R244 · SESSION 2 · SEARCH SPACE & OPTIMISATION

PetaBricks: A Language and Compiler for Algorithmic Choice

J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, S. Amarasinghe

MIT CSAIL · PLDI 2009

Joshua Fehn · 21 October 2026

MOTIVATION

No single sorting algorithm wins at every input size

Small input	Medium input	Very large input
Insertion sort	**Quicksort**	**Radix sort**

The best algorithm is a composition: decompose with quick or radix sort, then hand small sub-problems to insertion sort.

15

elements: where STL sort hard-codes its switch to insertion sort

60–150

elements: the cutoffs the authors found faster on current (back in 2009) machines

PRIOR WORK

Before PetaBricks, someone else fixed the choice

Library autotuners

ATLAS, FFTW, SPIRAL, OSKI tune a training phase per machine.

Only for the few algorithms the library designer wrote. The application cannot add its own.

Hard-coded cutoffs

STL sort: insertion sort below 15 elements.

LAPACK eigensolver (dstevd): switch at n = 25.

One number for every machine, buried in control flow.

PetaBricks

Any programmer lists the candidate algorithms in the language.

The compiler composes them at any recursion level and tunes the switch points per machine.

CORE IDEA

Make algorithmic choice a first-class construct of the language

The programmer says which algorithms are valid. The compiler and autotuner decide which one runs, on which region of the data, and when.

ARCHITECTURE

Four building blocks: language, compiler, autotuner, runtime

When you write it

Language

Transforms with several rules: the alternative algorithms.

Before it runs

Compiler

Analyses dependencies, builds the choice graph, emits C++.

Once per machine

Autotuner

Times candidates on the target and writes the config file.

Every run

Runtime library

Runs the chosen code in parallel; handles data and config.

The choice dependency graph links them: the compiler builds it based on the code written by the dev, the autotuner reads it to know what can be tuned, the runtime reads it to know what can run in parallel.

EXAMPLE · RUNNING SUM

Same result, two algorithms

Recompute: sum everything to the left

`B[i] = A[0] + … + A[i]`



additions: **9** cells busy at once: **4**

All unfinished cells add A[2] at once; B[2] is now final.

Running total: reuse the previous sum

`B[i] = A[i] + B[i−1]`



additions: **2** cells busy at once: **1**

B[2] waits for B[1]: only one cell can work.

being read being computed done

THE LANGUAGE

Rules declare dependencies, not order

```
transform RollingSum
from A[n]
to B[n]
{
  // rule 0: sum all elements to the left
  to(B.cell(i) b) from(A.region(0, i) in) {
    b = sum(in);
  }
  // rule 1: reuse the previous value
  to(B.cell(i) b) from(A.cell(i) a,
                       B.cell(i-1) leftSum) {
    b = a + leftSum;
  }
}
```

Rule 0 · recompute

$\Theta(n^2)$ work, but every output is independent, so it is data-parallel.

Rule 1 · running total

$\Theta(n)$ work, but each output waits for the one before it, so it is sequential.

Neither rule says when to run: the compiler finds a legal order, the autotuner picks a rule per region.

'where': limit a rule to cells that meet a condition, e.g. i == j

'tunable': expose a number to the autotuner, e.g. a merge-sort branching factor

Code: Figure 3 of the paper. Also in the language: rule priorities, template transforms, embedded C++.

COMPILER · STEPS 1–2

From rules to regions

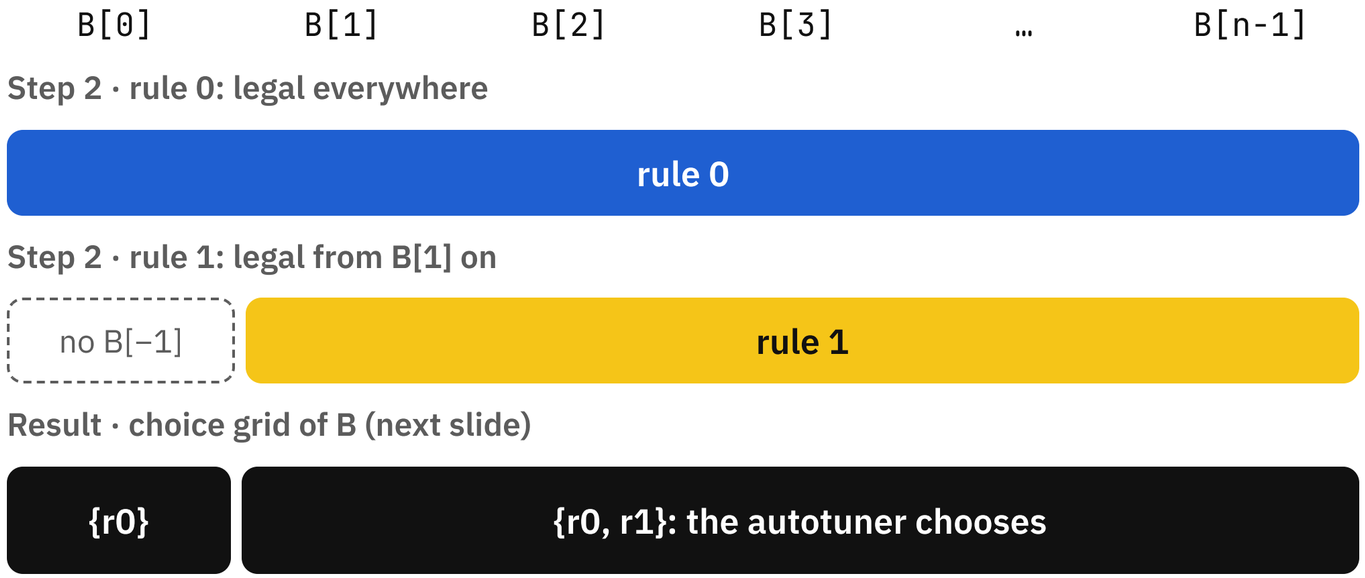
Parse & normalise → Applicable regions → Choice grid → Choice dependency graph → Code generation

1 Parse & normalise

Rewrite each dependency relative to the cell being computed, B[i]: rule 0 reads A[0..i], rule 1 reads A[i] and B[i−1].

2 Applicable regions

A rule may only run where everything it reads exists. There is no B[−1], so rule 1 cannot compute B[0].



COMPILER · STEPS 3–4

The choices form an explicit graph

Parse & normalise → Applicable regions → **Choice grid** → **Choice dependency graph** → Code generation

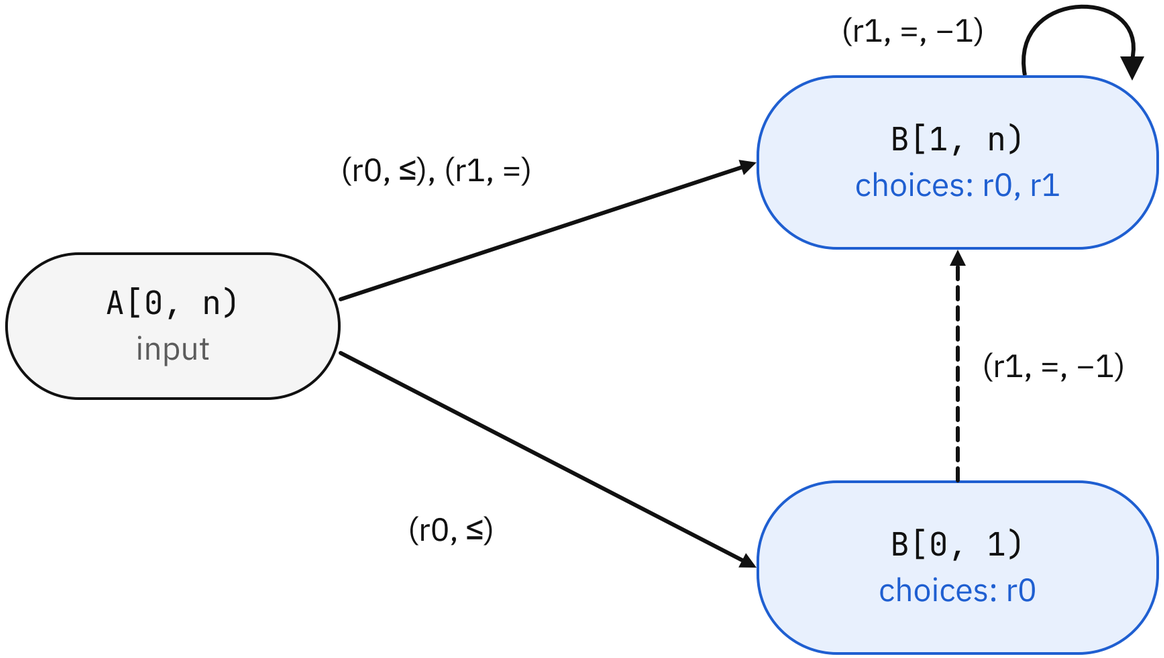
Choice grid for RollingSum's output B: rule 1 reads B[i−1], so it cannot compute B[0].

Region of B	Legal rules
[0, 1)	rule 0
[1, n)	rule 0, rule 1

Because every dependency is known

No deadlocks: a cycle with no valid loop direction is a compile error.

No races: only tasks with no path between them run in parallel.



COMPILER · OUTPUT AND MODES

Compile once, tune on the target, then fix the choices



Compiler mode	1 · Tuning (default)	2 · Static (config fed back)
Input	PetaBricks source	source + choice config file
Output	autotuning binary	static binary
Inside the binary	all choices + autotuner + runtime	chosen rules only + runtime
Choice made	at run time, read from the config	at compile time, built in
Speed	small overhead for switching	faster: C++ compiler sees fixed code

AUTOTUNER · IN THE SPIRIT OF A GENETIC ALGORITHM

The autotuner builds algorithms bottom-up

1

POPULATION

Seed with every single-algorithm candidate on a small input.

2

EVALUATE

Double the input size and time every candidate.

3

EXTEND

Add a level on top of the fastest ones: switch algorithm above a cutoff.

4

SELECT · REPEAT

Drop the slowest until the population is under a size limit.

Other parameters

n-ary search over cutoffs, block sizes and tunables; leaves of the graph first.

Consistency checking

All candidates run on the same input; their outputs must match (within a threshold for iterative methods).

No optimality guarantee

Assumes the best sub-solution stays best and prunes greedily, so it can stop at a local optimum.

AUTOTUNER · VARIABLE ACCURACY

Accuracy becomes a tuning dimension

The problem

Many numerical solvers are approximate: they repeat a step until the error is small enough. More steps give a more accurate answer, but take longer. Comparing them on speed alone is unfair.

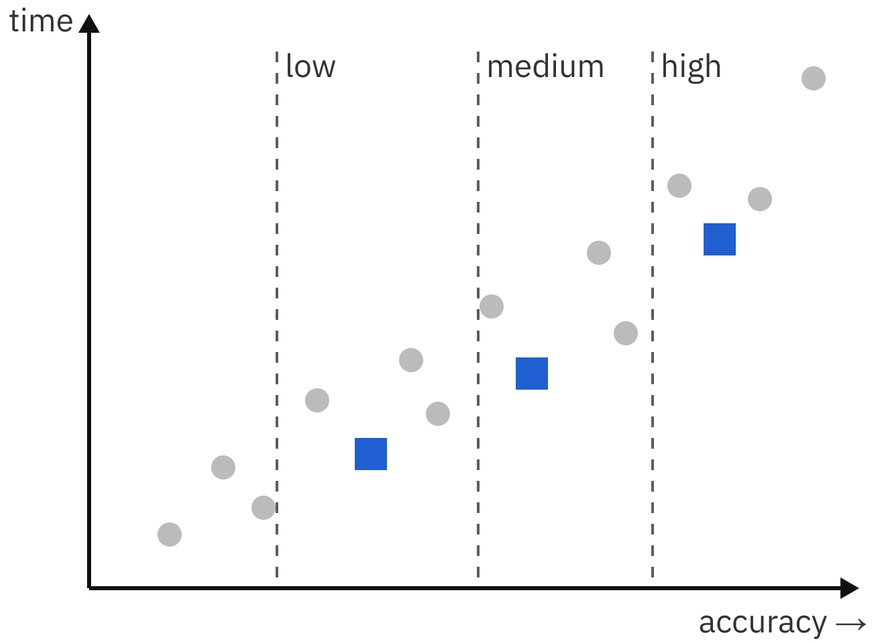
PetaBricks' answer

You state the accuracy you need. For each accuracy target, the tuner keeps the fastest candidate that reaches it.

Inside a recursion

A sub-step may use a cheaper, less accurate setting, as long as the final answer still meets the target.

Benchmark: a solver for Poisson's equation (heat or potential on a grid).



Dashed lines: accuracy targets. Blue: the fastest candidate that meets each target (after Fig. 9a).

RUNTIME LIBRARY

The runtime executes the choices in parallel

Work-stealing scheduler

One deque per core. A thread takes its newest task (depth-first, cache-friendly).

An idle thread steals the oldest, biggest task from a random other thread.

Two versions of each function

The compiler emits a parallel task version and a plain sequential one.

Below a tuned size, the runtime switches to sequential: no scheduler overhead.

Data and configuration

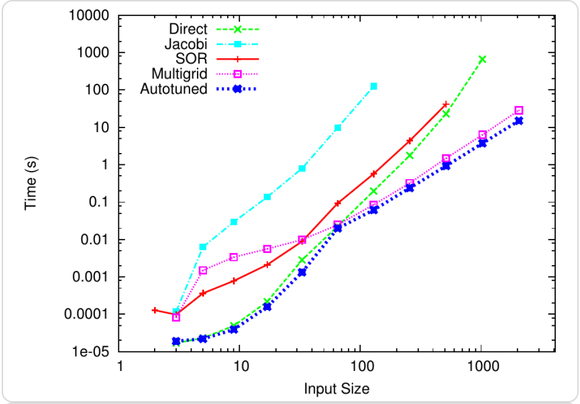
Matrices and regions, program input and output.

Loads the config file; hosts the autotuner in tuning mode.

Linked into every PetaBricks binary, including static ones.

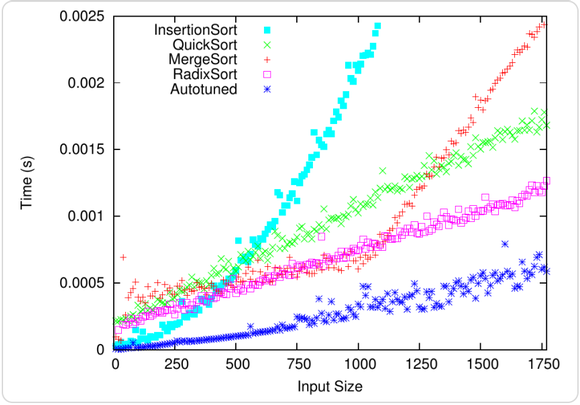
RESULTS · 8-CORE XEON · TUNED VERSION IN DARK BLUE ✱

Tuned hybrids beat every single algorithm



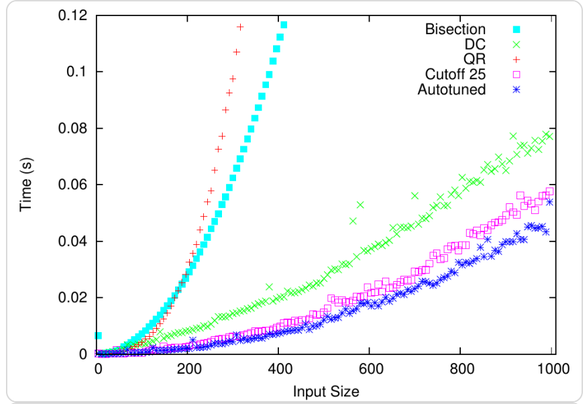
Poisson (log scale)

Direct solve for small grids, then multigrid.



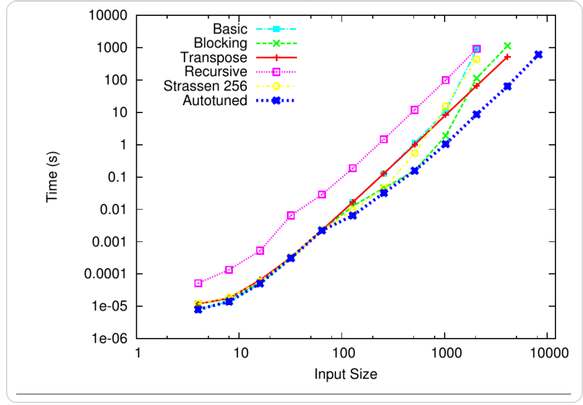
Sort

2-way merge → quicksort → insertion sort; radix sort unused.



Eigenproblem

Divide & conquer above n = 48, QR below; beats LAPACK's cutoff of 25.



Matrix multiply (log scale)

Mix of blocking, transposing and recursive splits.

PORTABILITY · SORT BENCHMARK

The best program depends on the machine

Running on the 8-core Xeon, tuned on …

… the 8-core Xeon itself

1.00×

… one core of the same Xeon

2.14× slower

… a different machine (Sun Niagara)

2.35× slower

1.68×

average slowdown when a configuration runs on a machine it was not tuned on

Porting means re-tuning, not rewriting: same source, tuned once per machine.

ASSESSMENT

Strengths and limitations

Strengths

Choice is part of the language

Any programmer can expose alternatives, at any recursion level.

Accuracy tuned alongside speed

Approximate algorithms get a fair comparison.

Portable performance

Porting means re-tuning, not rewriting.

Safety for free

No deadlocks or races, plus automatic consistency tests.

Limitations

Optimal substructure assumed

Best small solution assumed best inside larger ones; trained on random inputs only.

Parameters tuned one at a time

Interactions between them, e.g. cutoff × block size, are largely missed.

Tuning cost unknown

No tuning times, no scaling numbers, no comparison with other search methods.

Weak baselines

Compared with their own single-algorithm versions, not std::sort or LAPACK.

TAKEAWAY

Represent the choices first, then search them

POWERGRAPH

Graph = the data

PETABRICKS

Alternatives = the search space

TENSAT · X-RLFLOW

Graph = the computation, rewrites = the choices

NEXT IN THE COURSE

Better search: OpenTuner, Bayesian optimisation, RL, LLM agents

WHAT STAYED

Separate **what** from **how**: Halide, TVM, torch.compile

Make the alternatives explicit, then let a search pick per machine.

Questions?

REFERENCES

Sources

MAIN PAPER · ALL FIGURES AND CODE

PetaBricks: A Language and Compiler for Algorithmic Choice. J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, S. Amarasinghe. PLDI 2009.

ALSO MENTIONED

PowerGraph · Gonzalez et al., OSDI 2012
TENSAT · Yang et al., MLSys 2021
X-RLflow · He, Parker, Yoneki, MLSys 2023
OpenTuner · Ansel et al., PACT 2014
PyTorch 2 / torch.compile · Ansel et al., ASPLOS 2024
CherryPick · Alipourfard et al., NSDI 2017
KernelEvolve · Liao et al., arXiv 2026
Halide · Ragan-Kelley et al., PLDI 2013
TVM · Chen et al., OSDI 2018

TAKEAWAY

Represent the choices first, then search them

POWERGRAPH

Graph = the data

PETABRICKS

Alternatives = the search space

TENSAT · X-RLFLOW

Graph = the computation, rewrites = the choices

NEXT IN THE COURSE

Better search: OpenTuner, Bayesian optimisation, RL, LLM agents

WHAT STAYED

Separate **what** from **how**: Halide, TVM, torch.compile

Make the alternatives explicit, then let a search pick per machine.

Questions?