

Scheduling Parallel Programs

The Multi-MLton Parallel Framework

Daniel Spoonhower
CMU

OCR Workshop
28 May 2008

Framework For Parallel Scheduling: Goals

Preserve sequential semantics

Support many forms of “conjunctive” parallelism

- ▶ no speculation, no aborts
- ▶ e.g., fork-join, futures, &c.

Implement several scheduling policies

- ▶ both for performance and experimentation

Reason about performance of *source* code

MLton: Overview

Whole-program, optimizing compiler for SML

Generates *fast* code!

- ▶ 2–20× faster than other SML impl [mlton.org]
- ▶ within 2× of GCC [shootout.alioth.debian.org]

Stable, mature code base

- ▶ 10+ years of development
- ▶ 3 back-ends on 4 architectures; 6 OSs
- ▶ allocation, time profilers; libraries

MLton: Compilation

Whole program compilation:

- ▶ eliminate: modules, polymorphism, higher-order functions
- ▶ optimize via: inline, contify, constant prop, flatten, CSE, &c.
- ▶ optimizations operate on first-order SSA

Aggressive representation strategies

- ▶ e.g., arrays of unboxed ints, datatypes
- ▶ “imperative” stacks

MLton: An Opportunity for Parallelism

Generates fast sequential code

Rich control primitives

- ▶ callcc, user-level threads
- ▶ already understood by compiler

Whole-program \Rightarrow implement “run-time” as library

Simple, powerful foreign function interface

No runtime type information necessary

Multi-MLton Extensions

Runtime changes

- ▶ support fixed set of threads
- ▶ mutex around GC, shared resources
- ▶ thread-local allocation

SML Library

- ▶ core parallel primitives
- ▶ scheduling policies: prioritize tasks
- ▶ parallel constructs: fork-join, futures

Progress

Implementation

- ▶ framework in place, 3 scheduling policies
- ▶ focus on fork-join

Benchmarks

- ▶ sorting, convex hull, n -body sim., SAT

Progress

Implementation

- ▶ framework in place, 3 scheduling policies
- ▶ focus on fork-join

Benchmarks

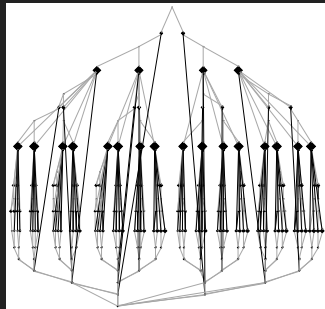
- ▶ sorting, convex hull, n -body sim., SAT

This summer: scheduling overheads & futures

Cost Semantics

Abstract measure of cost

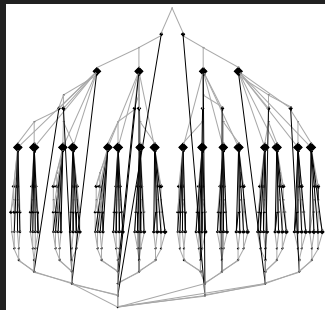
- ▶ here: two DAGs
- ▶ basis for prototype profiling tools



Cost Semantics

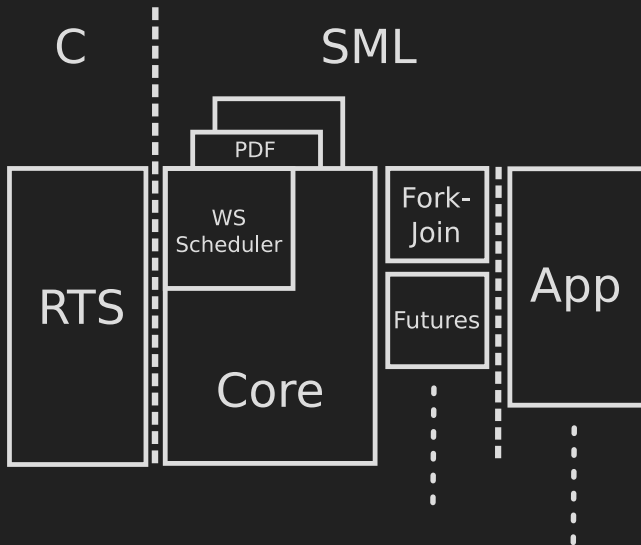
Abstract measure of cost

- ▶ here: two DAGs
- ▶ basis for prototype profiling tools



Found a space leak in generated code

Software Architecture



Core Primitives

Suspend and resume tasks

Wraps interface to scheduling policy

- ▶ factor out common code
- ▶ \Rightarrow easy to plug in new policies

Leave synchronization to constructs

Core Primitives (con't)

type work = unit \rightarrow void (** parallel task **)

type 'a susp (** suspension waiting for 'a **)

(** reify the calling context, like callcc **)

val capture : ('a susp \rightarrow work list) \rightarrow 'a

(** restart captured (and suspended) context **)

val resume : ('a * 'a susp) \rightarrow void

(** ends task **)

val return : work

Using Core Primitives

Simplest task:

```
return ()
```

Using Core Primitives

Simplest task:

```
return ()
```

Run function f in parallel:

```
capture (fn k => [fn () => resume (k, ()),  
               fn () => f (); return ()])
```

Further Details...

Derived “primitives”

- ▶ e.g., addWork, continue
- ▶ avoid extra work when possible

MLton Thread library

- ▶ includes one-time callcc

Further Details...

Derived “primitives”

- ▶ e.g., addWork, continue
- ▶ avoid extra work when possible

MLton Thread library

- ▶ includes one-time callcc

Wraps implementation of scheduling policy...

Scheduling Policy

Determines which task is next

(add new work to the queue *)*

val addWork : proc \rightarrow work list \rightarrow unit

(remove the next, highest priority work *)*

val getWork : proc \rightarrow work option

(mark the most recent unit of work as done *)*

val finishWork : proc \rightarrow unit

Scheduling Policy

Determines which task is next

(add new work to the queue *)*

val addWork : proc \rightarrow work list \rightarrow unit

(remove the next, highest priority work *)*

val getWork : proc \rightarrow work option

(mark the most recent unit of work as done *)*

val finishWork : proc \rightarrow unit

(is there higher priority work? *)*

val shouldYield : proc \rightarrow bool

Benchmarks

Focus on fork-join \Rightarrow divide-and-conquer

Smaller examples:

- ▶ sorting, convex hull, matrix ops

Barnes-Hut n -body simulation

Investigating parallel SAT

PDF Scheduling in X10

Similar “core” primitives already in place

- ▶ `pushFrame`
 - ▶ similar to `addWork`
 - ▶ may suspend until some policies
- ▶ `abortOnSteal`
 - ▶ needs to check `shouldYield`
 - ▶ also check local queue (as in X10)

PDF Scheduling in X10 (con't)

Exceptions currently used to abort

- ▶ too expensive in common case?

Closures might simplify code