The Lazy Task Creation Machine

Umut Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski

ERC Deepsea, Inria Gallium

Gallium Seminar - July 28th 2016
Parallelism in Theory and in Practice

**Parallel algorithmics**
- Design algorithms with good parallel complexity.
- Difficult, or even impossible for some problems.
- Use low-level machine models or informal arguments.

**Parallel programming**
- Implement parallel algorithms on real-world computers.
- Many low- and high-level abstractions and systems.
- Parallel programming has traditionally been done by experts.
Parallel algorithmics

- Design algorithms with good parallel complexity.
- Difficult, or even impossible for some problems.
- Use low-level machine models or informal arguments.

Parallel programming

- Implement parallel algorithms on real-world computers.
- Many low- and high-level abstractions and systems.
- Parallel programming has traditionally been done by experts.

*Translating a scalable algorithm into an efficient parallel program is difficult with today’s mainstream hardware architectures.*
Scheduling parallel programs

- **Static**: compile time; needs program and machine models in general.
- **Dynamic**: run time; few hypotheses; adds overhead.
Scheduling parallel programs

- **Static**: compile time; needs program and machine models in general.
- **Dynamic**: run time; few hypotheses; adds overhead.
Key Ingredients of Dynamic Scheduling

Load balancing
- Dispatch ready tasks over cores in order to maximize utilization.
- Intensely studied, from both theoretical and practical standpoints.
- Task handling incurs large overheads w.r.t. sequential execution.
- A popular technique: Work Stealing.

Granularity control
- Tries to minimize the creation of small tasks.
- Used in industrial runtime systems such as Cilk and Intel® TBB.
- Much less understood than load balancing.
- A popular technique: Lazy Task Creation.
Key Ingredients of Dynamic Scheduling

Load balancing
- Dispatch ready tasks over cores in order to maximize utilization.
- Intensely studied, from both theoretical and practical standpoints.
- Task handling incurs large overheads w.r.t. sequential execution.
- A popular technique: *Work Stealing*.

Granularity control
- Tries to minimize the creation of small tasks.
- Used in industrial runtime systems such as Cilk and Intel® TBB.
- Much less understood than load balancing.
- A popular technique: *Lazy Task Creation*. 
The Lazy Task Creation granularity control policy

- Trade potential parallelism for lower overhead.
  - Run newly created parallel tasks sequentially by default.
  - Create actual parallelism only when enough sequential work is done.
- Usually described in terms of concrete implementations.
- No analytic performance study.
Investigating Lazy Task Creation

The Lazy Task Creation granularity control policy

- Trade potential parallelism for lower overhead.
  - Run newly created parallel tasks sequentially by default.
  - Create actual parallelism only when enough sequential work is done.
- Usually described in terms of concrete implementations.
- No analytic performance study.

This Talk

- Lazy Task Creation as a non-standard cost semantics.
  - Big-step semantics for a strict $\lambda$-calculus with parallel pairs.
  - Extract parallelism from the stack in a lazy manner.
- Compare it to the fully sequential and fully parallel semantics.
  - Prove its soundness: does it compute the same thing?
  - Bound its overheads: what is its effect on performance?
Outline

1 Introduction

2 The Semantics of Lazy Task Creation

3 Efficiency and Soundness Results

4 Perspectives
Syntax of the Object Language

A standard $\lambda$-calculus with a non-standard notation for pairs.

\[
\text{Exp} \ni e ::= x \mid \lambda x.e \mid (e\ e) \mid (e \ || \ e) \mid \text{fst} \ e \mid \text{snd} \ e
\]

Values are closures or pairs of values.

\[
\text{Val} \ni v ::= (\lambda x.e)\{\sigma\} \mid (v, v)
\]

\[
\text{Env} \ni \sigma = \text{Var} \rightarrow^\text{fin} \text{Val}
\]
Syntax of the Sequential Abstract Machine

Frames are expression constructors with formal holes and partial results.

\[ Frame \ni f ::= \text{APPL}(\square, e, \sigma) | \text{APPR}((\lambda x.e)\{\sigma\}, \square) \]
\[ | \text{PAIRL}(\square, e, \sigma) | \text{PAIRR}(v, \square) \]
\[ | \text{FST}(\square) | \text{SND}(\square) \]

Stacks are list of frames.

\[ Kont \ni k ::= \text{TOP} | f :: k \]

Heads are either an expression in its local environment or a final value.

\[ Head \ni h ::= (e | \sigma) | (v | -) \]

Machines are formed by heads in front of stacks.

\[ Mach \ni m ::= \langle h | k \rangle \]
Transitions of the Sequential Abstract Machine

\[
\begin{array}{llll}
  m & \rightarrow & m \\
\end{array}
\]

\[
\begin{array}{llllll}
  \langle x \rangle & \sigma & k & \rightarrow & \langle \sigma(x) \rangle & \sigma & k \\
  \langle \lambda x. e \rangle & \sigma & k & \rightarrow & \langle (\lambda x. e)\{\sigma} \rangle & - & k \\
  \langle (e_1, e_2) \rangle & \sigma & k & \rightarrow & \langle e_1 \rangle & \sigma & \text{APPL}(\emptyset, e_2, \sigma) :: k \\
  \langle (\lambda x. e)\{\sigma} \rangle & - & \text{APPL}(\emptyset, e_2, \sigma') :: k & \rightarrow & \langle e_2 \rangle & \sigma' & \text{APPR}((\lambda x. e)\{\sigma}, \emptyset) :: k \\
  \langle v \rangle & - & \text{APPR}((\lambda x. e)\{\sigma}, \emptyset) :: k & \rightarrow & \langle e \rangle & \sigma[x \mapsto v] & k \\
  \langle (e_1 || e_2) \rangle & \sigma & k & \rightarrow & \langle e_1 \rangle & \sigma & \text{PAIRL}(\emptyset, e_2, \sigma) :: k \\
  \langle v_1 \rangle & - & \text{PAIRL}(\emptyset, e_2, \sigma) :: k & \rightarrow & \langle e_2 \rangle & \sigma & \text{PAIRR}(v_1, \emptyset) :: k \\
  \langle v_2 \rangle & - & \text{PAIRR}(v_1, \emptyset) :: k & \rightarrow & \langle (v_1, v_2) \rangle & - & k \\
  \langle \text{fst } e \rangle & \sigma & k & \rightarrow & \langle e \rangle & \sigma & \text{FST}(\emptyset) :: k \\
  \langle (v_1, v_2) \rangle & - & \text{FST}(\emptyset) :: k & \rightarrow & \langle v_1 \rangle & - & k \\
  \langle \text{snd } e \rangle & \sigma & k & \rightarrow & \langle e \rangle & \sigma & \text{SND}(\emptyset) :: k \\
  \langle (v_1, v_2) \rangle & - & \text{SND}(\emptyset) :: k & \rightarrow & \langle v_2 \rangle & - & k \\
\end{array}
\]
We assume that all the transitions of a machine are constant time.
Thus we take the cost of a sequence of transitions to be its length.
It will turn out to be convenient to have an explicit big-step judgment.

\[ m \Rightarrow_{\text{seq}} v; n \]
We assume that all the transitions of a machine are constant time.
Thus we take the cost of a sequence of transitions to be its length.
It will turn out to be convenient to have an explicit big-step judgment.

\[ m \Rightarrow_{seq} v; n \]

\[ \text{SEQVAL} \]

\[ \langle v \mid - \mid \text{TOP} \rangle \Rightarrow_{seq} v; 0 \]
The Fully Sequential Evaluation Judgment

- We assume that all the transitions of a machine are constant time.
- Thus we take the cost of a sequence of transitions to be its length.
- It will turn out to be convenient to have an explicit big-step judgment.

\[ m \Rightarrow_{\text{seq}} v; n \]

**SEQVAL**

\[ \langle v \mid - \mid \text{TOP} \rangle \Rightarrow_{\text{seq}} v; 0 \]

**SEQSIMPLESTEP**

\[ \langle h \mid k \rangle \rightarrow \langle h' \mid k' \rangle \quad \langle h' \mid k' \rangle \Rightarrow_{\text{seq}} v; n \]

\[ \langle h \mid k \rangle \Rightarrow_{\text{seq}} v; 1 + n \]
Assigning costs to parallel executions require two distinct notions.

- The *work* is the total number of elementary steps performed.
- The *span* is the length of the longest sequential subexecution.
Assigning costs to parallel executions require two distinct notions.

- The *work* is the total number of elementary steps performed.
- The *span* is the length of the longest sequential subexecution.

We thus define costs to be series/parallel directed acyclic graphs.

\[
\text{Cost} \ni g ::= 0 \mid 1 \mid (g \cdot g) \mid (g \parallel g)
\]
Assigning costs to parallel executions require two distinct notions.

- The *work* is the total number of elementary steps performed.
- The *span* is the length of the longest sequential subexecution.

We thus define costs to be series/parallel directed acyclic graphs.

\[
\text{Cost} \ni g ::= \quad 0 \mid 1 \mid (g \cdot g) \mid (g \parallel g)
\]

Work and span are defined inductively on the structure of graphs.

\[
\begin{align*}
  \text{work}(0) &= 0 \\
  \text{work}(1) &= 1 \\
  \text{work}(g_1 \cdot g_2) &= \text{work}(g_1) + \text{work}(g_2) \\
  \text{work}(g_1 \parallel g_2) &= \text{work}(g_1) + \text{work}(g_2) \\
  \text{span}(0) &= 0 \\
  \text{span}(1) &= 1 \\
  \text{span}(g_1 \cdot g_2) &= \text{span}(g_1) + \text{span}(g_2) \\
  \text{span}(g_1 \parallel g_2) &= \max(\text{span}(g_1), \text{span}(g_2))
\end{align*}
\]
Assigning costs to parallel executions require two distinct notions.

- The work is the total number of elementary steps performed.
- The span is the length of the longest sequential subexecution.

We thus define costs to be series/parallel directed acyclic graphs.

\[
\text{Cost} \ni g \ ::= \ 0 \mid 1 \mid (g \cdot g) \mid (g \parallel g)
\]

Work and span are defined inductively on the structure of graphs.

\[
\begin{align*}
\text{work}(0) &= 0 \\
\text{work}(1) &= 1 \\
\text{work}(g_1 \cdot g_2) &= \text{work}(g_1) + \text{work}(g_2) \\
\text{work}(g_1 \parallel g_2) &= \tau + \text{work}(g_1) + \text{work}(g_2)
\end{align*}
\]

\[
\begin{align*}
\text{span}(0) &= 0 \\
\text{span}(1) &= 1 \\
\text{span}(g_1 \cdot g_2) &= \text{span}(g_1) + \text{span}(g_2) \\
\text{span}(g_1 \parallel g_2) &= \tau + \max(\text{span}(g_1), \text{span}(g_2))
\end{align*}
\]

We assume that the overhead of synchronization is a fixed constant \(\tau \in \mathbb{N}\).
\[ m \Rightarrow_{\text{seq}} v; g \]

**SEQVAL**

\[ \langle v \mid - \mid \text{TOP} \rangle \Rightarrow_{\text{seq}} v; 0 \]

**SEQSIMPLESTEP**

\[ \langle h \mid k \rangle \rightarrow \langle h' \mid k' \rangle \quad \langle h' \mid k' \rangle \Rightarrow_{\text{seq}} v; n \]

\[ \langle h \mid k \rangle \Rightarrow_{\text{seq}} v; 1 \cdot g \]
The Fully Sequential Evaluation Judgment Revisited

\[
\begin{align*}
\text{SEQVal} & \quad \langle v \mid - \mid \text{TOP} \rangle \Rightarrow_{\text{seq}} v; 0 \\
\text{SEQSimpleStep} & \quad \langle h \mid k \rangle \rightarrow \langle h' \mid k' \rangle \\
& \quad \langle h' \mid k' \rangle \Rightarrow_{\text{seq}} v; n \\
& \quad \langle h \mid k \rangle \Rightarrow_{\text{seq}} v; 1 \cdot g
\end{align*}
\]

Work and span are always equal and do not depend on \( \tau \).
The Fully Parallel Evaluation Judgment

\[ m \Rightarrow_{\text{par}} v; g \]
The Fully Parallel Evaluation Judgment

\[ m \Rightarrow_{\text{par}} v; g \]

\[ \text{PARVAL} \]

\[ \langle v \mid - \mid \text{TOP} \rangle \Rightarrow_{\text{par}} v; 0 \]
The Fully Parallel Evaluation Judgment

\[ m \Rightarrow \text{par } v; g \]

**ParVal**

\[
\langle v \mid - \mid \text{TOP} \rangle \Rightarrow \text{par } v; 0
\]

**ParSimpleStep**

\[
\begin{align*}
 h \neq (- \parallel -) \mid - & \quad \langle h \mid k \rangle \rightarrow \langle h' \mid k' \rangle \quad \langle h' \mid k' \rangle \Rightarrow \text{par } v; g \\
 \langle h \mid k \rangle & \Rightarrow \text{par } v; (1 \cdot g)
\end{align*}
\]
The Fully Parallel Evaluation Judgment

\[ m \Rightarrow_{\text{par}} v; g \]

\textbf{ParVal}

\[ \langle v \mid - \mid \text{TOP} \rangle \Rightarrow_{\text{par}} v; 0 \]

\textbf{ParSimpleStep}

\[ h \neq (- \parallel -) \mid - \quad \langle h \mid k \rangle \rightarrow \langle h' \mid k' \rangle \quad \langle h' \mid k' \rangle \Rightarrow_{\text{par}} v; g \]

\[ \langle h \mid k \rangle \Rightarrow_{\text{par}} v; (1 \cdot g) \]

\textbf{ParPair}

\[ \langle e_1 \mid \sigma \mid \text{TOP} \rangle \Rightarrow_{\text{par}} v_1; g_1 \]

\[ \langle e_2 \mid \sigma \mid \text{TOP} \rangle \Rightarrow_{\text{par}} v_2; g_2 \quad \langle (v_1, v_2) \mid - \mid k \rangle \Rightarrow_{\text{par}} v; g_3 \]

\[ \langle (e_1 \parallel e_2) \mid \sigma \mid k \rangle \Rightarrow_{\text{par}} v; ((g_1 \parallel g_2) \cdot g_3) \]
Towards Lazy Task Creation

Defer task creation until enough sequential work has been done.
Towards Lazy Task Creation

*Defer task creation until enough sequential work has been done.*

Formally:

\[ N \in \mathbb{N} \]

Maintain a work counter \( n \in \mathbb{N} \) incremented at sequential steps.

\[ \frac{m}{n}, \quad N \Rightarrow \text{amo} \quad g \]

Look for \( \text{PAIRL}(\square, e_2, \sigma) \) frames to extract parallelism when \( n \geq N \).
Towards Lazy Task Creation

Defers task creation until enough sequential work has been done.

Formally:

- Assume “enough” is a fixed constant $N \in \mathbb{N}$. 

Towards Lazy Task Creation

Defer task creation until enough sequential work has been done.

Formally:

- Assume “enough” is a fixed constant $N \in \mathbb{N}$.
- Maintain a work counter $n \in \mathbb{N}$ incremented at sequential steps.

\[ m/n, N \xrightarrow{amo} v; g \]
Towards Lazy Task Creation

Deferring task creation until enough sequential work has been done.

Formally:

- Assume “enough” is a fixed constant $N \in \mathbb{N}$.
- Maintain a work counter $n \in \mathbb{N}$ incremented at sequential steps.

$$m/n, N \Rightarrow_{amo} v; g$$

- Look for PAIRL($\square, e_2, \sigma$) frames to extract parallelism when $n \geq N$.

$$\langle h \mid k_1 \quad \circ \quad \text{PAIRL}(\square, e_2, \sigma) :: k_2 \rangle$$
Towards Lazy Task Creation

*Defer task creation until enough sequential work has been done.*

Formally:

- Assume “enough” is a fixed constant $N \in \mathbb{N}$.
- Maintain a work counter $n \in \mathbb{N}$ incremented at sequential steps.

\[
m/n, N \Rightarrow_{amo} v; g
\]

- Look for PAIRL($\Box, e_2, \sigma$) frames to extract parallelism when $n \geq N$.

\[
\langle \begin{array}{c}
h \\
k_1
\end{array} \quad \@ \quad \text{PAIRL($\Box, e_2, \sigma$)} :: k_2 \\
\text{Rest of left branch}
\rangle
\]
Towards Lazy Task Creation

*Defer task creation until enough sequential work has been done.*

Formally:

- Assume “enough” is a fixed constant $N \in \mathbb{N}$.
- Maintain a work counter $n \in \mathbb{N}$ incremented at sequential steps.

$$m/n, N \Rightarrow_{amo} v; g$$

- Look for $\text{PAIRL}(\Box, e_2, \sigma)$ frames to extract parallelism when $n \geq N$.

\[
\left\langle \begin{array}{c}
h \mid k_1 \\
\text{Rest of left branch}
\end{array} \right\rangle \@ \begin{array}{c}
\text{PAIRL}(\Box, e_2, \sigma) \\
\text{Right branch}
\end{array} :: k_2
\]
Towards Lazy Task Creation

*Defer task creation until enough sequential work has been done.*

Formally:
- Assume “enough” is a fixed constant $N \in \mathbb{N}$.
- Maintain a work counter $n \in \mathbb{N}$ incremented at sequential steps.

$$\frac{m}{n}, N \Rightarrow_{amo} v; g$$

- Look for PAIRL($\square, e_2, \sigma$) frames to extract parallelism when $n \geq N$.

\[\langle h \mid k_1 \atop \text{Rest of left branch} \atop \text{Right branch} \atop \text{Final continuation} \rangle \]
The Amortized Evaluation Judgment

\[ m/n, N \Rightarrow_{amo} v; g \]
The Amortized Evaluation Judgment

\[ \frac{m/n, N \Rightarrow_{\text{amo}} v; g}{\text{AMOVal}} \]

\[ \frac{\langle v | - | \text{TOP} \rangle/n, N \Rightarrow_{\text{amo}} v; 0}{\text{AMOVal}} \]
The Amortized Evaluation Judgment

\[ \frac{m/n, N \Rightarrow_{amo} v; g}{AmoVal} \]

\[ \frac{\langle v | - | TOP \rangle/n, N \Rightarrow_{amo} v; 0}{AmoSimpleStep} \]

\[ \frac{n < N \lor PAIRL(\Box, -, -) \notin k}{\langle h | k \rangle \rightarrow \langle h' | k' \rangle \quad \langle h' | k' \rangle/n+1, N \Rightarrow_{amo} v; g}{\langle h | k \rangle/n, N \Rightarrow_{amo} v; (1 \cdot g)} \]
The Amortized Evaluation Judgment

\[
\begin{align*}
m/n, N & \Rightarrow_{amo} \nu; g \\
\hline
\begin{align*}
\text{AmoVal} & \\
\langle \nu \mid - \mid \text{TOP} \rangle/_{n, N} & \Rightarrow_{amo} \nu; 0 \\
\hline
\text{AmoSimpleStep} & \\
n < N \lor \text{PAIRL}(\square, -,-) \notin k \\
\langle h \mid k \rangle & \rightarrow \langle h' \mid k' \rangle \\
\langle h' \mid k' \rangle/_{n+1, N} & \Rightarrow_{amo} \nu; g \\
\langle h \mid k \rangle/_{n, N} & \Rightarrow_{amo} \nu; (1 \cdot g) \\
\hline
\text{AmoPromote} & \\
n \geq N \\
\text{PAIRL}(\square, -,-) \notin k_2 \\
\langle e_2 \mid \sigma \mid \text{TOP} \rangle/_{0, N} & \Rightarrow_{amo} \nu_2; g_2 \\
\langle (\nu_1, \nu_2) \mid - \mid k_2 \rangle/_{0, N} & \Rightarrow_{amo} \nu; g_3 \\
\langle h \mid k_1 \circ \text{PAIRL}(\square, e_2, \sigma) :: k_2 \rangle/_{n, N} & \Rightarrow_{amo} \nu; ((g_1 || g_2) \cdot g_3)
\end{align*}
\end{align*}
\]
1 Introduction
2 The Semantics of Lazy Task Creation
3 Efficiency and Soundness Results
4 Perspectives
Work Bound

Statement

If \( \langle e \mid \emptyset \mid \text{TOP} \rangle_{0,N} \Rightarrow_{\text{amo}} v; g_a \), there is \( g_s \) s.t. \( \langle e \mid \emptyset \mid \text{TOP} \rangle \Rightarrow_{\text{seq}} v; g_s \).

Moreover, we have:

\[
W_a \leq (1 + \frac{\tau}{N}) \cdot W_s
\]

with \( W_a \) and \( W_s \) the work of \( g_a \) and \( g_s \), respectively.
Statement

If $\langle e \mid \emptyset \mid \text{TOP} \rangle_{/0,N} \Rightarrow_{\text{amo}} v; g_a$, there is $g_s$ s.t. $\langle e \mid \emptyset \mid \text{TOP} \rangle \Rightarrow_{\text{seq}} v; g_s$. Moreover, we have:

$$W_a \leq (1 + \frac{\tau}{N}) \cdot W_s$$

with $W_a$ and $W_s$ the work of $g_a$ and $g_s$, respectively.

Proof ingredients:

- The “inductive” bound is $W_a \leq (1 + \frac{\tau}{N}) \cdot W_s + n \cdot \frac{\tau}{N}$. 
Work Bound

Statement

If $\langle e \mid \emptyset \mid \text{TOP} \rangle_{/0,N} \Rightarrow_{\text{amo}} v; g_a$, there is $g_s$ s.t. $\langle e \mid \emptyset \mid \text{TOP} \rangle \Rightarrow_{\text{seq}} v; g_s$. Moreover, we have:

$$W_a \leq (1 + \frac{\tau}{N}) \cdot W_s$$

with $W_a$ and $W_s$ the work of $g_a$ and $g_s$, respectively.

Proof ingredients:

- The “inductive” bound is $W_a \leq (1 + \frac{\tau}{N}) \cdot W_s + n \cdot \frac{\tau}{N}$.
- If $\langle h \mid k_1 \rangle \rightarrow \langle h' \mid k'_1 \rangle$ then $\langle h \mid k_1 \odot k_2 \rangle \rightarrow \langle h' \mid k'_1 \odot k_2 \rangle$. 
### Statement

If \( \langle e \mid \emptyset \mid \text{TOP} \rangle_{/0,N} \Rightarrow_{\text{amo}} v; g_a \), there is \( g_s \) s.t. \( \langle e \mid \emptyset \mid \text{TOP} \rangle \Rightarrow_{\text{seq}} v; g_s \).

Moreover, we have:

\[
W_a \leq (1 + \frac{\tau}{N}) \cdot W_s
\]

with \( W_a \) and \( W_s \) the work of \( g_a \) and \( g_s \), respectively.

### Proof ingredients:

- The “inductive” bound is \( W_a \leq (1 + \frac{\tau}{N}) \cdot W_s + n \cdot \frac{\tau}{N} \).
- If \( \langle h \mid k_1 \rangle \rightarrow \langle h' \mid k'_1 \rangle \) then \( \langle h \mid k_1 @ k_2 \rangle \rightarrow \langle h' \mid k'_1 @ k_2 \rangle \).
- The following rule is derivable.

\[
\text{SEQCUT}  \\
\langle h \mid k_1 \rangle \Rightarrow_{\text{seq}} v_1; g_1 \quad \langle v_1 \mid - \mid k_2 \rangle \Rightarrow_{\text{seq}} v_2; g_2  \\
\langle h \mid k_1 \circledast k_2 \rangle \Rightarrow_{\text{seq}} v_2; g_1 \cdot g_2
\]
Span Bound

Statement

If $\langle e \mid \emptyset \mid \text{TOP} \rangle_{/0,N} \Rightarrow_{\text{amo}} v; g_a$, there is $g_p$ s.t. $\langle e \mid \emptyset \mid \text{TOP} \rangle \Rightarrow_{\text{par}} v; g_p$. Moreover, we have:

$$S_a \leq (1 + \frac{N}{\tau}) \cdot S_p$$

with $S_a$ and $S_p$ the span of $g_a$ and $g_p$, respectively.
If \( \langle e \mid \emptyset \mid \text{TOP} \rangle_{/0,N} \Rightarrow_{\text{amo}} v; g_a \), there is \( g_p \) s.t. \( \langle e \mid \emptyset \mid \text{TOP} \rangle \Rightarrow_{\text{par}} v; g_p \).

Moreover, we have:

\[
S_a \leq (1 + \frac{N}{\tau}) \cdot S_p
\]

with \( S_a \) and \( S_p \) the span of \( g_a \) and \( g_p \), respectively.

Proof ingredients:

- The “inductive” bound is \( S_a \leq (1 + \frac{N}{\tau}) \cdot S_p - (p \ ? \ min(n, N) : 0) \cdot \tau \)
  with \( p \) true iff \( \text{PAIRL}(\square, -, -) \in k \).
Span Bound

Statement

If $\langle e \mid \emptyset \mid \text{TOP} \rangle_{/0, N} \Rightarrow_{amo} \nu; g_a$, there is $g_p$ s.t. $\langle e \mid \emptyset \mid \text{TOP} \rangle \Rightarrow_{par} \nu; g_p$.

Moreover, we have:

$$S_a \leq \left(1 + \frac{N}{\tau}\right) \cdot S_p$$

with $S_a$ and $S_p$ the span of $g_a$ and $g_p$, respectively.

Proof ingredients:

- The “inductive” bound is $S_a \leq \left(1 + \frac{N}{\tau}\right) \cdot S_p - \left(p \ ? \ min(n, N) : 0\right) \cdot \tau$ with $p$ true iff $\text{PAIRL}(\Box, \_, \_) \in k$.

- The proof uses an extended parallel judgment which eagerly evaluates frames of shape $\text{PAIRL}(\Box, e_2, \sigma)$ in parallel.
Span Bound

Statement

If \( \langle e \mid \emptyset \mid \text{TOP} \rangle_{/0,N} \Rightarrow_{\text{amo}} v; g_a \), there is \( g_p \) s.t. \( \langle e \mid \emptyset \mid \text{TOP} \rangle \Rightarrow_{\text{par}} v; g_p \).

Moreover, we have:

\[
S_a \leq (1 + \frac{N}{\tau}) \cdot S_p
\]

with \( S_a \) and \( S_p \) the span of \( g_a \) and \( g_p \), respectively.

Proof ingredients:

- The “inductive” bound is \( S_a \leq (1 + \frac{N}{\tau}) \cdot S_p - (p \oplus \min(n, N) : 0) \cdot \tau \) with \( p \) true iff \( \text{PAIRL}(\Box, -, -) \in k \).

- The proof uses an extended parallel judgment which eagerly evaluates frames of shape \( \text{PAIRL}(\Box, e_2, \sigma) \) in parallel.

- If \( \langle h \mid k_1 \odot k_2 \rangle \Rightarrow_{\text{par}} v; g \) with \( \text{PAIRL}(\Box, -, -) \notin k_2 \), there is a value \( v_1 \) s.t. \( \langle h \mid k_1 \rangle \Rightarrow_{\text{par}} v_1; g_1 \) and \( \langle v_1 \mid - \mid k_2 \rangle \Rightarrow_{\text{par}} v; g_2 \) with \( g = g_1 \cdot g_2 \).
Soundness

**Sequential-to-amortized**

If $m \Rightarrow_{\text{seq}} v; g_s$ then for any $n, N$ there is $g_a$ s.t. $m/n, N \Rightarrow_{\text{amo}} v; g_a$.

**Parallel-to-amortized**

If $m \Rightarrow_{\text{par}} v; g_p$ then for any $n, N$ there is $g_a$ s.t. $m/n, N \Rightarrow_{\text{amo}} v; g_a$.

**Amortized-to-amortized**

If $m/n, N \Rightarrow_{\text{amo}} v; g_a$ then for any $n', N'$ there is $g'_a$ s.t. $m/n', N' \Rightarrow_{\text{amo}} v; g'_a$.

Remarks:
- We have not yet proved amortized-to-amortized soundness.
- The other results mostly rely on stack decomposition lemmas.
Soundness

Sequential-to-amortized

If $m \Rightarrow_{\text{seq}} v; g_s$ then for any $n, N$ there is $g_a$ s.t. $m/n, N \Rightarrow_{\text{amo}} v; g_a$.

Parallel-to-amortized

If $m \Rightarrow_{\text{par}} v; g_p$ then for any $n, N$ there is $g_a$ s.t. $m/n, N \Rightarrow_{\text{amo}} v; g_a$.

Amortized-to-amortized

If $m/n, N \Rightarrow_{\text{amo}} v; g_a$ then for any $n', N'$ there is $g'_a$ s.t. $m/n', N' \Rightarrow_{\text{amo}} v; g'_a$.

Remarks:

- We have not yet proved amortized-to-amortized soundness.
- The other results mostly rely on stack decomposition lemmas.
Definitions, statements, and proofs have been mechanized in Coq.

- The complete development spans 758 LoC at the moment.
- No big complication with respect to pen-and-paper proofs.
  - No binder issues since substitutions are not involved.
- All results were first proved in Coq except for the work bound.
Outline

1 Introduction

2 The Semantics of Lazy Task Creation

3 Efficiency and Soundness Results

4 Perspectives
A Library Implementation of Lazy Task Creation

- Written in C++ with a little bit of x86 assembly.
- Much less naive than what I presented.
  - Compiled rather than interpreted.
  - Relies on techniques used for implementing control operators.
  - Measure execution time much more concretely (wall-clock time).
- Offers parallel for loops in addition to parallel pairs.
  - Generalize binary fork/join to n-ary fork/join.
  - Expose more information to the runtime system.
- Performance looks at least as good as in comparable systems.
Future Work

In the short term:

▶ Finish and clean up the Coq development.
▶ Add and study n-ary fork/join, probably as a looping construct.
▶ Complement the theoretical study with more empirical data.

Longer-term possibilities:

▶ Understand the limitations of the technique beyond fork/join.
▶ Justify the cost semantics with respect to a lower-level semantics.
▶ Study the metatheory of cost semantics?
Conclusion

I presented a formal description of Lazy Task Creation, a granularity control technique used in parallel runtime systems.

- The description takes the form of a relatively simple mix of big-step and small-step operational semantics.
- Defining the cost of each execution makes it possible to study the (idealized) performance of programs in a high-level way.
I presented a formal description of Lazy Task Creation, a granularity control technique used in parallel runtime systems.

- The description takes the form of a relatively simple mix of big-step and small-step operational semantics.
- Defining the cost of each execution makes it possible to study the (idealized) performance of programs in a high-level way.

This formal description makes it possible to prove the first analytic bounds on the overhead of Lazy Task Creation.
Conclusion

I presented a formal description of Lazy Task Creation, a granularity control technique used in parallel runtime systems.

- The description takes the form of a relatively simple mix of big-step and small-step operational semantics.
- Defining the cost of each execution makes it possible to study the (idealized) performance of programs in a high-level way.

This formal description makes it possible to prove the first analytic bounds on the overhead of Lazy Task Creation.

In particular, we showed that with $N$ set to $\tau$, Lazy Task Creation properly amortizes scheduling overheads while still preserving parallelism.