Synthesis of Divide and Conquer Parallelism for Loops [*]

Azadeh Farzan and Victor Nicolet [†]

University of Toronto

Abstract

Divide-and-conquer is a common parallel programming skeleton supported by many cross-platform multithreaded libraries, and most commonly used by programmers for parallelization. The challenges of producing (manually or automatically) a correct divide-and-conquer parallel program from a given sequential code are two-fold: (1) assuming that a good solution exists where individual worker threads execute a code identical to the sequential one, the programmer has to provide the extra code for dividing the tasks and combining the partial results (i.e. joins), and (2) the sequential code may not be suitable for divide-and-conquer parallelization as is, and may need to be modified to become a part of a good solution. We address both challenges in this paper. We present an automated synthesis technique to synthesize correct joins and an algorithm for modifying the sequential code to make it suitable for parallelization when modifications are necessary. We focus on a class of loops that traverse a read-only collection and compute a scalar function over that collection. We present theoretical results for when the necessary *modifications* to sequential code are possible, theoretical guarantees for the algorithmic solutions presented here, and experimental evaluation of the approach's success in practice and the quality of the produced parallel programs.

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

Keywords Divide and Conquer Parallelism, Program Synthesis, Homomorphisms

1. Introduction

Despite big advances in optimizing and parallelizing compilers, *correct and efficient* parallel code is often hand-crafted in a difficult and error-prone process. The introduction of libraries like Intel's TBB [39] was motivated by this difficulty and with the aim of making the task of writing well-performing parallel programs easier for an average programmer. These libraries offer efficient implementations for commonly used *parallel skeletons*, which makes it easier for the programmer to write code in the style of a given skeleton without having to make special considerations for important performance factors like scalability of memory allocation or task scheduling. Divide-and-conquer parallelism is the most commonly used of these skeletons.

```
sum = 0;
for (i = 0; i < |s|; i++) {
   sum = sum + s[i];
}
```

Consider the function sum that returns the sum of the elements of an array of integers. The code on the right is a sequential loop that computes this function. To compute the sum, in the style of divide and conquer parallelism, the computation should be structured as illustrated in Figure 1. The array s of length $|s|$ is partitioned into $n+1$ chunks, and sum is individually computed for each chunk. The results of these partial sum computations are *joined* (operator ⊙) into results for the combined chunks at each intermediate step, with the join at the root of the tree returning the result of sum for the entire array. The burden of a correct design is to come up with the correct implementation of the join operator. In this example, it is easy to quickly observe that the join has to simply return the sum of the two partial results. In general, it could be tricky to reformulate an arbitrary sequential computation in this style. Recent advances in program synthesis [2] demonstrate the power of synthesis

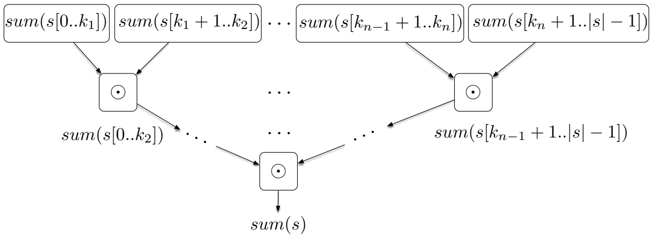


Figure 1: Divide and conquer style computation of sum.

[*] An extended version of this paper including proofs of theorems can be found at www.cs.toronto.edu/~azadeh/papers/pldi17-ex.pdf

[†] Funded by Ontario Ministry of Research's Early Researcher Award and NSERC Discovery Accelerated Supplement Award

PLDI'17 June 18–23, 2017, Barcelona, Spain

ACM ISBN 978-1-nnnn-nnnn-n/17/06…$15.00

DOI: http://dx.doi.org/10.1145/3062341.3062355

in producing non-trivial computations. A natural question to ask is whether this power can be leveraged for this problem.

In this paper, we focus on a class of divide-and-conquer parallel programs that operate on *sequences* (lists, arrays, or in general any collection type with a linear traversal iterator) in which the divide operator is assumed to be the default sequence *concatenation* operator (i.e. divide s into s_1 and s_2 where $s = s_1 \bullet s_2$). In Section 4, we discuss how we use syntax-guided synthesis (SyGuS) [2] *efficiently* to synthesize the join operators. Moreover, in Section 7, we discuss how the proofs of correctness for synthesized join operations can be automatically generated and checked. This addresses a challenge that many SyGuS schemes seem to bypass, in that, they only guarantee the synthesized artifact be correct for the set of examples used in the synthesis process (or boundedly many input instances), and do not provide a proof of correctness for the entire (infinite) data domain of program inputs.

A general divide-and-conquer parallel solution is not always as simple as the diagram in Figure 1. Consider the function *is-sorted*(s) which returns true if an array is sorted, and false otherwise. Providing the partial computation results, a boolean value in this case, from both sides (of a join) will not suffice. If both sub-arrays are sorted, the join cannot make a decision about the sortedness of the concatenated array. In other words, a join cannot be defined solely in terms of the sortedness of the subarrays.

To a human programmer, it is clear that the join requires the last element of the first subarray and the first element of the second subarray to *connect the dots*. The extra information in this case is as simple as remembering two extra values. But, as we demonstrate with another example in Section 2, the extra information required for the join may need to be *computed* by worker threads to be available to the join. Intuitively, this means that worker threads have to be modified (compared to the sequential code) to compute this extra information in order to guarantee the existence of a join operator. We call this modification of the code, *lifting* [1], for short, after the identical standard mathematical concept as illustrated in the diagram above, where A stands for the extra (auxiliary) information that needs to be computed by the loop.

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

The necessity of lifting in some cases raises two questions: (1) does such a lifting always exist? And, (2) can the overhead from lifting and the accompanying join overshadow the performance gains from parallelization? In Section 5, we address both questions. The challenge for automation is to modify the original sequential loop in a way that it computes *enough* additional information such that (i) a join does exist, (ii) the join is efficient, and (iii) the overhead of lifting is not unreasonably high. We lay the theoretical foundations to answer these questions, and in Section 6, we present an algorithm for producing this lifting automatically that satisfies all aforementioned criteria. In summary, the paper makes the following contributions:

- We present an algorithm to synthesize a join for divide-and-conquer parallelism when one exists. Moreover, these joins are accompanied by automatically generated machine-checked proofs of correctness (Sections 4, 7).
- We present an algorithm for automatic *lifting* of non-parallelizable loops (where a join does not exist) to transform them into parallelizable ones (Section 6).
- We lay the theoretical foundations for when a loop is *efficiently* parallelized (divide-and-conquer-style), and explore when an efficient *lift* exists and when it can be automatically discovered (Sections 5, 6).
- We built a tool, ParSynt, and present experimental results that demonstrate the efficacy of the approach and the efficiency of the produced parallel code (Section 8).

2. Overview

We start by presenting an overview of our contributions by way of two examples that demonstrate the challenges of converting a sequential computation to divide-and-conquer parallelism and help us illustrate our solution. We use *sequences* as linear collections that abstract any collection data type with a linear traversal iterator.

Second Smallest. Consider the loop implementation of the function $min2$ on the right, that returns the second smallest element of a sequence. Functions `min` and `max` are used for brevity and can be replaced by their standard definitions $min(a, b) = a < b\ ?\ a : b$ and $max(a, b) = a > b\ ?\ a : b$. Here, `m` and `m2` keep track of the smallest and the second smallest elements of the sequence respectively.

Figure 2: Second Smallest

```
m = MAX_INT;
m2 = MAX_INT;
for (i = 0; i < |s|; i++) {
   m2 = min(m2, max(m,s[i]));
   m = min(m, s[i]);
}
```

For a novice, in devising a join for a divide and conquer parallelization of this example, it is easy to make the mistake of using the incorrect updates illustrated on the right[2].

$$\boxed{\begin{aligned} \mathtt{m} &= min(\mathtt{m}_l, \mathtt{m}_r) \\ \mathtt{m2} &= min(\mathtt{m2}_l, \mathtt{m2}_r) \end{aligned}}$$

The correct join operator computes the combined smallest and second smallest of two subsequences according the following two equations, where the l and r subscripts distinguish the values coming from the *left* and the *right* subsequences (respectively):

$$\mathtt{m} = min(\mathtt{m}_l, \mathtt{m}_r)$$
$$\mathtt{m2} = min(min(\mathtt{m2}_l, \mathtt{m2}_r), max(\mathtt{m}_l, \mathtt{m}_r))$$

[1] Not to be confused with lambda lifting or the informal use of it in [24].

[2] A significant percentage of undergraduate students in an elementary algorithms class routinely make this mistake when given this exercise.

We use syntax-guided synthesis, to synthesize the correct join operators for sequential loops like this one. We use a template which is based on the code of the loop body with strategically introduced unknowns to define the state space of synthesis, and we use homomorphisms to introduce a sufficient correctness specification for synthesis (more on this in Section 4) that lends itself to efficient synthesis and facilitates automatic generation of correctness proofs for the synthesized code (more on this in Section 7).

Maximum Tail Sum. Consider the function mts that returns the maximum suffix sum of a sequence of integers (positive and negative). For example, $mts([1, -2, 3, -1, 3]) = 5$, which is associated to sum of the suffix $[3, -1, 3]$.

```
mts = 0;
for (i = 0; i < |s|; i++) {
   mts = max(mts + s[i], 0)
}
```

The code on the right illustrates a loop that implements mts. To parallelize this loop, the programmer needs to come up with a correct join operator. In this case, even the most clever programmer would be at a loss, since no correct join operator exists. Consider the partial sequence $[1, 3]$, when *joined with* two different partial sequences $[-2, 5]$ and $[0, 5]$. We have:

$$\begin{array}{r|r}
mts([1,3]) = 4 & mts([1,3]) = 4 \\
mts([-2,5]) = 5 & mts([0,5]) = 5 \\
mts([1,3,-2,5]) = 7 & mts([1,3,0,5]) = 9
\end{array}$$

The values of mts for both pairs of partial sequences are the same, yet, the values of the mts for the two concatenations are different. If a join *function* $\odot$ exists such that $mts(s \bullet t) = mts(s) \odot mts(t)$, then this function would have to produce two different results for $4 \odot 5$ in the two instances above. In other words, the mts value of the concatenation is *not computable* solely based on the mts values of partial sequences. What is to be done?

At this point, a clever programmer makes the observation that the loop is not *parallelizable* in its original form. She discovers that beyond knowing $mts([1, 3]) = 4$ and $mts([-2, 5]) = 5$, she needs to know that $sum([-2, 5]) = 3$, in order to conclude that $mts([1, 3, -2, 5]) = 7$.

```
mts = 0;
sum = 0;
for (i = 0; i < |s|; i++) {
   mts = max(mts + s[i], 0);
   sum = sum + s[i];
}
```

Consider a modification of the previous sequential loop for mts on the right. There is an additional loop variable, sum, that records the sum of all the sequence elements. In the sequential loop that computes mts, this is a *redundant* computation. We call sum an *auxiliary* accumulator, and the new loop a *lifting* of the sequential loop.

$$\boxed{\begin{array}{l} \texttt{sum} = \texttt{sum}_l + \texttt{sum}_r \\ \texttt{mts} = max(\texttt{mts}_r, \texttt{sum}_r + \texttt{mts}_l) \end{array}}$$

The lifted code can now be parallelized using the join operator on the right. For loops like this, the burden of the programmer is more than just coming up with the correct join implementation. She has to modify the original sequential code *to make it parallelizable*, and then devise the correct join for the lifted sequential loop. The algorithm presented in Section 6 does exactly that. Let us illustrate how the algorithm *discovers* the *auxiliary* accumulator sum.

Consider the general case of computing $mts(s)$ sequentially, where sequence s is of length n. Imagine a point in the middle of the computation, when the sequence s has been processed up to and including the index i computing the value mts_i. We have the following sequence of *recurrence*-like equations that represent the first two unfoldings of this sequential loop starting from index $i + 1$:

$$mts_{i+1} = max(mts_i + s[i + 1], 0) \tag{1}$$

$$\begin{aligned} mts_{i+2} &= max(mts_{i+1} + s[i + 2], 0) \\ &= max(max(mts_i + s[i + 1], 0) + s[i + 2], 0) \end{aligned} \tag{2}$$

When the (unfolded) computation above starts, the initial value of mts_i is not available to it, since it is being computed by a different thread simultaneously. The challenge is to perform the computation above, without having the value of mts_i in the beginning, and then *adjust* the computed value accordingly once the value becomes available (at join time). Looking at Equation 1, if the value of $s[i + 1]$ is known to the join operator, then it can plug values mts_i, $s[i + 1]$, and 0 (a program constant) into the expression on the righthand side and compute the correct value of mts_{i+1}. At this stage, our algorithm (presented in Section 6) *conjectures* $s[i + 1]$ as an auxiliary value and + as its accumulator, since it is actively looking for *accumulators* to *learn*, and + appears next to $s[i + 1]$ in the expression.

Once an auxiliary accumulator is conjectured, the algorithm turns its attention to the next unfolding, namely Equation 2 to test its *conjecture*. Here, the unknown mts_i is sitting one level *deeper* in the expression. This means that the join has to perform more steps to compute the value of mts_{i+2}. It is easy to see that the depth of mts_i will linearly increase for mts_{i+3} and so on. At this point, the tool decides to use a set of standard algebraic equalities to rewrite the righthand side of Equation 2 to a different expression where mts_i appears at *a lower depth*, following the insight that the closer it is to the root, the less the amount of work left to the join operator. The step by step rewriting of the expression happens as follows:

$$\begin{aligned} mts_{i+2} &= max(max(mts_i + s[i + 1], 0) + s[i + 2], 0) \\ &= max(max(max(mts_i + s[i + 1], s[i + 1]), 0) + s[i + 2], 0) \\ &= max(max(max(mts_i + s[i + 1] + s[i + 2], \\ &\qquad s[i + 1] + s[i + 2]), s[i + 2]), 0) \\ &= max(mts_i + s[i + 1] + s[i + 2], \\ &\qquad max(s[i + 1] + s[i + 2], s[i + 2], 0)) \\ &= max(mts_i + s[i + 1] + s[i + 2], mts([s[i + 1..i + 2]])). \end{aligned}$$

The structure of the expression in the last line provides a clue to how the parallel computation can be organized. While one thread is computing mts_i (i.e. $mts(s[0..i])$), the second thread computes $mts(s[i + 1..i + 2])$. To compute the overall value of $mts(s[0..i + 2])$, the join would need the partial results mts_i **and** $mts(s[i+1..i+2])$, **and** the value of $s[i+1]+s[i+2]$. The algorithm decides at this point that $s[i+1]+s[i+2]$ is the auxiliary value that needs to be computed, and realizes that the *conjectured* accumulator from the previous round makes this information already available. Therefore, not having seen anything new, it stops and concludes that the auxiliary accumulator $sum = sum + s[k]$ (where k is the current iteration number) is the new auxiliary computation required to make the loop parallelizable.

3. Background and Notation

This section introduces the notation used in the remainder of the paper. It includes definitions of some new concepts, and some already known in the literature.

3.1 Sequences and Functions

We assume a generic type Sc that refers to any scalar types used in typical programming languages, such as `int`, `float`, `char`, and `bool` whenever the specific type is not important in the context. The significance of the type is that scalars are assumed to be of *constant* size, and conversely, any constant-size representable data type is assumed to be scalar in this paper. Moreover, we assume all operations on scalars to have constant-time complexity.

Type $\mathcal{S}$ defines the set of all *sequences* of elements of type Sc. For any sequence x, we use $|x|$ to denote the length of the sequence. $x[i]$ (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). Concatenation operator $\bullet : \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ is defined over sequences in the standard way, and is associative. The sequence type stands in for many collection types such as *arrays*, *lists*, or any other collection that admits a linear iterator and an *associative* composition operator (i.e. concatenation).

In this paper, our focus is on *single-pass computable* functions on sequences (of scalars to scalars). The assumption is that a sequential implementation for the function is given as an imperative sequential loop. Below, we formally define what a *single-pass computable* function is.

Definition 3.1. *A function $h : \mathcal{S} \rightarrow D$ is called rightwards iff there exists a binary operator $\oplus : D \times Sc \rightarrow D$ 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).

Definition 3.2. *A function $h : \mathcal{S} \rightarrow D$ is called leftwards iff there exists a binary operator $\otimes : Sc \times D \rightarrow D$ such that for all $x \in \mathcal{S}$ and $a \in Sc$, we have $h([a] \bullet x) = a \otimes h(x)$.*

Associativity of $\otimes$ is likewise not well-defined.

$$
\begin{array}{llrl}
e \in \mathsf{Exp} ::= & e \circ e' & e, e' \in \mathsf{Exp} & \text{Expressions} \\
& |x \mid k & x \in \mathsf{Var}, k \in \mathbb{Z} & \\
& |s[e] & s \in \mathsf{SeVar}, e \in \mathsf{Exp} & \\
& |\mathsf{if}\ be\ \mathsf{then}\ e\ \mathsf{else}\ e' & & \\
& & & \\
be \in \mathsf{BExp} ::= & e \olessthan e' & e, e' \in \mathsf{Exp} & \text{Boolean Exps} \\
& |be \curlywedge be' | \neg be & be, be' \in \mathsf{BExp} & \\
& |t[e] & t \in \mathsf{BSVar}, e \in \mathsf{Exp} & \\
& |\mathsf{true} \mid \mathsf{false} & & \\
\mathsf{Program} ::= & c; c' & c, c' \in \mathsf{Program} & \\
& |x \texttt{:=} e & x \in \mathsf{Var}, e \in \mathsf{Exp} & \\
& |b \texttt{:=} be & b \in \mathsf{BVar}, be \in \mathsf{Exp} & \\
& |\texttt{if}(e)\{c\}\texttt{else}\{c'\} & be \in \mathsf{Exp}, c, c' \in \mathsf{Program} & \\
& |\texttt{for}(i \in \mathcal{I})\{c\} & i \in \mathsf{Iterator} &
\end{array}
$$

Figure 3: Program Syntax . The binary $\circ$ operator represents any arithmetic operation $(+, -, *, /)$, $\olessthan$ operator represents any comparator $(<, \leq, >, \geq, =, \neq)$. $\mathcal{I}$ is an iteration domain, and $\curlywedge$ operator represents any boolean operation $(\wedge, \vee)$.

Definition 3.3 (*Single-Pass Computable*)**.** *Function $h : \mathcal{S} \rightarrow D$ is single-pass computable iff it is rightwards* **or** *leftwards.*

3.2 Homomorphisms

Homomorphisms are a well-studied class of mathematical functions. In this paper, we focus on a special class of homomorphisms, where the source structure is a set of sequences with the standard concatenation operator.

Definition 3.4. *A function $h : \mathcal{S} \rightarrow D$ is called $\odot$-homomorphic for binary operator $\odot : D \times D \rightarrow D$ iff for all sequences $x, y \in \mathcal{S}$ we have $h(x \bullet y) = h(x) \odot h(y)$.*

Note that, even though it is not explicitly stated in the definition above, $\odot$ is necessarily associative on D since concatenation is associative (over sequences). Moreover, $h([])$ (where $[]$ is the empty sequence) is the unit of $\odot$, because $[]$ is the unit of concatenation. If $\odot$ has no unit, that means $h([])$ is undefined. For example, function $head(x)$, that returns the first element of a sequence, is not defined for an empty list. $head(x)$ is $\circledast$-homomorphic, where $a \circledast b = a$ (for all a, b) but $\circledast$ does not have a left unit element.

3.3 Programs and Loops

For the presentation of the results in this paper, we assume that our sequential program is written in a simple imperative language with basic constructs for branching and looping. We assume that the language includes scalar types `int` and `bool`, and a collection type `seq`. The syntax of our input programs is illustrated in Figure 3. We forego a semantic definition since it is standard and intuitively clear. For readability, we use a simple iterator and an integer index (instead of the generic $i \in \mathcal{I}$), and use the standard array random access notation $a[i]$ to refer to each element of a collection type. In principle, any collection with an iterator and a split function (that implements inverse of concatenation) works. There has been a lot of research on iteration spaces and iterators (e.g.

[51] in the context of translation validation and [25] in the context of partitioning) that formalize complex traversals by abstract iterators.

State and Input Variables. Every variable that appears on the lefthand side of an assignment statement in a loop body is called a *state variable*, and the set of state variables is denoted by SVar. Every other variable is an *input variable*, and the set of input variables is denoted by IVar. The sequence that is being read by the loop is considered an *input variable*, since it is only read and not modified. Note that $\mathsf{SVar} \cap \mathsf{IVar} = \varnothing$.

Model of a Loop Body

We introduce a formal model for the body of a loop which has no other loops nested inside it. The loop body, therefore, consists of assignment and conditional statements only.

Let $\mathsf{SVar} = \{s_1, \ldots, s_n\}$. The body is modelled by a system of (ordered) recurrence equations $E = \langle E_1, \ldots E_n \rangle$, where each equation E_i is of the form $s_i = \mathsf{exp}_i(\mathsf{SVar}, \mathsf{IVar})$.

Remark 3.5. *Every non-nested loop in the program model in Figure 3 can be modelled by a system of recurrence equations (as defined above).*

This can be achieved through a translation of the loop body from the imperative form to the functional form. In [15], we include a description of how this can be done, but also refer the interested reader to [3, 26] for a more complete explanation. The essence of this translation, which is through transformation of conditional statements into conditional expressions, is illustrated in the example below.

Example 3.6. Consider the long version of the second smallest example from Section 2 on the right, where the auxiliary functions *min* and `max` are replaced with their definitions as conditional statements and where the code complies with the syntax in Figure 3.

```
m = MAX_INT;
m2 = MAX_INT;
for (i = 0; i < |s|; i++) {
  if m > s[i] then
     if m2 > m then m2 = m;
   else
     if m2 > s[i] then m2 = s[i];
   if m > s[i] then m = s[i];
}
```

The loop body is then modelled by recurrence equations on the right, through the transformation of the conditional statements above into the assignment statements that use conditional expressions ($\mathsf{if}\ be\ \mathsf{then}\ e\ \mathsf{else}\ e'$).

```
m2 = if m2 < (if m > s[i] then m else s[i])
     then m2
     else (if m > s[i] then m else s[i]);
m = if m < s[i] then m else s[i];
```

4. Synthesis of Join

The premise of this section is that the sequential loop under consideration is parallelizable, in the sense that a join operation for divide-and-conquer parallelism exists. We use a system of recurrence equations E in the style of Section 3.3 to model the body of a loop for all formal and algorithmic developments in this paper, and therefore, we use the term *loop body* to refer to E whenever appropriate.

4.1 Parallelizable Loops

We start by formally defining when a loop is parallelizable. This is intuitively related to when the computation performed by the loop body is a *homomorphic* function. First, we define a function f_E that models E (the body of the loop). The reader who is not interested in the details can fast-forward to Definition 4.1 with an intuitive understanding of function f_E.

Let the system of recurrence equations $E = \langle s_1 = \mathsf{exp}_1(\mathsf{SVar}, \mathsf{IVar}), \ldots, s_n = \mathsf{exp}_n(\mathsf{SVar}, \mathsf{IVar}) \rangle$ represent the body of a loop. Let $\mathsf{IVar} = \{\sigma, \iota, i_1, \ldots, i_k\}$ where σ is the sequence variable, and ι is the current index (which we assume to be an integer for simplicity) in the sequence, and $\vec{i} = \langle i_1, \ldots, i_k \rangle$ captures the rest of the input variables in the loop body. Let $\mathcal{I}$ be the set of all such k-ary vectors.

For a loop body E, define function $f_E = f_1 \times f_2 \times \ldots \times f_n$, where each $f_i : \mathcal{S} \times \mathit{int} \times \mathcal{I} \to \mathit{type}(s_i)$ is a function such that $f_i(\sigma, \iota, \vec{i})$ returns the value of s_i at iteration ι with input values σ and $\vec{i}$. It is, by definition, straightforward to see that $\langle f_1(\sigma, \iota, \vec{i}), \ldots, f_n(\sigma, \iota, \vec{i}) \rangle$ represents the state of the loop at iteration ι.

Definition 4.1. *A loop, with body E, is parallelizable iff f_E is $\odot$-homomorphic for some $\odot$.*

Definition 4.1 basically takes the encoding of the loop as a *tupled* function, and declares the loop parallelizable if this tupled function is homomorphic. It is important to note that *parallelizable* here is not used in the broadest sense of the term. It is limited to the particular scheme of divide-and-conquer parallelism where the divide operator is the inverse of concatenation.

4.2 Syntax-Guided Synthesis of Join

We use syntax-guided synthesis (SyGuS)[2] to synthesize the join operator for parallelizable loops. The goal of program synthesis is to automatically synthesize a correct program based on a given specification. SyGuS is an emerging field with several existing solvers. The task of the user of these solvers is to define (1) a correctness specification for the program to be synthesized, and (2) provide syntactic constraints that define the state space of possible programs (mainly for tractability). In this section, we describe what the correctness specification and syntactic constrains that we use. Beyond that, the specific design of these two elements, this work does not make any new contributions to SyGuS.

Note that we focus on synthesizing *binary* join operators in this paper. The restriction is superficial, in the sense that any other statically fixed number would work. And, it is not hard to imagine an easy generalization to a parametric join.

Correctness Specification

The homomorphism definition 3.4 provides us with the correctness specification to synthesize a join operator $\odot$ for a loop body E (or rather f_E to be precise). In case of the specific SyGuS solver used in this paper (and many others similar to it), a bounded set of possible inputs are required, and therefore, the correctness specification is formulated on symbolic inputs of bounded length K. A join $\odot$ is a solution of the synthesis problem if for all sequences x, y of length less than K, we have $f_E(x \bullet y) = f_E(x) \odot f_E(y)$.

There is a tension between having a small enough K for the solver to scale, and having a large enough K for the answer to be correct for inputs that are larger than K. We use small enough values for the solver to scale, and take care of general correctness by automatically generating proofs of correctness (see Section 7).

Syntactic Restrictions

The syntactic restrictions are defined through a pair of a *sketch* and a grammar for expressions. A *sketch* is a partial program containing *holes* (i.e. unknown values) to be completed by expressions in the state space defined by the grammar. The sketch is an ordered set of equations $E = \langle s_1 = \mathsf{exp}_1(\mathsf{SVar}, \mathsf{IVar}), \ldots, s_n = \mathsf{exp}_n(\mathsf{SVar}, \mathsf{IVar})\rangle$. Intuitively, it is produced from the loop body by replacing occurrences of variables and constants by holes. Note that the inputs to a join are the results produced from two worker threads, to which, we refer as *left* and *right* threads. To contain the state space of solutions described by this sketch, we distinguish two different types of holes.

Left holes $??_{LR}$ are holes that can be completed by an expression over variables from both the left and the right threads. *Right holes* $??_R$ will be filled with expressions over variables from the right thread only.
We define a compilation function $\mathcal{C}$ as

$$\begin{array}{rcl} \mathcal{C}\,(c) & = & ??_R \\ \mathcal{C}\,(x) & = & \begin{cases} ??_R & \text{if } x \in \mathsf{IVar} \\ ??_{LR} & \text{if } x \in \mathsf{SVar} \end{cases} \\ \mathcal{C}\,(x[e]) & = & ??_R \\ \mathcal{C}\,(op(e_1, .., e_n)) & = & op\,(\mathcal{C}(e_1), ..., \mathcal{C}(e_n)) \end{array}$$

where e is an expression, *op* is an operator from the input language, x is a variable, and c is a constant. The sketch for the join code will then be

$$\mathcal{C}(E) = \langle s_1 = \mathcal{C}(\mathsf{exp}_1(\mathsf{SVar}, \mathsf{IVar})), \ldots, s_n = \mathcal{C}(\mathsf{exp}_n(\mathsf{SVar}, \mathsf{IVar}))\rangle$$

where each hole in $\mathcal{C}(\mathsf{exp}_i)$ $(1 \le i \le n)$ can be completed by expressions in a predefined grammar that is suitable for a given class of programs. For the experiments in this paper, the grammar in Figure 4 is used, where the expression depth d is gradually increased until a solution is found.

Example 4.2. Consider the second smallest example from Section 2. The sketch (for the code in Figure 2) is illustrated on the right. It is clear that the join presented in Section 2 can be simply discovered using this Sketch when the unknowns are filled by instances of state variables.

```
m2 = min(??LR, max(??LR, ??R));
m = min(??LR, ??R);
```

4.3 Efficacy of Join Synthesis

Syntactic limitations imposed for scalability in SyGuS run the risk of leaving a correct candidate out of the search space.

$$\begin{array}{lll} ne_0 & ::= x \mid c & x \in nVars, c \text{ a numeric constant} \\ ne_{d>0} & ::= \; ne_{d-1} \,\oplus\, ne_{d-1} \mid -ne_{d-1} & \\ & \mid \texttt{if}\;(be_{d-1})\; ne_{d-1}\; \texttt{else}\; ne_{d-1} & \\ & & \\ be_0 & ::= b \mid \mathit{true} \mid \mathit{false} & b \in bVars \\ be_{d>0} & ::= \; be_{d-1} \,\owedge\, be_{d-1} \mid \neg be_{d-1} & \\ & \mid ne_{d-1} \,\olessthan\, ne_{d-1} & \\ & \mid \texttt{if}\;(be_{d-1})\; be_{d-1}\; \texttt{else}\; be_{d-1} & \end{array}$$

$\oplus$	:=	$+, -, min, max, \times, \div$	binary numeric
$\olessthan$	:=	$>, >=, <, <=, =$	comparisons
$\owedge$	:=	$\wedge, \vee$	binary boolean

Figure 4: Grammar of expressions used for $??_{LR}$ and $??_R$ holes. ne_d and be_d correspond to expressions of depth up to and equal to d. $nVars$ and $bVars$ stand for numeric and booleans variables.

To characterize the scope of expressivity of our compilation function $\mathcal{C}$, we provide formal conditions under which the synthesis approach of Section 4.2 is successful in discovering a correct join. This also facilitates a comparison of our join synthesis with the parallelization approach of [34].

Definition 4.3. *Function g is a weak right inverse of function f iff $f \circ g \circ f = f$.*

Each function may have many weak right inverses. Intuitively, g produces a sequence $s^{'}$ (out of a result $r = f(s)$) such that $f(s^{'}) = r = f(s)$. Therefore, $s^{'}$ (if much shorter than s) can be viewed as a summary of s with respect to the computation of f on s. In [34], this very notion of a weak right inverse is used to produce parallel implementations of functions. It is required that g's output is bounded or equally that $s^{'}$ is always a constant length sequence (i.e. independent of $|s|$). Our join synthesis is also guaranteed to succeed under the same assumption. Note that, however, in contrast to [34], we do not require *both* left and right implementations of the function as input, and either one alone suffices.

Proposition 4.4. *If the loop body E is parallelizable and the weak right inverse of f_E is constant time computable (that guarantees that it returns a list of bounded length), then there is a $\odot$ in the space described by the sketch of E such that f_E is $\odot$-homomorphic.*

Proposition 4.4 (see [15] for a proof sketch) provides us with the guarantee that under the conditions given, the state space defined for the join, through the sketch compilation function $\mathcal{C}$ does not miss an existing solution. Note that the compilation function $\mathcal{C}$ maintains the structure of the expressions when it compiles a sketch (Remember Example 4.2). When the weak inverse is constant-time computable, the *third homomorphism theorem* [19] guarantees that a solution faithful to this structure exists. The computation of the weak inverse itself is done through the plugged expressions for the unknowns $??_L$ and $??_{LR}$ where a fully generic grammars as illustrated in Figure 4 is used, that includes all possible constant-time computations at an appropriate depth.

Proposition 4.4 characterizes only a subset of functions, for which join synthesis succeeds. There are many (simple) functions whose weak right inverse is not bounded, where a join is successfully synthesized. For example, function $length$ is a simple example, where the weak right inverse always has to be a list of the same length as the original list (which would lead to an inefficient parallel implementation). If join synthesis fails, our tool un-constrains the compiled sketch to include more expressions gradually until a join is found. In practice, this was not necessary for any of our benchmarks. In Section 6.3, we comment on how some feedback from the algorithm presented in Section 6.1 helps with constructing an effective sketch when generalization beyond the sketch compilation function $\mathcal{C}$ may be required.

5. Parallelizability

The mts (maximum tail sum) example from Section 2 demonstrates that a loop is not always parallelizable. Here (and later in Section 6), we discuss how a loop can be *lifted* to become parallelizable.

5.1 Homomorphisms and Parallelism

A strong connection between homomorphisms and parallelism is well-known [18, 20, 34], specifically that, homomorphic functions admit efficient parallel implementations.

Theorem 5.1 (First Homomorphism Lemma [9])**.** *A function $h : \mathcal{S} \to Sc$ is a homomorphism if and only if h can be written as a composition of a map and a reduce operation.*

The *if* part of Theorem 5.1 basically states that a homomorphism has an efficient parallel implementation, since efficient parallel implementations for *maps* and *reductions* are known. The *only if* part is equally important, because it indicates that a large class of *simple and easy to formulate* parallel implementations of functions ought to be homomorphisms. Theorem 5.1 gives rise to an important research question: If a function is not a homomorphism, can the sequential code be modified (lifted) so that it corresponds to a homomorphic function to facilitate easy parallelization? In Section 6, we answer the question in the affirmative. But first in this section, we argue why this lifting should be done carefully to maintain performance advantages of parallelization.

Non-homomorphic Functions. There is a simple observation, which was made in [21] (and probably also in earlier work), that every non-homomorphic function can be made homomorphic by a rather trivial lifting:

Proposition 5.1. *Given a function $f : \mathcal{S} \to Sc$, the function $f \rtimes \iota$ is $\diamondsuit$-homomorphic where*

$$\iota(x) = x \quad \wedge \quad (a, x) \diamondsuit (b, y) = f(x \bullet y)$$

Operation $\rtimes$ denotes *tupling* of the two functions in the standard sense $f \rtimes g\ (x) = (f(x), g(x))$. Basically, the ι component of the tuple *remembers* a copy of the string that is being processed and the partial computation results $f(x)$ and $f(y)$ are discarded by $\diamondsuit$, which then computes $f(x \bullet y)$ from scratch. It is clear that a parallel implementation based on this idea is less efficient than the sequential implementation of f. Therefore, just making the function homomorphic will not result in a good solution for parallelism. Next, we identify a subset of homomorphic liftings that are computationally efficient.

5.2 Computationally Efficient Homomorphisms

Let us assume that $f : \mathcal{S} \to D$ is a *single-pass linear time computable* (see Definition 3.3) function and elements of D are tuples of scalars. The assumption that the function is linear time comes from the fact that we target loops without any loops nested inside, which (by definition) are linear-time computable (on the size of their input sequence).

Consider the (rightwards) sequential computation of f that is defined using the binary operator $\otimes$ as follows:

$$f(y \bullet a) = f(y) \otimes a$$

where $a \in Sc$, $f : \mathcal{S} \to D$, and $\otimes : D \times Sc \to D$. The fact that f is single-pass linear time computable implies that the time complexity of computing f at every step (of the recursion above) is constant-time. Let f' be a lifting of f that is $\odot$-homomorphic (for some $\odot$).

Proposition 5.2. *The (balanced) divide-and-conquer implementation based on f' has the same asymptotic complexity as f (i.e. linear time) iff the asymptotic complexity of $\odot$ is sub-linear (i.e $o(n)$).*

The simplest case is when $\odot$ is constant time. We call these *constant homomorphisms* for short. Note that the synthesis routine presented in Section 4 synthesizes *constant time* join operators exclusively. In the next section, we present an algorithm that lifts a non-homomorphic function to a *constant homomorphism* if one exists.

Let us briefly consider the super-constant yet sub-linear case for joins, to justify why this paper only focuses on the constant-time joins and ignores the super-constant joins. Based on the definition of homomorphism, we know:

$$f'([x_1, \dots x_n]) = f'([x_1, \dots x_k]) \odot f'([x_{k+1}, \dots x_n])$$

For $\odot$ to be super-constant in n, $f'([x_1, \dots x_k])$ (respectively, $f'([x_{k+1}, \dots x_n])$) has to produce a super-constant size output. Constant size inputs (to $\odot$) cannot yield super-constant time executions. Since the output of f is assumed to be constant size (scalars and tuples of them are by definition constant size), the output to f' is super-constant only due to the additional *auxiliary* computation that makes f' a homomorphism, but is not part of the sequential computation of f. We believe the automatic discovery of auxiliary information of this nature can be a fundamentally hard problem. The sub-linear (super-constant) auxiliary information is often the result of a clever algorithmic trick that improves over the trivial linear auxiliary (i.e. remembering the entire sequence as was discussed in Proposition 5.1). The field of efficient *data streaming algorithms* [1, 4] includes a few examples of such clever algorithms. Join synthesis can be adapted to handle such cases if the auxiliary information is available. However, since discovery of super-constant auxiliary information is a necessary step for automation, and extremely difficult to do automatically, we only target constant homomorphisms in this paper.

6. Synthesizing Parallelizable Code

Let us assume that the synthesis of the join operator from Section 4.2 fails. The reason could be that either (1) the function is not a homomorphism, or (2) the function is a homomorphism, but syntactic restrictions for synthesis exclude the correct join operator. We deal with case (1) by proposing an algorithm that *lifts* the function to a homomorphism by adding new computation to the loop body, and briefly comment on case (2) in Section 6.3.

6.1 The Algorithm

```
bal = true;
ofs = 0;
for (i = 0; i < |s|; i++) {
  if s[i] == '(' then
     ofs = ofs + 1;
  else
     ofs = ofs - 1;
  bal = bal && (ofs >= 0);
}
```

In Section 2, we used the mts example to show how inspecting unfoldings of the computation could result in the discovery of discovery of the *auxiliary accumulator* sum. There, the intuitive idea was to inspect expressions computing mts values to find an equivalent expression where the unknown appeared at a lower *depth*. Here, we use a new example, to introduce our algorithm, and provide insight for why the algorithm goes beyond just lowering the depth of the unknowns. The code above corresponds to a sequential loop that checks if an input string is a balanced sequence of parentheses. The integer variable `ofs` (for offset) keeps track of the difference in the number of open and closed parentheses seen so far, and the boolean variable `bal` maintains the status of balancedness of prefix read so far. At the end of the loop, the string is balanced if $ofs = 0 \wedge bal = true$.

If the loop only consisted of `ofs`, then it would correspond to a homomorphism. It is easy to see that $ofs(x \bullet y) = ofs(x) + ofs(y)$. The same is not true for `bal`. If x is balanced and y is not, then $x \bullet y$ could be balanced (or not). To determine this, information beyond the boolean values of `bal` and the integer values of `ofs` for x and y is required.

Let us see how our algorithm discovers the required auxiliary information. Consider breaking the sequential computation of the above code into two threads as illustrated in Figure 5. The value of $\langle ofs, bal\rangle(s[0..k])$ (illustrated in red), that is the result of the computation on the left, will not be known to the computation on the right when it is needed at the very beginning. This dependency is the killer of parallelism. The idea behind Algorithm 1 is to see if we can *rearrange* the computation on the right so that it can be performed with a constant known initial value, instead of the unknown $\langle ofs, bal\rangle(s[0..k])$. Then at the end, when the values become known (through the left thread), the *join operator* would *adjust* the result based on this known value.

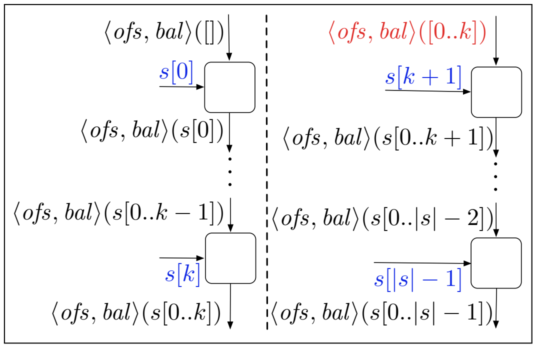

$\langle ofs, bal\rangle([])$
$s[0]$
$\langle ofs, bal\rangle(s[0])$
$\langle ofs, bal\rangle(s[0..k-1])$
$s[k]$
$\langle ofs, bal\rangle(s[0..k])$
$\langle ofs, bal\rangle([0..k])$
$s[k+1]$
$\langle ofs, bal\rangle(s[0..k+1])$
$\langle ofs, bal\rangle(s[0..|s|-2])$
$s[|s|-1]$
$\langle ofs, bal\rangle(s[0..|s|-1])$

Figure 5: Sequential computation broken into two threads.

Algorithm 1 illustrates our main algorithm. The algorithm (in function *Lift()*) inspects each state variable separately to see if its divide-and-conquer computation requires introduction of new auxiliary accumulators (by calling *Solve()*). In our example, the algorithm discovers that `ofs` requires no new auxiliary accumulators, but `bal` does.

In function *Solve()*, it starts simulating the loop from an arbitrary state $\langle s_0^0, \ldots, s_n^0\rangle$, mirroring the arbitrary break in Figure 5 where the red value is swapped with this arbitrary initial state. The while loop then iteratively unfolds the expression in the style of Equations 1 and 2 in Section 2 for the mts example. 'unfold' (in Algorithm 1) computes the k-th unfolding, for a given k. Let us focus on `bal`, which is the problematic part of the loop state, and consider the first few unfoldings of the computation of the right hand side thread. The first step is as follows:

$$\begin{aligned} bal(s[0..k+1]) &= bal(s[0..k]) \wedge (ofs(s[0..k+1]) \geq 0) \\ &= bal(s[0..k]) \wedge \\ &\quad\; ofs(s[0..k]) + ofs([s[k+1]]) \geq 0 \end{aligned}$$

using the equality $ofs(x \bullet y) = ofs(x) + ofs(y)$. The final expression above has both its unknown (red) values at the optimal depth in the expression tree, and does not seem to require any auxiliary information to compute it once the unknowns become knowns[3]. Let us look at the next unfolding (using the result of the first step above):

$$\begin{aligned} bal(s[0..k+2]) &= bal(s[0..k+1]) \wedge (ofs(s[0..k+2]) \geq 0) \\ &= [bal(s[0..k]) \wedge \\ &\quad (ofs(s[0..k]) + ofs([s[k+1]])) \geq 0] \wedge \\ &\quad [ofs(s[0..k]) + ofs(s[k+1..k+2]) \geq 0] \end{aligned}$$

Now, an alarming pattern seems to emerge. The number of occurrences of the unknown $ofs(s[0..k])$ has doubled, and to compute the expression once it is known, one needs to store the intermediate result for $ofs([s[k+1]])$ to use. Intuitively, it is clear that with 3 or more unfoldings, the pattern continues: the number of unknowns are arbitrary replicated,

[3] To be fully accurate, the algorithm guesses an auxiliary accumulator here which will later be declared to be redundant since it is effectively `ofs`.

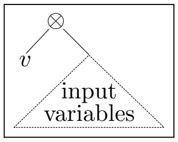
```
Algorithm 1: Homomorphic Lifting Computation
  Data: A set of recurrence equations $E$
  Result: A set of recurrence equations $E'$
  Function Lift($E$)
      $E' \leftarrow \emptyset$;
      for each $s_i = \mathsf{exp}_i$ in $E$ (in order) do
          if Solve(“$s_i = \mathsf{exp}''_i$”) = null then
              report failure
          else
              $E' \leftarrow E' \cup$ Solve(“$s_i = \mathsf{exp}''_i$”);
      return $E'$;
  Function Solve($s = \mathsf{Exp}(\mathsf{SVar}, \mathsf{IVar})$)
      Initially $k = 1$, $\mathsf{Aux} = \emptyset$, $\sigma_0 = \langle s_0^0, \dots, s_n^0 \rangle$
      while $\mathsf{Aux} \neq \mathsf{OldAux}$ do
          $\mathsf{OldAux} \leftarrow \mathsf{Aux}$ ;
          $\tau \leftarrow \mathsf{unfold}(\sigma_0, s, E, k)$;
          $\ell \leftarrow Normalize(\tau, E)$;
          if $\ell = empty$ then
              return $empty$;
          $\mathcal{E} \leftarrow \mathsf{collect}(\ell, \mathsf{SVar})$;
          for each $e$ in $\mathcal{E}$ do
              if $e$ already covered by something in Aux
              then
                  Add the accumulator and continue
              else
                  Create a new variable in Aux, with
                  expression $e$.
          $k \leftarrow k + 1$;
      $\mathsf{Aux} \leftarrow remove\text{-}redundancies(\mathsf{Aux})$;
      return Aux;
  Function Normalize($\tau, E$)
      Output: A set of expressions $\{e_1, \dots, e_k, e\}$ such
              that $\tau = f(e_1, \dots, e_k, e)$, each $e_i$ has
              exactly one occurrence of a state variable
              at depth 1.
```

and all intermediate values of $ofs([s[k+1..k+m]])$ need to be stored (for all $1 \leq m \leq n-k-1$), to retain the ability to compute the total `bal` at the end. This leads us to the second principle that we use in rearranging these expressions, namely *reducing the number of occurrences of the unknowns*. Using the last algebraic rule in Figure 6, the above expression can be rewritten as:

$$bal(s[0..k+2]) = bal(s[0..k]) \wedge$$
$$[ofs(s[0..k]) \geq max(-ofs([s[k+1]])), -ofs(s[k+1..k+2])]$$

This solves the problem. Instead of having to remember all intermediate values of $ofs([s[k+1..k+m]])$, the loop can just remember the maximum value of their negations. This can be added as an auxiliary accumulator to the sequential loop to lift it to a homomorphism. Algorithm 1 finds the equivalent expression above through the call to ‘*normalize*’, which is the most important step of the algorithm. The details of normalization are explained in the next section.

‘*normalize*’ takes the unfolded expression and returns an equivalent expression of the form $\mathsf{exp}(e_1, \dots, e_m, e)$ where e only refers to the input variables and each e_i is an expression of the form illustrated in the inset, where v is a state variable. The intuition is that other than the occurrences of the state variables (which is the *unknown*), *the remainder* of e_i needs to captured by an auxiliary accumulator and made available to the *join* operator. In our example, we would have $m = 2$, $e = true$, $e_1 = bal(s[0..k])$, and $e_2 = ofs(s[0..k]) \geq max(-ofs([s[k+1]])), -ofs(s[k+1..k+2])$. This leads to $max(-ofs([s[k+1]])), -ofs(s[k+1..k+2])$ being conjectured as an auxiliary accumulator from e_2, and nothing to be conjectured from e_1. Finally, ‘collect’ gathers these e_i’s in $\mathcal{E}$. For each e_i, we check if it can be assembled using existing auxiliary accumulators in Aux. If it can, the algorithm moves on. If no, a new auxiliary variable and accumulator is added. The while loop continues until no new auxiliaries are discovered.



Normalization

The most complex step of Algorithm 1 is the task performed by ‘*normalize*’. The description of ‘*normalize*’ in Algorithm is intentionally left as a functional specification only, without an implementation. This abstract version of *normalize is assumed to always succeeds when the appropriate auxiliary accumulator exist*. This is based on the simple idea that if the algebraic rules needed for simplification/manipulation of the expression are given, and effective strategy exists to explore the state space of expressions defined by these rules, then an existing expression in the space will always be found. We assume this idealized version of *normalize* to propose a crisp completeness theorem in Section 6.2.

The existence of an effective and efficient implementation (to replace this idealized version) depends on the given expression, operators that appear in it, and the algebraic equalities that hold for those operators. There has been a lot of research in the area of rewrite systems [29, 30, 35] that can inspire several heuristics for normalization. The problem can also be formulated using standard syntax-guided synthesis, although we suspect that this solution will not scale well since (in addition to standard scalability problems in SyGuS) non-linearity can easily be introduced, even in simple cases.

The main complication in a search for an equivalent expression (of a given form) is that the state space for this search can be infinite, with no particular structure to guide a finitary search. We propose a heuristic normalization procedure that utilizes a *cost function* to enforce a finitary search. The heuristic is simple and yet seems to work well (efficiently and effectively) for the benchmarks in this paper. Below, $dep_e(v)$ is the depth of the deepest occurrence of v and $occ_e(v)$ is the number of occurrences of v in expression e.

Cost-directed rewrite rules (apply the rule only if it reduces the cost of the expression):

$a \odot b$	→	$b \odot a$
$(a \odot b) \odot c$	→	$a \odot (b \odot c)$
$(a \odot b) \otimes c$	→	$(a \otimes c) \odot (b \otimes c)$
$(c\,?\,x\,:\,y) \odot z$	→	$c\,?\,(x \odot z)\,:\,(y \odot z)$
$c1\,?\,(c2\,?\,x\,:\,y)\,:\,z$	→	$c1 \wedge c2\,?\,x\,:\,(\neg c2\,?\,y\,:\,z)$
$\neg(a \wedge b)$	→	$(\neg a) \vee (\neg b)$
$-(a + b)$	→	$(-a) - b$
$a > c \vee b > c$	→	$max(a,\, b) > c$
$c > a \wedge c > b$	→	$c > max(a,\, b)$

Figure 6: $\odot$ and $\otimes$ stand in for the appropriate arithmetic ones. For brevity, we only include one rule from each algebraic equality omitting the symmetric versions from the list.

Definition 6.1. *The* cost function $\mathsf{Cost}_V : \mathsf{exp} \to \mathsf{Int} \times \mathsf{Int}$ *assigns to any expression e, a pair (d, n) where d is $Max_{v \in V}\, dep_e(v)$, and n is $\sum_{v \in V} occ_e(v)$.*

The process of normalization can be formalized by a set of rewrite rules R that are derived from a set of standard algebraic equalities that hold for the operators appearing in the program text. Each algebraic equality gives rise to two rewrite rules (one for each direction of the equality). Figure 6 includes instances of the rules that we use in our implementation. Given a set of algebraic rules R, we define the *cost minimizing* normalization procedure as the successive application of rewrite rules in R in order to reduce an expression to an equivalent expression with minimal cost. The algorithm is a standard cost-driven search algorithm. Evaluation of this algorithm (with the set of rules provided in Figure 6) in Section 8 demonstrates that despite its simplicity, it is fast and effective in finding the fruitful normal forms. It fails to infer only 1 auxiliary accumulator (among 15), and the reason for that is the shortcoming of the set of rewrite rules in dealing with conditional statements. The algebraic rules regarding conditionals in Figure 6 can only normalize light instances of conditionals. Devising a sophisticated heuristic normalization is a topic of interest for future work.

Soundness of Algorithm 1

The following Proposition formally states the correctness of Algorithm 1. Note that in all formal statements about Algorithm 1, we assume the idealized version of '*normalize*' as illustrated in Algorithm 1.

Proposition 6.2. *If Algorithm 1 terminates successfully and reports no new auxiliary variables, then the loop body corresponds to a $\odot$-homomorphic function (for some $\odot$). If it terminates successfully and discovers new auxiliary accumulators, then the lifted loop body augmented with them corresponds to a $\odot$-homomorphic function (for some $\odot$).*

6.2 Completeness

Here, we formally discuss when Algorithm 1 can be expected to terminate and successfully generate a divide and conquer parallelization of the sequential loop.

Existence of Constant Homomorphic Liftings

First, independently of any algorithm, we deliberate on the question of existence of an efficient solution. A constant homomorphic lifting includes *constantly many constant size auxiliary variables* (see Section 5.2). Remember that lifting the loop to a homomorphism with a single linear-size auxiliary variable (Proposition 5.1) is always possible.

Theorem 6.1. *If f is leftwards and rightwards linear time, then there exists a binary operator $\odot$ and a $\odot$-homomorphic lifting of f that is a constant homomorphism.*

Theorem 6.1 reduces the existence of a constant homomorphic lifting to the fact that function f can be computed in *single-pass linear-time* both leftwards and rightwards on a sequence. Note that this condition is strictly stronger than a function being leftwards and rightwards computable. A function could be sing-pass linear-time computable in one direction, and yet more expensive to compute in the opposite direction on the sequence. In [38], the problem of language recognition of a string that is divided between two agents is discussed, and a complexity argument is provided that a restriction to a single-pass (in a certain direction) could increase the time complexity exponentially. Here, we provide a simple example to pass the intuition on to the reader.

Remark 6.3. *Let $h_n : \{0,1\}^* \to$ bool be defined as $h_n(w) = \mathsf{true}$ iff $w[n] = 1$ (i.e. the n-th letter). h is rightwards but not leftwards linear time computable in n.*

For a function like h, there does not exist a *constant* homomorphic lifting.

Proposition 6.4. *If a function h is not leftwards (rightwards) linear-time computable, then there exists no homomorphic lifting of h that is linear-time computable.*

This is a simple consequence of the *Third Homomorphism Theorem* [19]. A homomorphism naturally induces a leftwards and a rightwards function. Existence of a constant homomorphic lifting would imply that the function should be single-pass linear-time computable leftwards and rightwards which is a contradiction.

Corollary 6.5. *There exists a linear time loop for which there does not exist a linear time homomorphic lifting.*

This corollary is significant, because it implies that not every simple sequential loop (i.e. one that models a single-pass linear-time computable function) necessarily has a constant homomorphic lifting.

Completeness of the Algorithm

Since not every single-pass linearly computable function can be lifted into a constant homomorphism, we can conclude that there exists no algorithm that is complete for the entire class of these function. It remains to determine how 'complete' Algorithm 1 is, for those that can be lifted to constant homomorphisms (see Theorem 6.1). First, we characterize what such a constant homomorphic lifting looks like.

Theorem 6.2. *For a function f, if there exists a constant $\odot$-homomorphic lifting $f \propto g$, then there exists a $\circledast$-homomorphic lifting where*

$$(f(x), g(x)) \circledast (f(y), g(y)) = \\ (\mathit{exp}(f(x), f(y), g(y)), \mathit{exp}'(f(x), g(x), f(y), g(y)))$$

that is, the value of $f(x \bullet y)$ component of the join does not depend on $g(x)$.

Theorem 6.2 is very significant. It is the key in the completeness argument for Algorithm 1. The way the algorithm operates is that it tries to discover $\mathit{exp}(f(x), f(y), g(y))$ through the unfoldings of $f(x \bullet y)$, and does not have access to $g(x)$. To make any claims about completeness of Algorithm 1, one has to argue that it is sufficient to look at $f(x)$, $f(y)$, and $g(y)$ **and not** $g(x)$.

Theorem 6.3 (Completeness)**.** *If there exists a* constant homomorphic *lifting of the loop body E, then there is a finite set of algebraic rules $\mathcal{R}$, and a run of Algorithm 1 where the algorithm succeeds in discovering this lifting.*

Note that completeness is contingent on the availability of a sufficient set of algebraic rules and an effective search strategy that would lead to the correct candidate. Theorem 6.3 guarantees the existence of the rules and the reachability of the candidate through them. However, any efficient heuristic for this search may incur incompleteness by foregoing part of the search space in the interest of (fast) convergence, since the problem is known to be undecidable [12] under certain conditions for the set of algebraic rules.

6.3 Feedback to Join Synthesis

There is an important observation that Algorithm 1 includes information that can be helpful to join synthesis. When Algorithm 1 terminates successfully and discovers no new auxiliary accumulators, it certifies the loop as a homomorphic function (Proposition 6.2). If join synthesis has previously failed to synthesize a join for this loop, the conclusion is that syntactic restrictions for join must have been too strict. In this instance, the *normalized* expression used for the discovery of the accumulators contains hints about the shape of the join operator, and these hints can be used for re-instantiating join synthesis. These hints are structure information (i.e. a template for the expression tree) that can be extracted from exp($e_1, \ldots, e_m, e$), the result of '*normalize*'. This expression provides a recipe for how partial results (for bounded unfoldings) can be combined at join time, and therefore, can be used as a cheat sheet for devising a join sketch. In our benchmarks, join synthesis succeeded on all instances and this trick was never used.

7. Correctness of the Synthesized Programs

We use a SyGus solver that relies on bounded checks to synthesize join operators, and therefore, correctness of the synthesized join is not guaranteed for all input sequences. In this section, we discuss a heuristic scheme to generate proofs of correctness for the general case automatically. Automatic verification of code is known to be a hard problem. Full automation often comes at the cost of incompleteness. The method presented here is no exception to this rule. The claim is that based on a few key observations, the boundaries of automation can be extended to include many of the typical benchmarks that are in scope for this work.

We use an example to introduce our automated proof construction scheme. Let us assume that we want to verify the correctness of the join operator synthesized to parallelize the mts example. We use Dafny [28] program verifier to encode and check the proof of correctness.

> *A full functional correctness proof of the parallel program is not necessary, rather, it suffices to show that the join forms a homomorphism together with the (lifted) sequential code.*

The sequential code together with Definition 3.4 constitute the specification of correctness for the join operator. Note that, by piggybacking on the assumption that original sequential code is correct, only this *one* type of property (e.g. formation of a homomorphism) has to ever be proved for any given program. This conveniently limits the scope of automated verification. As mentioned in Section 3.3, we construct a functional variant of the loop body, which is used at every stage of our approach as its representation. This is very helpful for proof construction.

> *Proofs are constructed for program fragments in our intermediate functional form, and therefore, there is no need to handle loops or synthesize loop invariants.*

We use this intermediate functional form to model the loop body as a collection of functions in Dafny. The general rule is that every state variable in the loop body is modelled by a unique Dafny function. Figures 7(a) and 7(b) illustrate the Dafny functions corresponding to state variables `sum` and `mts` respectively. Proofs are constructed for these recursively defined functions, where more often than not, the correctness specifications for functions, specially for simpler ones, are *inductive*. This means that no further (human provided annotations) are required for the proof, which is in harsh contrast to imperative looped programs where loop invariants have to be provided even for the simplest of loops. It is important to note that instances do exist that a strengthening of the specification (in form of injection of known invariants) is required to make the specification inductive. However, we did not encounter any such example among our benchmarks.

The next step is to model the synthesized join. Each state variable has a distinct join function. In this example, we introduce a Dafny function to model the join for `Sum` as illustrated in Figure 7(d), and another one for `Mts` as

```
function Sum(s: seq<int>): int
{ if s == [] then 0 else
    Sum(s[..|s|-1]) + s[|s|-1] }
```
(a)

```
function Mts(s: seq<int>): int
{  if s == [] then 0 else
    Max(Mts(s[..|s|-1]) + s[|s|-1], 0)}
```
(b)

```
function Max(x: int, y: int): int
{ if x < y then y else x }
```
(c)

```
function SumJoin(sum_l: int, sum_r: int): int
{  sum_l + sum_r  }
```
(d)

```
function MtsJoin(mts_l: int, sum_l: int, mts_r: int, sum_r: int): int
{ Max(mts_r, mts_l + sum_r) }
```
(e)

```
lemma HomomorphismSum(s: seq<int>, t: seq<int>)
  ensures Sum(s + t) == SumJoin(Sum(s), Sum(t))
{
  if t == [] { assert(s + t == s); }
  else {
      calc {
        Sum(s + t);
        == {assert (s + t[..|t|-1]) + [t[|t|-1]] == s + t;}
        SumJoin(Sum(s), Sum(t)); }
  }
}
```
(f)

```
lemma HomorphismMts(s: seq<int>, t: seq<int>)
  ensures Mts(s + t) == MtsJoin(Mts(s), Sum(s), Mts(t), Sum(t))
{
  if t == []  { assert(s + [] == s); }
  else {
    calc { Mts(s + t);
           == {HomomorphismSum(s,t[..|t|-1]);
              assert (s + t[..|t|-1]) + [t[|t|-1]] == s + t;}
           MtsJoin(Mts(s), Sum(s), Mts(t), Sum(t)); }
  }
}
```
(g)

Figure 7: The encodings of *sum* (a) *mts* (b), *max* (c). The encoding of join in two parts for *sum* (d) and *mts* (e). The proofs that the joins form a homomorphism in two parts for *sum* (f) and *mts* (g). `s + t` denotes the concatenation of sequences `s` and `t` in Dafny, `[]` is the empty sequence while `|s|` stands for the length of the sequence `s`. The lemmas are proved by induction through the separation of the base case `t == []` and the induction step. The `calc` environment provides the hints necessary for the induction step proof.

illustrated in 7(e). Each function corresponds to an update (a line of code) synthesized for the (overall) join operator (as presented in Section 2).

> *The proof is modularly constructed for each state variable.*

There exists a natural decomposition for homomorphism proofs of tupled functions, that is, each component of the tuple can be shown to be a homomorphism independently. Therefore, the proof argument can then be modularly constructed for each state variable of the loop. The overall proof, therefore, consists of simpler (to construct) proofs of the join operator producing the correct result for each state variable. In our example, we use two Dafny lemmas accordingly. The lemma in Figure 7(f) states correctness of join for `Sum`, and the one in Figure 7(g) states the correctness of join for `Mts`.

Let us focus on `HomomorphismSum` lemma in Figure 7(f). This lemma states that `Sum` is `SumJoin`-homomorphic (as defined in Definition 3.4). Dafny cannot prove this lemma automatically, **if** the body of the lemma is left *empty*. It can prove the lemma, however, if it is given guidance about how to formulate the argument, in other words, if the body of the lemma (as illustrated in Figure 7(f)) is provided.

> *Each homomorphism proof is an induction argument over the length of the input sequence(s).*

This fact is true independent of the specific function under consideration and enables us to devise a generic (induction) proof template that can be instantiated for each function and its corresponding join operator. Let us inspect this guidance (i.e. the body of the lemma in Figure 7(f)) more closely. The body basically tells Dafny that the proof is an induction argument on the length of `t` (the second sequence argument), with the base case of an empty sequence (reflected in the `if` condition). In the base case, Dafny should be aware of the fact that an *empty* `t` simplifies the reasoning about `s + t` to reasoning about `s`. In the induction step (i.e. the `else` part), Dafny is guided to peel off the last element of `t` to get a smaller instance for induction.

This guidance is generic, i.e. not dependent on the nature of the function `Sum`, and is applicable to any other function. Consider the proof for the correctness of `MtsJoin` (Figure 7(g)). This proof is identical to that of `HomomorphismSum` lemma, with the minor difference of recalling of (the already proved) `MtsSum` lemma (bright pink text). Note that `Mts` calls `Sum` in its definition, and therefore the proof of correctness of the join operator for `Mts` has to assume the correctness of the join for `Sum` (which is exactly the `HomomorphismSum` lemma). This rule, namely, if value of u depends on value of v then recall the homomorphism lemma for v in the proof of homomorphism for u, is generically applied in all constructed proofs.

Our tool follows the simple rules illustrated by this example, and generates proofs like the one illustrate in Figure 7 automatically once the synthesis task is finished. It is important to note that if the proof checks, then correctness is guaranteed. If it fails, then it can mean that either problem-specific invariants were required, or the bounded synthesizer has synthesized a wrong solution. We did not have any instances of failure in our benchmarks. Our proposed solution in case of failure is to do a more extensive bounded check to discover counterexamples or acquire more coverage for testing. A backup plan like this is required due to the boundaries of applicability of automated proof generation techniques.

8. Experiments

Our parallelization technique is implemented in a prototype tool called PARSYNT, for which we report experimental results in this section.

8.1 Implementation

We use CIL [36] to parse C programs, do basic program analysis, and locate inner loops. The loop bodies are then converted into functional form, from which a sketch and

	sum	min	max	average	hamming	length	2nd-min	mps	mts	mss	mts-p	mps-p	poly	is-sorted	atoi	dropwhile	balanced-()	0*1*	count-1's	line-sight	0after1	max-block-1
Aux required?	no	no	no	no	no	no	no	yes	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	yes
Join Synt time?	1.6	2.0	1.7	22.9	1.5	1.6	6.1	3.1	3.9	29.4	82.2	77.1	115.2	7.0	84.4	3.7	19.8	1.4	3.0	6.6	7.6	7.5
#Aux required	–	–	–	–	–	–	–	–	1	2	1	–	–	1	1	1	1	2	1	1	1	1*

Table 1: Experimental results for performance of PARSYNTover all benchmarks. Times are in seconds. "–" indicates that no relevant data can be reported in this case. *: tool succeeds in finding 1 out 2 necessary auxiliaries. Auxiliary synthesis and proof generation/checking times negligible in all cases.Hardware: laptop with 8G RAM and Intel dual core m3-6Y30.

the correctness specification is generated. ROSETTE [47] is used as our backend solver, with a custom grammar for synthesizable expressions. In addition to the narrowing of the search space by using *left* and RIGHT holes (unknowns) as discussed in Section 4, we use type information to reduce the state space of the search for the solver. We also bound the size of the synthesizable expressions, especially when the loop body contains *non-linear operators*, which are difficult for solvers to handle. To sidestep this problem, we produce a more constrained join sketch with a smaller search space for when non-linear operators are involved.

8.2 Evaluation

Benchmarks. We collected a diverse set of benchmarks to evaluate the effectiveness of our approach. Table 1 includes a complete list. The benchmarks are all in C:

- min, max, length, sum, is-sorted and average are standard functions over lists. 2nd-min is the second smallest example discussed in Section 2.
- mps (maximum prefix sum), mts (maximum tail sum), and mss (maximum segment sum) are programming pearls from [34, 43]. mps-p and mts-p are the variations (from [43]) where in addition to the value of the corresponding sum, the position that defines it is also returned.
- poly computes the value of a polynomial at a given point when the coefficients are given in a list. atoi is the standard (string to integer) function from C.
- balanced-() checks if a string of brackets is balanced. count-1's counts the number of blocks (i.e. contiguous sequences) of 1's in a sequence of 0's and 1's. max-block-1 returns the length of the maximum block of (contiguous) 1's. 0after1 is a language recognizer that accepts strings in which a 0 has been seen after a 1. 0*1* is a regular expression filter. hamming computes the hamming distance between two strings.
- dropwhile (from Haskell) is a *filter* that removes from the beginning of a sequence all the elements that do not satisfy a given predicate. line-sight (from [34]), determines if a building is visible from a source in a line of other buildings of various heights.

Performance of PARSYNT Table 1 lists the times spent in join synthesis, which dominate the total time. The times for auxiliary accumulator synthesis are negligible (about 1-2ms on average). Note that if a benchmark is parallelizable (in original form), then no attempt to discover auxiliaries is made (indicated by "–" in the table). When auxiliary variables were required, Table 1 reports how many were discovered. With one exception (i.e. max-block-1), the auxiliary synthesis always succeeds. In this case, 1 of the 2 auxiliaries required is discovered, but the tool fails to discover the second one. Manual inspection of the failed procedure convinced us that the discovery of the second one would be possible if the rules in Figure 6 were enriched with more sophisticated rules involving conditional statements. Finally, the time to generate and check proofs is negligible compared to the join synthesis times, with most cases taking about or under 1s and the most expensive case (mss) taking about 7s.

Quality of the Synthesized Code We manually inspected all synthesized programs, and as programmers, we cannot produce a better version for any of them other than 0*1*. For this one, only one of the two auxiliary accumulators discovered was strictly necessary, and the other was redundant. There is, however, a negligible performance difference between this version and the optimized one. Here, we evaluate the performance of the synthesized programs. The quality of a parallel implementation, depends on many parameters beyond the algorithmic design (our target). We use Intel's Thread Building Blocks (TBB) [39] as the library to implement the divide-and-conquer parallel solutions that we synthesize. TBB is a popular runtime C++ library, that offers improved performance scalability by dynamically redistributing parallel tasks across available processors, and accommodates portability across different platforms. It supports divide-and-conquer parallelism, so transforming our solutions (from ROSETTE) into a TBB-based implementation became a simple mechanical task. Evaluation of the quality of the generated parallel code was done on a Proliant DL980 G7 with 8 eight-core Intel X6550 processors (64 cores total) and 256G of RAM running 64-bit Ubuntu.

Figure 8 illustrates the speedups of our parallel solutions over the input sequential programs. The size of the input arrays is about 2bn elements and the grain size is set at 50k. Note that for those benchmarks that auxiliary computation was required for parallelization, the parallel version is more expensive per iteration than the original sequential input. It is clear that the speedups are linear on the number of cores

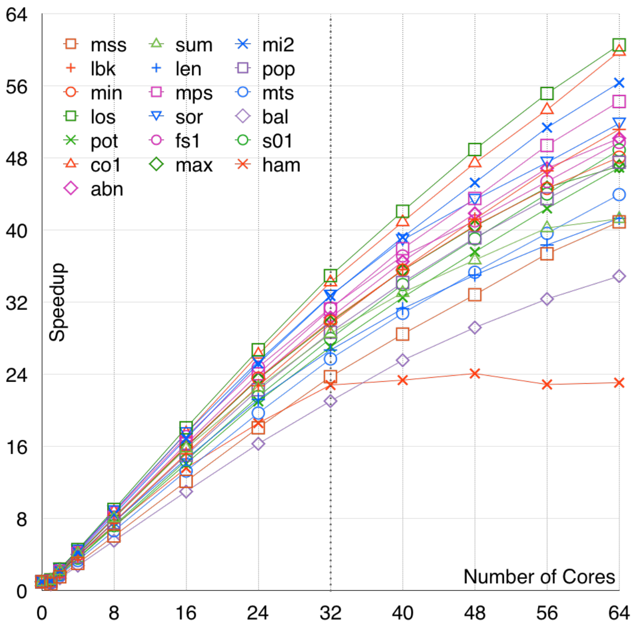


Figure 8: Speedups relative to the sequential implementation

up to around 32 cores. A study [14] of TBB's performance has shown that not scaling well above 32 cores is a known problem with TBB. It is due to TBB's scheduling overhead and not due to a design problem in our produced parallel programs. We separately measured the overhead of TBB, by limiting the number of cores to 1. The slowdown (over sequential) is on average negligible; the average slowdown is close to 1, with a standard deviation of 0.04.

9. Related Work

Homomorphisms and Parallelism Most closely related our work are the approaches that use homomorphisms for parallelization. There has been previous attempts in using the derivation of list homomorphisms for parallelization, such as methods based on the third homomorphism theorem [18, 20], those based on function composition [17], their less expressive/more practical variant based on matrix multiplication [43], methods based on the quantifier elimination [32] as well as those based on recurrence equations [8]. We will discuss the most closely related one here.

In [34], the *third homomorphism theorem* and the construction of the *weak right inverse* are used to derive parallel programs. The approach in [34] requires much more information from the programmer compared to our approach. The programmer needs to provide the leftwards **and** rightwards sequential implementations of a function to get the parallel one. This is often unreasonable, specially when the parallel version has a twist, since coming up with the leftwards implementation of a function could be as complex (and time consuming) as parallelizing it in the first place. Intuitively, by providing the reverse computation, the programmer is providing the information that we compute automatically here in Section 6. By contrast, we only need one (reference) sequential implementation as input. In Section 4.3, we make a more technical comparison against [34].

Later, the theoretical ideas of [34] were extended to trees in [33], and generalized for lists in [31]. But these papers do not provide a practical way of producing parallel code. In [13], a study of how functions may be extended to homomorphisms is presented, but no algorithm is provided.

Loop Parallelization
More recently in [42], symbolic execution is used to identify and break dependences in loops that are hard to parallelize. Since we produce correct parallel implementations, we do not have to incur the extra cost of symbolic execution at runtime. That said, the scope of applicability of [42] and our approach are not comparable. In the related area of distributed computation, there has been research on producing MapReduce programs automatically, for example through using specific rewrite rules [41] or synthesis [44]. The most relevant and recent work is GraSSP [16] which uses synthesis to parallelize a reference sequential implementation by analyzing data dependencies. To the best of our knowledge, since the (constant size) prefix information used in [16] is only a special case of our auxiliary accumulators, our approach subsumes theirs.

Synthesis and Concurrency Synthesis techniques have been leveraged for the parallel programs before. Instances include synthesis of distributed map/reduce programs from input/output examples [44], optimization and parallelization of stencils [24, 45], concurrent data structures synthesis [46], concurrency bug repair [49]. Other than use of synthesis, these problem areas and the solutions have very little in common with this paper.

Parallelizing Compilers and Runtime Environments
Automatic parallelization in compilers is a prolific field of research, with source-to-source compilers using highly sophisticated methods to parallelize generic code [50, 11, 5, 37] or more specialized nested loops with polyhedral optimization [6, 7, 48]. There is a body of work specific to reductions and parallel-prefix computations [10, 22, 27] that deal with dependencies that cannot be broken. Breaking static dependencies at runtime, Galois [40] being a good example of this category, is another type of approaching the auto-parallelization problem. Handling irregular reductions, when the operations in the loop body are not immediately associative, has been explored through employing techniques such as data replication or synchronization [23]. Unfortunately, there is not enough space to do the mountain of research on parallelizing compilers justice here. 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) in a space that includes many incorrect programs. This facilitates discovery of equivalent parallel implementations that are not reachable through generally correct program transformations.

References

[1] Alon, N., Matias, Y., and Szegedy, M. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing* (1996), STOC '96, pp. 20–29.

[2] Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M. M. K., Raghothaman, M., Saha, S., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. In *Dependable Software Systems Engineering*. 2015, pp. 1–25.

[3] Appel, A. W. Ssa is functional programming. *SIGPLAN Not. 33*, 4 (Apr. 1998), 17–20.

[4] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (2002), PODS '02, pp. 1–16.

[5] Bacon, D. F., Graham, S. L., and Sharp, O. J. Compiler transformations for high-performance computing. *ACM Comput. Surv. 26*, 4 (Dec. 1994), 345–420.

[6] Bastoul, C. Efficient code generation for automatic parallelization and optimization. In *Proceedings of the Second International Conference on Parallel and Distributed Computing* (2003), ISPDC'03, pp. 23–30.

[7] Bastoul, C. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (2004), PACT '04, pp. 7–16.

[8] Ben-Asher, Y., and Haber, G. Parallel solutions of simple indexed recurrence equations. *IEEE Trans. Parallel Distrib. Syst. 12*, 1 (Jan. 2001), 22–37.

[9] Bird, R. S. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design* (1987), pp. 5–42.

[10] Blelloch, G. E. Prefix sums and their applications.

[11] Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., and Tu, P. Parallel programming with polaris. *Computer 29*, 12 (Dec. 1996), 78–82.

[12] Boone, W. W. The word problem. *Proceedings of the National Academy of Sciences of the United States of America 44* (1958), 1061–1065.

[13] Chin, W.-N., Takano, A., and Hu, Z. Parallelization via context preservatio. In *Proceedings of the 1998 International Conference on Computer Languages* (1998), ICCL '98, pp. 153–.

[14] Contreras, G., and Martonosi, M. 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* (2008), pp. 57–66.

[15] Farzan, A., and Nicolet, V. Automated synthesis of divide and conquer parallelism.

[16] Fedyukovich, G., Maaz Bin Safeer, A., and Bodik, R. Gradual synthesis for static parallelization of single-pass array-processing programs. In *PLDI* (2017).

[17] Fisher, A. L., and Ghuloum, A. M. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (1994), PLDI '94, pp. 135–146.

[18] Geser, A., and Gorlatch, S. Parallelizing functional programs by generalization. In *Proceedings of the 6th International Joint Conference on Algebraic and Logic Programming* (1997), ALP '97-HOA '97, pp. 46–60.

[19] Gibbons, J. The third homomorphism theorem. *J. Funct. Program. 6*, 4 (1996), 657–665.

[20] Gorlatch, S. Systematic extraction and implementation of divide-and-conquer parallelism. In *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs* (1996), PLILP '96, pp. 274–288.

[21] Gorlatch, S. Extracting and implementing list homomorphisms in parallel program development. *Sci. Comput. Program. 33*, 1 (Jan. 1999), 1–27.

[22] Hillis, W. D., and Steele Jr, G. L. Data parallel algorithms. *Communications of the ACM 29*, 12 (1986), 1170–1183.

[23] Hwansoo, H., and Chau-Wen, T. A comparison of parallelization techniques for irregular reductions. In *Parallel and Distributed Processing Symposium., Proceedings 15th International* (2001), pp. 8 pp.–.

[24] Kamil, S., Cheung, A., Itzhaky, S., and Solar-Lezama, A. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2016), PLDI '16, pp. 711–726.

[25] Kejariwal, A., D&#39;Alberto, P., Nicolau, A., and Polychronopoulos, C. D. 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* (2005), LCPC'04, pp. 102–116.

[26] Kelsey, R. A. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (1995), IR '95, pp. 13–22.

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

[28] Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (2010), pp. 348–370.

[29] Marché, C. Normalized rewriting: an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation 21*, 3 (1996), 253–288.

[30] Marché, C., and Urbain, X. *Termination of associative-*

commutative rewriting by dependency pairs. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 241–255.

[31] MORIHATA, A. A short cut to parallelization theorems. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013* (2013), pp. 245–256.

[32] MORIHATA, A., AND MATSUZAKI, K. 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* (2010), pp. 321–336.

[33] MORIHATA, A., MATSUZAKI, K., HU, Z., AND TAKEICHI, M. The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009* (2009), pp. 177–185.

[34] MORITA, K., MORIHATA, A., MATSUZAKI, K., HU, Z., AND TAKEICHI, M. Automatic inversion generates divide-and-conquer parallel programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), PLDI '07, pp. 146–155.

[35] NARENDRAN, P., AND RUSINOWITCH, M. *Any ground associative-commutative theory has a finite canonical system*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991, pp. 423–434.

[36] NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*. 2002.

[37] PADUA, D. A., AND WOLFE, M. J. Advanced compiler optimizations for supercomputers. *Commun. ACM 29*, 12 (Dec. 1986), 1184–1201.

[38] PAPADIMITRIOU, C. H., AND SIPSER, M. Communication complexity. *J. Comput. Syst. Sci. 28*, 2 (1984), 260–269.

[39] PHEATT, C. Intel® threading building blocks. *Journal of Computing Sciences in Colleges 23*, 4 (2008), 298–298.

[40] PINGALI, K., NGUYEN, D., KULKARNI, M., BURTSCHER, M., HASSAAN, M. A., KALEEM, R., LEE, T.-H., LENHARTH, A., MANEVICH, R., MÉNDEZ-LOJO, M., PROUNTZOS, D., AND SUI, X. The tao of parallelism in algorithms. *SIGPLAN Not. 46*, 6 (June 2011), 12–25.

[41] RADOI, C., FINK, S. J., RABBAH, R., AND SRIDHARAN, M. Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (2014), OOPSLA '14, pp. 909–927.

[42] RAYCHEV, V., MUSUVATHI, M., AND MYTKOWICZ, T. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 153–167.

[43] SATO, S., AND IWASAKI, H. Automatic parallelization via matrix multiplication. *SIGPLAN Not. 46*, 6 (June 2011), 470–479.

[44] SMITH, C., AND ALBARGHOUTHI, A. Mapreduce program synthesis. *SIGPLAN Not. 51*, 6 (June 2016), 326–340.

[45] SOLAR-LEZAMA, A., ARNOLD, G., TANCAU, L., BODIK, R., SARASWAT, V., AND SESHIA, S. Sketching stencils. *SIGPLAN Not. 42*, 6 (June 2007), 167–178.

[46] SOLAR-LEZAMA, A., JONES, C. G., AND BODIK, R. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2008), PLDI '08, pp. 136–148.

[47] TORLAK, E., AND BODÍK, R. Growing solver-aided languages with rosette. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013* (2013), pp. 135–152.

[48] VASILACHE, N., BASTOUL, C., AND COHEN, A. Polyhedral code generation in the real world. In *Proceedings of the 15th International Conference on Compiler Construction* (2006), CC'06, pp. 185–201.

[49] ČERNÝ, P., HENZINGER, T. A., RADHAKRISHNA, A., RYZHYK, L., AND TARRACH, T. Regression-free synthesis for concurrency. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559* (2014), pp. 568–584.

[50] WILSON, R., FRENCH, R., WILSON, C., AMARASINGHE, S., ANDERSON, J., TJIANG, S., LIAO, S., TSENG, C., HALL, M., LAM, M., AND HENNESSY, J. The suif compiler system: A parallelizing and optimizing research compiler. Tech. rep., Stanford, CA, USA, 1994.

[51] ZUCK, L. D., PNUELI, A., FANG, Y., GOLDBERG, B., AND HU, Y. Translation and run-time validation of optimized code. *Electr. Notes Theor. Comput. Sci. 70*, 4 (2002), 179–200.

A. Conversion to a system of equations

We show that we can use a simple procedure to convert a loop body given in the input langauge described in 3 without loops into a system of equation. We proceed statement by statement, updating expressions on the right-hand side of an equation system and merging branches of conditionnals (φ functions in static single assignments). First, let us remark that we can convert all `if-then` statements to `if-then-else` statements by adding an empty else statement.

We start with a system where each varaible in SVar is assigned to itself : $E = \langle s_0 = s_0, \dots, s_n = s_n \rangle$, and update it by visiting each statement following the control flow graph edges (we do not have loops). We transform E into the new system E', depending on what statement we see:

- an assignment $s_i = \mathsf{Exp}(\mathsf{SVar}, \mathsf{IVar})$. The expression of s_i in E is updated by $\mathsf{Exp}(\mathsf{SVar}, \mathsf{IVar})[s_j \leftarrow exp_j \in E]$, E' is E with $s_i = \mathsf{Exp}$ once all the variables appearing in Exp have been replaced by their expression in E.
- a conditional on expression c : we apply the procedure recursively, each branch is converted to a system of equations E_{if} and E_{else}. The systems are merged into :
$$E_{merged} = \langle \dots s_i = (c \,?\, \mathsf{exp}_i^{if} \,:\, \mathsf{exp}_i^{else}), \dots \rangle$$
where exp_i^{if} is the expression of s_i in E_{if} and exp_i^{else} is the expression of s_i in E_{else}.
We update E to be:
$$E' = \langle \dots \mathsf{exp}_i^{merged}[s_j \leftarrow \mathsf{exp}_j \in E] \dots \rangle$$
with the susbtitution of variables in the expressions of the merged system by their expression in E.

And then we proceed to the next statement.

This procedure would yield systems with expressions of non optimal size, but this supports our claim that we can convert any input loop body into a system of equations.

B. Proof Sketch of Proposition 4.4

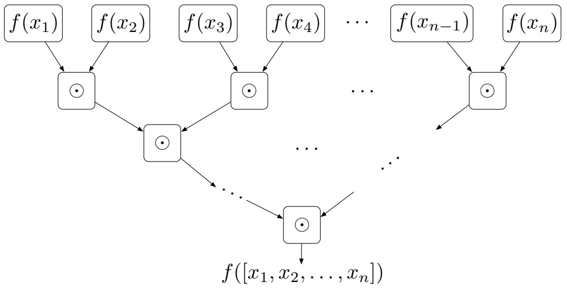
If the theorem of parallelization with the weak right inverse in [34] holds and the weak-right inverse returns a list of constant length, then there is a solution to the synthesis problem. The theorem states that the join $\odot$ for the loop body function f_E can be built using the following construct :

$$a \odot b = f_E(f_E^0\, a \bullet f_E^0\, b)$$

where f_E^0 is the weak-right inverse of function f_E. The recursive application of f_E on the two concatenated lists $f_E^0 a$ and $f_E^0 b$ will then yield a vector of expressions, in which all the occurences of the state variables in the expresssions Exp_i will contain an expression containing variables from inputs a and b, whereas the last input read contains only variables from the right input b. The solution given by this procedure is therefore a completion of the sketch for the ⟶ problem. (AZADEH SAYS: needs to be corrected)

C. Proof of Proposition 5.2

Consider the figure below that illustrates the parallel computation of a $\odot$-homomorphic function f' over the list $x = [x_1, \dots, x_n]$:



One can use the recurrence

$$J(n) = 2J(n/2) + t_j(n)$$

to compute the time complexity of performing the join operations, where n corresponds to the size of input handled by join, and $ct_j(n)$ captures the cost of performing one join of size n. The entire cost of computing $f'(x)$ in parallel (as illustrated in the diagram above) will then be obtained by adding to $J(n)$ the cost of computing $f'(x_i)$ for each x_i, which is constant for each $f'(x_i)$ and therefore linear overall. Therefore the cost of computing f' in parallel is in total:

$$T_p(n) = J(n) + cn$$

To have an efficient parallel implementation (with constantly many processors[4]), we need to our parallel asymptotic time complexity for f' not to be larger than our asymptotic sequential time complexity for f, that is $T_p(n) \in O(n)$.

The Master theorem then suggests that for $T_p(n) \in O(n)$ to hold, we have to have $t_j(n) \in o(n)$. That is, the cost of join should be sub-linear.

D. Proof of Theorem 6.1

A more detailed version of the theorem statement is:

If $f : \mathcal{S} \to Sc$ is both leftwards and rightwards linear time, then there exists a tuple of functions $\langle g_1, \dots, g_k \rangle$ for a constant k, where $g_i : \mathcal{S} \to Sc$ $(1 \le i \le k)$, and an operator $\odot$ such that $f \times g_1 \times \dots \times g_k$ is $\odot$-homomorphic.

By the third homomorphism theorem, we know that f is $\odot$-homomorphic for some $\odot$. All that remains to show is that there exists a constant-time $\odot$.

If f is rightwards linear time then

$$f(y \bullet a) = f(y) \otimes a$$

such that $\otimes$ is constant time computable. Similarly, if f is leftwards linear time then

$$f(a \bullet y) = a \oplus f(y)$$

[4] This is just to clarify that the discussion is avoiding common scenarios in the PRAM model that the number of processors available can be a function of the input size, such as $\log(n)$.

such that $\oplus$ is constant time computable.

We prove the claim by double induction on the sizes of the split sequences. Consider the base case $y = []$: if $f([])$ exists, then since $f(x) \odot f([]) = f(x)$, it is obviously constant time computable (there is nothing to be done if $f(x)$ is available). If $f([])$ does not exist, the $y = a$ is the base case and $f(x) \odot f([a]) = f(x) \otimes a$, and by the assumption above, it is constant-time computable.

The symmetric left argument proves the base cases $x = []$ and $x = a$ for x.

Let us assume $f(x) \odot f(y)$ is constant-time computable for $|x|, |y| < n$.

$$\begin{aligned} f(x \bullet (y \bullet a)) &= f(x) \odot f(y \bullet a) \\ &= f(x \bullet y) \otimes a \\ &= (f(x) \odot f(y)) \otimes a \end{aligned}$$

The last term is constant time computable by induction hypothesis and the assumptions. Therefore, $f(x) \odot f(y \bullet a)$ is constant time computable. So, we have just proven the induction step for y. Similarly:

$$\begin{aligned} f((a \bullet x) \bullet y) &= f(a \bullet x) \odot f(y) \\ &= a \oplus f(x \bullet y) \\ &= a \oplus (f(x) \odot f(y)) \end{aligned}$$

The last term is constant time computable by induction hypothesis and the assumptions. Therefore, $f(a \bullet x) \odot f(y)$ is constant time computable. And, now we have proven the induction step for x as well.

E. Proof of Theorem 6.2

Lemma E.1 (from [19]). *A function $h : \mathcal{S} \rightarrow Sc$ is a homomorphism iff for all sequences x,y, v, and w, we have*

$$h(x) == h(v) \ \wedge \ h(y) = h(w) \implies h(x \bullet y) = h(v \bullet w)$$

Assume a rightwards $f : \mathcal{S} \rightarrow Sc$ is not homomorphic, and is extended by a function g into a $\odot$-homomorphic function $f \propto g$.

According to the third homomorphism theorem [19], if a function is homomorphic, then it is both leftwards and rightwards. We will use the fact that f is rightwards, and that $h = f \propto g$ is both leftwards and rightwards in the argument below.

Assume that for four arbitrary sequences x, y, v and w, we have

$$f(x) = f(v) \wedge h(y) = h(w)$$

the latter being logically equivalent to $f(y) = f(w) \wedge g(y) = g(w)$. Then, we have:

$$\begin{aligned} &(f(x \bullet y), g(x \bullet y)) \\ &= h(x \bullet y) && \text{by definition} \\ &= h(x \bullet w) && \text{by Lemma E.1} \\ &= (f(x \bullet w), g(x \bullet w)) && \text{by definition} \\ &= (f(v \bullet w), g(x \bullet w)) && \text{by } f \text{ being rightwards} \end{aligned}$$

We have just argued for a variation of Lemma E.1:

$$\begin{aligned} f(x) = f(v) \ \wedge \ f(y) = f(w) \ \wedge \ g(y) = g(w) \\ \implies f(x \bullet y) = f(v \bullet w) \end{aligned}$$

which effectively says that the value of $f(x \bullet y)$ only depends on values of $f(x)$, $f(y)$, and $g(y)$; and, specifically not $g(x)$. Now, let us consider the join operator $\odot$ for $f \propto g$. Since the return values of f and g are considered to be scalars, we have:

$$\begin{aligned} &(f(x), g(x)) \odot (f(y), g(y)) = \\ &(\mathit{exp}(f(x), g(x), f(y), g(y)), \mathit{exp}'(f(x), g(x), f(y), g(y))) \end{aligned}$$

The argument above indicates that the dependency of *exp* on $g(x)$ must be non-existent or superficial. In other words, the join can be restructured in the following form:

$$\begin{aligned} &(f(x), g(x)) \circledast (f(y), g(y)) = \\ &(\mathit{exp}(f(x), f(y), g(y)), \mathit{exp}'(f(x), g(x), f(y), g(y))) \end{aligned}$$

which brings us to the proof of Theorem 6.2.

F. Proof of Theorem 6.3

Let f be a rightwards function (which can be a tuple of many functions) that is sequentially defined as

$$f(y \bullet a) = f(y) \otimes a$$

Let us assume that there is a function g (which can be a tuple of many functions) such that $f \propto g$ is $\odot$-homomorphic (constant) for some $\odot$. By Theorem 6.2, then we know that there exist $\circledast$, exp, and exp' such that:

$$\begin{aligned} &f \propto g(x \bullet y) = f \propto g(x) \circledast f \propto g(y) = \\ &(\mathsf{exp}(f(x), f(y), g(y)), \mathsf{exp}'(f(x), g(x), f(y), g(y))) \end{aligned}$$

The assumptions of the theorem about the existence of the algebraic rules implies that there is a sequence of rules that if applied step by step, they would transform the lefthand side of the above equation to the righthand side. Therefore, we can assume that the exist a run of *normalize* that does exactly that, and returns the righthand side.

Since $f \propto g$ is a homomorphism, by the third homomorphism theorem, it is rightwards. Therefore, there exists $\oplus$ such that:

$$f \propto g(y \bullet a) = f \propto g(y) \oplus a$$

this means that g can be computed sequentially along f. Note that without this argument, it would not be the case that g corresponds to an accumulator. Once we know it does, it is straightforward to see that if *normalize* returns the righthand side to the *solve*, then *solve* will discover the correct accumulator.

G. Proof of Remark 6.3

The rightwards function corresponds to the regular language $L_1 = \{w | \ w[n] = 1\}$. It is clear that a DFA for this language should just be able to count up to n using n states, and then check one letter.

The leftwards function corresponds to the regular language $L_2 = \{w | \ n\text{-}th \ bit \ from \ the \ end \ is \ 1\}$. This language cannot be as efficiently as L_1. The DFA for L_2 needs to record all the last n bits in its state, so that with every new input letter, it can update the window of size n, being prepared for the unknown end of the string to come at any time. Once the string is finished, then it has to look at the first letter in the n-letter window that it has memorized to see if it is a 0 or a 1.

To prove that one cannot do better than the above, a simple distinguishability argument will do. Let u denote the string consisting of the last n letters of the input string w. Consider now that the input string is extended by a sequence of strings $v_0, v_2, \ldots v_{n-1}$ where each v_i has length i. For each v_i, for the DFA to correctly accept/reject wv_i, it has to be able to distinguish the cases $u[i] = 0$ and $u[i] = 1$. So, if the DFA is in state q after reading letter $u[i-1]$, it has to go to two different states for the two different values of $u[i]$. Therefore, the DFA needs at least 2^n different states to distinguish all cases that potentially would lead to different acceptance/rejection results depending on the extensions of the string.