PetaBricks: A Languages and Compiler for Algorithmic Choice

Jason Ansel et al.

Presented by Harri Bell-Thomas

19/11/2019

Motivating Example

Sorting — there is no *single best* sorting algorithm. It depends on context (input size, data types etc).

Optimal performance comes from composing different approaches. For example the GNU C++ standard library's `std::sort` method is a composition of three different sorting algorithms.

1. *Quicksort*, subdividing a maximum of $2\log_2(n)$ times.
2. Then continue sorting using *Heapsort*.
3. Any partition of 16 elements or fewer is sorted quickly using *Insertion sort*.

How do we choose these cutoff points?

Headline Concepts

It is common for the best solution to a problem to be a *hybrid algorithm*, exploiting different properties in different contexts.

Therefore;

- Give this compiler the ability to make *algorithmic choices.*
- Let the programmer specify *accuracy choices* to aid algorithmic optimisation — optimal efficiency for any level of accuracy.
- Automatically parallelise solutions where possible.
- Let the autotuner be aware of different hardware architectures and optimise for their strengths/consider their weaknesses.

PetaBricks Language

C++ derivative. Two major components; *transforms* and *rules*.

```
transform kmeans
from Points[n, 2] // Array of 2D points.
through Centroids[sqrt(n), 2]
to Assignments[n]
{
    // Rule 1. Points -> Centroids.
    to (Centroids.column(i) c) from (Points p) {
        c = p.column(rand(0, n));
    }

    // Rule 2. Points and Centroids to Assigments.
    to (Assignments a) from (Points p, Centroids c) {
        while (true) {
            int change;
            AssignClusters(a, change, p, c, a);
            if (change == 0) return; // Reached fixed point.
            NewClusterLocations(c, p, a);
        }
    }
    ...
}
```

Components

1. Source-to-source compiler (PetaBricks $\rightarrow$ C++).

   Performs static analysis of transforms and encodes choices/tunable parameters into the output code.

2. Runtime library to aid generated code.

3. Autotuning system and choice framework.

   Autotuning happens at either compile time (via a configuration file) or installation time.

Compiler

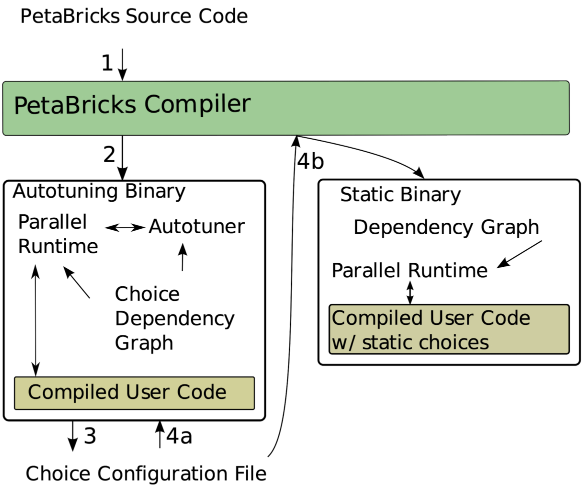


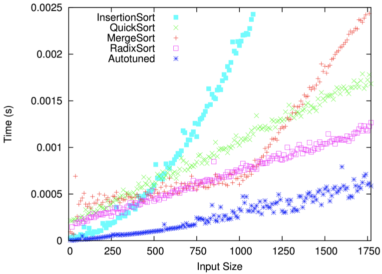
Figure: PetaBricks compiler overview.

Comments

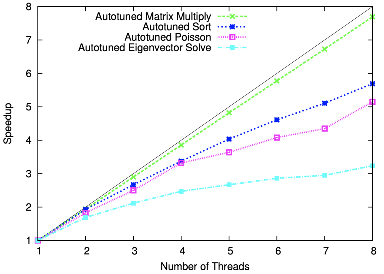
- Autotuning.
  - Bottom-up approach.
  - “Spirit of a genetic tuner”.
  - (No mention of example compile-times).
- Automated consistency checks ✓
- No deadlocks or race conditions ✓
- Automated/implicit parallelising of output code — runtime library operates a work-stealing dynamic scheduler, makes use of “continuation points”. ✓

Results

Very thorough analysis based on real-world problems, good to see.[1]



(a) Performance of the autotuned PetaBricks-sort algorithm.



(b) Scalability factor for example PetaBricks programs.

[1] *cough* Google *cough* Facebook *cough*

Critique

In summary — excellent design and promising results.

**with a few questionable parts subtly glossed over.*

There are a number of novel and important contributions to note, including;

- Automated performance tuning before this did not consider exploring optimisations on top of dynamic algorithm combinations.
- It is claimed that this is the first algorithm-composition engine that considers and optimises for desired accuracy.
- A viable approach for creating high-performance optimised programs that are also *portable*.
- Nothing is assumed about the hardware prior to optimisation. As a result programs can be re-optimised as architectures change over time.

Criticism

- No mention of compile time — the original PhD thesis reveals that this in the order of half to an entire day for the example algorithms.
  - Does this greatly limit the contexts in which this can be usefully applied? How long to optimise on a Raspberry Pi?
- The paper claims the approach is suitable for trees and sparse structures, but offers no explanation or evidence for this. The implementation specifics appear highly dependent on matrix-based representations.
  - There is an implicit reliance on algorithms that are sub-dividable or can produce compatible partial solutions/intermediate forms — mentioned in passing halfway through.
- Most content in this paper is copied verbatim from the original thesis, leaving out key explanations. Without a working knowledge of domain-specifics this is a tricky read.