Gradual Synthesis for Static Parallelization of Single-Pass Array-Processing Programs

Grigory Fedyukovich Maaz Bin Safeer Ahmad Rastislav Bodík
University of Washington Paul G. Allen School of Computer Science & Engineering, USA
{grigory, maazsaf, bodik}@cs.washington.edu

Abstract

Parallelizing of software improves its effectiveness and productivity. To guarantee correctness, the parallel and serial versions of the same code must be formally verified to be equivalent. We present a novel approach, called GRASSP, that automatically synthesizes parallel single-pass array-processing programs by treating the given serial versions as specifications. Given arbitrary segmentation of the input array, GRASSP synthesizes a code to determine a new segmentation of the array that allows computing partial results for each segment and merging them. In contrast to other parallelizers, GRASSP *gradually* considers several parallelization scenarios and certifies the results using constrained Horn solving. For several classes of programs, we show that such parallelization can be performed efficiently. The C++ translations of the GRASSP solutions sped performance by up to 5X relative to serial code on an 8-thread machine and Hadoop translations by up to 10X on a 10-node Amazon EMR cluster.

CCS Concepts • **Theory of computation → Parallel algorithms**; **Verification by model checking**; **Automated reasoning**

Keywords Program Synthesis, Automatic Parallelization, Inductive Invariants, Constrained Horn Clauses

1. Introduction

Data parallelism is one of the most crucial requirements for modern software. Large amounts of data produced by industrial applications make existing serial programs both slow and inefficient. Due to nontrivial recurrent computations on the back-end of those programs, developers often prefer to completely rewrite serial code to enable efficient computation on a distributed programming platform, such as MapReduce [5], Dryad [14], Spark [34] or Hadoop [33].

In this paper, we consider a scenario, in which data is partitioned into a sequence of segments, each segment is processed separately, and the partial outputs for the segments are finally merged. Many tasks of such parallelism fall into category of the conventional MapReduce methodology, in which the order of the given data segments is not fixed. The others are more challenging and require the dependencies among segments to be taken into account. Developers attempt to overcome the particular task-specific parallelization requirements *gradually*, by considering the most simple and intuitive solutions first. If after an extensive stage of testing and verification the parallel program appears to be unsatisfactory, a more complicated solution is required. In this paper, we present an attempt to make this process more automated.

State-of-the-art synthesis tools use the Counter-Example-Guided Inductive Synthesis (CEGIS) [28] paradigm: they assume a space of candidate implementations and check whether there exists a candidate among them that matches a given specification. Typically, getting specifications in a machine-readable form is hard since they are usually written by and for humans. But for program parallelization, the situation improves. Serial program, assumed to be well-tested and trustworthy, itself can be treated as the specification, and the desired parallel program can be expected to comply with the serial one for any possible inputs.

We formulate our synthesis task as a search for a new parallel program that preserves equivalence with the serial one, i.e., the pairwise equivalence of inputs implies the pairwise equivalence of outputs. We propose a novel approach, Gradual Synthesis for Static Parallelization (GRASSP), that effectively performs the search. The key idea behind GRASSP is to mirror the gradual software development process and to consider potential solutions in stages, that is, exploiting the type of data dependencies that persist in the serial code. Its most distinguishing feature is the ability to break tangled dependencies by supporting the cases where data cannot be split into arbitrary segments.

PLDI'17, June 18–23, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062382

Driven by automated formal methods, GRASSP manages data approximations and performs automatic reasoning about data dependencies. Consequently, its results are general enough and can be adapted for use in many programming platforms, from cloud-computing to ordinary laptops. GRASSP handles arrays and higher-order operations over arrays. For the sake of efficiency, the synthesis is performed for a bounded number of both, array elements and segments. To ensure soundness for larger inputs, GRASSP provides SMT-based verification capabilities: the bounds could be increased, the results be validated, and (if needed) the parallel code be re-synthesized. Finally, we propose a way of inductive-invariant-based certification of the results using constrained Horn solving.

We implemented GRASSP in an SMT-based programming language, ROSETTE [30, 31], and evaluated it on a set of looping programs in C++. We demonstrate that synthesis with GRASSP is orders of magnitude faster than running the programs themselves for a realistic amount of data. The C++ translations of the GRASSP solutions showed up to a 5X speedup relative to serial code performance on an 8-thread machine, and the Hadoop translations up to a 10X on a 10-node Amazon EMR cluster.

Summary. This paper contributes a new approach, called GRASSP, for automatically parallelizing single-pass array-processing programs. GRASSP provides the following features:

- It carefully treats dependencies among data segments and supports different parallelization scenarios determined by the type of dependency in serial code.
- It employs SMT-based synthesis for array-handling functions that maintains array lengths lazily; it then certifies the solutions for arrays of any length.
- Its solutions are translatable to C++ multithreading code and Hadoop tasks and achieve up to a linear performance speedup relative to serial code.

The rest of the paper is structured as follows. We present the intuition behind GRASSP using examples (in Sect. 2 and Sect. 3). We then explain its architecture (Sect. 4) and the theoretical conepts (Sect. 5-7). Sect. 8 provides low-level details about the synthesis routine, and Sect. 9 describes our evaluation. Finally, we present the related work in Sect. 10 and conclusions in Sect. 11.

2. Motivating Example

Consider a function that calculates the number of matches of pattern $1(0)^*2$ in a string of elements from $\{0, 1, 2\}$. To ease comprehension, we show the corresponding Finite State Transducer (FST) in Fig. 1a and its C-encoding in Fig. 2. Note that the input is split among several files, each of which contains a separate *segment* to be processed in a specific order. The program has N loops to read each file line-by-

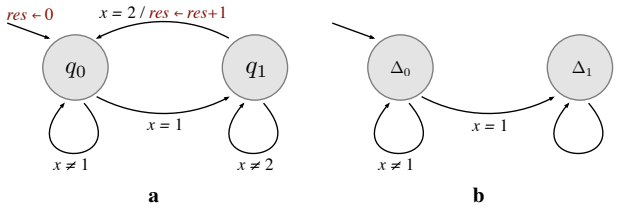

Figure 1: (a) Original FST and (b) Finite State Machine (Δ-FSM).

```
enum state { q_0, q_1 } ;
struct st_pair { state q; int res; };
void fst_transition (st_pair* st, int v) {
    if (v == 1 && st->q == q_0) { st->q = q_1; }
    else if (v == 2 && st->q == q_1)
            { st->q = q_0; st->res++; }
}
// serial application:
st_pair* st = new st_pair();
while (file_1 >> v) fst_transition (st, v);
...
while (file_N >> v) fst_transition (st, v);
return st->res;
```

Figure 2: C-function for the FST from Fig. 1a and serial processing of N files.

```
enum delta { d_0, d_1 };
struct dt_pair { delta d; bool found; };
bool search_bnd (int v) { return v == 2; }
void fsm_delta_transition (dt_pair* dt, int v) {
    if (search_bnd (v)) { dt->found = true; }
    else if (dt->delta == d_0 && v == 1)
                { dt->delta = d_1; }
}
// parallel application, for each file_i:
st_pair* st_i = new st_pair();
dt_pair* dt_i = new dt_pair();
while (file_i >> v) {
    if (!dt->found)
        { fsm_delta_transition (dt_i, v); }
    else { fst_transition (st_i, v); }
}
```

Figure 3: Synthesized C-functions for the boundary search, the Δ-FSM (Fig. 1b), and processing one of the N files in parallel.

```
int total_res = 0;
state q = q_0;
while (++i < N) {
    total_res += st_i->res;
    q = (q == q_0 &&
            dt_i->delta == d_0) ? q_0 : q_1;
    if (dt_i->found) {
        if (q == q_1) total_res++;
        q = st_i->state;
    }
}
return total_res;
```

Figure 4: C snippet for getting partial results from parallel processes, updating, and then merging them.

line and invoke function `fsm_transition`. This scenario facilitates the handling of large amounts of data that often make memory allocation infeasible.

The order of segments is crucial; for example, for the following input, the computed output is expected to be 3:

$$segment_1 = \{1,0,0,0\} \quad segment_2 = \{0,0,0,0\}$$
$$segment_3 = \{0,2,1,2\} \quad segment_4 = \{1,0,2,0\}$$

Notice that the number of matches of pattern $1(0)^*2$ is at most the number of occurrences of element "2"; therefore if a segment does not have "2" (like $segment_1$ and $segment_2$), then the counter `res` does not change its value. Additionally, if a segment does not have "1" (like $segment_2$), then the FST is looping, i.e., it does not change the value of `state`. This observation is exploited while parallelizing each loop from Fig. 2: functions `search_bnd` and `fsm_delta_transition` (both shown in Fig. 3) search for elements "2" and "1" respectively. The latter function encodes a Finite State Machine (or a Δ-FSM, shown in Fig. 1b) that behaves similarly to the original FST from Fig. 1a but recognizes only the subset $\{0,1\}$ of its language and does not perform counting. Thus, Δ-FSM takes as input only the *prefixes of segments* until the first element "2" is found, and the original FST takes as input only the remaining string.

For example, for $segment_1$, the prefix identified by `search_bnd` is $segment_1$ itself, and Δ-FSM takes it as input and terminates in state `d_1` (that is, at least one instance of "1" appeared in $segment_1$). For $segment_2$, Δ-FSM also takes the entire segment and terminates in `d_0` (that is, no instances of "1" in $segment_2$). For $segment_3$, the prefix is restricted only to its first element, and Δ-FSM terminates in `d_0`. The remaining elements of $segment_3$ are processed by the original FST, which sets the value of the counter to 1. Finally, for $segment_4$, the prefix contains all but one element, Δ-FSM terminates in `d_1`, and then the original FST sets the value of the counter to 0.

Note that during the parallel run, there are multiple Δ-FSMs, one for each processor. They operate in isolation and do not share knowledge about their states. However, once they all terminate, their outputs get merged, as shown in Fig. 4. That is, the final counter `total_res` gets the sum of the partial counters for $segment_3$ and $segment_4$, and then gets updated with respect to elements from all prefixes: `d_1` and `d_0`, respectively, for $segment_1$ and $segment_2$ followed by "2" in $segment_3$ reveal an instance of $1(0)^*2$ that spans three segments; `d_1` for $segment_4$ followed by "2" in $segment_4$ reveals the last instance of $1(0)^*2$. In general, parallel execution for four processors is shown in Fig. 6. Performance speedup can be linear: the parallel execution is up to four times faster than the serial execution, shown in Fig. 5.

3. GRASSP Overview

This section outlines how an arbitrary recurrent function f can be effectively parallelized using the functions synthesized by GRASSP. To simplify presentation, in this section, we assume that data is stored in an n-sized array. This lets us apply the higher-order function $fold$, a well-known mechanism taken from functional programming languages.

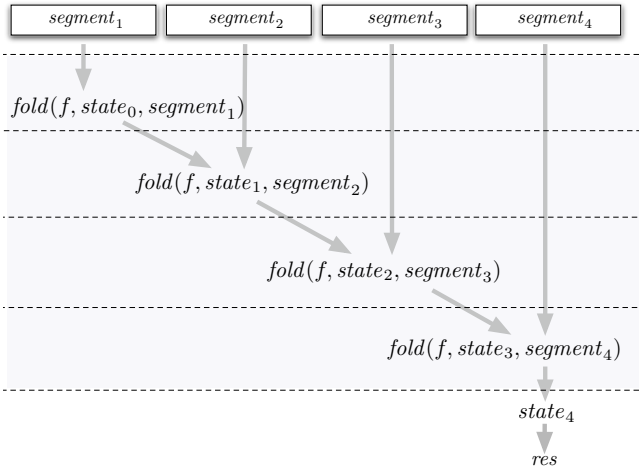

Figure 5: Executing a serial function.

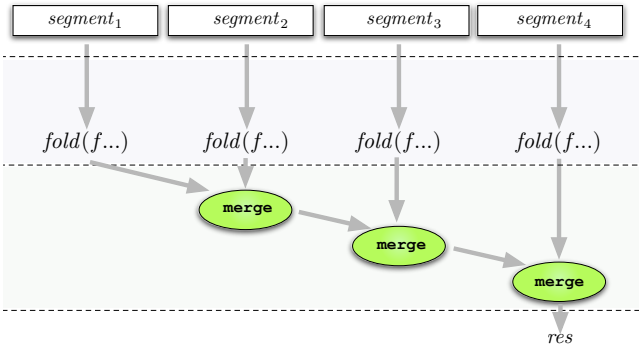

Figure 6: Best-case scenario – merging partial results without prefixes.

We focus our attention on functions that change the state iteratively for each element of a given array A. We denote the results of such iterations as $fold(f, state_0, A)$, where $state_0$ collects initial values of state variables of f. Throughout the paper, we write $fold(f\ldots)$ when the remaining arguments are clear from the context. We use primed notation for the "small-step" updates of $state_0$, i.e., after applying f exactly once. To avoid confusion, the "big-step" updates of $state_0$, i.e., after applying $fold(f\ldots)$, are denoted by the use of indexed notation. For example, the result of $fold(f, state_0, A)$ is mnemonically denoted $state_1$.

Serial computation (specification). In the rest of the paper, we consider a scenario in which the data array A is split into segments. A serial run of $fold(f\ldots)$ for four segments $\{segment_i\}$ is depicted in Fig. 5. Function $fold(f\ldots)$ takes each $segment_i$ as input and requests the result of $fold(f\ldots)$ for $segment_{i-1}$. That is, the computation starts with $state_0$, and it gets updated to $state_1$ for $segment_1$. Then, $state_2$ is computed by applying $fold(f\ldots)$ to $segment_2$; $state_3$ by applying $fold(f\ldots)$ to $segment_3$; and $state_4$ – by applying

$fold(f\ldots)$ to $segment_4$. The final output (denoted res_4) is extracted from $state_4$ and returned to the user.

The drawback of serial computation is its long processing time. Indeed, the application of $fold(f\ldots)$ to each $segment_i$ waits until $segment_{i-1}$ is processed, which in turn waits until $segment_{i-2}$ is processed, and so on. Thus, assuming the running time to process each data segment is estimated as $O(n/4)$, the running time for the entire process of obtaining res_4 using the serial computation is estimated as $O(n)$. Ideally, we wish to optimize the computation to behave as shown in Fig. 6: to break the data dependencies due to f. The next paragraphs describe several possible scenarios for such parallelization and the novel ways of attaining them using GRASSP.

Split-based computation. The motivation to reduce the waiting time while performing serial computation over the data segments is straightforward. GRASSP seeks to improve computational efficiency, and formally ensures that no precision is lost for the final output. Examples of parallel *split-based* functions synthesized by GRASSP are shown in Figs. 6, 7, and 8.

The key difference of the split-based against the serial computation is that the final output is composed from several partial outputs with the help of a function called $merge$. The main insight for getting partial outputs is breaking variable dependencies inside a state. Specifically, if there were a way to get a partial output for each $segment_i$ using a common input $state_0$, then the i-th processor would not wait until results from preceding processors were computed.

In the best-case scenario (Fig. 6), the running time for the parallel process of obtaining each partial output is estimated as $O(n/4)$, and the running time for merging the partial outputs is $O(3)$. This scenario occurs if the choice of segment boundaries does not affect the result of $merge$. For example, while calculating the number of elements in an array, it is sufficient to calculate the number of elements in each given segment of the array and then to calculate the sum of these partial outputs.

In a *worse*-case scenario (Fig. 7), partial results computed for the given segments are *incomplete*, which means they could not get merged as described above. It occurs, e.g., when checking if array elements are equal to each other. In this case, it is not sufficient to check that this property holds for each given segment; indeed, all elements of some $segment_i$ could be equal to 0, while all elements of remaining segments could be equal to 1. Therefore, extra processing of the partial results is needed. In fact, to complete each partial output, it is sufficient to pick one element of $segment_{i+1}$ and check whenever it also equals all elements of $segment_i$. In the diagram, this extra processing is performed by a function called *split* that for some pre-determined constant k identifies the first k elements of each $segment_i$, which is further referred to as a k-sized $prefix_i$. Once $fold(f, state_0, segment_i)$ is computed, the

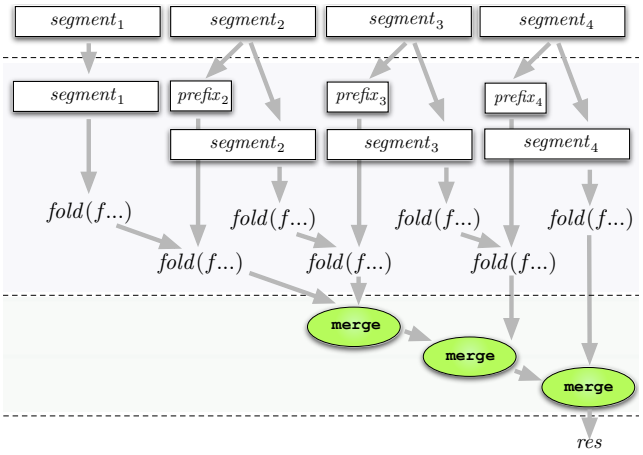

Figure 7: Worse-case scenario – merging results with constant prefixes.

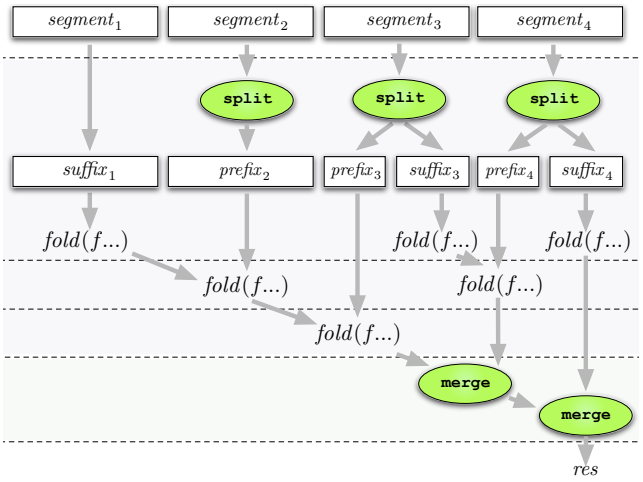

Figure 8: Worst-case scenario – merging results with conditional prefixes.

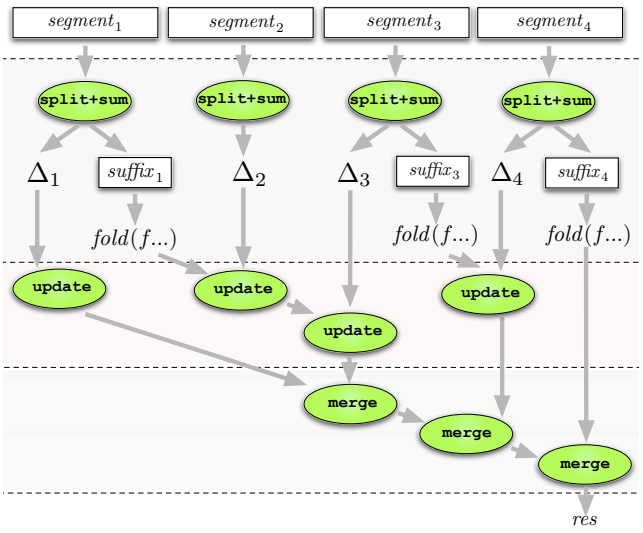

Figure 9: Worst-case scenario mitigated with summaries – updating partial results before merging.

resulting $state_i$ is taken as input by the (i+1)-st processor that iterates over $prefix_{i+1}$ and eventually completes $fold(f, state_i, prefix_{i+1})$. Finally, the *completed* outputs are taken as input by the $merge$ function that calculates the final output. The running time for the parallel process of obtaining each partial output is estimated as $O(n/4 + k)$, and the running time for merging the partial outputs is $O(3)$.

In the worst-case scenario (Fig. 8), constant k needed for a set of prefixes does not exist, and partial outputs computed for the given segments are incomplete. This occurs, e.g., when counting the number of matches of a regular expression of unknown length (e.g., $1(0)^*2$). As described in Sect. 2, the prefix can be expressed via a condition over the array elements: the first element equal to "2" in $segment_i$ witnesses the boundary between $prefix_i$ and $suffix_i$. A naive way of completing the computation of each partial output is to keep running $fold(f \ldots)$ for the prefix of a subsequent segment. However, the running time for this approach would depend on the existence of boundaries for each segment; but even in the most unlucky situation, the running time still would not exceed the time for the serial computation $O(n)$.

Split+Sum+Update-based computation. The split-based computation of f is optimal only if the conditional prefix is empty for all segments. Evidently, this is not the case for our example in Fig. 8, in which $prefix_2$ is equivalent to $segment_2$ itself; consequently, the entire computation over $segment_2$ depends on results of the computation over $suffix_1$. GRASSP counters such inefficiency by: (a) capturing the most relevant information about all elements seen during the explicit search for the segment boundary (with a function called sum), and (b) using this summary to instantaneously update the preceding state (with a function called $update$).

A diagram of *split+sum+update-based* computation corresponding to the example in Sect. 2 is shown in Fig. 9. There are two main differences from split-based computation: (a) $fold(f \ldots)$ is applied only to each $suffix_i$ and to the initial state, and (b) each $state_i$ (except the last one) is updated with respect to summary Δ_{i+1}, providing the same result as could be found after processing $prefix_{i+1}$. By construction, each Δ-update is a single-step procedure (as opposed to an iterative procedure of traversing the consequent $prefix$). In the context of $1(0)^*2$-counting, each Δ_{i+1} reflects the appearance of "1" in the given prefix; and each call $update(state_i, \Delta_{i+1})$ matches Δ_{i+1} with the fact that there is (or there is no) "2" after the last appearance of "1" in $suffix_i$. The entire running time for *split+sum+update-based* computation including the merging of partial outputs, is estimated as $O(n/4 + 3)$.

4. Gradual Synthesis

GRASSP seeks to gradually find the simplest way (among the ones listed in Sect. 3) to parallelize computation for a given function f and all possible segmentations of input

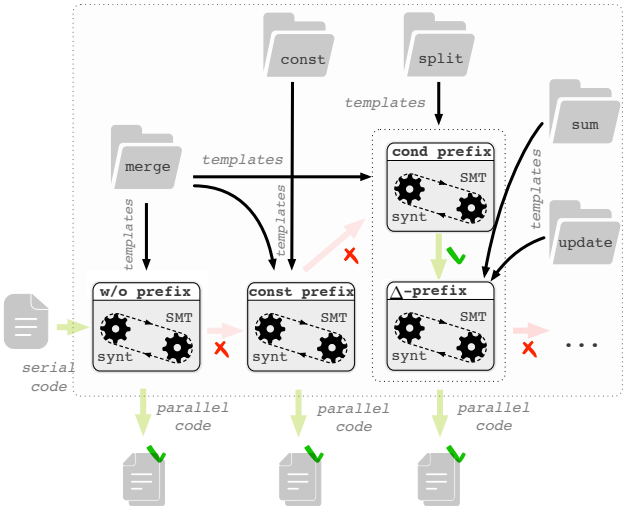

Figure 10: GRASSP architecture.

data. The search is gradual in the purest sense: GRASSP checks whether the best-case (Fig. 6) or worse-case (Figs. 7,9) scenarios hold, and jumps between stages only on demand. The increasing complexity of synthesis in different stages of GRASSP is the key to saving time when parallelizing and to delivering as simple a solution as possible.

The GRASSP architecture is depicted in Fig. 10. Internally, GRASSP models input data as arrays and treats them nondeterministically, i.e., it allows them to contain only symbolic elements. Thus, when parallelizing a given function, the algorithm considers all possible resolutions of nondeterminism for the symbolic elements of the array, and, if a solution is found, it is guaranteed to be general enough to satisfy all arrays containing numeric elements. To ensure the finiteness of the search space of the solutions, the input arrays are bounded by a user-defined number of elements. The bounds can, in fact, be maintained lazily: a round of synthesis for a chosen bound can be followed by a round of verification for a bigger bound and then (if needed) by yet another round of synthesis. Finally, to ensure correctness of the synthesized solutions in an unbounded case, GRASSP relies on a solver of satisfiability of systems of constrained Horn clauses (CHC). That is, each system of CHCs encodes the equivalence condition between the serial and parallel code for arrays of any size (see Sect. 8.2). Satisfiability of such system (i.e., existence of the inductive invariant) entails the soundness of the parallelization.

GRASSP is parameterized by libraries of templates (i.e., partial programs with holes) for the functions $split$, $merge$, sum, and upd, required in different stages of the approach. The libraries contain predefined templates that help synthesizing single-pass array-processing functions (see Sect. 9) and can also be populated with new, user-defined templates to enlarge the search space.

Three main stages allow GRASSP to gradually synthesize parallel implementation of a given serial function. For

all stages, the serial code is treated as specification, and the synthesized parallel code is asked to fulfil this specification, so that both serial and parallel code produce equivalent results. Each synthesis stage is performed on a subset of templates and at a low level relies on an off-the-shelf SMT solver. Whenever a synthesis stage succeeds, the parallel code is delivered to the user; otherwise, GRASSP proceeds to the next synthesis stage. Thus, GRASSP's modular architecture allows the addition of more synthesis stages that in the future could support other types of parallelization.

The first stage is built on the hypothesis that a given function can be parallelized without segment prefixes. Thus, it requires templates for the $merge$ function only and checks for instantiations of holes for one of those templates to witness the hypothesis. The entire synthesis problem for this stage is described in detail in Sect. 6.1.

The second stage is built on the hypothesis that a given function can be parallelized with constant prefixes. It also requires templates for the $merge$ function, as well as a finite set of constants (to constitute the lengths of prefixes which are the same for each data segment). The entire synthesis problem for this stage is described in detail in Sect. 6.2.

The third stage is the most challenging. It consists of two sub-stages. The first is built on the hypothesis that a given function can be parallelized with conditional prefixes and requires the use of templates for the $merge$ and $split$ functions. Synthesized implementations of these functions raise the issue of how to optimize the use of prefixes via summaries and updates. Thus, the required templates for this stage are the sum and upd functions. We elaborate on the synthesis problems for this stage in Sect. 6.3 and further in Sect. 7.

5. Notation and Specification for Synthesis

Before describing the synthesis problems outlined in Sect. 4, we formally introduce the functional notation for single-pass array-processing programs, namely, those that take a single finite-sized array as input and recurrently compute a single output. In the following three sections, we use terms "function" and "program" interchangeably.

Let D, In, and Out be any types, element $d : D$ is called *state*; elements $in : In$ and $out : Out$ are called *input* and *output* respectively. This paper considers function f, which updates a state for a given input, and function h, which converts a state to an output:

$$f : D \times In \to D \qquad h : D \to Out$$

An n-sized *array* is a finite sequence of inputs, $A : In^n$. Function f can be iteratively applied to the elements of A by means of the higher-order function $fold$:

$$fold : (D \times In \to D) \times D \times In^n \to D$$

For example, let $A = (in_1, \ldots, in_n)$ be an array of n elements; in the first iteration, f would be applied to element in_1 and initial state d_0. The updated state would then be updated by f again with respect to element in_2. The result of n iterative applications of f to d_0 is called a *final* state, represented as the following first-order *recurrent relation*:

$$fold(f, d_0, A) = f(in_n, f(in_{n-1}, \ldots, f(in_2, f(in_1, d_0))))$$

Conventionally, out must be computed only for the final state:

$$out \stackrel{\text{def}}{=} h(fold(f, d_0, A))$$

Throughout the paper, we rely on an operator that concatenates m arrays:

$$append : In^{n_1} \times \ldots \times In^{n_m} \to In^{n_1+\ldots+n_m}$$

It is important to ensure the following functional property of $append$:

$$fold(f, d_0, append(A_1, \ldots A_m)) = \quad (1)$$
$$fold(f, fold(f, \ldots fold(f, d_0, A_1), \ldots, A_{m-1}), A_m)$$

The left side of (1) denotes initial state d_0 iteratively updated by f with respect to all elements of $append(A_1, \ldots A_m)$. The right side of (1) consists of m groups of consequent applications $fold(f \ldots)$ to each of the arrays $\{A_i\}$. That is, the final state obtained for an i-th $fold(f \ldots)$ is further used for the $(i+1)$-th $fold(f \ldots)$. We refer to this property as *sequential recurrence decomposition* since it guarantees equivalence between a single application of $fold$ to $append(A_1, \ldots A_m)$ and m recurrent applications $fold(f \ldots)$.

The equivalence of final states entails equivalence of outputs computed from these states.

6. Synthesizing Functions $merge$ and $split$

Parallel application of $fold(f \ldots)$ to each of the arrays in $\{A_i\}$ requires decomposition of the recurrent relation. We assume that each application of $fold(f \ldots)$ to A_i takes the same initial state d_0 and refer to the corresponding final states $\{d_i\}$ as the *partial* states.

$$\forall i \cdot d_i \stackrel{\text{def}}{=} fold(f, d_0, A_i)$$

The question is how to merge partial states, so that the result is equivalent to the output of sequential computation:

$$merge : D^m \to Out$$

In the rest of the section, we establish the property of *parallel* (as opposed to sequential) *recurrence decomposition* that binds together all ingredients of the parallel processing of m arrays. Interestingly, there are several possible ways to define this property, depending on the existence of $merge$ for each particular f and h. We consider three such cases.

6.1 Best-Case Scenario (No Need for Prefixes)

The first case assumes the existence of a function $merge$ that is *directly* applicable to all partial states obtained after applications of $fold(f \ldots)$ to each A_i.

Synthesis problem. Given functions f and h, and initial state d_0, we wish to find a function $merge$, such that for any possible sequence of input arrays, the output of $merge$ is equivalent to the output of sequential computation:

$$\exists merge, \forall\, A_1, \ldots A_m \cdot \quad (2)$$
$$h(fold(f, d_0, append(A_1, \ldots A_m))) = merge(d_1 \ldots, d_m)$$

6.2 Worse-Case Scenario (Need for Constant Prefixes)

We say that two arrays A' and A'' are respectively *prefix* and *suffix* of A if $A = append(A', A'')$. We allow the prefix to be an empty array. We denote by $prefix_\ell(A)$ the prefix of A of the predetermined length ℓ. We call this the *constant prefix*.

When no $merge$ function meeting (2) exists, then computations $fold(f, d, A_i)$ and $fold(f, d, A_{i+1})$ depend on each other and cannot be *correctly* performed from the same initial state. However, incorrect executions could be repaired by recomputing the affected prefix. The scheme of this subsection performs such a repair on a constant-size prefix, if one exists. First, computations $fold(f, d, A_i)$ are performed in parallel from the initial state $d = d_0$, yielding a partial state d_i. Subsequently, the scheme reruns the computations on a constant-size prefix of A_{i+1}, starting from d_i. State $d_i^{\ell-repaired}$ obtained after processing the prefix, is then supplied to a suitable function $merge$ instead of d_i:

$$\forall i \cdot d_i^{\ell-repaired} \stackrel{\text{def}}{=} fold(f, d_i, prefix_\ell(A_{i+1}))$$

Note that computing each $d_i^{\ell-repaired}$ requires processing $prefix_\ell(A_{i+1})$ twice, with the second processing serialized after d_i has been computed. The inefficiency is mitigated by the observation that the prefixes can be processed in parallel, so the critical path of the computation grows only by the processing of the constant prefix.

Synthesis problem. Given functions f and h, and initial state d_0, we wish to find a function $merge$ and a constant ℓ, such that for any possible sequence of input arrays, the output of $merge$ applied to the ℓ-repaired partial states is equivalent to the output of sequential computation:

$$\exists\, \ell, merge, \forall\, A_1, \ldots A_m \cdot \quad (3)$$
$$h(fold(f,d_0, append(A_1, \ldots A_m))) =$$
$$merge(d_1^{\ell-repaired}, \ldots, d_{m-1}^{\ell-repaired}, d_m)$$

Note that the partial state d_m produced for the last array A_m is always repaired since there is no subsequent array A_{m+1}.

6.3 Worst-Case Scenario (Need for Conditional Prefixes)

When there is no function $merge$ and constant ℓ that satisfy (2) or (3), we wish to find another way to shift array boundaries. Similar to the scheme presented in Sect. 6.2, the solution would rerun the computations on the affected prefixes. However, the length of a prefix of some A might vary from case to case. The problem of identifying such *conditional* prefixes can be reduced to: (a) searching for a predicate $prefix_{cond}$ over the array elements, and then (b) iteratively evaluating $prefix_{cond}$ for each element of the array:

$$prefix_{cond} : In \to Bool$$

Formally, the procedure to get the conditional prefix is iterative and can be formulated as the application of the higher-order function $fold$. Let $\langle found, pos\rangle : Bool \times Int$ be a tuple containing a boolean flag $found$ and a position number pos. The entire length of a conditional prefix of A is calculated by applying function bnd_{cond} to $\langle false, 0\rangle$ and to all elements of A:

$$bnd_{cond}(\langle found, pos\rangle, el) \stackrel{\text{def}}{=}$$
$$ite((prefix_{cond}(el) \wedge \neg found), \langle true, pos\rangle,$$
$$ite(\neg found, \langle false, pos + 1\rangle, \langle found, pos\rangle))$$

$$\forall i \cdot length_{prefix_i} \stackrel{\text{def}}{=} fold(bnd_{cond}, \langle false, 0\rangle, A_i)$$

In other words, for each i, the value of $length_{prefix_i}$ is the position number k of some element in A such that $prefix_{cond}$ evaluates to $true$ for the k-th element, and $prefix_{cond}$ evaluates to $false$ for each j-th element in A so that $j < k$.

Predicate $prefix_{cond}$ gives rise to a function $cond_{split}$ that splits each array into two parts:

$$cond_{split} : (In \to Bool) \times In^n \to In^* \times In^*$$

where the sum of sizes of two arrays in the image equals n. Thus, for $A_1 \ldots A_m$ we have m pairs of prefixes and suffixes:

$$\forall i \cdot prefix_i \stackrel{\text{def}}{=} first(cond_{split}(prefix_{cond},\, A_i))$$
$$\forall i \cdot suffix_i \stackrel{\text{def}}{=} second(cond_{split}(prefix_{cond},\, A_i))$$
$$\forall i \cdot A_i = append(prefix_i, suffix_i)$$

Note that for some A_i, $prefix_{cond}$ could evaluate to $false$ for each element of A_i. In other words, $prefix_i = A_i$, and $suffix_i$ is empty. This case is crucial for split-based parallelization since the repair of $fold(f, d_{i-1}, prefix_i)$ would require a subsequent repair with respect to $prefix_{i+1}$, and the partial state d_i would not simply be computed and used at all (recall $prefix_2$ in Fig. 8). In the rest of this subsection, we formally describe this case.

Let us fix a sequence $Ind_{prefix_{cond}} = \{p_1, \ldots, p_s\} \subseteq \{1, \ldots, m\}$ that contains indexes of arrays $A_1, \ldots, A_m$ for

which the suffixes are not empty. It is easy to see that the computation over each $suffix_{p_i}$ should be sequentially repaired by $p_{i+1} - p_i$ prefixes:

$$\forall p_i \cdot d_{p_i}^{repaired} \stackrel{\text{def}}{=} fold(f, d_0, \\ append(suffix_{p_i}, prefix_{p_i+1}, \ldots, prefix_{p_{i+1}}))$$

Finally, the computation over the first array should also be repaired as follows:

$$d_0^{repaired} \stackrel{\text{def}}{=} fold(f, d_0, \\ append(prefix_1, \ldots, prefix_{p_1-1}, prefix_{p_1}))$$

Intuitively, this definition practically finds a new segmentation of the input arrays $\{A_i\}$ based on $prefix_{cond}$ that allows merging the local states. Each $d_{p_i}^{repaired}$ is the result of computing f on such a segment.

We are now ready to formulate the synthesis problem for the "*worst-case*" parallelization scenario. The main difference with the previous "*worse-case*" scenario is that it considers sequences (of various length) of repairs, and the total number of repaired states could be less than the number of the input arrays (due to multiple possible empty prefixes).

Synthesis problem. Given functions f and h, and initial state d_0, we wish to find functions $merge$ and $prefix_{cond}$, such that for any possible sequence of input arrays, the output of $merge$ applied to repaired partial states is equivalent to the output of sequential computation:

$$\exists\, prefix_{cond}, merge, \forall\, A_1, \ldots A_m \cdot \\ h(fold(f, d_0, append(A_1, \ldots A_m))) = \\ merge(d_0^{repaired}, d_{p_1}^{repaired}, \ldots, d_{p_s}^{repaired}) \tag{4}$$

7. Synthesizing Functions sum and upd

The parallelization scenarios so far considered the original function f for both, main computation and the prefix-based repair. Clearly, an iterative repair of each $prefix_i$ may lead to a further inefficiency since the elements of A_i are processed twice: for calculating the prefix and for completing the preceding state, as pointed out in (4). To avoid such double processing over array elements, function bnd_{cond} could be augmented by summarization capabilities that would memorize the appearances of the elements from the prefix being constructed. Once computed, such summaries would replace the applications $fold(f \ldots)$ by a code that produces the same effect in one step. Another important requirement for such summaries is that they should be applicable also for the case where for some i, $length_{prefix_i}$ equals the length of A_i.

Thus, we are looking for an alternative "small-step" repair of computations over the non-empty suffixes specified in $Ind_{prefix_{cond}}$:

$$\forall p_i \cdot d_{p_i} \stackrel{\text{def}}{=} fold(f, d_0, suffix_{p_i})$$

Let D' be a type, which is expressive enough to capture the effect of updates to all possible suffixes with respect to all possible prefixes. That is, instead of performing a sequence of small-step updates to each d_{p_i} with respect to $prefix_{p_i+1}, \ldots, prefix_{p_{i+1}}$ using $fold(f \ldots)$, we wish to find functions sum and upd and perform just big-step updates for $\Delta_{p_i+1}, \ldots, \Delta_{p_{i+1}} \in D'$ created by the iterative application of sum for all elements of the corresponding prefixes (recall, e.g., repairing $suffix_1$ by Δ_2 and Δ_3 in Fig. 9):

$$sum : D' \times In \to D' \qquad upd : D \times D' \to D$$

The idea is to compute each $\Delta_j \in D'$ by iterative applications of sum to all first elements of A_j until the first element for which $prefix_{cond}$ evaluates to $true$ is found. Note that this procedure can be naturally embedded into the procedure for computing the corresponding $prefix_j$:

$$\Delta_{cond}(\langle found, \Delta\rangle, el) \stackrel{\text{def}}{=} \\ ite((prefix_{cond}(el) \wedge \neg found), \langle true, \Delta\rangle, \\ ite(\neg found, \langle false, sum(\Delta, el)\rangle, \langle found, \Delta\rangle)))$$

$$\forall j \cdot \Delta_j \stackrel{\text{def}}{=} fold(\Delta_{cond}, \langle false, \Delta_{init}\rangle, A_j)$$

Finally, the result of sequential updates of d_{p_i} with respect to $\Delta_{p_i+1} \ldots \Delta_{p_{i+1}}$ could be computed as follows:

$$\forall p_i \cdot d_{p_i}^{\Delta} \stackrel{\text{def}}{=} upd(\Delta_{p_{i+1}}, (upd \ldots (upd(\Delta_{p_i+1}, d_{p_i}))))$$

Synthesis problem. Given functions f and h, and initial state d_0, for which functions $prefix_{cond}$ and $merge$ exist and (4) is satisfied, we wish to find functions sum and upd, such that for any possible sequence of input arrays, the output of $merge$ applied to "updated+summarized" partial states is equivalent to the output of sequential computation:

$$\exists\, sum, upd, \forall\, A_1, \ldots A_m \cdot \\ h(fold(f, d_0, append(A_1, \ldots A_m))) = \\ merge(d_0^{\Delta}, d_{p_1}^{\Delta}, \ldots, d_{p_s}^{\Delta}) \tag{5}$$

Alternatively, functions sum and upd can be synthesized by exploiting the already satisfied condition (4):

$$\exists\, sum, upd \cdot \forall\, A_1, \ldots A_m, \forall p_i \cdot d_{p_i}^{\Delta} = d_{p_i}^{repaired}$$

8. Core Machinery behind GRASSP

GRASSP uses the well-established approach to program synthesis that searches for a solution within a space of predetermined candidates. Since a direct reasoning on the level of formulas (2), (3), (4), and (5) is hard, GRASSP fixes the number of arrays and the length of each array and searches for an "approximate" solution in such a bounded setting.

Our synthesis procedure is reduced to a sequence of equivalence checks between each candidate and the serial

$$
\begin{cases}
inv(s\_id, r, r_1 \dots r_m) \leftarrow s\_id = 1, r = init, r_1 = init, \dots, r_m = init \\
inv(s\_id', r', r'_1 \dots r'_m) \leftarrow inv(s\_id, r, r_1 \dots r_m), (s\_id' = s\_id \vee s\_id' = s\_id + 1), r' = f(nondet, r), \\
\qquad\qquad s\_id = 1 \implies r'_1 = f'(nondet, r_1) \wedge r'_2 = r_2 \wedge \dots \wedge r'_m = r_m, \\
\qquad\qquad \dots \\
\qquad\qquad s\_id = m \implies r'_1 = r_1 \wedge \dots \wedge r'_{m-1} = r_{m-1} \wedge r'_m = f'(nondet, r_m), \\
\bot \leftarrow inv(s\_id, r, r_1 \dots r_m), s\_id \leq m, h(r) \neq merge(r_1 \dots r_m)
\end{cases}
$$

Figure 11: Encoding PA to CHCs.

$$
\begin{cases}
cnt(s\_id, r, r_1 \dots r_m) \leftarrow s\_id = 1, r = 0, r_1 = 0, \dots, r_m = 0 \\
cnt(s\_id', r', r'_1 \dots r'_m) \leftarrow cnt(s\_id, r, r_1 \dots r_m), (s\_id' = s\_id \vee s\_id' = s\_id + 1), r' = r + 1, \\
\qquad\qquad s\_id = 1 \implies r'_1 = r_1 + 1 \wedge r'_2 = r_2 \wedge \dots \wedge r'_m = r_m, \\
\qquad\qquad \dots \\
\qquad\qquad s\_id = m \implies r'_1 = r_1 \wedge \dots \wedge r'_{m-1} = r_{m-1} \wedge r'_m = r_m + 1, \\
\bot \leftarrow cnt(s\_id, r, r_1 \dots r_m), s\_id \leq m, r \neq r_1 + \dots + r_m
\end{cases}
$$

Figure 12: Example of encoding to CHCs ("*counting elements*").

program (i.e., the specification). Both program are symbolically executed, which yields two quantifier-free first order formulas. Then these two formulas are conjoined with the pairwise equivalence of the input variables and the negation of the equivalence of the output variables. Unsatisfiability of the resulting formula means the candidate is sufficient for the chosen bound, and satisfiability means the search should continue for the next candidate.

In the lower level, GRASSP implements an instance of the Counter-Example-Guided Inductive Synthesis (CEGIS) [28] paradigm, in which the satisfiability checks are delegated to an SMT solver. The key insight is to use satisfying assignments produced for each formula (i.e., counter-examples showing that a candidate does not meet the specification) and pruning the remaining search space based on them. Since the technique is standard, we refer the reader to [30, 31] for more details.

8.1 Applying (Un)-Bounded Verification for Synthesis

Approximate solutions for the synthesis problems are easy to find if the pre-defined bounds are sufficiently small. For bigger bounds, each solution should be re-verified and, if the verification failed, re-synthesized from scratch. Ideally, a solution needs to be verified without respect to any bound to guarantee the complete equivalence between the serial and the parallel programs.

In an unbounded setting, a verification condition is encoded into a system of constraints that involves uninterpreted predicates. Consequently, verification results rely on a decision procedure that can find an interpretation for the predicates, which represents a solution for the corresponding system. In first-order logic, this can be done by checking satisfiability of constrained Horn clauses (CHCs).

Let us fix a vocabulary of interpreted symbols that consists of sets $\mathcal{F}$ and $\mathcal{P}$, fixed-arity function and predicate symbols, respectively. Suppose we are given a set $\mathcal{R}$ of uninterpreted fixed-arity relation symbols, disjoint from $\mathcal{F}$ and $\mathcal{P}$, and a set $\mathcal{X}$ of variables. A constrained Horn clause is a formula:

$$H \leftarrow I_1, I_2 \wedge, \dots, \wedge I_n, S \tag{6}$$

where each I_i is an application of a relation symbol $r \in \mathcal{R}$ to first-order terms over $\mathcal{F}$,$\mathcal{X}$; S is a constraint over $\mathcal{F}, \mathcal{P}, \mathcal{X}$; and H is either an application of $r \in \mathcal{R}$ to first-order terms over $\mathcal{F}$.

Following the constraint logic programming convention, the right side of (6) is a conjunction of terms and is called *body* of the clause. The left side of (6), or H, is called *head* of the clause. A clause is called *query* if its head is $\mathcal{R}$-free; and otherwise, it is called *rule*. A rule with body $true$ is called *fact*. A clause is *linear* if its body contains at most one predicate symbol; otherwise, it is called *non-linear*. A set of CHCs over predicates $\mathcal{R}$ is called satisfiable (or solvable) if there is an interpretation of the predicates $\mathcal{R}$, called *inductive invariant*, such that the universal closure of every clause holds.

8.2 Certifying GRASSP Solutions

CHCs are served to encode the equivalence conditions between the serial and the parallel programs. If satisfiable, the corresponding inductive invariant certifies the parallel program delivered by GRASSP. We present a system of linear CHCs that encodes a product automaton (PA) that simultaneously executes both programs for the same (potentially un-

bounded) array of input data and requires equivalence of the outputs after each step. CHC-encoding proceeds in a way similar to [21], and in particular, results in one fact, one rule, and one query. The fact encodes an assignment to the initial state of PA, the rule encodes a transition relation of PA, and the query encodes the violation of the equivalence condition.

Intuitively, PA maintains one state variable r, which contains the serial function result for the data array traversed so far, and m other state variables, which contain partial results $\{r_i\}$ for all segments of the data array. Each time PA reads an element from the array (treated nondeterministically), it updates r and strictly one of $\{r_i\}$. Our goal is to inductively prove that the application of $merge$ to all partial results $\{r_i\}$ is equivalent to r.

The key insight for encoding equivalence conditions is the use of an uninterpreted predicate inv, a template for an invariant that exists if and only if the equivalence holds. Note that the rule has inv twice: it describes the pre-states and corresponding post-states of a transition. Additionally, inv appears: (a) in the fact, to ensure that it covers initial states, and (b) in the query, to ensure that after any transition made by PA, inv is strong enough to guarantee that the error (i.e., a violation of the equivalence) is unreachable.

To encode the segmentation of an arbitrary array, we introduce a helper integer variable s_id which contains the current segment's index. That is, for a fixed number of possible segments m, s_id can either increment its value or maintain. Further, s_id identifies which partial result r_i gets updated. Finally, in the query, s_id is forced to be less or equal to m, thus ensuring that the search exhausted all possible segmentations of the input array up to length m.

A skeleton for the system of CHCs that encodes the verification condition is shown in Fig. 11. It has f and h, which represent the specification, and f' and $merge$ (that embeds sum and upd, if needed), which represent the solution by GRASSP.

Example. Fig. 12 shows an instantiation of the system of CHCs at Fig. 11 for the *best-case* parallelization scenario (i.e., sum and upd are not needed) of the program that calculates the count of the array elements. Trivially, $f = f' = merge$ = "+"; h is the identity function, $init = 0$; and $nondet$ symbol is ignored (i.e., no matter what are the array elements, the program just counts them). The system of CHCs is clearly satisfiable since there is an inductive invariant $cnt(s\_id, r, r_1 \dots r_m) \stackrel{\text{def}}{=} r = r_1 + \dots + r_m$ that makes each implication in the system of CHCs true.

An invariant can be obtained by guessing various formulas, substituting them to the system, and checking validity of each implication. Alternatively, the solvers based on Property Directed Reachability (PDR) [13, 17] perform fixed-point computation by alternating interpolation and quantifier elimination and incrementally build the invariant. In our experiments, PDR detected invariants for nearly all programs in our benchmark set (i.e., the ones expressible in linear arithmetic), thus proving the equivalence between the given and synthesized programs in the unbounded setting. Most of the invariants (especially for *Split+Sum+Update-based* computations) are not human-readable, so we do not present them in the paper.

9. Evaluation

We implemented GRASSP in an SMT-based programming language, ROSETTE [30, 31], and supplied the template libraries for the $merge$, sum, upd and $split$ functions. GRASSP treats a serial implementation of an array-handling function as specification. It traverses the search space of candidate implementations populated by explicit instantiations of the templates and model-checks whether a current parallel candidate produces equivalent results to the serial one.

9.1 Benchmarks

We evaluated GRASSP on a set of ROSETTE implementations for some array-handling problems inspired by [3, 4, 25, 27]. Table 1 lists programs, which model wide classes of real-world analytics problems. For example, "*maximal distance between ones*" can be interpreted as "*longest period between commits in github*"; "*checking if the array is sorted*" can be interpreted as either "*checking if the order of system log files is consistent with system time*" or "*checking if there is arithmetic overflow of some counter*"; "*counting instances of* $(1)^*2$" can be interpreted as "*counting the number of purchases of the item strictly after searching for it several times*", and so on.

In Table 1, the benchmarks are distributed across four groups that correspond to the three stages of GRASSP in Fig. 10 and properties of function $merge$:

B1. First stage of GRASSP (with empty prefixes) with a trivial implementation of function $merge$ (to be explained in the rest of this subsection)

B2. First stage of GRASSP (with empty prefixes) with a nontrivial implementation of function $merge$

B3. Second stage of GRASSP with implementations of constant-sized prefix computation and function $merge$

B4. Third stage of GRASSP with implementations of conditional-prefix computation and functions $merge$, sum and upd

We say that function $merge$ has a *trivial* implementation for parallel code with m segments if: (1) it takes m partial states, and (2) it produces the final output in $m - 1$ steps (see Sect. 9.2 for examples). We separate groups B1 and B2 to show that the search within a GRASSP stage can also be made gradually. That is, GRASSP attempts to synthesize a trivial $merge$ first, and, if the attempt did not succeed, it then proceeds to synthesize any other $merge$.

Benchmarks from groups B3 and B4 require shifting the segments boundaries via constant and conditional prefixes,

Table 1: Applying GRASSP to parallelize looping programs of different types; and runtime evaluation of the synthesized results.

	Benchmark description	GRASSP performance (synt time)	Parallel code performance data size (Gb)	time (serial)	% Speedup
no prefix + trivial merge	counting elements	1.056s	126	18m 23.083s	3.6X
	counting elements greater than a constant	1.393s	95	26m 37.630s	4.8X
	search for an element	1.349s	95	26m 42.664s	4.8X
	sum of elements	2.030s	95	25m 58.071s	4.9X
	sum of even elements	1.770s	95	26m 27.372s	4.7X
	sum elements greater than a constant	2.109s	95	25m 58.071s	4.9X
	minimal element	1.473s	100	29m 56.584s	4.7X
	maximal element	1.628s	100	29m 47.601s	4.7X
	maximal absolute value	1.744s	100	29m 27.438s	4.7X
no prefix + nontrivial merge	second maximal element	6.230s	100	30m 23.832s	4.7X
	delta between maximal and minimal elements	4.626s	100	30m 0.074s	5.0X
	average integer value	2.319s	95	27m 5.305s	4.8X
	counting maximal elements	3.090s	100	29m 52.567s	4.9X
	counting minimal elements	3.228s	100	30m 27.702s	4.8X
	equal number of zeroes and ones	1.490s	95	26m 0.901s	4.8X
	counting distinct elements	1.774s	66	23m 19.956s	14.5X
const prefix	checking if all elements are equal to each other	1.455s	88	17m 38.415s	4.9X
	checking if the array is sorted	1.395s	87	16m 57.736s	5.1X
	checking if the array is alternation of 0 and 1	1.542s	91	14m 36.051s	3.6X
conditional prefix + summaries	counting instances of $(1)^*$	2.276s	110	17m 38.024s	3.8X
	counting instances of $(1)^*2$	1.860s	110	17m 53.466s	4.3X
	counting instances of $1(0)^*2$	2.157s	110	20m 14.363s	4.5X
	counting instances of $(1)^*(2)^*3$	11.695s	93	15m 42.181s	3.9X
	counting instances of $1(0)^*2(0)^*3$	5.940s	93	15m 3.485s	3.8X
	checking if 0 (1) is only in the first (last) position	5.531s	110	17m 50.585s	4.0X
	maximal distance between ones	1.830s	110	17m 36.824s	3.9X
	maximal sum between zeros	5.024s	110	17m 27.023s	3.8X

$$
\begin{aligned}
\boldsymbol{eq}(x) &= x = C \,\big|\, x \neq C \\
\boldsymbol{cmp}(x,y) &= x = y \,\big|\, x \neq y \,\big|\, x < y \,\big|\, x > y \,\big|\, x \leq y \,\big|\, x \geq y \,\big|\, eq(x) \wedge eq(y) \\
\boldsymbol{iE}(x,y) &= C \,\big|\, x \,\big|\, y \,\big|\, iE(x,y) + iE(x,y) \,\big|\, min(x,y) \,\big|\, max(x,y) \,\big|\, ite\big(cmp(x,y), iE(x,y), iE(x,y)\big) \\
\boldsymbol{iE^+}(\langle x,y\rangle, z) &= (let\ w = iE(x,z), \langle w, iE(y,w)\rangle) \\
\boldsymbol{iE^2}(\langle x,y\rangle, \langle z,w\rangle) &= ite(cmp(x,z), \langle iE(x,z), iE(y,w)\rangle, \langle iE(x,z), iE(y,w)\rangle)
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{prefix}_{\boldsymbol{cond}}(x) &= eq(x) \\
\boldsymbol{sum}(x,y) &= iE(x,y) \\
\boldsymbol{upd}(\langle x,y\rangle, z) &= iE^+(\langle x,y\rangle, z) \\
\boldsymbol{merge}(x_1,\ldots,x_m) &= m \,\big|\, (x_1 + \ldots + x_m) \,\big|\, min(x_1,\ldots,x_m) \,\big|\, max(x_1,\ldots,x_m) \\
\boldsymbol{merge}^+(A_1,\ldots,A_m) &= h(f(append(A_1,\ldots,A_m))) \;\big|\; h\big(\langle merge(A_1[1],\ldots,A_m[1]), merge(A_1[2],\ldots,A_m[2])\rangle\big) \;\big|\; \\
&\quad fold(iE^2, \langle C, C\rangle, append(A_1,\ldots,A_m))
\end{aligned}
$$

Figure 13: Grammars for template generation.

respectively. Unlike B1 and B2, the order of segments (except "*equality of elements*") must be preserved. Intuitively, the implementation of function upd seeks to bind a particular suffix with a particular prefix; and if the corresponding segments are not consecutive, then the synthesized parallelization would give incorrect results.

9.2 Templates

Fig. 13 shows the grammars for the template generation that are currently implemented in GRASSP. These templates enable synthesis of all programs from our benchmark set without extra help from the user. However, in order to speed the synthesis up, the user might empirically restrict some of the templates, thus strangulating the search for the particular tasks. In the rest of the subsection, we report on our experience with creation of the templates, that the user can follow in order to create own ones.

For benchmarks from B1 and B3, types D, In, and Out are instantiated to integers, and function h is simply the identity. Thus, all is needed for GRASSP is to consider trivial merge functions for integers ($\boldsymbol{merge}$ in Fig. 13) which

Table 2: Hadoop performance on some GRASSP solutions.

Name	Running Time in 10 nodes (sec.)	Speedup (X)
average integer value	912	9.01X
counting all elements	803	9.65X
counting under a condition	862	9.36X
counting maximal elements	899	9.09X
counting minimal elements	935	8.78X
maximal element	899	9.24X
maximal element	918	8.8X
minimal element	821	10.09X
search for an element	904	8.89X
second maximal element	855	9.74X
sum of elements	802	10.3X
sum of even elements	860	9.7X
max / min delta	945	8.82 X
equality among elements	883	9.26X

contain number of elements, summation, min / max element computation.

For benchmarks from B2 and B4, type D is composed of two or more integers, requiring the merging of states to be more complicated ($\boldsymbol{merge}^+$ in Fig. 13). For instance, for benchmarks of types "*second maximal*" or "*counting distinct elements*", the partial states are arrays themselves. GRASSP attempts appending those arrays first; and then processes the resulting array similarly to the specification (i.e., using f and h). For benchmarks of types "*average value*", "*delta between*", or "*equal numbers*", the final state is obtained by applying a trivial merge separately to sequences of the first and the second elements of partial states, and then by applying a given function h. Finally, the $\boldsymbol{merge}$ template for benchmarks "*counting minimal / maximal elements*" requires synthesizing and applying its own $fold$ function.

For benchmarks from B4, we designed $\boldsymbol{prefix}_{cond}$, $\boldsymbol{sum}$, and $\boldsymbol{upd}$. It is sufficient for predicate $\boldsymbol{prefix}_{cond}$ to be either equality or disequality of an element to some constant. Recall (Sect. 6.3) that $prefix_{cond}$ is the building block for $split$, i.e., it is iteratively applied while traversing prefixes. Type D', required for $\boldsymbol{sum}$ and $\boldsymbol{upd}$, is integer, thus the templates gather various combinations of (possibly nested) operations from linear integer arithmetic. In our experience, most of the synthesized $\boldsymbol{sum}$ and $\boldsymbol{upd}$ functions have the form of nested ite-operations.

9.3 GRASSP Performance

We ran GRASSP on a Mac machine, 2.7 GHz Intel Core i7 with 16 GB 1600 MHz DDR3. For all considered serial benchmarks, GRASSP gradually synthesized parallel versions. The typical synthesis time ranged from 1 to 11 seconds. GRASSP found appropriate templates from our predefined template library (see Sect. 9.2) and completed them.

9.4 Parallel Code Performance

GRASSP code is sufficiently general to be ported onto different platforms and to exploit benefits on different hardware. We present two case studies, for multithreading in C++ and MapReduce-like jobs in Hadoop.

Multithreading in C++. The code generated by GRASSP was translated into C++ that uses POSIX Threads library. For efficiency, it exploits demand paging via memory-mapped file I/O which allows the quick handling of large quantities of data. Note the gap between GRASSP synthesized code, that uses arrays, and the running code in C++ that performs "reading-and-processing" on the fly.

For compilation, we used LLVM clang-700 with -O3 optimizations (for both serial and parallel code);[1] for running, we used 8 threads (2 physical cores). As shown in Table 1, we found an up to 5X speedup when running the parallel code against serial code over the same amount and distribution of data (including reading). The experiment confirms that parallel disk (in our case, SSD) reads help for the I/O-bound problems. When the files are mmapped, reading them on eight threads improves the read bandwidth from 55MB/sec to 280MB/sec. Non-mmaped implementation does not benefit from parallel reads. We also tried the strategy of opening in parallel eight mmaped files, followed by a serial read of these files in single thread, but this *did not lead to speedup*.

One exceptional benchmark was "*counting distinct elements*", which outperformed the serial code for 14.5X. However, the speedup exceeded the theoretically calculated maximal one for eight threads. The serial code handled an intermediate data structure for storing the distinct elements, whose size was calculated once all segments were processed. Notably, adzading an element to the intermediate data structure proved computationally expensive: it required searching to determine whether the element was already added to the data structure. Obviously, the total number of iterations in the serial code depends on the order of segments: imagine the case where the first segment has a thousand of distinct elements, but other segments have just one. In the parallel code, such intermediate data structures are local to each thread and therefore can be populated faster than the shared one. Thus, the observed 14.5X speedup was not due only to parallelism, but to the more successful algorithm delivered as a by-product of our parallelization.

MapReduce jobs in Hadoop. We show that the code generated by GRASSP can be translated into tasks for MapReduce framework if the order of processing data segments is not crucial. We evaluate the subset of our benchmarks that fulfill this requirement (mostly from groups B1 and B2) using a 10-node cluster of Amazon EMR m3.xlarge instances. The cluster uses the Hadoop Distributed File System (HDFS) to store input files of 200 GB. Table 2 summarizes a performance gain after switching from serial versions on one node to the corresponding parallel versions on 10 nodes. When the parallel code was delivered by GRASSP and was translated to Hadoop, we re-ran the experiment to fully ex-

[1] Our experimental setup for running the parallel code was chosen intentionally to be the same as for synthesis to demonstrate that synthesis can be performed by arbitrary users on ordinary machines.

ploit the data parallelism. As shown in Table 2, we found an up to 10X speedup in this setting.

10. Related Work

The problem of parallelizing recurrent programs dates back to The 1970s [16]. Today, from purely mathematical solutions, parallelizing matured to address large-scale industrial applications within MapReduce [5], Dryad [14], Spark [34], and Hadoop [33]. These distributed programming platforms are unfortunately unable to automatically parallelize serial code: they require users to write the code for both mappers and reducers. To automatically generate MapReduce programs, [24] proposes to translate serial code into parallel one based on a set of rewrite rules. Alternatively, MapReduce program synthesis can be driven by input/output examples [27].

The closest work to GRASSP is [25]. It considers the same type of parallelism as GRASSP, and it also uses symbolic summaries over the data segments processed in *runtime*. In contrast, GRASSP produces summaries *at compile time* and on demand, if the worst-case scenario happens, letting the other cases use simpler solutions. Finally, since GRASSP delivers the code itself (as opposed to [25]), it can be reused to process other data inputs.

In our previous work [7], we also performed synthesis in stages, but ignored the most problematic case, in which prefixes span the entire segments. Current work subsumes all the contributions from [7] and adds: (a) gradual synthesis of sum and $update$ functions, (b) certification of the solutions with constrained Horn solving, and (c) extensive evaluation of the solutions using C++ multithreading and Hadoop tasks.

The most recent approach to synthesis of parallel code [6] proceeds similarly to GRASSP, by discovering so called *auxiliary accumulators* which play the role of sum functions and enable parallelizing challenging tasks. Despite looking at the parallelization scenario from a different angle, the authors managed to end up with the results consistent to ours. On the technical level, GRASSP differs from [6] in the way it proves soundness of the solutions: it uses an invariant synthesizer, while [6] uses a deductive verifier.

Other approaches to parallelization include Speculation [15, 20, 23], which does not scale for applications on large data. The kind of parallelism useful in regular expression matching [29, 32] and SAT solving [12] is running multiple inputs in parallel (in contrast to single inputs for GRASSP). Some work was proposed to parallelize FSMs [22] by enumerating transitions from all possible states on each input symbol. While the tasks supported by GRASSP can also be viewed as parallelizing FSMs, most of them use counters, making the approach by [22] inapplicable.

Recently, synthesizing programs by examples (PBE) proved extremely successful; see: [1, 2, 10, 11, 27]. Since these applications do not require explicit specification (unlike GRASSP), PBE is orthogonal to our synthesis approach. However, one can still find a connection between bounded synthesis and PBE. Indeed, given a bound n, GRASSP implicitly considers all possible examples in which arrays of length $k \leq n$ constitute the inputs and the serial-code's results constitute the outputs; and then it delivers the parallel code that obviously matches all those examples.

Automated formal methods are now influencing parallel and array-handling computing: [3] studies the commutativity problem of MapReduce, and [4] proposes an approach to prove counting properties in unbounded array-handling programs. In the future, it would be interesting to explore the role these recent verification methods can play in our synthesis engine. Finally, equivalence checking is one of the most intriguing branches of formal methods (e.g., [8, 9, 18, 19, 21, 26]). Orthogonally, GRASSP targets the construction of parallel programs that are equivalent to serial ones. We believe that recent advances in unbounded equivalence checking may open new dimensions in program synthesis and automatic parallelization.

11. Conclusion

In this paper, we addressed the challenge of parallelizing serial code that operates on big data. Our novel approach, GRASSP, applies to cases where data is partitioned into a sequence of segments, each segment is processed separately, and partial outputs for the segments are merged together. GRASSP automatically rewrites the existing serial code such that the decomposition of data dependencies in loop iterations becomes easy. For this, it gradually considers several parallelization scenarios and attempts to find easier solutions first.

We are approaching the goal of automated parallelization through formal verification and synthesis: numerous ingredients of parallel code can be discovered using SMT-based synthesis techniques. We proposed a way of using constrained Horn solving to certify results of *bounded* synthesis and evaluated GRASSP on various classes of single-pass array-processing programs. We showed that the parallel results can achieve up to linear performance speedup relative to serial code.

Acknowledgments

We thank Xi Wang, Rishabh Singh, Madanlal Musuvathi and all the reviewers for the constructive feedback.

This work is supported in parts by the SNSF Fellowship P2T1P2_161971, NSF Grants CCF–1139138, CCF–1337415, and NSF ACI–1535191, a Grant from U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences Energy Frontier Research Centers program under Award Number FOA–0000619, and grants from DARPA FA8750–14–C–0011 and DARPA FA8750–16–2–0032, as well as gifts from Google, Intel, Mozilla, Nokia, and Qualcomm.

References

[1] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. Synthesis through unification. In *CAV*, volume 9207 of *LNCS*, pages 163–179. Springer, 2015.

[2] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In *PLDI*, pages 218–228. ACM, 2015.

[3] Yu-Fang Chen, Lei Song, and Zhilin Wu. The commutativity problem of the MapReduce framework: A transducer-based approach. In *CAV*, volume 9780, pages 91–111, Part II. Springer, 2016.

[4] Przemyslaw Daca, Thomas A. Henzinger, and Andrey Kupriyanov. Array folds logic. In *CAV*, volume 9780, pages 230–248, Part II. Springer, 2016.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[6] Azadeh Farzan and Victor Nicolet. Automated synthesis of divide and conquer parallelism. In *PLDI*. ACM, 2017. to appear.

[7] Grigory Fedyukovich and Rastislav Bodík. Approaching symbolic parallelization by synthesis of recurrence decompositions. In *SYNT*, volume 229 of *EPTCS*, pages 55–66, 2016.

[8] Grigory Fedyukovich, Ondrej Sery, and Natasha Sharygina. eVolCheck: Incremental Upgrade Checker for C. In *TACAS*, volume 7795 of *LNCS*, pages 292–307. Springer, 2013.

[9] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. Property directed equivalence via abstract simulation. In *CAV*, volume 9780, Part II, pages 433–453. Springer, 2016.

[10] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239. ACM, 2015.

[11] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[12] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009.

[13] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, volume 7317, pages 157–171. Springer, 2012.

[14] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72. ACM, 2007.

[15] Shmuel Tomi Klein and Yair Wiseman. Parallel huffman decoding with applications to JPEG files. *Comput. J.*, 46(5): 487–497, 2003.

[16] Peter M. Kogge. Parallel solution of recurrence problems. *IBM Journal of Research and Development*, 18(2):138–148, 1974.

[17] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV*, volume 8559 of *LNCS*, pages 17–34, 2014.

[18] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *DAC*, pages 263–268. IEEE, 1997.

[19] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *FSE*, pages 345–355. ACM, 2013.

[20] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Speculative parallel pattern matching. *IEEE Trans. Information Forensics and Security*, 6(2):438–451, 2011.

[21] Dmitry Mordvinov and Grigory Fedyukovich. Synchronizing Constrained Horn Clauses. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 338–355. EasyChair, 2017.

[22] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *ASPLOS*, pages 529–542. ACM, 2014.

[23] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. Safe programmable speculative parallelism. In *PLDI*, pages 50–61. ACM, 2010.

[24] Cosmin Radoi, Stephen J. Fink, Rodric M. Rabbah, and Manu Sridharan. Translating imperative code to MapReduce. In *OOPSLA*, pages 909–927. ACM, 2014.

[25] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *SOSP*, pages 153–167. ACM, 2015.

[26] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. Data-driven equivalence checking. In *OOPSLA*, pages 391–406. ACM, 2013.

[27] Calvin Smith and Aws Albarghouthi. MapReduce program synthesis. In *PLDI*, pages 326–340. ACM, 2016.

[28] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.

[29] Peter Sutton. Partial character decoding for improved regular expression matching in fpgas. In *FPT*, pages 25–32. IEEE, 2004.

[30] Emina Torlak and Rastislav Bodík. Growing solver-aided languages with Rosette. In *Onward!*, pages 135–152. ACM, 2013.

[31] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, pages 530–541. ACM, 2014.

[32] Giorgos Vasiliadis, Michalis Polychronakis, Spyros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *RAID*, volume 5758 of *LNCS*, pages 265–283. Springer, 2009.

[33] Tom White. *Hadoop - The Definitive Guide: MapReduce for the Cloud*. O'Reilly, 2009.

[34] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28. USENIX Association, 2012.