DAG-Calculus: A Calculus for Parallel Computation

Umut Acar
Carnegie Mellon University

Arthur Charguéraud
Inria & LRI Université Paris Sud, CNRS

Mike Rainey
Inria

Filip Sieczkowski
Inria

Gallium Seminar
February 2016
Parallel Computation

• Crucial for efficiency in the age of multicores

• Many different modes of use and language constructs

• Stress generally on efficiency, semantics not as well understood
Fibonacci with fork-join

function fib (n)
    if n <= 1 then n else
        let (x, y) = forkjoin (fib (n-1), fib (n-2)) in
        x + y

1. Perform two recursive calls in parallel
2. Evaluate the result
function fib (n)
  if n <= 1 then n else
  let (x,y) = (ref 0, ref 0) in
  finish { async (x := fib (n-1));
      async (y := fib (n-2)) }
  ![x] + ![y]

1. Perform two recursive calls in parallel
2. Synchronise on completion of parallel calls
3. Read results of calls and compute final result
Fibonacci with futures

function fib (n)
    if n <= 1 then n else
    let (x,y) = (future fib (n-1), future fib (n-2))
    in (force x) + (force y)

1. Perform two recursive calls in parallel

2 & 3. Demand results from the parallel calls, and evaluate the final result
Motivating questions

Is there a unifying model or calculus that can be used to express and study different forms of parallelism?

Can such a calculus be realised efficiently in practice as a programming language, which can serve, for example, as a target for compiling different forms of parallelism?
Parallelism patterns: fork-join, async-finish

\[
\text{fib}(n) \rightarrow \text{fib}(n-1), \text{fib}(n-2) \rightarrow K[ ]
\]
Parallelism patterns: futures

fib(n) -> fib(n-1) -> fib(n-2) -> K[force(x)]

x -> K[force(y)]
Core idea — reify the dependency edges
Computing with DAGs

- State of computation: \((V, E, \sigma)\)
- \(V\) — a set of DAG vertices, each with associated program and status
- \(E\) — a set of DAG edges
- \(\sigma\) — shared mutable state
- Side conditions enforce edges form DAG, status of vertices, etc.
Manipulating the DAG

- Four commands that are used to dynamically modify the computation DAG
  - `newTd(e)` creates a new vertex in the DAG
  - `newEdge(e, e')` creates an edge from $e$ to $e'$
  - `release(e)` is called to mark that the vertex $e$ is now set up and can be scheduled
  - `transfer(e)` is a parallel control operator that transfers the outgoing edges of calling thread to $e$
The life-cycle of a vertex

- Vertex can have one of four status values: New, Released, eXecuting or Finished (N, R, X, F)
- Node created by running `newTd` has status N
- After calling `release`, its status is changed to R
- At this point it can be *scheduled* for execution, which sets the status to X
- After the execution terminates, the status is set to F
Operational Semantics (1)

\[
V(t) = (K[\text{newTd } e], X) \quad t' \text{ fresh} \\
V, E, \sigma \rightarrow V[t \mapsto (K[t'], X)][t' \mapsto (e, N)], E, \sigma
\]

**NEWTD**

\[
V(t) = (K[\text{release } t'], X) \\
V(t') = (e, N) \\
V, E, \sigma \rightarrow V[t \mapsto (K([ ]), X)][t' \mapsto (e, R)], E, \sigma
\]

**RELEASE**

\[
V(t) = (e, R) \quad \{t' \mid (t', t) \in E\} = \emptyset \\
V, E, \sigma \rightarrow V[t \mapsto (e, X)], E, \sigma
\]

**START**

\[
V(t) = (e_1, X) \quad \sigma_1, e_1 \rightarrow e_2, \sigma_2 \\
V, E, \sigma_1 \rightarrow V[t \mapsto (e_2, X)], E, \sigma_2
\]

**STEP**

\[
V(t) = (v, X) \quad E' = E \setminus \{(t, t') \mid t' \in \text{dom}(V)\} \\
V, E, \sigma \rightarrow V[t \mapsto ([ ], F)], E', \sigma
\]

**STOP**
Operational Semantics (2)

\[
V(t) = (K[\text{newEdge } t_1 t_2], X) \quad t_1, t_2 \in \text{dom}(V) \\
\text{status}(V(t_2)) \in \{N, R\} \quad E' \text{ cycle-free} \\
E' = E \cup \{(t_1, t_2)\} \\
V, E, \sigma \rightarrow V[t \mapsto (K(\_), X)], E', \sigma \quad \text{NEW EDGE}
\]

\[
V(t) = (K[\text{transfer } t'], X) \quad \text{status}(V(t')) = N \quad \{t'' \mid (t', t'') \in E\} = \emptyset \\
E' = E \setminus \{(t, t'') \mid t'' \in \text{dom}(V)\} \cup \{(t', t'') \mid (t, t'') \in E\} \quad E' \text{ cycle-free} \\
V, E, \sigma \rightarrow V[t \mapsto (K(\_), X)], E', \sigma \quad \text{TRANSFER}
\]
Encoding fork-join

\[\text{forkjoin}(e_1, e_2)\] =
\[
capture (\text{fn } k =>
\[
\text{let } l1 = \text{alloc}
\]
\[
\quad l2 = \text{alloc}
\]
\[
\text{t1 } = \text{newTd } (l1 := [e_1])
\]
\[
\text{t2 } = \text{newTd } (l2 := [e_2])
\]
\[
\text{t } = \text{newTd } (k (!l1, !l2))
\]
\][newEdge(t1, t); newEdge(t2, t)
\][transfer(t); release(t);
\][release(t1); release(t2))]
Encoding async-finish

\[
[t \mid \text{async}(e)] = \\
\text{let } t' = \text{newTd}( [t \mid e] ) \\
\text{in newEdge}(t', t); \text{release}(t')
\]

\[
[t \mid \text{finish}(e)] = \\
\text{capture (fn } k \Rightarrow \\
\text{let } t2 = \text{newTd}(k ()) \\
\text{t1} = \text{newTd}( [t2 \mid e] ) \\
\text{in newEdge}(t1, t2); \text{transfer}(t2); \text{release}(t2); \text{release}(t1))
\]
Encoding futures

\[
\text{future}(e) = \\
\text{let } l = \text{alloc} \\
\quad t = \text{newTd}() \\
\quad t' = \text{newTd}(l := [e]) \\
\quad \text{in newEdge}(t', t); \text{release}(t); \text{release}(t'); (t, l)
\]

\[
\text{force}(e) = \\
\text{let } (t, l) = [e] \\
\quad \text{in capture (fn } k \Rightarrow \\
\quad \text{let } t' = \text{newTd} (k (l!)) \\
\quad \text{in newEdge}(t, t'); \\
\quad \text{transfer}(t'); \text{release}(t')
\]
Proving the encodings correct: the technique

• Compiler-style proof of simulation

• Need *backwards*-simulation due to nondeterminism

• Problem: partially evaluated encodings do not correspond to any source terms

• Solution: an intermediate, annotated language, two-step proof

• Keep the structure *and* allow partial evaluation of parallel primitives
Also in the paper

- Scheduling DAG-calculus computations using work stealing
- Data-structures for efficient implementation of DAG edges
- Experimental evaluation of implementation
Conclusions

• A unifying calculus: common framework for expressing different modes of parallelism

• A low-level calculus: useful as an intermediate language/mental model rather than directly

• An efficient implementation using novel data-structures to handle high-degree vertices