bullet y) &= (\vec{s}, \vec{s'}) \odot \hat{h}^{D'}([a_1, \ldots, a_k]) \\ &= (\vec{s}, \vec{s'}) \odot \left(\cdots(((\vec{s_0}, \vec{s'_0}) \hat{\oplus} a_1) \hat{\oplus} \cdots) \hat{\oplus} a_k\right).\end{aligned} \tag{5}$$

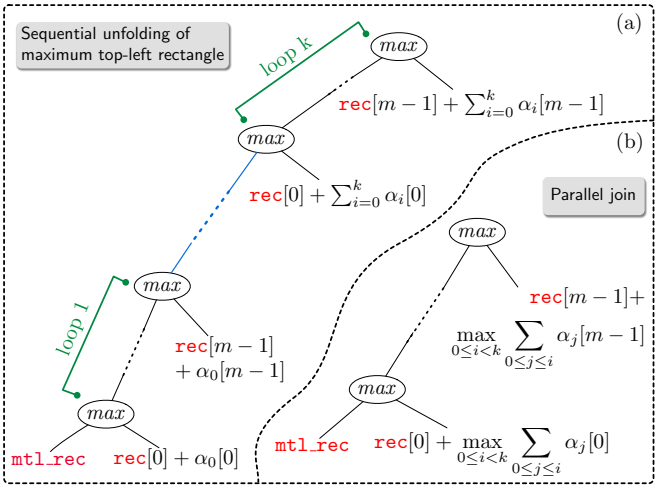


Figure 14. Sequential (a) vs parallel (b) computations.

First, the following proposition, adapted from Theorem 6.2 of [12], 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 5, 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:

$$\begin{aligned}&(\cdots((\vec{s} \oplus a_1) \oplus \cdots) \oplus a_k) = \\ &\qquad \theta(\vec{s},\ h([a_1, \ldots, a_k]),\ \pi_{D'} \circ \hat{h}^{D'}([a_1, \ldots, a_k]))\end{aligned} \tag{6}$$

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 [12] where it stood for an expression (i.e. loop-free code), which means the lifting algorithm in [12] is not applicable and a new lifting algorithm is required. Equation 6 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.

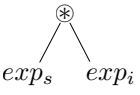
Example 8.2. Figure 14 illustrates the two sides of Equation 6 for the maximum top-left rectangle example of Section 2.2. Variables in red indicate the values of state variables from $\vec{s}$. Each α technically should include a field for `rec[]` and a field for `mtl_rec` of the saved states after summarization. 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 $max_{0\leq i<k} \sum_{0\leq j\leq i} \alpha_j[l]$ are available (i.e. computed before the join). These are exactly the auxiliary computation 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 6 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 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 $\oslash$*-recursive normal form* if it is defined recursively using an operator $\oslash$ as $e = e_c \mid e_c \oslash e$, where e_c is in constant normal form (the base case) and $\oslash$ satisfies the same condition as $\circledast$.

For example, in Figure 14(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. $\oslash$).

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 [29, 32] that can inspire several heuristics for normalization. We employ a cost-based search for the normal form as a heuristic to workaround the undecidability. Our algorithm uses a set of standard algebraic equalities as rewrite rules $\mathcal{R}$ and searches for the *shortest* normal form. The rewrite 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 [12], which is no coincidence since [12] focuses solely on constant normal forms.

In the second phase, the algorithm makes a guess about $\oslash$, inspired by expression e which is the result of the first phase, and attempts to rewrite e to a $\oslash$-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 $\oslash$. The algorithm chooses the simplest $\oslash$ such that $e = e_c \oslash e'$, where e_c is in constant normal form and e' is an arbitrary expression. For a fixed $\oslash$, a cost function is defined. It combines the sum of the sizes of expression not in constant normal form and the count of expressions in constant normal form. Formally:

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

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

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

Intuitively, the expression is in $\oslash$-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 $c_\oslash$ when *size* cannot be decreased and $size > 0$.

In the example of Figure 14, 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 [12]) 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 14(b). This expression is in *max*-recursive normal form with cost $(0, m + 1)$.

If the process fails to find a normal form for $\oslash$, then another operator $\oslash$ 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 10.

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 [26]. 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 [39]). 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 list of expressions $max_{0 \le i < k} \sum_{0 \le j \le i} \alpha_j[l]$ (for $0 \le l < m$) from Figure 14(b) can be computed in an auxiliary array `max_rec[]` using a *zip* operation and the state variable `rec[]` as follows:

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

The next section illustrates how this auxiliary can be found.

9 An extended example

In this section, we go through steps of the automated parallelization process for the maximum top-left subarray sum example of Section 2.2. The goal is to give a good intuition of how the heuristic algorithms described in this paper work on an example but not to describe them precisely.

The code is given in Figure 5(a). We call the corresponding function *mtls*, which is of type $int[][] \to int[] \times int \times int$ (before lifting) and the state is a triple of variables SVar = {`rec[]`, `row_sum`, `mtl_rec`}.

9.1 Summarizing the loop nest

The first step in the parallelization process is to synthesize the summarized loop ((II) in Figure 11). As stated in Section 7, the problem is similar to synthesizing a parallel join with a few modifications. We explain here how the sketch is generated and the solution found by the syntax-guided synthesis solver.

From the code in Figure 5(a) we synthesize the sketch that corresponds to finding a parallel join for the inner loop, with the complexity budget that is enforced by the budget for the summarized function. Remark that for this reason, the sketch resulting from the application of the compilation function of Section 7 can be repeated more than a constant number of times. Since there is a linear variable (`rec[]`), we will need a loop on the dimension of this linear variable. The solution will require a loop of the size of `rec[]`.

Figure 15 presents the sketch for the memoryless join $\circledcirc$ and a solution (each hole has been completed). Given the generic correctness specification $\varphi(\circledcirc, \emptyset_g) \equiv \forall d \in D, a \in int[], \mathbb{G}(d)(a) = a \circledcirc \mathbb{G}(\emptyset_g)(a)$, the sketch and the generic grammar of Figure 13, the solver finds the solution presented.

```
int mtl_rec = ??_LR;
int rec[] = ??_LR;
int row_sum = ??_LR;
for(j=0; j < m; j++){
  row_sum = ??_Rec + ??_R;
  rec[j] = ??_Rec + ??_Rec;
  mtl_rec = max(??_Rec, ??_Rec);
}
```

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

Figure 15. Sketch and solution for the memoryless join of *mtls* (Sec. 2.2)

Variables ending by `_l` are variables from the left input of the join and the variables ending by `_r` are the right input of the join. The identity state $\emptyset_g$ is $\{[0], 0, 0\}$.

Then, we obtain the summarized loop of Figure 5(b) by removing `row_sum` which is not necessary for the computation of `mtl_rec` anymore, and inserting the loop implementing $d \circledcirc$ `inner_loop[i]` where `inner_loop[]` is the conceptual array representing the results of mapping the inner loop instances to the input. Remark that in the general case, the elements of `inner_loop[]` are of type D, but in this case, the only element of D that is required is the variable `rec[]`.

In the case where the variables are linear and every cell is modified in the inner loop, we cannot reduce the complexity of the loop nest. However, the summarized loop abstracts all the unnecessary information that the inner loop has computed. This will become more apparent in the next section, where the unfoldings of the function on symbolic inputs need to be inspected. In this example, the elements of `rec` that were column sums of prefix sums are summarized as simple column sums. In the symbolic execution, we reduce the amount of information in the cells from quadratic to linear.

9.2 Lifting

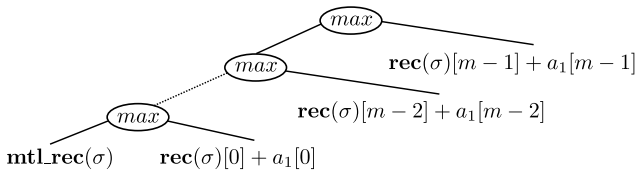
The first attempt at synthesizing a join will fail for this example since the summarized loop cannot be parallelized. We need to lift the summarized function to a homomorphism ((III) in Figure 11), if possible. The codomain of the summarized function h_{mtls} is $D = int[] \times int$ before lifting. We drop `row_sum` from the state, since it does not appear in the summarized loop of Figure 5(b).

Normalization First, we compute the unfolding of h_{mtls} over an input $\sigma \bullet [a_1, \ldots, a_k]$. Each a_i for $0 \leq i \leq$ is itself an integer array of size m. We denote the different projections of $h_{mtls}(\sigma)$ over the different parts of the codomain by $rec(\sigma)$ and $mtl\_rec(\sigma)$.

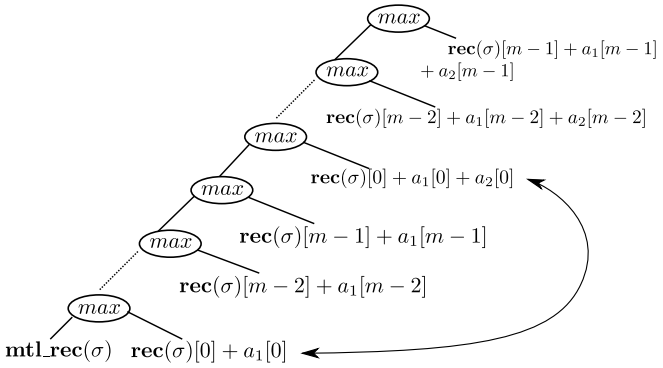
Let us start with $k = 1$:

$$\begin{aligned} mtl\_rec(\sigma \bullet [a_1]) \quad = \quad & max(a_1[m-1] + rec(\sigma)[m-1], \\ & max(a_1[m-2] + rec(\sigma)[m-2], \\ & \ldots, \\ & max(a_1[0] + rec(\sigma)[0], \\ & mtl\_rec(\sigma)) \ldots))) \end{aligned}$$

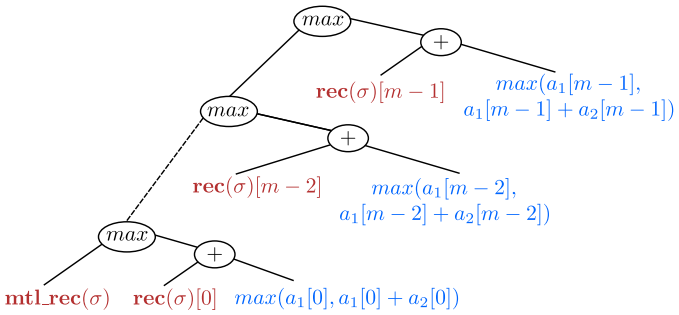
We can visualize the expression as a tree:



Remark that this expression is in *max*-recursive normal form, and computable with a loop of size m and it is already in the shortest (*max*)-recursive normal form. This is not surprising, since we only unfolded the function on one line of input.



(a) Expression tree after unfolding



(b) Normalized expression tree

Figure 16. Expression trees of $mtl\_rec(\sigma \bullet [a_1, a_2])$

The second unfolding will be computed with a loop of size $2m$: we need a loop of size m to compute $mtl\_rec(\sigma \bullet [a_1])$ and m additional iterations to compute the rest of the expression.

$$\begin{aligned} mtl\_rec(\sigma \bullet [a_1, a_2]) \quad = \quad & \\ & max(a_1[m-1] + a_2[m-1] + rec(\sigma)[m-1], \\ & max(a_1[m-2] + a_2[m-1] + rec(\sigma)[m-2], \\ & \ldots, \\ & max(a_1[0] + a_2[0] + rec(\sigma)[0], \\ & mtl\_rec(\sigma \bullet [a_1])) \ldots))) \end{aligned}$$

Remark that the expression is already in *max*-normal form, since each subexpression (of the form $a_1[j]+a_2[j]+rec(\sigma)[j]$) or $a_1[j] + rec(\sigma)[j]$)) is in constant normal form. However, this normal form does not represent a computationally efficient parallel join, since for an arbitrary k we would need km iterations, which is not acceptable under Proposition 6.1 (the join would be $O(m^n)$ for n lines of input). However, we can normalize the expression to a shorter normal form. We write the expression tree in Figure 16(a).

Some terms in this expression tree can be factored together using the associativity of *max* to reorganize the tree, and the factorization rule $max(a + b, a + c) \rightarrow a + max(b, c)$. For example, the two expressions linked by the double arrow

can be factored together: $max(\mathbf{rec}(\sigma)[0] + a_1[0], \mathbf{rec}(\sigma)[0] + a_1[0] + a_2[0]) = \mathbf{rec}[0] + max(a_1[0], a_1[0] + a_2[0])$. This rule decreases the number of occurrences of state variables (the $\mathbf{rec}[j]$) and is applied during the first phase of the normalization process. We can apply the same rule for all m pairs of subexpressions and this yields the expression in Figure 16(b).

No rewrite rule can be applied to this expression in the first phase, but the above is not in constant normal form. When starting the second phase, the max operator is picked and the matching expression cost is $(0, m + 1)$. The max-recursive normal form uses $m + 1$ max operators, and is computable using a loop of size m and one additional statement. Each of the subexpressions under the tree spine of max operators is in constant normal form: it is either only a state variable ($mtl\_rec(\sigma)$) or an expression that has the root operator + and on one side, the state variable $mtl\_rec(\sigma)[j]$ and on the other side the input-only dependent expression $max(a_1[j], a_1[j] + a_2[j])$, for $0 \leq j < m$. The second phase concludes immediately, and returns the expressions in max-recursive normal form.

Remark that this generalizes to any unfolding k (as seen in Section 8). We will have a max-recursive normal form with a spine of length $m + 1$, and the constant normal forms $e_c[j]$ that have input and state variables will be:

$$e_c[j] = mtl\_rec(\sigma)[j] + max(a_1[j], a_1[j] + a_2[j], \ldots, \sum_{l=1}^{k} a_l[j])$$

Recursion discovery In the next step, we want to extract the information that needs to be computed in the threads for the join from the expression in normal form. We denote the expressions that consitute this information by $u_k = \pi_{D'} \circ \hat{h}_{mtls}^{D'}([a_1, \ldots, a_k])$. In the expression tree, it corresponds to the blue input dependent expression in each of the constant normal forms. For any unfolding $k > 0$ we have:

$$u_k = \left\{ max(a_1[j], a_1[j] + a_2[j], \ldots, \sum_{i=1}^{k} a_i[j]) \mid 0 \leq j < m \right\}$$

We have collected u_k and now need to discover a recursive function that can compute them. The codomain of the function needs to be extended. Since in this case u_k is an integer collection of size m, we extend the codomain by an array of integers $D' = int[]$. We have to discover an operator $\boxplus : D' \times int[] \to D'$ such that $u_{k+1} = u_k \boxplus a_{k+1}$. To do so, we look for subtree isomorphisms (or, subexpression relations) between the elements of u_k and the elements of u_{k+1}.

For any $j \in [0..m]$ the following equality defines a family of subtree isomorphisms for each pair $u_{k+1}[j], u_k[j]$:

$$u_{k+1}[j] = max(u_k[j], \sum_{i=1}^{k+1} a_k[j])$$

This is not sufficient yet, since the sum term needs to be computed in non-constant time. But this sum is already part of the computation we are lifting, we have:

$$rec([a_1, \ldots, a_k, a_{k+1}])[j] = \sum_{i=1}^{k+1} a_k[j]$$

The operator of the lifting $\boxplus$ can in use parts of the existing function. In this case, the max_rec auxiliary is created, defined by:

$$max\_rec = \mathbf{zip}(max)\ rec\ max\_rec$$

Which concludes our description of the lifting. A description of the last step in the parallelization process, the join synthesis ((I) in Figure 11), has already been included as part of the example developed in Section 7.

10 Experimental Evaluation

Implementation Our methodology is implemented as an addition to our existing too ParSynt from [12]. We employ a new incremental approach to synthesizing the join to mitigate the large search problem. The state of the loop is partitioned into substates, and the join is synthesized for each substate separately, while preserving correctness. 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.

The main idea behind this optimization is that if a function $h : \mathcal{S}^n \to D$ is homomorphic, then any projection of h over a domain $D' \subset D$ will be homomorphic, if the projection is well defined. The projection on D' is well defined if there is no variable in D' that depends on a variable in $D \setminus D'$. We first define a sequence of sets $D_1 \subset D_2 \subset \cdots \subset D$ such that the projection of h over the D_i are well defined.
Then, we start by solving the synthesis problem for h projected on D_1. We continue to $\pi_{D_2} \circ h$. But at this point, we can use the solution of the previous problem to solve part of the current problem. Intuitively, only the part concerning the variables in $D_2 \setminus D_1$ needs to be dealt with. We go on incrementally with the next projections, and finally with h.

10.1 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. And, (2) the efficiency of SyGuS-based solution for parallel join generation and loop summarization; since it

	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: desktop 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.

is impossible to predict synthesis times for the search-based routine.

Benchmarks. We use a diverse set of benchmarks, where some implement highly non-trivial single-pass algorithms. 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.

The code of the benchmarks is available at https://github.com/victornicolet/parsynt-pldi19-benchmarks.

Performance of ParSynt . Table 1 presents the times spent in the two steps summarized loop and parallel join generation. The synthesis time have been measured on a laptop with an dual core Intel Core m3-6Y30 CPU and 6 Gb RAM running 64-bit Ubuntu 16.04. Our goal is to provide an implementation that can synthesize the parallel implementation of a sequential algorithm in reasonable time, and much faster than a programmer. It is not aimed at being integrated in a compiler, but rather in a programmer-aid tool.

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 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.

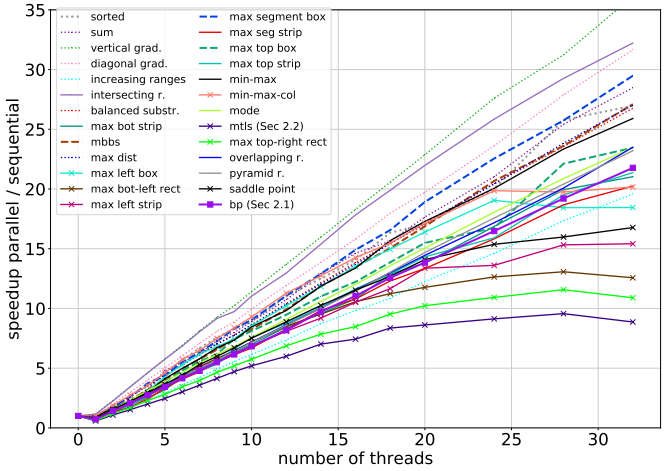


Figure 17. Speedups relative to the sequential implementation. *Hardware:* 8 eight-core Intel X6550 processors (64 cores total) and 256G of RAM running 64-bit Ubuntu

Quality of the Synthesized Code. A divide-and-conquer parallelization, in the style of this paper, is 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) [35] 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

Figure 17 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. We used `gcc 7.3.0` to compile the parallelized benchmarks. On average, the speedups are close to linear on the number of cores up to around 32 cores. Speedup is measured by dividing the running time of the parallel implementation with the

running time of the sequential implementation *without auxiliaries*. Therefore, the speedup reported for 0 is always 1, and the speedup for 1 core is the parallel implementation *with auxiliaries* running on a single thread (which explains the inflexion of the curve at the beginning of the graph). 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 [9] 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 [12] to produce correctness proofs verified through Dafny [28]. The majority of the programs were verified using the same proof generation scheme as [12]. 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, $\vec{x} + max(\vec{y}, \vec{z}) = max(\vec{x} + \vec{y}, \vec{x} + \vec{z})$.

11 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) [41], 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 [13].

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 not discovery 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 6.8 indicates.

Future work. The focus of this paper (and its predecessor [12]) has been on synthesizing homomorphisms which have fixed simple 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, this adds one more unknown dimension to the problem which can make it substantially more difficult/interesting to solve.

12 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, 31], and builds up on our recent work [12], 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 [12] 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, 31], function composition [15], and quantifier elimination [30], 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 [38], 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 [12], 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 [37] or synthesis [40]. 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 [12].

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 [40] 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, 34] or more specialized nested loops with polyhedral optimization [3, 8, 42, 43]. There is a body of work specific to reductions and parallel-prefix computations [5, 22, 27] 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 [36] 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 complimentary 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 (FMCAD), 2013*. 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] Cédric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think *(29 September - 3 October 2004, Antibes Juan-les-Pins, France)*. IEEE Computer Society, 7–16. https://hal.archives-ouvertes.fr/hal-00017260

[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. 1990. Prefix sums and their applications. (1990).

[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] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, Vol. 43. ACM, 101–113.

[9] Gilberto Contreras and Margaret Martonosi. 2008. Characterizing and improving the performance of Intel Threading Building Blocks. In *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*. 57–66.

[10] 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 *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE, 136–146.

[11] 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.

[12] Azadeh Farzan and Victor Nicolet. 2017. Synthesis of divide and conquer parallelism for loops. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 540–555.

[13] Azadeh Farzan and Victor Nicolet. 2019. Modular Synthesis of Divide-and-Conquer Parallelism for Nested Loops (Extended Version). arXiv:cs.PL/submit/2613711

[14] Grigory Fedyukovich, Ahmad Maaz Bin Safeer, and Rastislav Bodık. 2017. Gradual Synthesis for Static Parallelization of Single-Pass Array-Processing Programs. In *PLDI*.

[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] Arun Kejariwal, Paolo D'Alberto, Alexandru Nicolau, and Constantine D. Polychronopoulos. 2005. A Geometric Approach for Partitioning N-dimensional Non-rectangular Iteration Spaces. In *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing (LCPC'04)*. 102–116.

[26] 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.

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

[28] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 348–370.

[29] Claude Marché and Xavier Urbain. 1998. *Termination of associative-commutative rewriting by dependency pairs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 241–255.

[30] 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.

[31] 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.

[32] Paliath Narendran and Michaël Rusinowitch. 1991. *Any ground associative-commutative theory has a finite canonical system*. Springer Berlin Heidelberg, Berlin, Heidelberg, 423–434.

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

[34] David A. Padua and Michael J. Wolfe. 1986. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM* 29, 12 (Dec. 1986), 1184–1201.

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

[36] 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. *SIGPLAN Not.* 46, 6 (June 2011), 12–25.

[37] 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.

[38] 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.

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

[40] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. *SIGPLAN Not.* 51, 6 (June 2016), 326–340.

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

[42] 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.

[43] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 54.

[44] Lenore D. Zuck, Amir Pnueli, Yi Fang, Benjamin Goldberg, and Ying Hu. 2002. Translation and Run-Time Validation of Optimized Code. *Electr. Notes Theor. Comput. Sci.* 70, 4 (2002), 179–200.