Towards An Adaptive Framework for Performance Portability

Work in Progress (submission #23)

Patrick Maier Magnus Morton Phil Trinder

School of Computing Science
University of Glasgow

IFL 2015

Performance Portability?

Write once, run anywhere (a.k.a. the holy grail of portability)

1970s	portable operating systems	$\rightsquigarrow$ C
1990s	portable web applications	$\rightsquigarrow$ Java
2010s	portable linear algebra kernels	$\rightsquigarrow$ OpenCL

Performance Portability?

Write once, run anywhere (a.k.a. the holy grail of portability)

1970s	portable operating systems	$\rightsquigarrow$ C
1990s	portable web applications	$\rightsquigarrow$ Java
2010s	portable linear algebra kernels	$\rightsquigarrow$ OpenCL

It is accepted that portability incurs a performance hit.

- C compilers are very mature; performance hit vs. assembly is small.
- JIT compilation has brought Java performance within reach of C++.
- OpenCL code *can* perform as well as hand-written kernels.

Performance Portability?

Write once, run anywhere (a.k.a. the holy grail of portability)

1970s	portable operating systems	$\rightsquigarrow$ C
1990s	portable web applications	$\rightsquigarrow$ Java
2010s	portable linear algebra kernels	$\rightsquigarrow$ OpenCL

It is accepted that portability incurs a performance hit.

- C compilers are very mature; performance hit vs. assembly is small.
- JIT compilation has brought Java performance within reach of C++.
- OpenCL code *can* perform as well as hand-written kernels.

But: Performance of OpenCL code very sensitive to architecture.

- Necessary architecture-specific tuning defeats portability.

Performance Portability?

Write once, run anywhere (a.k.a. the holy grail of portability)

1970s	portable operating systems	$\rightsquigarrow$ C
1990s	portable web applications	$\rightsquigarrow$ Java
2010s	portable linear algebra kernels	$\rightsquigarrow$ OpenCL

It is accepted that portability incurs a performance hit.

- C compilers are very mature; performance hit vs. assembly is small.
- JIT compilation has brought Java performance within reach of C++.
- OpenCL code *can* perform as well as hand-written kernels.

But: Performance of OpenCL code very sensitive to architecture.

- Necessary architecture-specific tuning defeats portability.

Performance Portability

Same parallel code runs across different architectures with reasonable efficiency.

Grand Vision

Idea: Combine trace-based JIT compiler with demand-driven parallel scheduler.

- Language: Racket + skeleton library

Grand Vision

Idea: Combine trace-based JIT compiler with demand-driven parallel scheduler.

- Language: Racket + skeleton library

Slogan: Compile once, run anywhere **in parallel**.

- Performance portability hypothesis to be tested
  - by benchmarking problems with irregular parallelism
  - on several (CPU-centric) architectures (desktop, NUMA server, small cluster).

Grand Vision

Idea: Combine trace-based JIT compiler with demand-driven parallel scheduler.

- Language: Racket + skeleton library

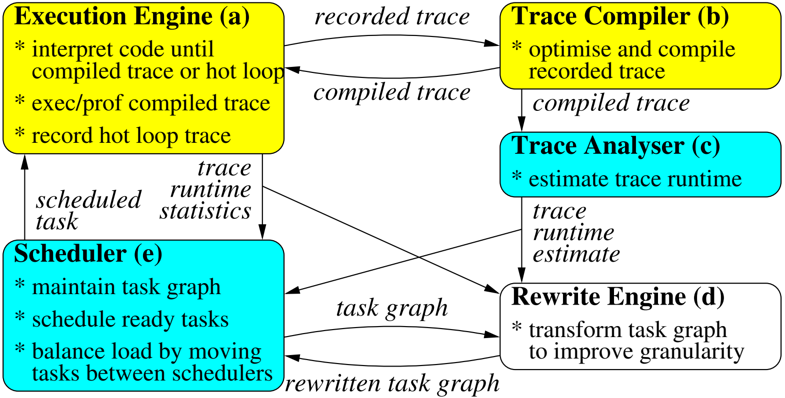
Slogan: Compile once, run anywhere **in parallel**.

- Performance portability hypothesis to be tested
  - by benchmarking problems with irregular parallelism
  - on several (CPU-centric) architectures (desktop, NUMA server, small cluster).

Some technical details:

- Focus parallelisation where it matters.
  - Using JIT compiler's hot code detection.
- Estimate task granularity by online profiling and/or static analysis of traces.
  - Linear structure of traces enables cheap yet accurate analyses.
- Adapt task granularity by online code transformation.
  - Rewriting according to programmable rules expressing semantic equivalences.

Grand Vision — Functional Block Diagram



Language and Compiler

Language: **Racket** (Scheme dialect)

- dynamically typed, strict functional language
- elaborate macro system
- concurrency, shared-memory parallelism, distributed computation, ...

Language and Compiler

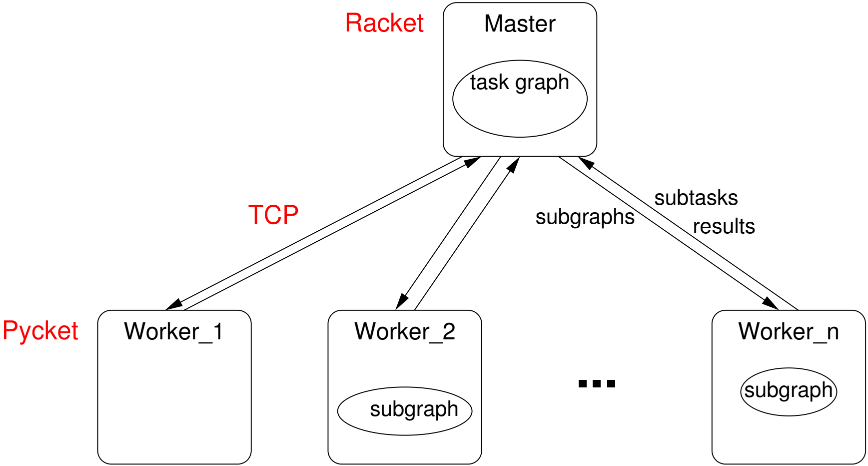
Language: **Racket** (Scheme dialect)

- dynamically typed, strict functional language
- elaborate macro system
- concurrency, shared-memory parallelism, distributed computation, ...

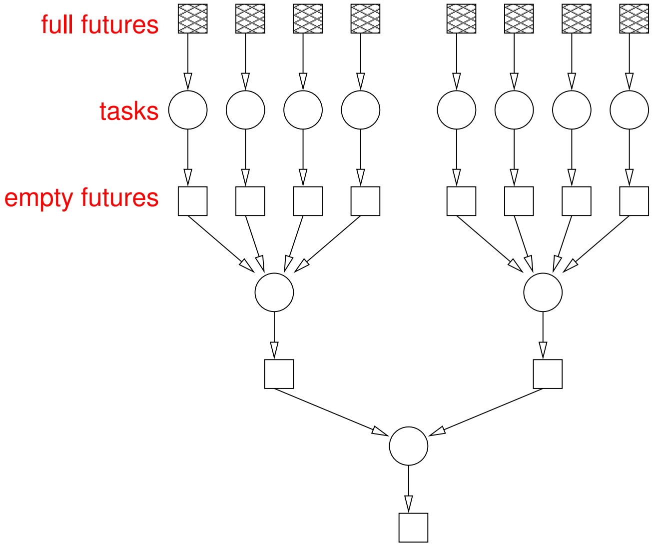
Compiler	JIT	language support
Racket *standard VM* *20 years development*	function-level	full language
Pycket *PyPy-derived VM* *1 year development* *often beats Racket*	trace-level	DOES NOT SUPPORT * concurrency (threads) * parallelism (futures) * distributed comp (places) * exceptions * sockets ...

Scheduler

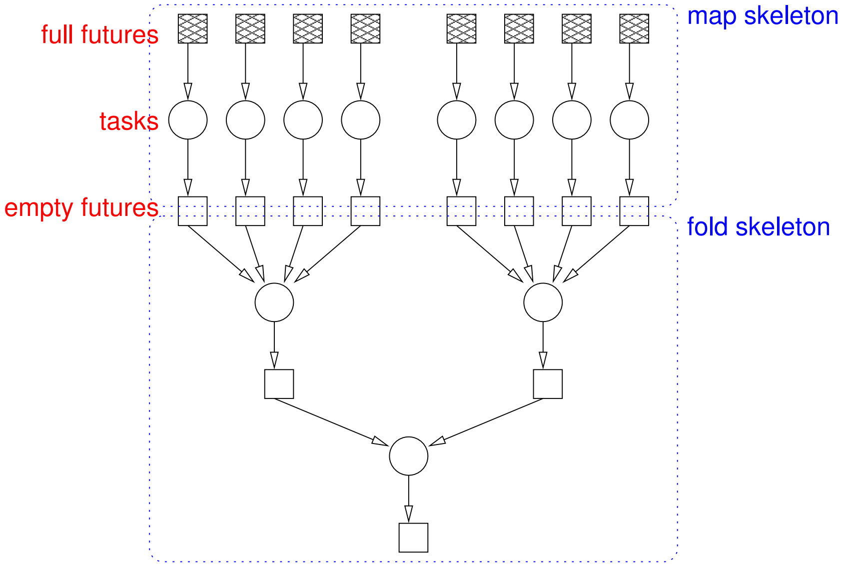
- Centralised control.
- Actor-like processes (no shared state, single threaded, message passing).



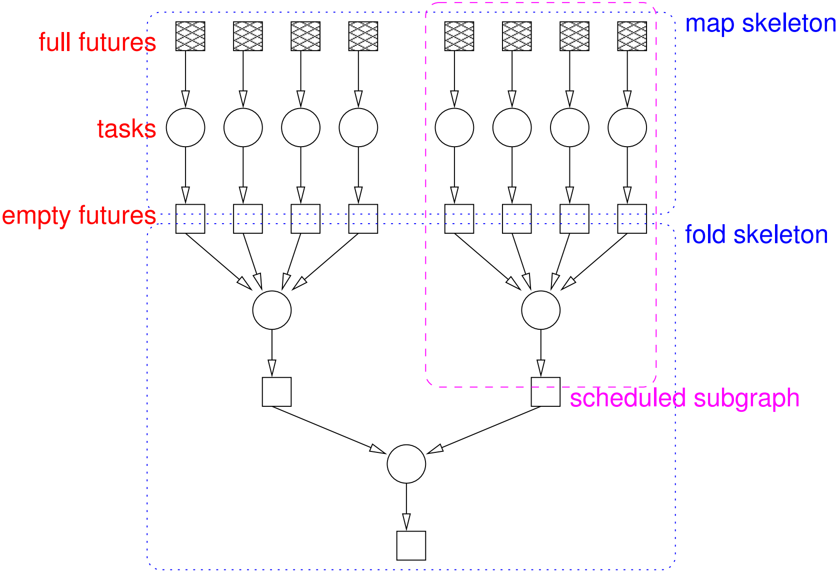
Task Graphs



Task Graphs


full futures
tasks
empty futures
map skeleton
fold skeleton

Task Graphs


full futures
tasks
empty futures
map skeleton
fold skeleton
scheduled subgraph

Skeletons

Map skeletons

```
par-map        ::        Closure ( a  ->  b ) -> [a] -> [b]
par-map/chunk  :: Int -> Closure ([a] -> [b]) -> [a] -> [b]
par-map/stride :: Int -> Closure ([a] -> [b]) -> [a] -> [b]
```

Fold skeletons

```
par-fold       ::        Closure ([a] -> a) -> [a] -> a
par-fold/depth :: Int -> Closure ([a] -> a) -> [a] -> a
```

Divide and conquer skeletons

```
par-d&c       ::        Closure (a -> b, a -> [a], [b] -> b) -> a -> b
par-d&c/depth :: Int -> Closure (a -> b, a -> [a], [b] -> b) -> a -> b
```

Skeletons Transformations

Skeletons are related by an equational theory.

Some map skeleton equations

```
(1) map f $ map g xs == map (x -> f $ g x) xs
(2) map f xs         == concat $ map (map f) $ chunk k xs

(3) map f xs                     == par-map (Closure f) xs
(4) concat $ map g $ chunk k xs == par-map/chunk k (Closure g) xs
```

Skeletons Transformations

Skeletons are related by an equational theory.

Some map skeleton equations

```
(1) map f $ map g xs == map (x -> f $ g x) xs
(2) map f xs         == concat $ map (map f) $ chunk k xs

(3) map f xs                     == par-map (Closure f) xs
(4) concat $ map g $ chunk k xs == par-map/chunk k (Closure g) xs
```

Equations can be used as bi-directional rewrite rules.

- Instantiate granularity parameter **k** when applying (2) from left to right.

Sample transformation

```
   par-map (Closure f) $ par-map (Closure g) xs
== map f $ map g xs
== map (x -> f $ g x) xs
== concat $ map (map (x -> f $ g x)) $ chunk 5 xs      guessed k=5
== par-map/chunk 5 (Closure (map (x -> f $ g x))) xs
```

Skeletons Transformations II

Transform task graph when observed task cost (i.e. runtime) distribution not in target range (10 – 100 milliseconds).

Transformation strategy:

1. Repeatedly
    - Rewrite task graph according to skeleton equations
        - randomised selection of rewrite rules;
        - cost model guided instantiation of granularity parameters.
    - Predict costs of rewritten tasks.
2. Select a task graph whose cost distribution falls within target range.

Compute cost model on the fly during JITting.

Use cost model

- to predict task execution time, and
- to infer suitable values for granularity parameters (e.g. chunk size).

Trace-based Cost Models

Tracing JIT compilers automatically produce

- traces (= sequences of instructions), and
- trace counters.

Simple cost model piggybacking on tracing JIT

$$cost(trace) = \sum_{inst \in trace} cost(inst)$$

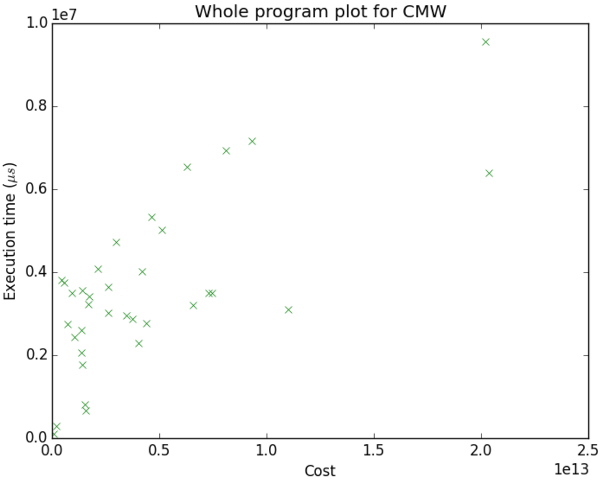
$$cost(task) = \sum_{trace \in task} count(trace) \cdot cost(trace)$$

Simple cost model parametric in cost of instructions.

- "Learn" cost of instructions by training cost model on a Pycket benchmark suite.

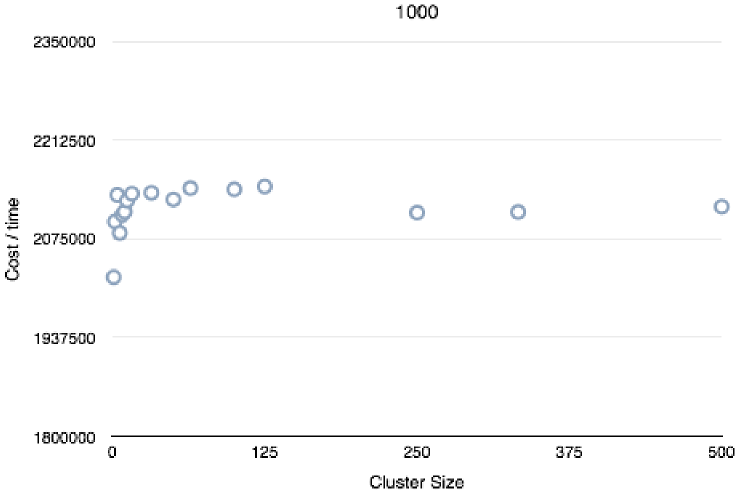
Trace-based Cost Models II

Bad news: Cost model not very accurate for comparing whole programs.



Trace-based Cost Models II

Bad news: Cost model not very accurate for comparing whole programs.
Good news: Cost model quite accurate for comparing task transformations.



Evaluating Scheduler

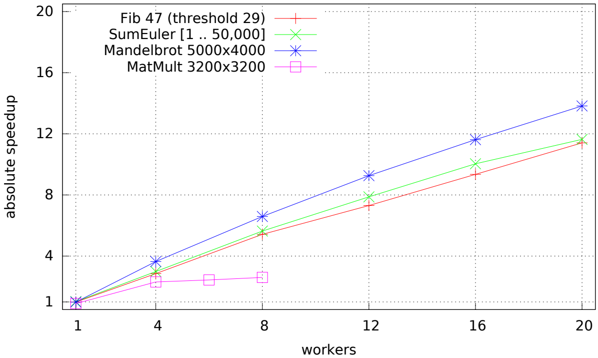
Limitations: Single server (max 24 cores).

Microbenchmarks	skeleton	irregular?	comm. volume	C gap
Fibonacci	divide/conquer	no	low	3.4×
SumEuler	parallel map	moderate	low	1.3×
Mandelbrot	parallel map	moderate	moderate	3.2×
Matrix multiplication	parallel map	no	high	1.2×

Evaluating Scheduler

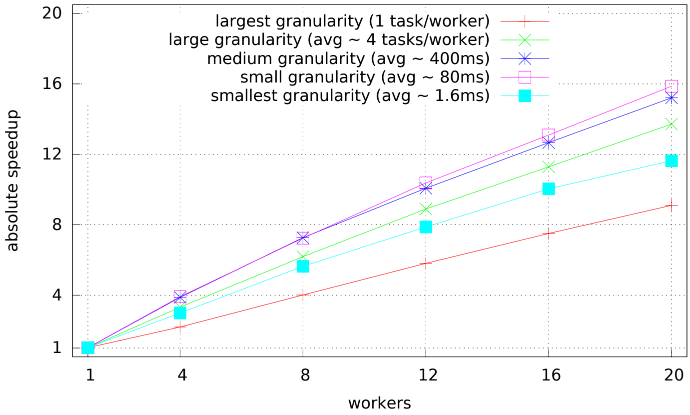
Limitations: Single server (max 24 cores).

Microbenchmarks	skeleton	irregular?	comm. volume	C gap
Fibonacci	divide/conquer	no	low	3.4×
SumEuler	parallel map	moderate	low	1.3×
Mandelbrot	parallel map	moderate	moderate	3.2×
Matrix multiplication	parallel map	no	high	1.2×



Impact of Transformations

SumEuler does not scale well because of low task granularity ($\approx$ 1.6 ms).

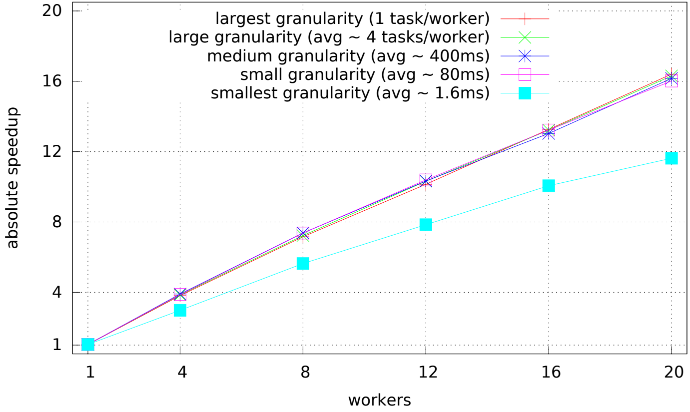


Transformation 1: Split input interval into even chunks.

- Irregular parallelism: scaling very sensitive to task size.
- Top speedup: 15.9 (up from 11.6)

Impact of Transformations

SumEuler does not scale well because of low task granularity ($\approx$ 1.6 ms).



Transformation 2: Stride through input interval.

- Fairly regular parallelism: scaling independent of task size.
- Top speedup: 16.4 (up from 11.6)

The End

Summary:

- Scheduler running parallel Racket code in Pycket.
- Skeleton transformations can speedup parallel code.
- Not yet demonstrated: best transformation dependent on architecture.

Current limitations:

- Task graph scheduling not fully implemented.
- Limited to single server architecture.
- High communication/serialisation overheads.

Work in progress:

- Hook cost analysis into JIT compiler.
- Task graph transformation engine.