Space Profiling for Parallel Functional Programs

Daniel Spoonhower¹, Guy Blelloch¹, Robert Harper¹, & Phillip Gibbons²

¹Carnegie Mellon University
²Intel Research Pittsburgh

23 September 2008
ICFP '08, Victoria, BC
Improving Performance – Profiling Helps!

Profiling improves functional program performance.
Improving Performance – Profiling Helps!

Profiling improves functional program performance.

Good performance in parallel programs is also hard.
Improving Performance – Profiling Helps!

Profiling improves functional program performance.

Good performance in parallel programs is also hard.

This work: space profiling for parallel programs
Naïve NESL code for matrix multiplication

\[
\text{function } \text{dot}(a,b) = \text{sum}\left(\{ a * b : a; b \}\right)
\]
\[
\text{function } \text{prod}(m,n) = \left\{ \left\{ \text{dot}(m,n) : n \right\} : m \right\}
\]
Example: Matrix Multiply

Naïve NESL code for matrix multiplication

```nesl
function dot(a,b) = sum ({ a * b : a; b })
function prod(m,n) = { { dot(m,n) : n } : m }
```

Requires $O(n^3)$ space for $n \times n$ matrices!

- compare to $O(n^2)$ for sequential ML
Example: Matrix Multiply

Naïve NESL code for matrix multiplication

function dot(a,b) = sum ({ a * b : a; b })
function prod(m,n) = { { dot(m,n) : n } : m }

Requires $O(n^3)$ space for $n \times n$ matrices!

► compare to $O(n^2)$ for sequential ML

Given a parallel functional program, can we determine, “How much space will it use?”
Example: Matrix Multiply

Naïve NESL code for matrix multiplication

```nesl
function dot(a,b) = sum ({ a * b : a; b })
function prod(m,n) = { { dot(m,n) : n } : m }
```

Requires $O(n^3)$ space for $n \times n$ matrices!

► compare to $O(n^2)$ for sequential ML

Given a parallel functional program, can we determine,

“How much space will it use?”

Short answer: It depends on the implementation.
Parallel programs admit many different executions
- not all impl. of matrix multiply are $O(n^3)$

Determined (in part) by scheduling policy
- lots of parallelism; policy says what runs next
Semantic Space Profiling

Our approach: factor problem into two parts.

1. Define **parallel structure** (as graphs)
   - circumscribes all possible executions
   - deterministic (independent of policy, &c.)
   - include approximate space use

2. Define **scheduling policies** (as traversals of graphs)
   - used in profiling, visualization
   - gives specification for implementation
Contributions of this work:

- **cost semantics** accounting for...
  - scheduling policies
  - space use
- **semantic** space profiling tools
- extensible **implementation** in MLton
Talk Summary

Cost Semantics, Part I: Parallel Structure
Cost Semantics, Part II: Space Use
Semantic Profiling
Talk Summary

Cost Semantics, Part I: Parallel Structure
Cost Semantics, Part II: Space Use
Semantic Profiling
Program Execution as a Dag

Model execution as directed acyclic graph (dag)

One graph for all parallel executions

- nodes represent units of work
- edges represent sequential dependencies
Program Execution as a Dag

Model execution as directed acyclic graph (dag)

One graph for all parallel executions
  ▶ nodes represent units of work
  ▶ edges represent sequential dependencies

Each schedule corresponds to a traversal
  ▶ every node must be visited; parents first
  ▶ limit number of nodes visited in each step
Program Execution as a Dag

Model execution as directed acyclic graph (dag)

One graph for all parallel executions
  ▶ nodes represent units of work
  ▶ edges represent sequential dependencies

Each schedule corresponds to a traversal
  ▶ every node must be visited; parents first
  ▶ limit number of nodes visited in each step

A policy determines schedule for every program
Program Execution as a Dag (con’t)
Program Execution as a Dag (con’t)

Graphs are **NOT** . . .
- control flow graphs
- explicitly built at runtime

Graphs are . . .
- derived from cost semantics
- unique per closed program
- independent of scheduling
Scheduling policy defined by:

- **breadth**-first traversal of the dag  
  \((i.e. \text{ visit nodes at shallow depth first})\)
- break ties by taking leftmost node
- visit at most \(p\) nodes per step  
  \((p = \text{ number of processor cores})\)
Breadth-First Illustrated \((p = 2)\)
Breadth-First Illustrated (\(p = 2\))
Breadth-First Illustrated \((p = 2)\)
Breadth-First Illustrated ($p = 2$)
Breadth-First Illustrated \((p = 2)\)
Breadth-First Illustrated ($p = 2$)
Breadth-First Illustrated \((p = 2)\)
Breadth-First Illustrated ($p = 2$)
Breadth-First Scheduling Policy

Scheduling policy defined by:

- breadth-first traversal of the dag (i.e. visit nodes at shallow depth first)
- break ties by taking leftmost node
- visit at most $p$ nodes per step ($p = \text{number of processor cores}$)

Variation implicit in impls. of NESL & Data Parallel Haskell

- vectorization bakes in schedule
Depth-First Scheduling Policy

Scheduling policy defined by:

- **depth**-first traversal of the dag
  \[(i.e. \text{ favor children of recently visited nodes})\]
- break ties by taking leftmost node
- visit at most \(p\) nodes per step
  \[(p = \text{ number of processor cores})\]
Depth-First Illustrated \((p = 2)\)
Depth-First Illustrated \((p = 2)\)
Depth-First Illustrated ($p = 2$)
Depth-First Illustrated ($p = 2$)
Depth-First Illustrated \((p = 2)\)
Depth-First Illustrated ($\rho = 2$)
Depth-First Illustrated ($p = 2$)
Depth-First Illustrated \((p = 2)\)
Depth-First Illustrated ($p = 2$)
Depth-First Scheduling Policy

Scheduling policy defined by:

- depth-first traversal of the dag (i.e. favor children of recently visited nodes)
- break ties by taking leftmost node
- visit at most $p$ nodes per step ($p =$ number of processor cores)

Sequential execution

= one processor depth-first schedule
"Work-stealing" means many things:

- idle procs. shoulder burden of communication
- specific implementations, e.g. Cilk
- implied ordering of parallel tasks

For the purposes of space profiling, ordering is important
- briefly: globally breadth-first, locally depth-first
Computation Graphs: Summary

**Cost semantics** defines graph for each closed program
- *i.e.* defines parallel structure
- call this graph **computation** graph

Scheduling polices defined on graphs
- describe behavior *without* data structures, synchronization, &c.
Talk Summary

Cost Semantics, Part I: Parallel Structure
Cost Semantics, Part II: Space Use
Semantic Profiling
Heap Graphs

Goal: describe space use independently of schedule
► our innovation: add heap graphs

Heap graphs also act as a specification
► constrain use of space by compiler & GC
► just as computation graph constrains schedule
Heap Graphs

Goal: describe space use independently of schedule
  ▶ our innovation: add heap graphs

Heap graphs also act as a specification
  ▶ constrain use of space by compiler & GC
  ▶ just as computation graph constrains schedule

Computation & heap graphs share nodes.
  ▶ think: one graph w/ two sets of edges
Cost for Parallel Pairs

Generate costs for parallel pair,

\{ e_1, e_2 \}
Cost for Parallel Pairs

Generate costs for parallel pair,

\{e_1, e_2\}
Generate costs for parallel pair, \( \{e_1, e_2\} \)
Cost for Parallel Pairs

Generate costs for parallel pair,

\{e_1, e_2\}
Cost for Parallel Pairs

Generate costs for parallel pair, 

\[ \{e_1, e_2\} \]
Cost for Parallel Pairs

Generate costs for parallel pair,

\{e_1, e_2\}
Cost for Parallel Pairs

Generate costs for parallel pair,

\{e_1, e_2\}

(see paper for inference rules)
Recall, schedule = traversal of computation graph
  ▶ visiting $p$ nodes per step to simulate $p$ processors

Each step of traversal divides set of nodes into:
  1. nodes executed in past
  2. notes to be executed in future
From Cost Graphs to Space Use

Recall, schedule = traversal of computation graph
  ▶ visiting $p$ nodes per step to simulate $p$ processors

Each step of traversal divides set of nodes into:
  1. nodes executed in past
  2. notes to be executed in future

Heap edges crossing from future to past are “roots”
  ▶ i.e. future uses of existing values
Determining Space Use
Determining Space Use
Determining Space Use
Determining Space Use
Determining Space Use
Determining Space Use
Heap Edges Also Track Uses

Heap edges also added as "possible last-uses," e.g.,

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3
\]
Heap Edges Also Track Uses

Heap edges also added as “possible last-uses,” e.g.,

if $e_1$ then $e_2$ else $e_3$
(where $e_1 \rightarrow^* \text{true}$)
Heap Edges Also Track Uses

Heap edges also added as “possible last-uses,” e.g.,

if $e_1$ then $e_2$ else $e_3$
(where $e_1 \rightarrow^* \text{true}$)
Heap edges also added as “possible last-uses,” e.g.,

if $e_1$ then $e_2$ else $e_3$  
(where $e_1 \rightarrow^* \text{true}$)
Heap edges also added as "possible last-uses," e.g.,

if $e_1$ then $e_2$ else $e_3$
(where $e_1 \rightarrow^* \text{true}$)
Heap Edges Also Track Uses

Heap edges also added as "possible last-uses," e.g.,

if $e_1$ then $e_2$ else $e_3$
(where $e_1 \mapsto^* \text{true}$)
Heap Edges Also Track Uses

Heap edges also added as “possible last-uses,” e.g.,

if $e_1$ then $e_2$ else $e_3$
(where $e_1 \rightarrow^* \text{true}$)
Heap edges also added as “possible last-uses,” e.g.,

```
if \( e_1 \) then \( e_2 \) else \( e_3 \)
```

(where \( e_1 \stackrel{*}{\rightarrow} \text{true} \))
Heap Graphs: Summary

Heap edge from $B$ to $A$ indicates a dependency on $A$

... *given knowledge* up to time corresponding to $B$
Heap Graphs: Summary

Heap edge from $B$ to $A$ indicates a dependency on $A$...

...given knowledge up to time corresponding to $B$

Some push back on semantics from implementation

- semantics must be implementable

- e.g., “true” vs. “provable” garbage
Example Graphs

Matrix multiplication

- computation graph on left; heap on right
Talk Summary

Cost Semantics, Part I: Parallel Structure
Cost Semantics, Part II: Space Use
Semantic Profiling
Semantic Profiling

Analysis of costs

- *not* a static analysis
Semantic Profiling

Analysis of costs

- *not* a static analysis

Semantics yields one set of costs per input

- run program over many inputs to generalize
Semantic Profiling

Analysis of **costs**

- *not* a static analysis

Semantics yields one set of costs per input

- run program over many inputs to generalize

Semantic → independent of implementation
Semantic Profiling

Analysis of costs

- *not* a static analysis

Semantics yields one set of costs per input

- run program over many inputs to generalize

Semantic ⇒ independent of implementation

- loses some precision

- acts as specification
Visualizing Schedules

Distill graphs, focusing on parallel structure

- coalesce sequential computation
- use size, color, relative position
- omit less interesting edges
Visualizing Schedules

Distill graphs, focusing on parallel structure
- coalesce sequential computation
- use size, color, relative position
- omit less interesting edges

Graphs derived from semantics,
... compressed mechanically,
... then laid out with GraphViz
Matrix Multiply (Breadth-First, \( p = 2 \))
Matrix Multiply (Work Stealing, $p = 2$)
Quick Hull
Quick Hull (Depth First, $p = 2$)
Quick Hull (Work Stealing, $p = 2$)
Space Use By Input Size

Matrix multiply w/ breadth-first scheduling policy:

- Space high-water mark (units)
- Input size (# rows/columns)

(work queue)

[work queue]

append (#1 lr, #2 lr)

(remainder)
Space Use By Input Size

Matrix multiply w/ breadth-first scheduling policy:

Scheduler Overhead

append (#1 lr, #2 lr)

⟨work queue⟩

[b,a]

[ m,u ]
Matrix multiply w/ breadth-first scheduling policy:
Verifying Profiling Results
Verifying Profiling Results

Implemented a parallel extension to MLton

- including three different schedulers
- compared predicted and actual space use
Quicksort – MLton Space Use

![Graph showing space use for Quicksort with different strategies: Depth-First, Work-Stealing, and Breadth-First. The y-axis represents max live space in MB, and the x-axis represents input size in elements. The graph illustrates how the space usage changes with increasing input size.]
Initial Quicksort Results

> predicted: *breadth-first* outperforms *depth-first*
Initial Quicksort Results

- predicted: *breadth-first* outperforms *depth-first*
- initial observation: same results!
Space Leak Revealed

Cause: reference flattening optimization
(representing reference cells directly in records)
Space Leak Revealed

Cause: reference flattening optimization
(representing reference cells directly in records)

Now fixed in MLton source repository
Space Leak Revealed

Cause: reference flattening optimization (representing reference cells directly in records)

Now fixed in MLton source repository

Without a cost semantics, there is no bug!
Also in the Paper

More details, including...

- rules for cost semantics
- discussion of MLton implementation
  - efficient method for space measurements
- more plots (profiling, speedup, &c.)
- application to vectorization (in TR)
Selected Related Work

Cost semantics

- Sansom & Peyton Jones. *POPL ’95*
- Bleloch & Greiner. *ICFP ’96*

Scheduling

- Bleloch, Gibbons, & Matias. *JACM ’99*
- Blumofe & Leiserson. *JACM ’99*

Profiling

- Runciman & Wakeling. *JFP ’93*
- *ibid. Glasgow FP ’93*
Conclusion
Semantic profiling for parallel programs... 
- accounts for scheduling, space use
- constrains implementation (and finds bugs!)
- supports visualization & predicts actual performance
Thanks to MLton developers, and
Thank you for listening!

Questions?
spoons@cmu.edu

Download binaries, source code, papers, slides:
http://www.cs.cmu.edu/~spoons/parallel/
svn co svn://mlton.org/mlton/...
branches/shared-heap-multicore mlton