Modernizing Parallel Code with Pattern Analysis

Roberto Castañeda Lozano
University of Edinburgh
School of Informatics
Edinburgh, United Kingdom
rcastae@inf.ed.ac.uk

Murray Cole
University of Edinburgh
School of Informatics
Edinburgh, United Kingdom
mic@inf.ed.ac.uk

Björn Franke
University of Edinburgh
School of Informatics
Edinburgh, United Kingdom
bfranke@inf.ed.ac.uk

Abstract

Fifty years of parallel programming has generated a substantial legacy parallel codebase, creating a new portability challenge: re-parallelizing already parallel code. Our solution exploits inherently portable *parallel patterns*, and addresses the challenge of identifying patternization opportunities in legacy parallel code via constraint matching on traced dynamic dataflow graphs. Notably, this makes the analysis source-independent and equally applicable to sequential and parallel legacy code. We identify various map and reduction patterns, including compositions, in Pthreads code. Experiments with the Starbench suite show that our analysis is effective (finding 86% of the patterns known in the literature), accurate (reporting actual patterns in 98% of the cases), and efficient (scaling linearly with the size of the execution traces). We re-express the found patterns via a parallel pattern library, making code freely portable across CPU/GPU systems and performing competitively with hand-tuned implementations at zero additional effort.

CCS Concepts: • **Theory of computation → Parallel algorithms**; **Constraint and logic programming**.

Keywords: parallel patterns, code modernization, dynamic analysis, pattern matching

1 Introduction

For fifty years, "parallelization of legacy software" has meant the challenge of transforming sequential code into equivalent but faster parallel form. With explicit parallel coding now mainstream, a new challenge arises: as complex heterogeneous architectures continue to evolve, how can we re-parallelize *legacy parallel code* so that it remains fit for purpose? This is not simply about semantic portability. Languages for the full spectrum of heterogeneous systems offer diverse conceptual models and may even require algorithmic rethinking for best performance.

If we are coding from scratch, a programming methodology based around *parallel patterns* can help considerably [12, 48, 49]. Patterned parallel code is inherently portable, and optimization, including algorithmic revision, is handled transparently by the ingenuity of the pattern architect [8, 36]. But what if legacy parallel code already exists? We address the missing link: our analysis examines legacy parallel code, and reports on the presence of code fragments which could be replaced by calls to known parallel pattern library abstractions.

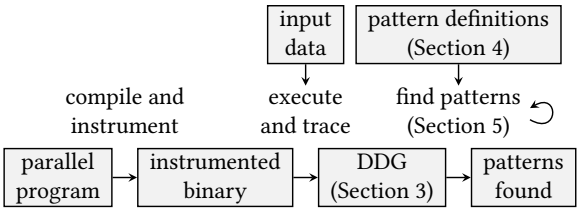


Figure 1. Overview of our parallel pattern finding analysis.

These instances are fed back to the programmer, indicating precisely where in the legacy code it will be possible to make this transformation, and therefore to gain the portability advantages of the known pattern implementations. It is no surprise that this is possible: legacy programmers will often have naturally gravitated towards the algorithmic strategies which underpin standard patterns, but coded them ad hoc. Our challenge is to find these code fragments and highlight them to the programmer.

We have a simple thesis: each pattern is fundamentally characterized by the topology of the dynamic dataflow it invokes, and the repetitions within it, rather than the static structure of the code which expresses it. This makes our work oblivious to whether the legacy code is sequential or parallel. Our representation also abstracts away from the bulk data structures used: we can analyze linked-list code as simply as array code. Our core technique is to define patterns as constrained subgraphs of dynamic dataflow graphs (DDGs), and to deploy a standard constraint programming system to find instances of these in the traces of instrumented legacy parallel programs. Unlike previous work on sequential code, it is crucial for us to to be able to reason about *different executions of the same instruction*, as these characterize the parallel forms of some patterns, as discussed in Section 2.

Our analysis is captured in Figure 1. Our compiler pass instruments the legacy program. The resulting traced DDG is matched against predefined patterns, employing heuristics which trade analysis time against completeness. A key feature is the cycle on pattern-finding: identifying and abstracting simple patterns can reveal more complex pattern nests and compositions. Since our initial implementation instruments Pthreaded C/C++ code, we evaluate our analysis on the Starbench benchmark suite [2]. Our patterns include maps, reductions, and compositions of these. We find 36

of the 42 expected instances in Starbench, including some which require iterated application of the matching phase. The only six missed instances raise interesting issues to be addressed in future work. In summary, our contributions are:

- we implement a compiler pass which instruments legacy parallel C code to generate DDGs;
- we characterize a range of map, reduction, and map-reduction patterns in terms of their dynamic dataflow;
- we describe a scalable, iterative pattern-finding analysis to identify instances of ad hoc coding of patterns in the DDGs of legacy parallel code; and
- we report excellent effectiveness and accuracy of our analysis on the Starbench benchmark suite, and strong performance and portability of the resulting programs.

Our strategy raises several meta-issues. Firstly, a dynamic analysis is only as good as the input used. In deployment, our system would require programmer confirmation of its suggestions. Experiments show that our approach is highly accurate: it reports actual patterns in 98% of cases, reflecting previous findings that "data-dependences in frequently executed loops are not changed noticeably with respect to different input" [45]. Secondly, our approach may generate a large number of pattern suggestions. In deployment, we could use related work such as hot-spot profiling [25] to focus on the most promising cases.

The remainder of the paper is structured as follows: Section 2 presents a motivational example, Section 3 introduces our DDGs and the instrumentation pass to generate them, Section 4 explains our use of constraint programming to capture patterns, Section 5 discusses our iterative pattern finder and the heuristics which make it scalable, and Section 6 presents our analysis of Starbench. We discuss related work in Section 7, and the current limitations in Section 8. We conclude with a discussion of future work.

2 Motivating Example

Figure 2a shows an excerpt from `streamcluster`, slightly simplified for clarity (our analysis also handles the original version, as shown in Section 6.1). `streamcluster` is a parallel C program that groups a stream of points into clusters using k-medians clustering. `streamcluster` uses low-level multi-threading (Pthreads) in different phases of the execution. This program is a popular benchmark, available for example in the PARSEC [6] and Starbench [2] suites.

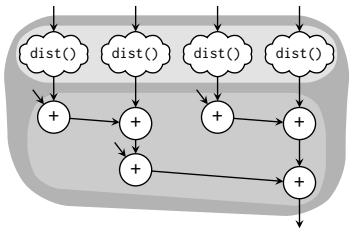
The particular code excerpt in Figure 2a computes, in parallel, the total distance between the first point (`P->p[0]`) and all other points. First, each thread `pid` computes the partial distance (`hizs[pid]`) between a subset `k1`...`k2` of the points and the first point. Then, the total distance `hiz` is computed by adding all partial distances. As highlighted in Figure 2a, the entire computation can be seen as a parallel instance of the compound *map-reduction* pattern, where the *map* captures the parallel computation of the distance between the

```
for (i = 0; i < nproc; i++) {
 // Run pkmedian in nproc threads.
 pthread_create(..);
}
..
for (i = 0; i < nproc; i++) {
 pthread_join(..);
}
..
float pkmedian(Points *P, .., int pid) {
 ..
 pthread_barrier_wait(..);
 float myhiz = 0;
 for (kk = k1; kk < k2; kk++) {
  myhiz += dist(P->p[kk], P->p[0]);
 }
 hizs[pid] = myhiz;
 pthread_barrier_wait(..);
 for (i = 0; i < nproc; i++) {
  hiz += hizs[i];
 }
 ..
}
```

(a) Original distance computation.

```
float pkmedian(Points *P, ..) {
 ..
 auto computeDist = MapReduce(
  [] (Point p) {
    return dist(p, P->p[0]);
  },
  [] (float h1, float h2) {
    return h1 + h2;
  }
 );
 hiz = computeDist(
  Vector<Point>(P->p, P->num));
 ..
}
```

(b) Modernized version.



(c) Partial, simplified DDG of the original parallel distance computation for four points and two threads (two points per thread).

Figure 2. Parallel distance computation in `streamcluster.c`. Found patterns are colored in light gray (*map*), gray (*reduction*), and dark gray (*map-reduction*).

first and all other points, and the *reduction* captures the intra- and inter-thread addition of distances.

The use of Pthreads makes the parallelization CPU-efficient but degrades its maintainability and precludes its portability to increasingly powerful parallel architectures such as GPUs. These properties could be improved by expressing the computation with a *map-reduction* pattern instead, however finding which patterns apply to a program is in general a tedious task carried out by experts. While there exist analyses to find parallel patterns in legacy *sequential* code, to the best of our knowledge no analysis exists to do so in *parallel* code as in this case. Doing so involves significant challenges:

1. The entire map-reduction is executed by multiple threads (challenging for static analysis).
2. The reduction is spread across multiple loops (challenging for analyses that examine one loop at a time).
3. The reduction is characterized by how the individual executions of the addition operations are arranged (challenging for existing dynamic analyses, which do not reason about individual operation executions).

4. The map is scattered across multiple translation units (challenging for analyses that examine one function or module at a time).
5. The reduction is obscured by data copying between `myhiz` and `hizs[pid]` (challenging for analyses based on idiom recognition).

Our analysis overcomes these challenges and finds the compound *map-reduction* pattern in Figure 2a. Using a seamless multi-thread tracing technique allows it to derive the same trace representation (a DDG) for sequential and parallel code (challenge 1). Decoupling the representation from its original code and applying powerful pattern matching allows it to find patterns across single program regions (challenge 2). Capturing each individual operation execution in the representation allows it to find patterns characterized by specific groups of operation executions, as is common in legacy parallel code (challenge 3). Analyzing whole-program traces allows it to find patterns across translation units and even libraries (challenge 4). Finally, abstracting away auxiliary computation such as data transfers and data-structure traversals allows it to find patterns regardless of the specific shape of the source code (challenge 5).

Figure 2c shows the DDG corresponding to the execution of the Pthreaded code for four points and two threads. Nodes correspond to executions of operations in Figure 2a, and arcs correspond to flow of data among the executed operations. For simplicity, each computation `dist()` is shown as a single subgraph node, and the initial values of `myhiz` and `hiz` (the addition identity `0`) are depicted with sourceless arcs. The *map*, *reduction*, and *map-reduction* patterns found by our analysis are highlighted. Our pattern definitions capture these patterns for varying number of points and threads.

After finding the map-reduction, the code can be modernized by replacing the low-level Pthreads code (thread management, synchronization, and workload splitting) with high-level pattern library constructs. Figure 2b shows a modernized C++ version adapted from the P^{3}ARSEC suite [11]. The modernized version implements the found map-reduction pattern using the `MapReduce` pattern construct provided by the SkePU 2 pattern library [16]. The modernized version involves a single call to `pkmedian`, and handles thread creation, destruction, and synchronization as well as workload splitting transparently without loss of performance [11]. The high-level nature of the modernized version allows it to seamlessly capitalize on the strengths of different hardware architectures. For example, the modernized `streamcluster` executed on a GPU is 56% faster than the original Pthreaded version executed on a 12-core CPU, which already offered a 9.6x speed up over a sequential baseline (see Section 6.3).

3 Dynamic Dataflow Graphs

The program representation in our pattern finding analysis is the dynamic dataflow graph (DDG), which can be seen as a dynamic dependency graph without control-flow information. More formally, a DDG is a directed acyclic graph where nodes correspond to specific executions of operations and there is an arc $\langle u, v \rangle$ for each operation execution v using a value defined by operation execution u [33]. DDGs differ from static dataflow graphs common in compilers in that in the former each node represents *a single* operation execution, whereas in the latter each node represents *all* possible operation executions. DDGs are an appealing representation for pattern finding, as they are recognized to capture the "essence of a program's computation" [33].

Our DDG nodes correspond to (executions of) operations from a compiler intermediate representation (IR). Using an IR allows us to handle different programming languages while retaining the ability to point to the exact location of the found patterns in the source code. Besides IR operations, the DDG contains nodes for calls to standard functions such as `pthread_create()`.

A desirable property of DDGs is to capture the computation that potentially characterizes the parallel patterns while abstracting away *pattern-independent computation* such as data transfer operations, memory address calculations, and data structure traversals. By construction, the DDG does not contain any notion of data location, and hence abstracts away data transferring. The effect of data transfer operations is only reflected implicitly on the shape of the DDG. Memory address calculations and data structure traversals are identified and removed during pattern finding as described in Section 5.

DDGs are generated by instrumenting and executing the program under analysis. During execution, the operations that define and use each value are traced. The tracing process is supported by a *shadow* memory that records the node that has defined the current value in each memory location [33]. Accesses to the shadow memory are synchronized to support seamless generation of DDGs from multi-threaded programs.

4 Pattern Definitions

This section defines the patterns studied in the paper (*map*, *linear/tiled reduction*, and *linear/tiled map-reduction*) as collections of subgraphs of DDGs that satisfy specific constraints. The patterns are introduced incrementally, starting with the definition of abstract patterns (Section 4.1); following with the definition of *map* as a basic pattern (Section 4.2); continuing with the definition of *linear reduction* as a basic pattern and *tiled reduction* as composition of linear reductions (Section 4.3); and finishing with the definition of *linear/tiled map-reduction* as compositions of maps and reductions (Section 4.4). Figure 3 illustrates the structure of these patterns.

The patterns are defined in a general form. Our pattern finder (Section 5), implements each definition as a combinatorial model with finite-domain variables and constraints.

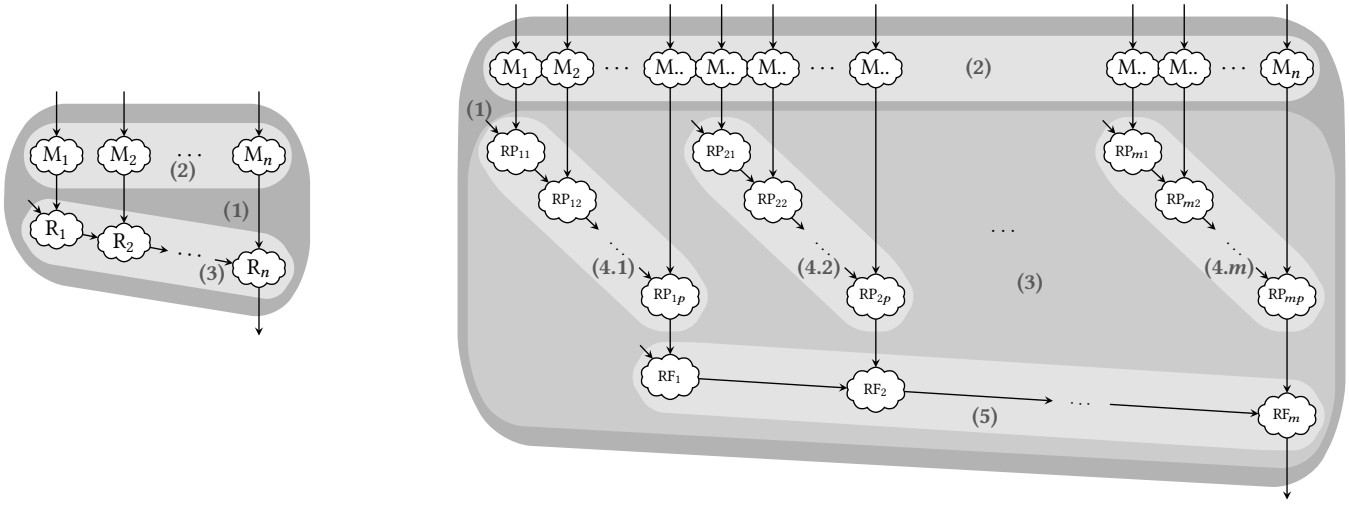


Figure 3. Patterns as DDGs. **Left**: linear map-reduction over n data elements **(1)** as a composition of a map **(2)** and a linear reduction **(3)**. **Right**: tiled map-reduction over n data elements **(1)** as a composition of a map **(2)** and a tiled reduction **(3)**, which is composed of m linear reductions **(4.1)**, **(4.2)**, …, **(4.*m*)** of p components each and a linear reduction **(5)** of m components.

All definitions assume a dynamic dataflow graph *DDG* where each node u is labeled with its operation $operation(u)$.

4.1 Patterns

Abstract patterns are groups of repeated operation executions. A *pattern* P within *DDG* is a sequence of *components* $(\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_n)$ subject to the constraints specified in (1a-1e).

Each component is an induced *DDG* subgraph (1a), formed by a subset of the nodes of *DDG* and all arcs between nodes in that subset. Components of a pattern do not share nodes (1b). Each component captures a particular repetition of the execution of the same operations; this is modeled by requiring that all components are isomorphic when they are labeled with the operation of each node (1c). The operations executed within each component must be related by their data flow; this is modeled by requiring that each component is *weakly connected*, that is, its corresponding undirected graph is *connected* (1d). Finally, patterns are *convex* in the sense that all paths between pattern nodes are internal to the pattern. This is modeled by forbidding pairs of nodes in P that both reach and are reached by nodes in *DDG* but not in P (1e).

$$
\begin{aligned}
& pattern(\mathrm{P}, DDG) \iff \\
& \forall \mathrm{c} \in \mathrm{P}\colon induced\text{-}subgraph(\mathrm{c}, DDG) && \text{(1a)} \\
\wedge\ & \forall i, j \in [1, n] \mid i \neq j\colon nodes(\mathrm{P}_i) \cap nodes(\mathrm{P}_j) = \emptyset && \text{(1b)} \\
\wedge\ & \forall \mathrm{c}, \mathrm{c}' \in \mathrm{P}\colon isomorphic(\mathrm{c}, \mathrm{c}', operation) && \text{(1c)} \\
\wedge\ & \forall \mathrm{c} \in \mathrm{P}\colon weakly\text{-}connected(\mathrm{c}) && \text{(1d)} \\
\wedge\ & \forall u, w \in nodes(\cup_{\mathrm{c}\in\mathrm{P}}\, \mathrm{c}), \forall v \in nodes(DDG - \cup_{\mathrm{c}\in\mathrm{P}}\, \mathrm{c})\colon \\
& \quad \neg\,(reaches(u, v) \wedge reaches(v, w)) && \text{(1e)}
\end{aligned}
$$

4.2 Map Patterns

Map patterns apply an independent operation over multiple data elements. A *map* $\mathrm{M} = (\mathrm{M}_1, \mathrm{M}_2, \ldots, \mathrm{M}_n)$ within *DDG* is a pattern specialization (2a) subject to component independence and input/output constraints as specified in (2b-2d).

Component independence is modeled by forbidding arcs between components (2b); transitive dependencies between components are prevented by pattern convexity (1e). Each map component takes an input data element and produces an output data element. This is modeled by requiring that each component has incoming (2c) and outgoing (2d) arcs. Multiple incoming and outgoing arcs are permitted to capture maps over non-scalar data.

$$
\begin{aligned}
& map(\mathrm{M}, DDG) \iff \\
& pattern(\mathrm{M}, DDG) && \text{(2a)} \\
\wedge\ & \forall i, j \in [1, n] \mid i \neq j, \nexists\, \langle u, v\rangle \in arcs(DDG)\colon \\
& \quad u \in nodes(\mathrm{M}_i) \wedge v \in nodes(\mathrm{M}_j) && \text{(2b)} \\
\wedge\ & \forall \mathrm{c} \in \mathrm{M}, \exists\, \langle u, v\rangle \in arcs(DDG)\colon \\
& \quad u \notin nodes(\mathrm{c}) \wedge v \in nodes(\mathrm{c}) && \text{(2c)} \\
\wedge\ & \forall \mathrm{c} \in \mathrm{M}, \exists\, \langle u, v\rangle \in arcs(DDG)\colon \\
& \quad u \in nodes(\mathrm{c}) \wedge v \notin nodes(\mathrm{c}) && \text{(2d)}
\end{aligned}
$$

Map variants. A conditional map is a special variant of the map pattern where components might produce output or not depending on a condition tested within the map operation. Conditional maps are modeled as regular maps except that only *some* of their components have outgoing arcs.

A fused map is a special map case consisting of two or more chained maps over the same data elements. Fusing

maps (sometimes referred to as *code fusion*) improves parallel efficiency by reducing synchronization and communication costs [31]. The map definition provided above readily captures fused maps of arbitrary size.

4.3 Reduction Patterns

Reduction patterns combine multiple data elements by applying an associative reduction operator to all elements. The associativity of the reduction operator introduces a large degree of freedom in the way reductions are arranged, which can be exploited for parallelization. This section defines two common reduction variants: *linear* and *tiled* reductions.

Linear reductions. Linear reductions apply the reduction operator in a single pass over all data elements. This pattern arises typically from sequential implementations [31].

A *linear reduction* $\mathrm{R} = (\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n)$ within *DDG* is a pattern specialization (3a) subject to component associativity, component chaining, and input/output constraints as specified in (3b-3f). Although the most common reduction operators are formed by a single operation (for example, integer addition), the definition provided in this paper allows capturing arbitrary reduction operators with multiple operations (for example, complex number addition).

Linear reduction components are required to be associative for efficient parallelization. The associativity test considers the dataflow structure of each component and the underlying operations of each node (3b). Commutativity might be modeled in a similar way, if required. The components of a linear reduction are structured as a chain, where each component in the chain defines data that is used by the next component only. This is modeled by requiring that all nodes in a component reach the nodes in the next component in the chain (3c), and forbidding arcs between non-consecutive components (3d). Besides the chain structure, each component takes an input data element, and the final component produces an output data element. This is modeled by requiring that each component has incoming arcs (3e) and the last component has outgoing arcs (3f).

$$
\begin{aligned}
&\mathit{linear\text{-}reduction}(\mathrm{R}, \mathit{DDG}) \iff \\
&\quad \mathit{pattern}(\mathrm{R}, \mathit{DDG}) && \text{(3a)}\\
&\wedge\ \forall \mathrm{c} \in \mathrm{R}\colon \mathit{associative}(\mathrm{c}, \mathit{operation}) && \text{(3b)}\\
&\wedge\ \forall i \in [1, n-1], \forall u \in \mathrm{R}_i, \forall v \in \mathrm{R}_{i+1}\colon \mathit{reaches}(u, v) && \text{(3c)}\\
&\wedge\ \forall i, j \in [1, n]\ |\ |i-j| > 1, \nexists \langle u, v\rangle \in \mathit{arcs}(\mathit{DDG})\colon \\
&\qquad u \in \mathit{nodes}(\mathrm{R}_i) \wedge v \in \mathit{nodes}(\mathrm{R}_j) && \text{(3d)}\\
&\wedge\ \forall \mathrm{c} \in \mathrm{R}, \exists \langle u, v\rangle \in \mathit{arcs}(\mathit{DDG})\colon \\
&\qquad u \notin \mathit{nodes}(\mathrm{c}) \wedge v \in \mathit{nodes}(\mathrm{c}) && \text{(3e)}\\
&\wedge\ \exists \langle u, v\rangle \in \mathit{arcs}(\mathit{DDG})\colon \\
&\qquad u \in \mathit{nodes}(\mathrm{R}_n) \wedge v \notin \mathit{nodes}(\mathrm{R}_n) && \text{(3f)}
\end{aligned}
$$

Tiled reductions. Tiled reductions apply the reduction operator in two phases: the first phase (*partial reduction*) computes a set of partial combinations, and the second phase (*final reduction*) combines the partial combinations into a single element. Tiled reduction can be seen as a compound pattern based on multiple linear reductions. This pattern arises often, but not exclusively, from the standard parallelization of a reduction, where multiple partial reductions are computed in parallel, and the partial results are combined sequentially in a final reduction [31].

A *tiled reduction* within *DDG* is a tuple $\langle \mathrm{RP}, \mathrm{RF}\rangle$ where RP is a sequence of linear reductions $(\mathrm{RP}_1, \mathrm{RP}_2, \ldots, \mathrm{RP}_m)$ (4a) of p components each, RF is also a linear reduction (4b), and both are subject to operation isomorphism and reduction channeling constraints as specified in (4c-4e).

In a tiled reduction, all partial and final reductions apply the same reduction operator, even if they might be implemented in different source locations. This is modeled by requiring that all components within all partial and final reductions are isomorphic when they are labeled with the operation of each node (4c). Finally, in this pattern each partial reduction produces an output data element that is taken as input by a specific component of the final reduction. This is modeled by requiring that all nodes in the last component of each partial reduction can reach the nodes of its corresponding component in the final reduction (4d), and forbidding all other arcs between partial and final reductions (4e).

$$
\begin{aligned}
&\mathit{tiled\text{-}reduction}(\langle \mathrm{RP}, \mathrm{RF}\rangle, \mathit{DDG}) \iff \\
&\quad \forall \mathrm{R} \in \mathrm{RP}\colon \mathit{linear\text{-}reduction}(\mathrm{R}, \mathit{DDG}) && \text{(4a)}\\
&\wedge\ \mathit{linear\text{-}reduction}(\mathrm{RF}, \mathit{DDG}) && \text{(4b)}\\
&\wedge\ \forall \mathrm{c}, \mathrm{c}' \in (\cup_{\mathrm{R}\in\mathrm{RP}}\, \mathrm{R}) \cup \mathrm{RF}\colon \mathit{isomorphic}(\mathrm{c}, \mathrm{c}', \mathit{operation}) && \text{(4c)}\\
&\wedge\ \forall i \in [1, m], \forall u \in \mathit{nodes}(\mathrm{RP}_{ip}), \forall v \in \mathit{nodes}(\mathrm{RF}_i)\colon \\
&\qquad \mathit{reaches}(u, v) && \text{(4d)}\\
&\wedge\ \forall i, j \in [1, m]\ |\ i \neq j, \nexists \langle u, v\rangle \in \mathit{arcs}(\mathit{DDG})\colon \\
&\qquad u \in \mathit{nodes}(\mathrm{RP}_{ip}) \wedge v \in \mathit{nodes}(\mathrm{RF}_j) && \text{(4e)}
\end{aligned}
$$

4.4 Map-Reduction Patterns

Map-reduction patterns fuse map and reduction patterns, potentially improving parallel efficiency by reducing synchronization and communication costs. This opportunity arises often enough in practice that many pattern libraries provide specific abstractions or optimizations [31]. Analogously to Section 4.3, this section defines *linear* and *tiled* map-reductions. Their formal definitions simply enforce a consistent interface between their map and reduction components, and are provided as supplementary material at https://github.com/robcasloz/llvm-discovery.

Linear map-reductions. A *linear map-reduction* within *DDG* is a tuple $\langle \mathrm{M}, \mathrm{R}\rangle$ where M is a map, R is a linear reduction, and each map component produces an output data

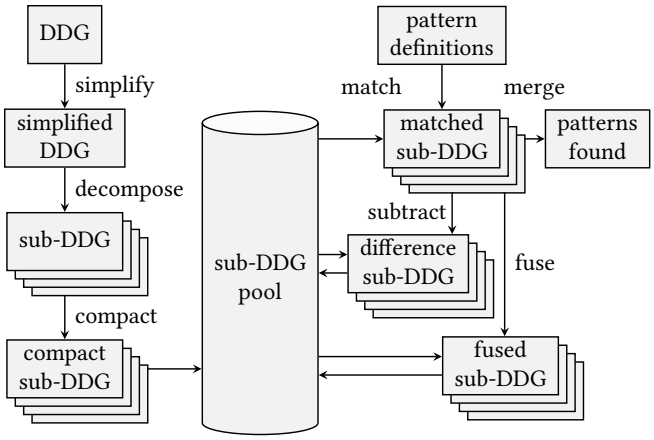


Figure 4. Overview of the iterative pattern finder.

element that is only taken as input by its corresponding reduction component.

Tiled map-reductions. A *tiled map-reduction* within *DDG* is a tuple $\langle M, \langle RP, RF\rangle\rangle$ where M is a map, $\langle RP, RF\rangle$ is a tiled reduction, and each map component produces an output data element that is only taken as input by its corresponding partial reduction component.

5 Iterative Pattern Finder

From a computability perspective, the patterns considered in this paper can be found on a given DDG by simply searching for all its subgraphs that satisfy the constraints in Section 4. Unfortunately, the size of the DDGs and the computational complexity of constrained subgraph matching render this direct approach impractical for all but the smallest problems. This section introduces a pattern finder that detects patterns underlying actual legacy sequential and parallel programs in a practical amount of time.

Figure 4 and Algorithm 1 outline our pattern finder. The pattern finder takes as input a DDG generated by tracing the program under analysis and a set of pattern definitions, and outputs a set of patterns found on the DDG. First, the DDG is reduced by a range of simplifications (line 1). The simplified DDG is then decomposed into a set of smaller sub-DDGs (line 2), based on the observation that most patterns are localized in specific regions of the DDG. Each of the decomposed sub-DDGs is compacted by grouping nodes with common features (line 3), and stored in a pool of sub-DDGs (line 4). The latest sub-DDGs added to the pool are marked as *active*. Each active sub-DDG in the pool is fed into a constraint solver that, according to a combinatorial model of each of the pattern definitions, searches for matching pattern subgraphs (line 7).

The matched patterns are used as feedback to generate new sub-DDGs that are stored in the pool, marked as active, and fed into the constraint solver, in an iterative scheme

Algorithm 1: Iterative pattern finding. Sets of DDGs or pattern definitions are given capital letters; single elements are given lowercase letters.

Input : A DDG g_{in}, a set D of pattern definitions.
Output: A set of found patterns.

```
   // Simplify and decompose g_in into a set of sub-DDGs.
1: g_s ← simplify(g_in);
2: G ← decompose(g_s);
   // Compact each decomposed sub-DDG.
3: G' ← {compact(g) | g ∈ G};
   // Initialize sub-DDG pool P and active subset A ⊆ P.
4: P ← G'; A ← G';
   // Initialize set of found patterns.
5: F ← {};
   // Match active sub-DDGs against pattern definitions,
      generate new active sub-DDGs by combining found
      patterns with already visited sub-DDGs, and repeat
      until no more active sub-DDGs are generated.
6: do
       // Match sub-DDGs against pattern definitions.
7:     M ← {g | g ∈ A, d ∈ D, match(g, d)};
       // Store found patterns (matched sub-DDGs).
8:     F ← F ∪ M;
       // Subtract new matches to pool sub-DDGs.
9:     A_S ← {subtract(g, g') | g ∈ P, g' ∈ M};
       // Fuse pool sub-DDGs and new matches.
10:    A_F ← {fuse(g, g') | g ∈ P, g' ∈ M,
                 adjacent(g, g'), compatible(g, g')};
       // Set new, unvisited sub-DDGs as active.
11:    A ← (A_S ∪ A_F) − P;
       // Store new active sub-DDGs.
12:    P ← P ∪ A;
13: while |A| > 0;
   // Merge and return the found patterns.
14: return merge(F);
```

(lines 6-13) that proceeds until a fixpoint is reached. Each iteration generates new sub-DDGs by applying *subtraction* and *fusion* operations to pairs of matched and previously generated sub-DDGs (lines 9-10). This is based on the observation that the difference between sub-DDGs might expose new patterns (such as maps in complex loops), and that patterns such as map-reductions can be found by fusing simpler patterns such as maps and reductions. As seen in Algorithm 1, subtraction and fusion are independent operations and can be computed in parallel.

The iterative scheme (lines 6-13) is guaranteed to terminate: the sub-DDG pool does not admit duplicates, so in the worst case the algorithm will terminate when all DDG subgraphs (a finite set) are explored. In practice (Section 6), only

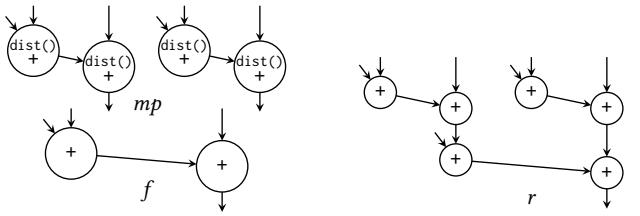


Figure 5. Compact sub-DDGs corresponding to the DDG in Figure 2c. *mp* and *f* are loop DDGs, *r* is an associative component DDG. The letters *m*, *r*, *p*, and *f* are used to abbreviate *map*, *reduction*, *partial* reduction, and *final* reduction.

a few iterations are usually required. After reaching a fixpoint, the matched patterns are merged across all sub-DDGs, and the resulting patterns are reported to the user, together with their exact source code location (line 14).

DDG Simplification. This phase removes memory address calculations and data structure traversals from the DDG to improve scalability. Memory address calculations are identified directly on the DDG, whereas linked and array data structure traversals are identified by generalized iterator recognition analysis [29] on the IR. This type of computation does not generally characterize a pattern in our representation, so removing it from the DDG preserves the information needed for pattern finding. In our experiments (Section 6), simplification reduces the size of DDGs by 3.82x on average.

DDG Decomposition. This phase partitions the simplified DDG into a set of sub-DDGs on which patterns can be found more easily. The union of the patterns found in the sub-DDGs approximates the patterns that could be found in the entire DDG. The decomposition is applied across two dimensions: *loops* and *associative components*.

Loop DDGs capture the computation within the dynamic scope of each loop. Loop decomposition aims at finding patterns based on single loops such as simple maps and linear reductions. Loops are a natural target for pattern finding, as they are the main repetition mechanism in imperative programming and capture most of a program execution time.

Associative component DDGs are weakly connected components with nodes of specific associative operations. Associative component decomposition aims at finding patterns such as linear and tiled reductions, whose components are connected and often formed by single associative operations.

Figure 5 shows the result of decomposing the DDG in Figure 2c. The resulting set of sub-DDGs includes two loop DDGs (*mp*, *f*) and an associative component DDG (*r*). In later phases of the pattern finder, loop and associative component sub-DDGs are subtracted (to expose new patterns such as maps hidden in complex loops) and fused (to find more complex patterns such as fused maps and map-reductions).

DDG Compaction. This phase collapses all nodes executed within each iteration of a loop DDG into single nodes. The resulting compact loop DDG has one node per iteration, and arcs corresponding to dataflow across iterations. The combination of decomposition and compaction is critical for scalability: in our experiments, disabling these two phases causes the constraint solver to exhaust its memory limit (32 GB) and fail for even the smallest benchmark.

In the running example (Figure 5), *mp* is compacted: it groups each subgraph resulting from a `dist()` call with its corresponding partial reduction (+) into a single node.

Pattern Matching. This phase searches, within each *active* sub-DDG in the pool, for pattern subgraphs that conform to Section 4's definitions.

The pattern definitions are implemented as combinatorial models with finite-domain variables and constraints [40]. This allows us to leverage the power of modern constraint solving and scale beyond tiny DDGs. For practical reasons, the models are implemented over sets of nodes rather than subgraphs. Hence, the primary variables are the sets of nodes within each pattern component. The constraint solver is given the task to find assignments of sets of nodes to components that satisfy the constraints in Section 4.

The isomorphism (1c, 4c), connectivity (1d), and associativity (3b) constraints are too costly to be modeled directly in the standard combinatorial modeling framework, and are thus approximated. Isomorphism and connectivity constraints are approximated using relaxations. In our experiments, this approximation does not lead to violations of the original pattern definitions. Associativity constraints in reductions are enforced by limiting each component to a single node whose operation is known to be associative. This underapproximation does not lead in practice to missing reduction patterns, as these often use simple operators such as addition or multiplication. Testing associativity automatically is generally hard, but the approximation could be refined [31].

Table 1 shows the result of applying pattern matching and the remaining phases to the sub-DDG pool in Figure 5.

DDG Subtraction. This phase computes the difference between each pool sub-DDG and each matched sub-DDG, that is, the graph induced by the difference of their nodes. The resulting sub-DDGs are stored back in the pool as active. Subtraction exposes additional patterns hidden by compaction, for example in loops that fuse multiple patterns.

Table 1 shows the result of applying subtraction in each iteration of the example. In it. 1, subtraction generates two sub-DDGs (*m* and *p*). The sub-DDG *m* is particularly useful, as it exposes a *map* pattern in the first loop by removing the partial reduction dataflow originally present in the loop. Exposing the map pattern, in its turn, enables finding the larger map-reduction pattern in the next iteration.

Table 1. Result of each pattern finding phase, starting from the sub-DDG pool in Figure 5. Active sub-DDGs (matching candidates) are colored in black, inactive sub-DDGs are colored in gray, and reported patterns are marked in bold.

it.	sub-DDG pool	phase	result
1	$\{mp, f, r\}$	match	f: linear reduction, r: tiled reduction
	$\{mp, f, r\}$	subtract	$m = \text{subtract}(mp, r)$, $p = \text{subtract}(r, f)$
	$\{mp, f, r\}$	fuse	-
2	$\{mp, f, r, m, p\}$	match	m: map
	$\{mp, f, r, m, p\}$	subtract	-
	$\{mp, f, r, m, p\}$	fuse	$mr = \text{fuse}(r, m)$
3	$\{mp, f, r, m, p, mr\}$	match	mr: tiled map-reduction
	$\{mp, f, r, m, p, mr\}$	subtract	-
	$\{mp, f, r, m, p, mr\}$	fuse	-
	$\{mp, f, r, m, p, mr\}$	merge	**mr: tiled map-reduction**

DDG Fusion. This phase computes the union of pool sub-DDGs and matched sub-DDGs, that is, the subgraph of the original DDG induced by the union of the nodes in both sub-DDGs. The resulting sub-DDGs are stored back in the pool as active. For efficiency, fusion is restricted to sub-DDG pairs adjacent in the original DDG (all arcs from one flow into the other) and where compatible patterns (a map flowing into any pattern) have been matched. Combined with the iterative scheme, fusion enables finding combinations of maps, reductions, and map-reductions of arbitrary complexity.

Table 1 shows the fusion result in each iteration of the example. In it. 2, fusion generates a new sub-DDG (mr) from r and m. This fusion is possible because m and r are adjacent (all arcs from m flow into r in the original DDG from Figure 2c) and match compatible patterns (m is known from it. 2 to match a *map* pattern, and r is known from it. 1 to match a *tiled reduction* pattern).

Pattern Merging. This phase combines all matched patterns into a final set to be reported to the user. Smaller patterns subsumed by larger patterns are discarded from the final set. For example, the *linear reduction*, *tiled reduction*, and *map* patterns matched in its. 1-2 (Table 1) are discarded, as they are subsumed by the *tiled map-reduction* pattern.

6 Experimental Evaluation

This section reports experimental results on the effectiveness, scalability, and potential benefits of our analysis.

```
29                                  pthread_barrier_wait(&barrier);
30
31                                  float myhiz = 0;
32                                  for (long kk=k1; kk < k2; kk++ ) {
33                                    myhiz += dist(p[kk], p[0]) * p[kk];
    tiled_map_reduction                     fadd,fmul
34                                  }
35                                  hizs[pid] = myhiz;
36
37                                  pthread_barrier_wait(&barrier);
38
39                                  for(int i = 0; i < T; i++)   {
40                                    hiz += hizs[i];
    tiled_map_reduction                     fadd
41                                  }
```

Figure 6. Screenshot of a pattern finder report.

Implementation. The instrumenting compiler is implemented using LLVM [26] v7, hence our DDG nodes correspond to executions of LLVM IR operations. The instrumentation pass and runtime support reuse LLVM's *DataFlowSanitizer* dynamic analysis framework [44]. Our implementation extends this framework with ~700 additional C/C++ LOC, including runtime calls on the execution of each LLVM IR operation and on loop boundaries, and runtime support to trace the dynamic loop scope of each DDG node. In our experiments, tracing takes a negligible amount of time compared to pattern finding. Hence, we have not yet had to apply advanced tracing techniques such as those of Kim *et al.* [22].

The pattern finder is implemented in ~2400 Python LOC, and run in parallel whenever possible. The pattern definitions are implemented in the MiniZinc constraint modeling language [35] (~600 LOC), and matched using the Chuffed solver [10] v0.10. A time limit of 60 seconds is used for each solver run; increasing it did not yield additional patterns.

Our implementation outputs HTML reports where the found patterns are highlighted in the source code, as shown in Figure 6. The instrumented compiler and the pattern finder are available at https://github.com/robcasloz/llvm-discovery.

Setup. The evaluation is based on Starbench [2], a parallel C/C++ benchmark suite that covers a broad range of application domains and includes a sequential and an optimized Pthreads version of each benchmark. Compared to other parallel benchmark suites such as PARSEC [6], Starbench uses Pthreads as the primary parallel programming model and hence it better represents legacy parallel code. The evaluation includes all benchmarks in the suite except `bodytrack` and `h264dec`, which follow patterns (pipelines) out of our current scope. Effectiveness is evaluated with respect to the patterns found by earlier manual studies [2, 11]. For pattern finding, input parameters have been chosen that exercise the main computation of each benchmark while yielding

Table 2. Input parameters for each Starbench benchmark.

benchmark	input	parameters
{c-ray, ray-rot}	analysis	7 objects, 8×4 pixels
	reference	192 objects, 1920×1080 pixels
md5	analysis	4 buffers, 2×2 B/buffer
	reference	128 buffers, 1024×4096 B/buffer
{rgbyuv, rotate, rot-cc}	analysis	4×4 pixels
	reference	8141×2943 pixels
kmeans	analysis	8 pt., 2 dim., 2 clusters
	reference	17695 pt., 18 dim., 2000 clusters
streamc.	analysis	4 pt., 2 dim., 2 clusters
	reference	200000 pt., 128 dim., 20 clusters

small DDGs. Table 2 details the parameters chosen for pattern finding in comparison to the reference parameters in Starbench. The chosen parameters are, on average, three orders of magnitude smaller than the reference input.

The pattern finding evaluation uses an Intel Xeon E5-2680 v3 architecture with 12 CPU cores and 64 GB of main memory. The portability study (Section 6.3) complements this *CPU-centric* architecture (which also has a low-end NVIDIA NVS 310 GPU) with a *GPU-centric* architecture that has only 4 CPU cores (Intel Core i7-4770) but a high-end NVIDIA GeForce GTX Titan GPU. All time measurements correspond to the median of 20 repetitions, where all repeated measurements exhibit a robust coefficient of variation [43] (interquartile range relative to the median) of less than 10%.

6.1 Pattern Finding Effectiveness

Our pattern finder detects 86% of the 42 Starbench patterns reported by earlier manual studies [2, 3, 11]. Table 3 lists the 36 patterns found and six patterns missed for the sequential and Pthreads version of each Starbench benchmark.

Among the 36 found patterns, 27 are found in the first iteration, seven are found in the second iteration, and the remaining two are found in the third iteration. The seven patterns found in the second iteration can be put into two classes: (conditional) maps in `ray-rot` and `streamcluster` that result from subtracting first-iteration reductions to loop DDGs (as exemplified in Table 1, it. 2), and fused maps in `rot-cc` that result from fusing simpler first-iteration maps. The two patterns found in the third iteration in `streamcluster` are map-reductions that result from fusing first-iteration reductions with second-iteration maps (as exemplified in Table 1, it. 3). In summary, most patterns are found in the first iteration, and the remaining iterations find increasingly fewer, but larger and potentially more profitable patterns.

30 of the found patterns are maps, of which 11 are conditional maps and two are fused maps. Conditional maps arise,

Table 3. Found and missed parallel patterns in Starbench; *m*, *c*, *f*, and *r* stand for *map*, *conditional*, *fused*, and *reduction* (*linear reduction* for sequential and *tiled reduction* for Pthread benchmarks). Final reported patterns are highlighted in bold.

bench.	version	found it. 1	found it. 2	found it. 3	missed
{c-ray, md5, rgbyuv}	(both)	**m**	-	-	-
rotate	(both)	**cm**	-	-	-
kmeans	(both)	**r**	-	-	m[1], mr[2]
rot-cc	(both)	m, cm	**fm**	-	-
ray-rot	seq.	**m**, **cm**	-	-	fm[3]
	Pthr.	**cm**	**m**	-	fm[3]
streamc.	(both)	**m**, **cm×3**, r	**m**, m	**mr**	-

[1] Pattern output identified as an address calculation and simplified away.
[2] Missed underlying maps.
[3] Loops of the underlying maps have mismatching iteration spaces.

for example, in the rotation computation in `rotate`, `rot-cc`, and `ray-rot`, where input pixels are transformed and output only if they appear in the final rotated image. The two fused maps arise naturally in `rot-cc` from the composition of rotation and color transformation computations applied to each pixel in the input image. Remarkably, the fused maps combine loops located in different translation units. The four reductions found in `kmeans` and `streamcluster` correspond to additions of distances or coordinates of individual points. Finally, the two map-reductions found in `streamcluster` combine the computation of distances between points (map) with their addition (reduction) as illustrated in Section 2.

The six missed patterns illustrate two current limitations in our heuristics. The first one affects the two maps in `kmeans` and inhibits finding their encompassing map-reductions, which form the core of the `kmeans` computation. The map pattern computes the index of the cluster assigned to each point. These indices are used exclusively in memory address calculations which are simplified away by the DDG simplification phase. This simplification removes the outgoing arcs from the candidate map pattern and precludes matching it, which in its turn precludes finding the encompassing map-reduction. The second limitation concerns the two missing fused maps in `ray-rot`. The loops corresponding to the underlying maps have mismatching iteration spaces: the ray-tracing loop ranges over the dimensions of the generated image, whereas the rotation loop ranges over the (larger) dimensions of the rotated image. This mismatch results in loop DDGs with different number of nodes, the fusion of which is not recognized as a fused map by the pattern finder. These

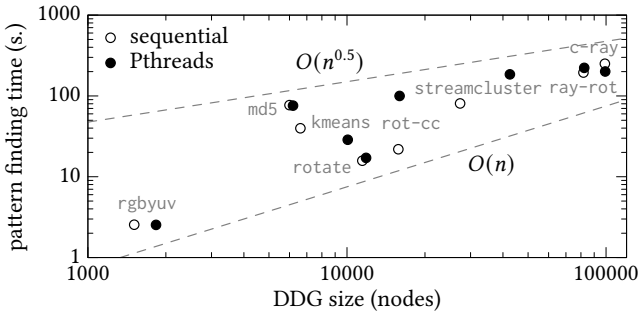


Figure 7. Pattern finding time by DDG size.

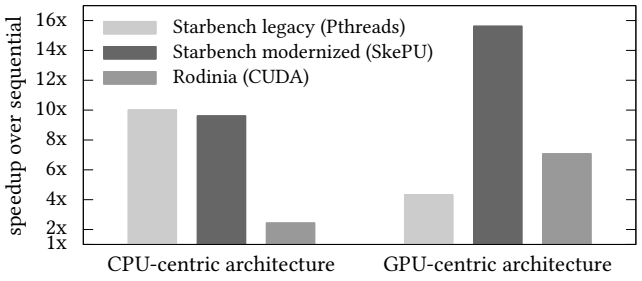


Figure 8. Speedup of Pthreads legacy, modernized, and Rodinia `streamcluster` on two architectures. The baseline is sequential execution time on the CPU-centric architecture.

two limitations are not fundamental issues in our core approach, but rather restrictions imposed by the simplification and decomposition heuristics introduced in Section 5.

Remarkably, the same patterns are found in both versions of all benchmarks. This shows that our analysis is oblivious to whether the code is sequential or parallel.

Accuracy. The pattern finder reports 50 additional patterns besides those in Table 3. A manual analysis of the additional patterns reveals that our approach is highly accurate, while confirming the need for human intervention: 48 of the 50 additional patterns are *true* patterns (they apply to any program input), whereas the remaining two are *false* patterns that only apply to the input used for pattern finding.

The 48 true patterns are formed by (conditional) maps (36), reductions (11), and one map-reduction. As can be expected from a well-studied suite such as Starbench, the additional patterns are in general unlikely to yield parallelization profit. The most remarkable cases are nested maps (for example in `c-ray` and `ray-rot`) that are not parallelized in the Pthreads versions but might be parallelized profitably on architectures that benefit from finer granularity, such as GPUs.

The two false patterns are maps in `streamcluster`. These patterns are reported on loops which also contain a conditional reduction that is not triggered by the program input. As a result, no dependency is registered across loop iterations, the component independence constraints (2b) are satisfied, and a map pattern is matched and reported.

6.2 Pattern Finding Scalability

Our pattern finder detects the patterns reported in Table 3 in seconds to minutes. Figure 7 shows pattern finding time, including tracing, for each benchmark version by DDG size. The results indicate that our pattern finder scales linearly with DDG size, as indicated by the dashed bounds in Figure 7. Other factors such as the number of distinct executed loops are also likely to affect pattern finding time.

Tracing and matching take 1% and 48% of the total time. The remaining 51% is spent on the other pattern finding phases, which have low computational complexity. This suggests that there is ample opportunity for optimization.

Pthreads versions yield DDGs that are on average 15% larger than their corresponding sequential versions, resulting in an average pattern finding slowdown of 28%.

6.3 Portability of the Modernized Code

This section studies the modernization of Pthreaded `streamcluster` as a demonstration of the benefits of applying our analysis to legacy parallel code. `streamcluster` is among the most complex benchmarks in Starbench, both in lines of code and in number and diversity of parallel patterns. The eight patterns found in this benchmark have been ported manually to SkePU, a parallel pattern library [36]. Automating the port itself is part of future work. The ported code is freely portable across CPUs and GPUs at execution time.

Figure 8 shows the parallel speedup of different parallelizations of `streamcluster` executed with its reference input on the two evaluation architectures. The results show that modernized code can leverage the resources of different hardware architectures and compete with implementations that are hand-tuned for a specific architecture. On the CPU-centric architecture the modernized version (executed on the CPU) achieves a speedup (9.6x) comparable to that of the legacy Pthreads version (10x). The GPU in this architecture has few cores and low memory bandwidth, which is reflected on the poor speedup (2.4x) achieved by a native CUDA implementation from the Rodinia benchmark suite [9].

The portability benefits of the modernized version are clearly seen when executing the same versions on a GPU-centric architecture. Here, the legacy Pthreads version performs poorly (4.3x) due to the low number of CPU cores, while the modernized version achieves the best speedup (15.6x) by fully exploiting the computational power of the GPU. Interestingly, while Rodinia's version also improves significantly on this architecture, it does not exploit the GPU to the same extent, and only achieves a speedup of 7.1x. The poorer performance of Rodinia's version compared to our modernized version might be due to Rodinia's use of specific optimizations and parameters for the GPU (NVIDIA GTX 280) that the code was originally written for.

7 Related Work

There is a large body of work on automatic and semi-automatic parallelization of *sequential* legacy code [32, 39], employing complex static and dynamic analysis to discover opportunities for simple *do-all* parallelism within sequential loops. In contrast, analysis of parallel code and discovery of parallel patterns are relatively unexplored areas.

Pattern Finding. Paraprox [42] analyzes OpenCL/CUDA kernels to detect the presence of six parallel patterns (map, scatter/gather, reduction, scan, stencil, partition). The analysis is used to transform the code into faster versions using approximation techniques in domains where these are acceptable. The recognition phase relies heavily on the highly structured nature of GPU kernels. In contrast, our analysis is fundamentally source independent, with any source-related techniques being merely performance enhancing heuristics.

Pattern detection has been exploited without subsequently tying back to the source code. Poovey [37] and Deniz [13] instrument parallel code with low-level counters and statistically correlate these to profiles of code representative of predefined patterns. The results are used to select architectural and low-level system software policies. Since the pattern analysis is statistical, the selected policies can only be allowed to impact performance, not correctness. The patterns identified are soft, in the style of Mattson [30], i.e. they capture informal characteristics of the algorithms and cannot be tied back to the source code. The technique has also been applied to synthetic benchmark generation, following the statistical profile of predefined pattern representatives [14]. Patterns can also be detected in legacy message-passing code [23, 24, 38]. A typical application is to find collective operations such as broadcasts, which the programmer has hand-coded with point-to-point operations. Detected instances are replaced with calls to a highly tuned collective library.

Dynamic Analysis. Dynamic Dataflow Graphs were introduced in Redux [33] and applied to debugging and slicing of sequential programs (see Section 3). Tracing and dynamic profiling have been used to capture application properties for benchmark characterization [15], collaborative runtime verification [7] and dynamic program optimization [5].

In common with our trace-then-match strategy, both [41] and [20] build dynamic graphs but aggregate data and control-flow information from multiple executions into single nodes. This assists with compaction and hence graph size, but is in tension with the discovery of less regular patterns. These approaches are only applicable to *sequential* code. DiscoPoP [28] builds upon Valgrind [34] to trace and build a dataflow analysis of executing sequential code, to determine the potential presence of simple parallel-do parallelism. Similarly, Parwiz [21] combines static and trace-driven dynamic analysis to provide parallelization hints to the programmer. Both DiscoPoP and Parwiz focus on detecting parallel-do parallelism. This limited scope allows them to aggregate all executions of each operation in their trace representation and to resort to custom greedy algorithms as opposed to more general constrained matching methods. Poly-Prof [17] is based in the polyhedral idiom, strengthening inter-procedural analysis through compact tracing, and feeding into a polyhedral compiler. Tracing has also been used to detect the potential presence of structured pipeline parallelism in sequential loops [46, 47]. All of these projects use dynamic tracing to augment static analysis, and are consequently, like our work, programmer-advisory in nature. The work on collective operation finding in MPI programs discussed above [23, 24, 38] dynamically traces point-to-point message passing calls.

8 Limitations

The design choices of using DDGs and applying the heuristics from Section 5 are key for effective and efficient pattern finding, but at the same time impose a number of limitations.

The main limitation of DDGs is that their size is proportional to execution time. Hence, input parameters need to be chosen carefully to exercise sufficiently a program during short executions. Such parameters might be readily available, for example, in the program's test suite. Otherwise, symbolic methods could be applied [4]. Another limitation of DDGs is the inability to match patterns based on computations such as swaps or min/max calculations, which are typically expressed as conditional data transfer operations. This limitation could be mitigated by applying *if-conversion* [1]. The limitations of the pattern finder heuristics, discussed in Section 6.1, could be mitigated by further refinements.

9 Conclusion and Future Work

This paper has introduced a dynamic analysis for modernizing legacy sequential and parallel code through parallel pattern finding. Experiments on Starbench benchmarks show that our analysis finds patterns effectively, accurately, and efficiently in legacy Pthreaded C/C++ code, and demonstrate how our approach facilitates re-targeting such code to exploit increasingly heterogeneous hardware.

Future work includes: addressing the limitations in Section 8; unifying the definition of *linear* and *tiled* patterns; characterizing more parallel patterns such as *pipeline* and *stencil*; further exploiting the dynamic nature of our analysis to propose *partial patterns* (which only apply under certain execution conditions) and *quasi-patterns* (which might be converted into patterns by simple transformations); and improving the scalability of the pattern finder with techniques from constraint modeling [40], frequent pattern mining [18], graph query processing [27], or instruction selection [19].

Acknowledgments

The authors are grateful for the support provided by C. Vasiladiotis, P. Metzger, A. Ernstsson, and RISE SICS; and for

helpful discussions with C. Kessler, C. McCreesh, J. Enright, and the anonymous reviewers. This research has been funded by the EPSRC (UK) under grant EP/P022642/1.

References

[1] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Principles of Programming Languages*. ACM.

[2] Michael Andersch, Chi Ching Chi, and Ben Juurlink. 2012. Using OpenMP superscalar for parallelization of embedded and consumer applications. In *Embedded Computer Systems*. IEEE.

[3] Michael Andersch, Ben Juurlink, and Chi Ching Chi. 2011. A benchmark suite for evaluating parallel programming models. In *Parallel Systems and Algorithms*. Springer.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (2018).

[5] Marc Berndl and Laurie Hendren. 2003. Dynamic Profiling and Trace Cache Generation. In *Code Generation and Optimization*. IEEE.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Parallel Architectures and Compilation Techniques*. ACM.

[7] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. 2007. Collaborative Runtime Verification with Tracematches. In *Runtime Verification*. Springer.

[8] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Parallel Architectures and Compilation Techniques*. IEEE.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*. IEEE.

[10] Geoffrey G. Chu. 2011. *Improving combinatorial optimization*. Ph.D. Dissertation. The University of Melbourne, Australia.

[11] Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. 2017. Bringing Parallel Patterns Out of the Corner: The P^{3}ARSEC Benchmark Suite. *ACM Transactions on Architecture and Code Optimization* 14, 4 (2017).

[12] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008).

[13] Etem Deniz and Alper Sen. 2016. Using Machine Learning Techniques to Detect Parallel Patterns of Multi-threaded Applications. *International Journal of Parallel Programming* 44, 4 (2016).

[14] E. Deniz, A. Sen, B. Kahne, and J. Holt. 2015. MINIME: Pattern-Aware Multicore Benchmark Synthesizer. *IEEE Trans. Comput.* 64, 8 (2015).

[15] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. 2003. Dynamic Metrics for Java. In *Object-Oriented Programing, Systems, Languages, and Applications*. ACM.

[16] August Ernstsson, Lu Li, and Christoph Kessler. 2018. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming* 46, 1 (2018).

[17] Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Antoine Moynault, Louis-Noël Pouchet, and Fabrice Rastello. 2019. Data-flow/Dependence Profiling for Structured Transformations. In *Principles and Practice of Parallel Programming*. ACM.

[18] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. 2007. Frequent Pattern Mining: Current Status and Future Directions. *Data Mining and Knowledge Discovery* 15, 1 (2007).

[19] Gabriel Hjort Blindell. 2016. *Instruction Selection: Principles, Methods, and Applications*. Springer.

[20] Zia Ul Huda, Rohit Atre, Ali Jannesari, and Felix Wolf. 2016. Automatic Parallel Pattern Detection in the Algorithm Structure Design Space. In *International Parallel and Distributed Processing Symposium*. IEEE.

[21] Alain Ketterlin and Philippe Clauss. 2012. Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization. In *IEEE/ACM International Symposium on Microarchitecture*. IEEE.

[22] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. 2010. SD3: A Scalable Approach to Dynamic Data-Dependence Profiling. In *IEEE/ACM International Symposium on Microarchitecture*. IEEE.

[23] Andreas Knüpfer, Dieter Kranzlmüller, and Wolfgang Nagel. 2004. Detection of Collective MPI Operation Patterns. In *European PVM/MPI Users' Group Meeting*. Springer.

[24] Dieter Kranzlmüller, Andreas Knüpfer, and Wolfgang Nagel. 2004. Pattern Matching of Collective MPI Operations. In *Parallel and Distributed Processing Techniques and Applications*. CSREA Press.

[25] Jim Kukunas. 2015. *Power and Performance*. Morgan Kaufmann.

[26] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization*. IEEE.

[27] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-Depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proceedings of the VLDB Endowment* 6, 2 (2012).

[28] Zhen Li, Ali Jannesari, and Felix Wolf. 2013. Discovery of Potential Parallelism in Sequential Programs. In *Parallel Processing*. IEEE.

[29] Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. 2018. Generalized Profile-Guided Iterator Recognition. In *Compiler Construction*. ACM.

[30] Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*. Addison-Wesley.

[31] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann.

[32] S. Midkiff. 2012. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool.

[33] Nicholas Nethercote and Alan Mycroft. 2003. Redux: A Dynamic Dataflow Tracer. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003).

[34] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Programming Language Design and Implementation*. ACM.

[35] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *Principles and Practice of Constraint Programming*. Springer.

[36] Tomas Öhberg, August Ernstsson, and Christoph Kessler. 2019. Hybrid CPU-GPU execution support in the skeleton programming framework SkePU. *The Journal of Supercomputing* (2019).

[37] Jason A. Poovey, Brian P. Railing, and Thomas M. Conte. 2011. Parallel Pattern Detection for Architectural Improvements. In *Hot Topics in Parallelism*. USENIX.

[38] R. Preissl, B. R. d. Supinski, M. Schulz, D. J. Quinlan, D. Kranzlmüller, and T. Panas. 2010. Exploitation of Dynamic Communication Patterns through Static Analysis. In *Parallel Processing*. IEEE.

[39] S. Prema, Rupesh Nasre, R. Jehadeesan, and B. K. Panigrahi. 2019. A study on popular auto-parallelization frameworks. *Concurrency and Computation: Practice and Experience* 31, 17 (2019).

[40] Francesca Rossi, Peter van Beek, and Toby Walsh (Eds.). 2006. *Handbook of Constraint Programming*. Elsevier.

[41] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. 2010. A Profile-based Tool for Finding Pipeline Parallelism in Sequential Programs. *Parallel Comput.* 36, 9 (2010).

[42] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *Architectural Support for Programming Languages and Operating Systems*. ACM.

[43] Howard M. Shapiro. 2003. *Practical Flow Cytometry*. Wiley.

[44] The Clang Team. 2020. *DataFlowSanitizer*. https://clang.llvm.org/docs/DataFlowSanitizer.html

[45] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. 2007. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *IEEE/ACM International Symposium on Microarchitecture*. IEEE.

[46] Georgios Tournavitis and Björn Franke. 2010. Semi-automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information. In *Parallel Architectures and Compilation Techniques*. ACM.

[47] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. 2009. Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping. In *Programming Language Design and Implementation*. ACM.

[48] Michael Voss, Rafael Asenjo, and James Reinders. 2019. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. Apress.

[49] Mani Zandifar, Mustafa Abdul Jabbar, Alireza Majidi, David Keyes, Nancy M. Amato, and Lawrence Rauchwerger. 2015. Composing Algorithmic Skeletons to Express High-Performance Scientific Applications. In *International Conference on Supercomputing*. ACM.