Parallelizing Parallel Programs: A Dynamic Pattern Analysis for Modernization of Legacy Parallel Code

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

1 INTRODUCTION

Parallelization traditionally refers to the challenge of analyzing legacy sequential code, to generate equivalent, higher performing parallel code. Emerging mainstream parallelism has created a new challenge: *legacy parallel code*, typically hand-optimized for a particular system, becomes outdated as the hardware evolves. Modernizing such code line-by-line is not effective: instead we need to understand the program's overall algorithmic intent, to re-express it for the new target. Re-expression can be simplified and future-proofed by coding with parallel patterns [4, 8, 9]. We address the remaining challenge of identifying the algorithm patterns in the original parallel source. We describe a novel dynamic approach to the identification of implicit algorithmic patterns. Our core principle is that the essence of a pattern is to be found in the dynamic data flows it invokes between operations and their repetition, rather than any specific source level encoding. This makes it neutral with respect to source code and hence immediately applicable to both legacy sequential and parallel code.

The high level structure of our approach is captured in Figure 1. Our LLVM pass instruments the legacy program, whose traced execution generates a *dynamic dataflow graph (DDG)*. Our constraint-programming based pattern finding tool analyzes the DDG against a library of pattern definitions, reporting found instances back to the programmer.

2 PARALLEL PATTERN FINDING

Figure 2 steps through our approach with an example extracted from a legacy parallel program, `streamcluster`, from the PARSEC [2] benchmark suite. The algorithm underlying `streamcluster` applies the MapReduce pattern [4]. Figure 2a shows key excerpts from the original code, including the threading infrastructure (note that function `pkmedian` is invoked by each thread in parallel), and highlighted, the key data operations which implicitly encode the pattern. Figure 2b visualizes the resulting DDG, obtained after some simplifications discussed below. In this simple example the MapReduce pattern is already clear to the experienced eye, but its automatic identification is out of the scope of existing methods. Finally, Figure 2c shows the modernized code, here calling

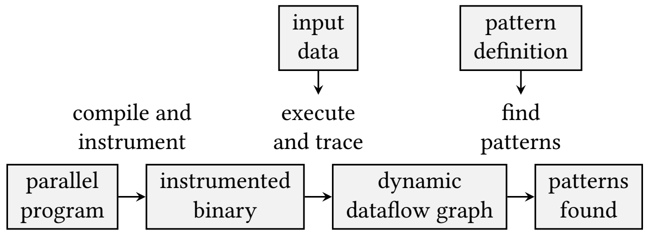


Figure 1: Overview of our parallel pattern finding analysis.

the `MapReduce` function from the SkePU 2 pattern library [5]. The code transformation between the two versions would be carried out either manually, informed by the feedback from our tool, or ideally within a refactoring tool armed with the same pattern knowledge.

Dynamic dataflow graphs. Our DDGs are generated by execution of legacy code which has been instrumented by a new LLVM compiler pass. In contrast to conventional DDGs, our raw graphs (inspired by [6]) have a node for *every dynamic execution* of an operation, rather than simply for each static operation, and edges for *every flow of data between these*. This makes them very bulky, and we employ a number of post-hoc simplifying heuristics to reduce size, including removal of sections corresponding to address calculation and data-structure traversal, since these are artifacts of the particular source encoding. In line with our core principle, these guide the operations which generate the true dataflow, but they are not part of it, and are therefore orthogonal and irrelevant to the patterns. We can also, heuristically, condense sub-graphs corresponding to entire loop iterations. These heuristics have been applied to the DDG sketched in Figure 2b.

Pattern definitions. Our pattern definitions are implemented in the MiniZinc constraint modeling language [7] (~400 LOC), and matched using the Chuffed constraint solver [3]. MiniZinc allows us to express, as properties of an encoding of the DDG, both what it means to be a pattern in general, and what further constraints are required to capture the specifics of each distinct pattern. Our current implementation has rules capturing a variety of flavors of both Map and Reduce patterns, as reported in Table 1.

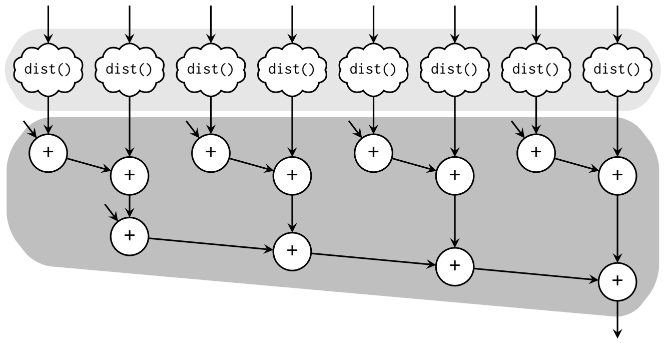
3 EXPERIMENTAL RESULTS

Table 1 presents our results for multithreaded code from the Starbench suite [1], omitting two programs, `bodytrack` and `h264dec`, which follow a pattern (pipeline) that is out of the scope of our current analysis. We found all but one expected patterns, known by

PACT '20, October 3–7, 2020, Virtual Event, GA, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8075-1/20/10.
https://doi.org/10.1145/3410463.3414663

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

(a) Original distance computation with `nproc` parallel calls to `pkmedian` and manual thread management, synchronization, and workload splitting.



(b) Partial, simplified DDG of the original parallel distance computation for eight points and four threads.

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

(c) Modernized distance computation with a single call to `pkmedian` and transparent parallelization.

Figure 2: Parallel distance computation in `streamcluster.c`. The patterns found by our analysis are colored in light gray (*map*) and dark gray (*reduction*).

Table 1: Found and missed parallel patterns in Starbench.

benchmark	found	missed
`c-ray`	map	-
`md5`	map	-
`rgbyuv`	map	-
`rotate`	conditional map	-
`kmeans`	reduction	map
`rot-cc`	map, conditional map	-
`ray-rot`	map, conditional map	-
`streamcluster`	map×4, conditional map×3, reduction	-

manual inspection and from the literature, and understand the reason for missing one map in kmeans, which will inform subsequent versions. Our analysis ran from a few seconds to a few minutes, depending upon the size and complexity of the DDGs. The resulting SkePU parallelized code performed well, comparably to hand-coded performance from benchmark implementations (e.g. 9.6x speed-up over sequential on a Xeon E5-2680 v3 with twelve hyperthreaded cores, 15.6x on an NVIDIA GeForce GTX Titan with 2688 cores), and was trivially portable across a range CPU/GPU systems.

ACKNOWLEDGMENTS

The authors are grateful for the support provided by Christos Vasiladiotis, Paul Metzger, August Ernstsson, and RISE SICS; and for helpful discussions with Christoph Kessler, Ciaran McCreesh, and Jessica Enright. This research has been funded by the Engineering and Physical Sciences Research Council (UK) under grant EP/P022642/1.

REFERENCES

[1] M. Andersch, C. C. Chi, and B. Juurlink. 2012. Using OpenMP superscalar for parallelization of embedded and consumer applications. In *Embedded Computer Systems*. IEEE.
[2] 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.
[3] Geoffrey G. Chu. 2011. *Improving combinatorial optimization.* Ph.D. Dissertation. The University of Melbourne, Australia.
[4] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008).
[5] 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).
[6] Nicholas Nethercote and Alan Mycroft. 2003. Redux: A Dynamic Dataflow Tracer. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003).
[7] 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.
[8] Michael Voss, Rafael Asenjo, and James Reinders. 2019. *Pro TBB: C++ Parallel Programming with Threading Building Blocks.* Apress.
[9] Mani Zandifar, Mustafa A. Jabbar, Alireza Majidi, David Keyes, Nancy M. Amato, and Lawrence Rauchwerger. 2015. Composing Algorithmic Skeletons to Express High-Performance Scientific Applications. In *Int. Conf. on Supercomputing*. ACM.