DAG-Calculus: A Calculus for Parallel Computation

Umut A. Acar
Carnegie Mellon University and Inria
umut@cs.cmu.edu

Arthur Charguéraud
Inria
arthur@inria.fr

Mike Rainey
Inria
rainey@inria.fr

Filip Sieczkowski
Inria
filip@inria.fr

Abstract

Hardware advances in the last decade has brought parallelism to masses, leading to the development of programming languages for writing parallel programs. As in other concurrent systems, these parallel languages offer several different abstractions for parallelism, such as fork-join, async-finish, futures, etc. While they may seem similar, these abstractions lead to different semantics, language design and implementation decisions.

We consider the question of whether it would be possible to unify different approaches to parallelism. To this end, we propose a calculus, called *DAG-calculus* that can encode existing approaches to parallelism based on fork-join, async-finish, and futures paradigms and possibly others. We describe the DAG calculus and its semantics, establish translations from the aforementioned paradigms, and present concurrent data structures for realizing it in practice. We show that the approach is realistic by presenting an implementation in C++ and performing an empirical evaluation, which shows the proposed techniques to be competitive with specific custom, highly optimized systems such as Cilk.

Keywords Calculus, operational semantics, parallelism, proofs, experiments, concurrent data structures.

1. Introduction

Hardware advances in the past decade made parallel chips, such as multicore chips, widely available. This development led much work on the design of programming languages and systems for writing parallel programs, especially for shared-memory parallel machines such as multicore computers. Example languages include Cilk [16], Fork/Join Java [29], Habanero Java [24], NESL [7], parallel Haskell [11, 26, 31], parallel ML [15, 43], TPL [30], and X10 [12].

These programming languages, typically developed as an extension to an existing language (such as C, Haskell, Java, and ML), provide support for parallelism primarily by using several broadly accepted paradigms, including: fork-join parallelism, async-finish parallelism, and futures. The fork-join and async-finish constructs, which can be used to express nested-parallel computations, are similar but async-finish is more flexible, because it allows arbitrarily many computations to synchronize. The Cilk programming language is primarily based on fork-join parallelism but also offers limited support for async-finish, disallowing for example their nesting. The X10 and the Habanero Java languages support both async-finish and futures. In functional languages such as Haskell and ML, futures offers a powerful mechanism for expressing parallelism, primarily because they are first-class values. In fact, futures have originated from work on functional programming language (e.g., [18]); they can now be found in many languages including C [1] and Java. The Parallel Haskell and Parallel ML languages support both futures and fork-join parallelism.

The diversity of parallelism abstractions raise several 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 realized efficiently in practice as a programming language, which can serve, for example, as a target for compiling different forms of parallelism?

In this paper, we answer these questions affirmatively. We propose a calculus, called DAG calculus, that allows expressing a broad class of parallel computations (Section 3). We then prove that DAG calculus can serve as a target language for programs written in fork-join (Section 4.1), async-finish (Section 4.2), and futures (Section 4.3) by showing translations

[Copyright notice will appear here once 'preprint' option is removed.]

from parallel programming languages with these constructs to DAG calculus. We show that DAG calculus is realizable in practice by designing and implementing the key data structures and algorithms needed (Section 5), and by evaluating its effectiveness (Section 6).

The starting point for DAG calculus is a well-known idea: many parallel computations can be represented as directed-acyclic graphs (or DAG's), where vertices represent sequential pieces of computation, variously called threads, tasks, strands, fibers etc., and edges represent the dependencies between them. This idea is broadly exploited in algorithms research to express algorithms [25] and also to analyze crucial scheduling algorithms used for parallel programming languages (e.g., [3, 9, 17]). In both approaches, the idea is to represent mathematically a parallel computation as a DAG and study the algorithmic properties of interest.

This algorithmic approach, however, does not work for a calculus, because it represents a computation with a known and unchanging DAG, which corresponds a specific execution. In contrast, a calculus or a programming language must treat computations abstract and discover their structure via evaluation. Since we want to capture parallel computations as broadly as possible, we propose DAG calculus as a system where evaluation dynamically constructs a DAG and where evaluation itself controlled by the structure of the DAG. This gives programmer full control over the structure of the DAG and thus over crucial aspects of parallelism.

Vertices of the DAG are labeled with expressions represent threads (units of parallelism) and edges represent dependencies between threads. Evaluation proceeds by taking the set of vertices whose parents (dependencies) have been completed and evaluates in some non-deterministic order their associated expressions. When evaluated an expression performs some computation and also possibly creates new vertices and edges. Evaluation can thus be viewed as a concurrent interleaving of conventional computation and updates to the structure of the DAG. We note that the operational semantics model scheduling via non-determinism; it is thus compatible with any appropriately defined scheduling algorithm.

We formalize this intuitive description of the DAG calculus by specifying its syntax and semantics (Section 3). DAG calculus includes expressions for creating vertices and edges, as well as another primitive called `transfer` that can be used to replace a vertex with a whole another DAG, essentially allowing parallel computations to be nested arbitrarily deeply, which is essential to expressiveness [6]. In addition, DAG calculus includes several standard control operators for capturing and operating on continuations. Continuations are not required for programming in DAG calculus, but are useful for encoding high-level parallelism constructs such as fork-join, async-finish, and futures.

A practical, efficient realization of DAG calculus raises a key challenge: how to operate on the DAG concurrently and efficiently in parallel. In particular, the data structure must avoid expensive atomic operations such as blocking locks, minimize contention, and a high degree of parallelism. This leads to three subproblems: (1) concurrent insertion of edges, (2) parallel traversal of the outgoing edges of an executed vertex, and (3) fast detection of vertices that become ready for execution. We solve these problems by presenting a non-blocking, concurrent DAG data structure (Section 5).

Based on the concurrent DAG data structure, we give an actual implementation of DAG calculus, written as a C++ library. We implement the proposed encodings for fork-join, async-finish, and futures in this library, empirically verifying that they indeed work as theoretically established. We perform an evaluation to show that the proposed techniques perform well in practice (Section 6). In addition to a suite of benchmarks, we also present evaluation of our data structures under high contention.

The contributions of the paper include the following.

- DAG-calculus: the specific constructs and their semantics.
- Encodings of fork-join, async-finish, and futures in DAG calculus, and their proof of correctness.
- Concurrent data structures for computation DAG's.
- An optimized C++ implementation and its evaluation.

2. Background

We discuss, by means of a simple example, the different abstractions for parallelism used in different languages and present a brief comparison. The knowledgeable reader can skip this section. We use a simple pseudocode language based on a strict functional language such as the ML family.

Fork-join. Consider the very simple example of computing Fibonacci numbers recursively. In fork-join parallelism, we can use the primitive **forkjoin** to perform two computations in parallel, and return the pair of their results.

```
function fib (n)    (* Fibonacci with fork–join *)
  if n <= 1 then n else
    let (x,y) = forkjoin (fib (n-1), fib (n-2)) in
    x + y
```

Although calls to fork-join may be nested to obtain more than two parallel threads, the async-finish construct provides a more direct, more flexible, and possibly more efficient way of spawning subcomputations in arbitrary number.

Async-finish. The async-finish construct (called "spawn-sync" in Cilk) provides a common join point for unbounded number of threads. This construct is more powerful than the fork-join pattern, yet at the same time less-structured: an **async**'ed computation has no return value—such return values must be communicated through the memory store. The scoping is dynamic: we may use **async** within an expression to spawn a thread within the **finish** block that encloses the expression. The **finish** block only returns once all **async**'ed computations started in its block have completed. The code for parallel Fibonacci can be expressed as follows.

$$
\begin{aligned}
e &::= v \mid e \oplus e \mid e \otimes e \mid \langle e, e \rangle \mid \pi_i\ e \mid\ e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \\
&\quad \texttt{alloc} \mid e := e \mid\ !e \mid \texttt{capture}\ e \mid \texttt{abort}\ e \mid \\
&\quad \texttt{newTd}\ e \mid \texttt{release}\ e \mid \texttt{newEdge}\ e\ e \mid \texttt{transfer}\ e \\
v &::= x \mid \ell \mid \texttt{t} \mid \texttt{n} \mid \texttt{()} \mid (v, v) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \\
K &::= \bullet \mid \texttt{let}\ x = K\ \texttt{in}\ e \mid K := e \mid v := K \mid\ !\,K \mid \texttt{release}\ K \\
&\quad \mid \texttt{newEdge}\ K\ e \mid \texttt{newEdge}\ v\ K \mid \texttt{transfer}\ K \mid \ldots
\end{aligned}
$$

Figure 1. Abstract syntax for DAG-Calculus.

```
function fib (n)        (* Fibonacci with async−finish *)
  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
```

Futures. Futures are perhaps the most functional form of parallelism. A future is a first-class value that captures a computation. It is created by an expression of the form **future** `e`. The body of the future, the expression `e`, can be evaluated in parallel as soon as the future has been created. The result of the future may be obtained using the **force** operation, which blocks until the body of the future completes.

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

Comparison. Async-finish parallelism and futures are strictly more powerful than fork-join parallelism: fork-join programs are easily expressible both in terms of async-finish and in terms of futures, but not the other way around. Between async-finish and futures, however, none of the two constructs appears to be strictly superior to the other. Intuitively, they are somewhat dual: futures impose synchronization by enforcing a data dependency but leave the control flow implicit, whereas async-finish leaves data dependencies implicit and makes parallelism explicit in the control flow. It is thus perhaps not surprising that both futures and async-finish have remained popular in many programming languages.

3. DAG Calculus

In this section we present a calculus for parallel computations, named DAG-calculus, and sometimes written as λ^{DC} for short. The calculus extends an imperative λ-calculus with primitives for dynamically creating a Directed Acyclic Graph (DAG) that represents the computation. We refer to such a DAG as a *computation DAG*, or simply as a *DAG*.

Abstract Syntax. The syntax of our calculus appears in Figure 1. Meta-variables x, y, f denote variables, `n` denotes a natural number, ℓ denotes a location, and `t` denotes the identity of a thread. The calculus includes a unit value (written `()`), pairs, general recursion, references. It also includes control operators, called `capture` and `abort`, for capturing the (sequential) continuation of a thread. We use these operators to encode high-level parallel constructs, e.g. fork-join, into our DAG calculus. Note that we do not use these operators in their full generality: we only require one-shot continuations, which admit an efficient implementation. For operating on the DAG, we introduce four language constructs, called `newTd`, `release`, `newEdge`, `transfer`.

The expression `newTd` e creates a new thread for evaluating the expression e, and returns the corresponding thread identifier. The expression `release` e, where e evaluates to a thread identifier t, is used to mark the end of the setup phase for the thread t. In other words, each call to `newTd` is paired with a subsequent call to `release`, to indicate that the freshly created thread is ready to be scheduled for evaluation —without such a pairing, a freshly created thread could be scheduled before its dependency edges are added.

Next, the expression `newEdge` $e_1\ e_2$ inserts an edge from thread e_1 to the thread e_2. We explain shortly afterwards the restrictions that apply to the addition of edges into the DAG.

Last, `transfer` e, where e evaluates to a thread identifier t, transfers all the outgoing edges of the thread which executes the expression `transfer` e to the thread t. Here, we assume that t has no outgoing edge and that t is not yet ready to execute. Note that `transfer` is meant to be called at most once per thread execution. This transfer operation can be thought of as capturing the *parallel continuation* of the currently-running thread, and attaching it to another tread t.

Dynamic Semantics. We present the dynamic semantics for the DAG calculus as a contextual semantics, with a reduction relation over computation DAG's. Formally, a computation DAG is a pair (V, E). Vertices are described by the map V, which binds each thread identifier to an expression and to a *status*. A thread status, written s, is one of: *new*, *released*, *executing*, and *finished*, written as N, R, X, and F, respectively. Edges are described by the set E, which contains pairs of valid thread identifiers.

The dynamic semantics for λ^{DC}, described in Figure 2, involves two judgments: one for *thread-local reductions* and one for *DAG reductions*. The first judgment, written $\sigma, e \rightarrow v, \sigma$, relates an input store and an expression with an output value and an output store. Its reduction rules, which have no impact on the computation DAG, are standard. They include the evaluation relation for pairs (Fst, Snd), application (Apply), as well as rules for memory allocation, dereference, assignment (Alloc, Deref, Assign), which are the only instructions that modify the store.[1] The rules Capture and Abort are explained further.

The judgment for parallel steps, written $V, E, \sigma \twoheadrightarrow V', E', \sigma'$, describes the evolution of the DAG, represented with V and E, and of the store σ. The rule Start selects

[1] For simplicity, we here assume a sequentially-consistent store. However, our semantics could accommodate a weaker memory model, e.g., for x86-TSO, the store σ can include the write buffer of each processor [34].

$$\frac{}{\sigma, \texttt{fst}\ (v_1, v_2) \to v_1, \sigma}\ \textsc{Fst} \qquad \frac{}{\sigma, \texttt{snd}\ (v_1, v_2) \to v_2, \sigma}\ \textsc{Snd} \qquad \frac{l \notin \mathrm{dom}(\sigma)}{\sigma, \texttt{alloc} \to l, \sigma[l \mapsto ()]}\ \textsc{Alloc} \qquad \frac{\sigma(l) = v}{\sigma, !\, l \to v, \sigma}\ \textsc{Deref}$$

$$\frac{l \in \mathrm{dom}(\sigma)}{\sigma,\ (l := v) \to (), \sigma[l \mapsto v]}\ \textsc{Assign} \qquad \frac{}{\sigma, ((\texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end})\ v) \to e[f \mapsto \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}][x \mapsto v], \sigma}\ \textsc{Apply}$$

$$\frac{k\ =\ \texttt{fun}\ f\ x\ \texttt{is abort}\ (K[x])\ \texttt{end}}{\sigma, K[\texttt{capture}\ e] \to e\ k, \sigma}\ \textsc{Capture} \qquad \frac{}{\sigma, K[\texttt{abort}\ e] \to e, \sigma}\ \textsc{Abort}$$

$$\frac{\sigma, e \to e', \sigma' \qquad \forall K', e''.\ e \notin \{K'[\texttt{capture}\ e''], K'[\texttt{abort}\ e'']\}}{\sigma, K[e] \to K[e'], \sigma'}\ \textsc{Context}$$

$$\frac{V(t) = (e, \mathsf{R}) \qquad \{t' \mid (t', t) \in E\} = \emptyset}{V, E, \sigma \twoheadrightarrow V[t \mapsto (e, \mathsf{X})], E, \sigma}\ \textsc{Start} \qquad \frac{V(t) = (e_1, \mathsf{X}) \qquad \sigma_1, e_1 \to e_2, \sigma_2}{V, E, \sigma_1 \twoheadrightarrow V[t \mapsto (e_2, \mathsf{X})], E, \sigma_2}\ \textsc{Step}$$

$$\frac{V(t) = (v, \mathsf{X}) \qquad E' = E \setminus \{(t, t') \mid t' \in \mathrm{dom}(V)\}}{V, E, \sigma \twoheadrightarrow V[t \mapsto ((), \mathsf{F})], E', \sigma}\ \textsc{Stop} \qquad \frac{V(t) = (K[\texttt{newTd}\ e], \mathsf{X}) \qquad t'\ \ \mathsf{fresh}}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[t'], \mathsf{X})][t' \mapsto (e, \mathsf{N})], E, \sigma}\ \textsc{NewTd}$$

$$\frac{\begin{array}{c} V(t) = (K[\texttt{release}\ t'], \mathsf{X}) \\ V(t') = (e, \mathsf{N}) \end{array}}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[()], \mathsf{X})][t' \mapsto (e, \mathsf{R})], E, \sigma}\ \textsc{Release} \qquad \frac{\begin{array}{c} V(t) = (K[\texttt{newEdge}\ t_1\ t_2], \mathsf{X}) \qquad t_1, t_2 \in \mathrm{dom}(V) \\ \mathrm{status}(V(t_2)) \in \{\mathsf{N}, \mathsf{R}\} \qquad E' \text{ cycle-free} \\ E'\ =\ E \uplus (\text{if } \mathrm{status}(V(t_1)) = \mathsf{F} \text{ then } \emptyset \text{ else } \{(t_1, t_2)\}) \end{array}}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[()], \mathsf{X})], E', \sigma}\ \textsc{NewEdge}$$

$$\frac{\begin{array}{c} V(t) = (K[\texttt{transfer}\ t'], \mathsf{X}) \qquad \mathrm{status}(V(t')) = \mathsf{N} \qquad \{t'' \mid (t', t'') \in E\} = \emptyset \\ E'\ =\ E \setminus \{(t, t'') \mid t'' \in \mathrm{dom}(V)\} \uplus \{(t', t'') \mid (t, t'') \in E\} \qquad E' \text{ cycle-free} \end{array}}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[()], \mathsf{X})], E', \sigma}\ \textsc{Transfer}$$

Figure 2. Dynamic semantics for λ^{DC}. Recall that thread status are: N (new), R (released), X (executing) and F (finished). Besides, we write "status($V(t)$)" to denote the status of thread t, i.e. the second component of $V(t)$.

a thread that is ready to execute, i.e. that has status R (released) and that has no incoming edges, and sets its status to X (executing). The rule Step reduces an expression in a thread t with status X (executing). The rule Stop applies to a thread t whose computation has completed. It updates the thread status from X (executing) to F (finished), and removes from the DAG all the outgoing edges from t. Note that the value produced by the thread, typically the unit value, is "dropped on the floor" and is not communicated to other threads along the dependency edges. Indeed, our calculus, which aims at being close to an implementation, requires values to be communicated through the memory store.

The rule NewTd creates a new thread identified as t', associates with it the expression e and the status N (new), and returns t'. The rule Release changes the status of the thread specified, namely t', from N (new) to R (released).

The rule NewEdge adds a dependency edge between two given threads, called t_1 and t_2. It is the programmer's responsibility to not introduce cyclic dependencies, otherwise the program will be stuck. Furthermore, we restrict the addition of an edge by requiring that the target vertex t_2 has not already started executing. Typically, the programmer would only request the addition of edges when it can be deduced from the invariants of the program that the target vertex has at least one incoming edge, and that this edge cannot be concurrently removed. Besides, note that the vertex t_1 at the source of the edge might already have completed its execution when `newEdge` is called. In this case, the operation is simply a no-op, in the sense that no edge is being added to the DAG.

The rule Transfer describes the migration of the set of outgoing edges associated with the currently-running thread, called t, to another given thread, called t'. This operation is only permitted when t' has status N (new) and has no outgoing edges. These two restrictions are driven by our use of `transfer` in practice, and they allow for efficient implementation of this operation, in particular avoiding the need to merge sets of outgoing edges.

The rule Capture and Abort describe the standard semantics of these control operators. The capture operator was introduced by Felleisen and Friedman [14] (where it is written C). It reifies the sequential evaluation context as a first-class value, and passes it to its argument. Formally, by rule Capture, an expression of a form K[`capture` e] for some context K reduces to the application $e\ k$, where k denotes the first-call reification of the context. The continuation k is defined as $\lambda x.$ `abort` ($K[x]$), which involves the `abort` operator, used to drop the context in which the `abort` expression evaluates. More precisely, during the evaluation of $e\ k$, we may eventually reach an expression of the form $K'[k\ v]$. This expression beta-reduces to K'[`abort` ($K[v]$)], which, by rule Abort, evaluates to $K[v]$. This transition thus restores the original context K, in which `capture` was called, plugging in the value v to which the continuation k was applied.

4. Parallelism in the DAG Calculus

In this section, we show how to translate three classic, high-level parallel constructs into our DAG calculus.

4.1 Fork Join

To describe the translation of fork-join into the DAG calculus, we consider as source language a pure lambda-calculus extended with a fork-join construct, written $e_1 \parallel e_2$. In this construct, also called *parallel pair*, reduction can take place on either the left or the right branch. We present the language in A-normal form, with evaluation context written K.

$$e ::= v \mid \texttt{fst}\ v \mid \texttt{snd}\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid v\ v \mid (e \parallel e)$$
$$v ::= x \mid n \mid (v, v) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}$$
$$K ::= \bullet \mid \texttt{let}\ x = K\ \texttt{in}\ e \mid (K \parallel e) \mid (e \parallel K)$$

Reduction rules are standard. A parallel pair whose both components have terminated is reduced to a conventional pair by the evaluation rule: $(v_1 \parallel v_2) \rightarrow (v_1, v_2)$.

Translation The translation from this source language to our DAG calculus is written $[\![e]\!]$. Its definition is entirely structural. For constructs other than parallel pairs, we have, in particular:

$$[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] = \texttt{let}\ x = [\![e_1]\!]\ \texttt{in}\ [\![e_2]\!]$$
$$[\![\texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}]\!] = \texttt{fun}\ f\ x\ \texttt{is}\ [\![e]\!]\ \texttt{end}$$
$$[\![v_1\ v_2]\!] = [\![v_1]\!]\ [\![v_2]\!].$$

The translation of parallel pairs is shown below.

```
1 [[ e1 || e2 ]] =
2   capture (fun _ k is
3     let l1 = alloc
4         l2 = alloc
5         t1 = newTd (l1 := [[e1]])
6         t2 = newTd (l2 := [[e2]])
7         t  = newTd (k (!l1, !l2))
8     in newEdge (t1, t); newEdge (t2, t); transfer t;
9        release t; release t1; release t2)
```

We allocate two memory cells, `l1` and `l2`, for storing the results of the branches (line 3-4). We then create two threads, `t1` and `t2`, for the two branches (lines 5–6), and also creates a thread, called `t`, for the *join* thread (line 7). The role of this join thread is to deliver the pair made of the two results, which we read in locations `l1` and `l2`, back to the context in which the parallel pair began its execution. This context is reified as the first-class continuation `k`, using the capture operator (line 2).

The synchronization between the branches `t1` and `t2` and the join thread `t` is realized by the creation of two edges (line 8). The transfer operation (line 8) migrates the outgoing edges from the currently-running thread, to the join thread `t`. Intuitively, just like we are transferring the current *sequential continuation* `k` to the join thread, we also need to transfer the current *parallel continuation*, as represented by the set of outgoing edges of the currently-running thread, to the join thread. The release operations (line 9) go in pair with the thread creation operations. In particular, they allow `t1` and `t2`, which have no incoming edges, to begin their execution.

Theorem 1 (Correctness)**.** *Let t be the identifier of the main thread, e be the source program stored in this thread, and l be a designated location in which to store the final result. For any integer result n, final state σ such that $\sigma(l) = n$, and final set of vertices V, assuming that all threads t' in V are finished (i.e.* status $(V(t')) = \mathsf{F}$*), we have:*

$$[t \mapsto (l :=[\![e]\!], \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^* V, \emptyset, \sigma \quad \Rightarrow \quad e \rightarrow^* n.$$

Furthermore, divergence in the DAG calculus entails divergence in the source language:

$$[t \mapsto (l :=[\![e]\!], \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^\infty \quad \Rightarrow \quad e \rightarrow^\infty .$$

The proof may be found in the technical appendix. This proof, like the other two correctness proofs presented further, involves one key difficulty, related to administrative reduction steps, i.e. to the fact that one reduction step in the source language may correpond to several reduction steps in the target language. Most compiler proofs deal with administrative reduction steps using well-known proofs techniques, typically based on simulation diagrams. However, we have found that these techniques were not directly applicable to the parallel semantics of a language such as that the fork-join language considered here. We next explain why.

When the target program takes a reduction step, this step corresponds either to an administrative step, or to a real step from the source program. Consider the latter case. With a sequential semantics, when the target program takes a real step, the target program then typically corresponds exactly to the translation of the source program. However, with a parallel semantics, this might not be the case. For example, since the two branches of a parallel pair may reduce independently, one branch may take a real step while the other branch is in the middle of performing a sequence of administrative steps. Although the target program takes a real step, it is not, at this point, the translation of any source program. Thus, we cannot easily close a simulation diagram.

To address this challenge, we introduce an instrumented programming language, similar to the source programming language, except that each parallel pair that began executing gets annotated with information about identities associated with the representation of the parallel pair in the target language: number of administrative steps already performed, thread identifiers, locations for the results, etc. We then set up a two-layer simulation diagram: the first layer relates the source program with the instrumented program, while the second layer relates the instrumented program with the target program. We are able to reason about both layers independently, using conventional simulation diagrams, and then conclude by relating the source and the target programs.

4.2 Async-Finish

We now describe the translation of async-finish into our DAG calculus. To that end, we consider an imperative lambda-calculus extended with two constructs: `async`(e) and `finish`($\{e\}$). The general form for the finish construct is written `finish`(S), where S denotes the set of all expressions that evaluate in parallel within the scope of this `finish` block considered. The grammar below, presented in A-normal form, includes contexts for reduction, written K, and contexts not traversing `finish` blocks, written L.

$$\begin{aligned} e &::= v \mid \texttt{fst}\ v \mid \texttt{snd}\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid v\ v \\ &\quad \mid \texttt{alloc} \mid v := v \mid !\,v \mid \texttt{async}(e) \mid \texttt{finish}(S) \\ v &::= x \mid n \mid l \mid (v, v) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \\ S &\equiv \{e_1, e_2, \ldots, e_n\} \\ K &::= \bullet \mid \texttt{let}\ x = K\ \texttt{in}\ e \mid \texttt{finish}(\{K\} \uplus S) \\ L &::= \bullet \mid \texttt{let}\ x = L\ \texttt{in}\ e \end{aligned}$$

The operational semantics is standard. We show below the reduction rules for `async` and `finish`, omitting the state since it is not altered by these rules. The first rule spawns an `async` task withing its enclosing `finish` block, adding it to the set of tasks already present. The second rule removes a completed `async` task. The third rule closes a `finish` block when all the tasks spawned in his scope have completed.

$$\begin{aligned} \texttt{finish}(\{L[\texttt{async}(e)]\} \uplus S) &\to \texttt{finish}(\{L[\texttt{()}]\} \uplus \{e\} \uplus S) \\ \texttt{finish}(\{\texttt{()}\} \uplus S) &\to \texttt{finish}(S) \\ \texttt{finish}(\emptyset) &\to \texttt{()} \end{aligned}$$

Translation The translation from this source language to our DAG calculus is written $[\![t \mid e]\!]$. It takes, in addition to the expression e, an argument t that denotes the identity of the DAG vertex that corresponds to the immediately enclosing finish block. The translation thus have a destination-passing style flavor. It is entirely structural. In particular, we have:

$$\begin{aligned} [\![t \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] &= \texttt{let}\ x = [\![t \mid e_1]\!]\ \texttt{in}\ [\![t \mid e_2]\!] \\ [\![t \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}]\!] &= \texttt{fun}\ f\ (x, t')\ \texttt{is}\ [\![t' \mid e]\!]\ \texttt{end} \\ [\![t \mid v_1\ v_2]\!] &= [\![t \mid v_1]\!]\ ([\![t \mid v_2]\!], t). \end{aligned}$$

In order to translate a complete program e, we introduce a dummy thread identifier t_0, and compute $[\![t_0 \mid e]\!]$. If reaching an async outside of any finish block, then the program is considered stuck by the semantics.

The translation of `async` is shown below. We create a new vertex, named `t'`, describing the spawned computation, and we add a new edge from `t'` to the vertex `t` associated with the enclosing finish block. We then release the vertex `t'`.

```
1 [[t | async(e)]] =
2   let t' = newTd [[t | e]] in newEdge (t', t); release t'
```

The translation of `finish` is shown below and explained next. Since we only need to translate surface programs, we do not need to translate the general form `finish`(S).

```
1 [[t | finish({e})]] =
2   capture (fun _ k is
3     let t2 = newTd (k ())
4         t1 = newTd ([[t2 | e]])
5     in newEdge (t1, t2); transfer t2;
6        release t2; release t1)
```

We create a thread, named `t2`, for the continuation `k`, and create a thread, named `t1`, for the body of the `finish` block. Note that the identity of the thread `t2`, which represents the finish block, is provided to the translation of e when computing $[\![\texttt{t2} \mid e]\!]$. We then set a dependency edge between `t1` and `t2`. The remaining operations, `transfer` and `release`, play exactly the same role as earlier in the case of fork-join.

Theorem 2 (Correctness)**.** *Let t be the identifier of the main thread, e be the source program stored in this thread, l be a designated location in which to store the final result, and t_0 be a dummy thread identifier. For any integer result n, final state σ such that $\sigma(l) = n$, and final set of vertices V, assuming that all threads t' in V are finished (i.e.* $\text{status}(V(t')) = \mathsf{F}$*), we have (for some source-language heap h):*

$$\begin{aligned} &[t \mapsto (l := [\![t_0 \mid e]\!], \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^* V, \emptyset, \sigma \Rightarrow \emptyset, e \to^* n, h \\ &[t \mapsto (l := [\![t_0 \mid e]\!], \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^\infty \qquad \Rightarrow \emptyset, e \to^\infty . \end{aligned}$$

4.3 Futures

We now describe the translation of futures into our DAG calculus. We consider a pure lambda-calculus extended with two constructs: `future` and `force`. We present it in A-normal form, and let f denote the identity of a future.

$$\begin{aligned} e &::= v \mid \texttt{fst}\ v \mid \texttt{snd}\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid v\ v \\ &\quad \mid \texttt{future}(e) \mid \texttt{force}(v) \\ v &::= x \mid n \mid f \mid (v, v) \mid \texttt{fun}\ g\ x\ \texttt{is}\ e\ \texttt{end} \\ K &::= \bullet \mid \texttt{let}\ x = K\ \texttt{in}\ e \end{aligned}$$

The operational semantics is defined using two judgments. The first judgment, written $e \circ\!\!\to e'$, captures the reduction relation for all expressions that are neither a `future` nor a `force`. Its rules are standard. The second judgment, written $M \to M'$, describes transition over *configurations*. A configuration, written M, associates an expression (possibly a value) with the identity of each allocated future. For simplicity, we view the main program expression as the body of a particular future, identified as f_0. The initial configuration thus takes the form of a singleton map $[f_0 \mapsto e]$. A configuration M is final when every future in M is bound to a value.

We show below the reduction rules that defines the judgment $M \to M'$. The first rule is for expressions other than `future` and `force`. The second rule reduces an expression `future`(e) to a fresh identity, called f', and adds in the current configuration M a binding from f' to e. The third rule reduces an expression `force`(f') to a value v, assuming that f'

is bound to v in the current configuration M. Note that the expression `force`(f') cannot reduce until the future f' has completed its evaluation.

$$\frac{M(f) = e \quad e \bigcirc\!\!\!\rightarrow e'}{M \rightarrow M[f \mapsto e']} \qquad \frac{M(f) = K[\texttt{future}(e)] \qquad f' \text{ fresh}}{M \rightarrow M[f \mapsto K[f']][f' \mapsto e]}$$

$$\frac{M(f) = K[\texttt{force}(f')] \qquad M(f') = v}{M \rightarrow M[f \mapsto K[v]]}$$

Translation The translation from the language with futures to our DAG calculus is written $[\![e]\!]$. It is entirely structural. In particular, constructs other than `future` and `force` are translated exactly like in the case of fork-join (recall Section 4.1). In our DAG calculus, we represent futures as pairs made of a thread identifier and of a location. The former is used for synchronization, while the later is used for communicating the value computed by the future.

The translation of an expression `future`(e) appears below. We first allocate a location, `l`, and a thread `t`, whose pair will be the representation of the future, as returned on line 5. We then allocate a separate thread, named `t'`, for describing the computation associated with the body of the future. (For technical reasons, `t'` and `t` need to be two distinct vertices.)

```
1 [[future(e)]] =
2   let l = alloc
3       t = newTd (())
4       t' = newTd (l := [[e]])
5   in newEdge (t', t); release t; release t'; (t, l)
```

The translation of an expression `force`(v) appears below. The argument v is expected to be the representation of a future, that is, a pair of the form (`t`, `l`). In the translation, we create a thead `t'`, whose body consists of applying the captured continuation `k` to the value computed by the future. This value can be read at location `l`, but only after the thread `t`, which describes the future, has completed. The required synchronization is implemented by the creation of an edge from `t` to `t'`. Note that this edge creation operation is a no-op in case the future has already completed.

```
1 [[force(v)]] =
2   let (t, l) = [[v]]
3   in capture (fun _ k is
4     let t' = newTd (k (!l))
5     in newEdge (t, t'); transfer t'; release t')
```

Remark: the translation above can be optimized by testing, before capturing the current continuation and setting up the continuation thread, whether the future has already completed; in this case, `force`(v) may immediately return the value `!l`.

Theorem 3 (Correctness)**.** *Let t be the identifier of the main thread, e be the source program stored in this thread, and l be a designated location in which e stores its final result. For any number n, final state σ such that $\sigma(l) = n$, and final set of vertices V, assuming that all threads t' in V are finished (i.e.* status $(V(t')) = \mathsf{F}$*), if*

$$[t \mapsto (l := [\![e]\!], \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^* V, \emptyset, \sigma$$

then there is a configuration M such that $[f_0 \mapsto e] \rightarrow^* M$ and $M(f_0) = n$. Furthermore, divergence is preserved:

$$[t \mapsto (l := [\![e]\!], \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^\infty \quad \Rightarrow \quad [f_0 \mapsto e] \rightarrow^\infty .$$

5. The concurrent computation DAG

Representation of vertices and edges. We begin by describing the representation of vertices and edges. The corresponding signatures appear in the first half of Figure 3. Each vertex is represented as an object with a `run` method, an *incounter*, an *outset*, and a `releaseHandle` field (explained later). The *incounter* keeps track of the number of incoming edges. The *outset* morally keeps track of the set of vertices pointed at by the outgoing edges.

The incounter provides two methods: `increment` and `decrement`. These operations can occur concurrently in any order, as long as two invariants are satisfied. First, each call to decrement must match an anterior call to increment. Second, when the counter reaches zero, no further increment can be performed. When reaching zero, the incounter is responsible for adding its associated vertex to the work queue. To enable key optimizations, the operation `increment` returns a *handle*, which points into the (tree-shaped) structure of the incounter. The matching `decrement` call must provide this handle.

The outset provides two methods which may be called concurrently: `add` and `parallelNotify`. The `add` operation is used for storing an outgoing edge. More precisely, it stores a handle returned by the increment operation performed on the incounter associated of the target vertex of the outgoing edge. This operation may return `false` in case the vertex has already terminated, in which case the edge is not created. The `parallelNotify` operation is called exactly once by the scheduler, when the execution of the corresponding vertex completes. Its purpose is to call `decrement` on all the handles stored, i.e. those for which the add operation returned `true`, possibly in parallel if many handles were stored.

Implementation of primitive DAG operations. We next explain the operations whose execution modify the DAG. The corresponding code appears in the second half of Figure 3. For simplicity, we omit the description of deallocation operations. Note that the code is agnostic to the scheduler used for performing load balancing and selecting which vertex to schedule among ready ones.

Each processor runs the code in `schedulerLoop`, which repeatedly picks a vertex from the work queue, stores its address in a variable called `current`, executes the vertex, and then calls `parallelNotify` to decrement the incounter of the vertices associated with all the outgoing edges of the vertex. We next describe the four DAG primitive operations.

The `newTd` construct is implemented using the function `createThread`, which allocates a vertex and assigns the

computation body, a fresh incounter, and a fresh outset. We increment the incounter to ensure that the vertex is considered for scheduling when the `release` operation is called on the vertex. We save the handle returned by this increment operation into the `releaseHandle` field of the vertex. When the operation `release` is called on a vertex, it simply calls `decrement` on the `releaseHandle` of the vertex, possibly resulting in pushing the vertex into the work queue.

The operation `newEdge` creates an edge. It first increments the incounter of the target vertex, obtaining an incounter handle, and then stores this handle into the outset of the source vertex. This operation, however, might fail if the source vertex has already completed. In this case, the increment operation needs to be undone, by decrementing the incounter.

The operation `transfer`, when applied to a vertex `v`, swaps the outset of `v`, which is assumed to be empty, with the outset of the currently-running vertex, namely `current`. This constant-time swap operation effectively transfers the set of outgoing edges from the vertex `current` to `v`.

Concurrent data structures for incounters. An incounter, which features concurrent `increment` and `decrement` operations, can be naively represented as a single cell updated with atomic fetch-and-add (or compare-and-swap) operations. This approach works well on multicore hardware when the in-degree of DAG vertices remains small. In fact, we use such a representation of incounters for vertices created by our translation of fork-join. (For optimization purposes, we exploit the flexibility of choosing the representation of incounters and outsets on a per-vertex basis.)

However, if the number of incoming edges grows large, the many operations performed on the shared counter may lead to high contention and dramatically slow down the program. In such a situation, we need a more elaborated concurrent data structure able to avoid contention even under high load. The SNZI data structure [13], introduced by earlier work, almost addresses the problem. We briefly present how we might use it as an incounter, explain why it would nevertheless be unrealistic to do so, and then explain how we generalize SNZI to obtain an efficient and scalable incounter data structure.

In short, a SNZI tree consists of a fixed-arity tree with $O(P)$ leaves, where P denotes the number of processors. Each node stores an integer. An increment operation is performed by incrementing a randomly-chosen leaf, the leaf being returned as handle. Each non-leaf node stores the number of immediate children that have a non-zero value. In particular, the incounter has value zero if and only if the root has value zero. To reflect on a leaf update, the tree is traversed upwards towards the root, in time $O(\log P)$, using atomic operation to update the nodes, and stopping as soon as the status no longer changes —typically, most updates execute in constant time.

The main limitation of the SNZI approach is that it requires $O(P)$ space. Clearly, we cannot afford to pay that much for every vertex in the DAG. We are seeking instead for a solution with consuming no more than $O(n)$ space,

```
1  class vertex
2    void run()
3    incounter* in
4    outset* out
5    incounterHandle* releaseHandle
6
7  class incounterHandle // implementation−dependent
8
9  class incounter
10   incounterHandle* increment(vertex* v)
11   static void decrement(incounterHandle* h)
12     // when reaching zero, calls workQueue.push on
13     // the vertex which has this structure as incounter;
14     // the address of this vertex is stored at the root
15     // of the tree data structure of which h is part of.
16
17 class outset
18   void parallelNotify()
19     // calls decrement on all handles that were added
20   bool add(incounterHandle* h)
21     // may return false when the vertex is already finished
22
23 processor_local queue<vertex*> workQueue
24 processor_local vertex* current
25
26 void schedulerLoop() // executed by each processor
27   while true
28     if workQueue.empty()
29       // implementation−dependent load balancing
30       acquireWork() // blocking call
31     current = workQueue.pop()
32     current→run()
33     current→out→parallelNotify()
34
35 // "newTd e" short for "createThread(fun () ⇒ e)"
36 vertex* createThread(runMethod)
37   vertex* v = new vertex
38   v→run = runMethod
39   v→in = new incounter
40   v→out = new outset
41   v→releaseHandle = v→in→increment(v)
42   return v
43
44 void release(vertex* v)
45   v→in→decrement(v→releaseHandle)
46
47 void newEdge(vertex* v1, vertex* v2)
48   incounterHandle* h = v2→in→increment()
49   bool success = v1→out→add(h)
50   if not success // vertex s has already completed
51     v2→in→decrement(h)
52
53 void transfer(vertex* v) // v→out is assumed empty
54   swap(current→out, v→out)
```

Figure 3. Implementation of primitive DAG operations.

where n denotes the number of increment operations. Our solution consists of dynamically growing a SNZI-like tree, up to some maximal depth (roughly, $\log P$). Each node in our tree plays both the role of a SNZI leaf (storing a count) and of a SNZI node (storing a number of non-zero children). For each increment operation, we go down the tree following a random path until either reaching the maximal depth, in which case we reuse the leaf reached, or reaching a leaf, in which case we grow the tree by atomically attaching a leaf.

We implement one crucial optimization for reducing the number of node allocations. Rather than growing the tree node by node, we wait until a constant number of operations have been performed on a given leaf before growing subtrees from it. (We use an extra field for counting these operations.) For example, we may wait until 8 fetch-and-add operations are performed on the root, and then allocate an array of size 15 (with padding to avoid false sharing), in which we represent a binary tree with 8 leaves. More generally, we can grow the tree by repeatedly attaching fixed-size, complete binary subtrees allocated in arrays. Doing so, we only slightly increase contention, and at the same time save a large number of allocations.

Concurrent data structures for outsets. We next discuss the representation of outsets, i.e. concurrent bags featuring an `add` and a `parallelNotify` operation. Like for incounters, as long as the number of operations remains small, naive approaches apply. In such situations, a locked list or, even better, a non-blocking concurrent stack [20] may be used. However, such structures would suffer from high contention when employed in computation DAG's featuring vertices with large out-degree. In such case, we might consider using a data structure involving one stack per processor (e.g., [45]). However, processor-indexed structures require a prohibitive $O(P)$ space overhead per vertex. We are seeking for a linear space solution.

As we could not find a data structure meeting our requirements, we devised a concurrent bag data structure based on a dynamically-growing tree. It is simpler than our incounter data structure because bag elements are not removed one by one, nevertheless we need to carefully handle possible races between the iteration over the elements performed by the `parallelNotify` operation and concurrent add operations.

To describe our structure in more details, we first present the basic idea, then explain the optimizations that we perform. In the basic version, we use a binary tree in which each node stores a single element from the bag. Each `add` operation follows a random path in the tree and atomically attach a fresh leaf to the tree. The `parallelNotify` operation performs a parallel traversal of the tree, and attemps to atomically tag the leaves so as to prevent the tree from growing further from this leaf. We implement this parallel traversal using lazy binary splitting [46], which is particularly effective on balanced trees.

To achieve efficiency in practice, we apply three key optimizations. First, in order to reduce the number of indirections when traversing the tree, we increase the arity of the nodes, from 2 to a larger constant, say 16. Second, in order to reduce the number of allocations, instead of storing a single element in each node, we store a fixed number of them, say 16. To that end, each node contains a buffer, i.e. a fixed-capacity concurrent stack, and uses an atomically-updated field to keep track of the index of the first free cell. We grow branches from a node only when its stack is full.

A third optimization is useful for very large bags. When the number of add operations already performed exceeds a constant fraction of the number of processors, we allocate a processor-indexed array of buffers. Processors can push elements into their privately-owned buffers. They commit full buffers at once into the tree, thereby preserving the ability to efficiently parallel process the elements. This third optimization greatly reduces the number of atomic operations and the number of traversals through the tree.

6. Experiments

Implementation and experimental setup. We implemented our DAG calculus as an extension to a C++ library called PASL. PASL provides us the work-stealing scheduler [2] that we use to perform load balancing for our benchmark programs.[2]

Comparison with Cilk. The first part of this study shows results from three benchmarks taken from the Problem-Based Benchmark Suite (PBBS) [8]. The first one is a parallel sample sort. Each input consists of 240 million doubles. The second is a suffix-array algorithm that takes as input a string S and returns an equal-length array A that specifies in sorted order the suffixes of S. We use an input with 10 million characters. The third is the K-nearest neighbors algorithm, which for n points in two or three dimensions and parameter k, returns for each point its k nearest neighbors. For each input, we used for n 10 million points. We implemented each of these algorithms in our DAG system and compared results with the original codes in Cilk. Results are shown in Figure Figure 4(a). The results show that encodings in our DAG system perform nearly as well as the Cilk-based counterparts, even though the Cilk system has benefited from significant, long-term engineering effort.

Our next experiment compares a Cilk-based implementation of the Gauss-Seidel heat-transfer simulation to our implementation in the DAG calculus. We obtained the Cilk

[2] We compiled all programs with GCC version 5.2, using optimizations `-O2 -march=native`. For the measurements, we considered an Ubuntu Linux machine with kernel v3.13.0-66-generic. Our benchmark machine has four Intel E7-4870 chips and 1Tb of RAM. Each chip has ten cores and shares a 30Mb L3 cache. Each core runs at 2.4Ghz and has 256Kb of L2 cache and 32Kb of L1 cache. Additionally, each core hosts two SMT threads, giving a total of eighty hardware threads. However, to avoid complications with hyperthreading, we did not use more than forty threads. For each data point, we report the average running time over thirty runs.

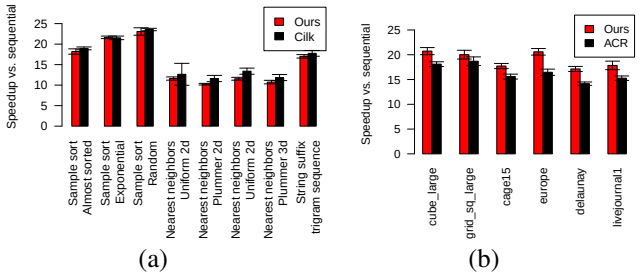


Figure 4. Results for (a) fork-join programs: sample sort, string suffix, nearest neighbors, and (b) Parallel DFS, both using 40 cores.

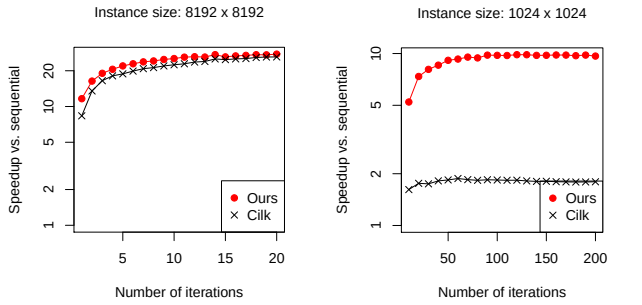


Figure 5. Gauss-Seidel application, using 40 cores.

code unmodified, from the authors of another study [38]. The algorithm simulates the propagation of heat through a plane using a five-point stencil. The purpose of this study is to demonstrate that flexible synchronization primitives that sit outside of async/finish or futures (1) can be encoded in our DAG calculus and (2) can deliver same or better performance than a well-optimized code.

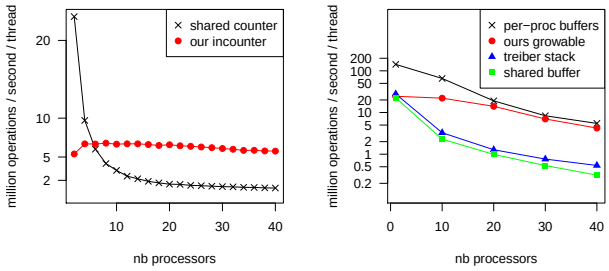
The Cilk implementation of this algorithm proceeds by advancing a hyperplane through a three-dimensional space (first two dimensions are spatial and the third is time), exploiting parallelism within the processing of a hyperplane. The Cilk code uses its spawn/sync primitives to enforce a barrier between the processing of successive hyperplanes. Their use of spawn/sync is equivalent to a function-local, non-nested async/finish block. To implement this synchronization mechanism, the Cilk system uses a single atomic fetch-and-add counter. For block size, we selected the best setting on our machine. Our algorithm uses the same algorithm to handle processing inside a base-case block, but a completely different, relaxed-syncronization technique to enforce data dependencies between blocks. We enforce inter-block dependencies by allocating one incounter per block (to be used for all time steps) and using each incounter to count the number of unsatisfied data dependencies of each block computation. Since each block depends on just a small, fixed number of surrounding blocks, we used for incounter the simple, atomic fetch-and-add cell. Because we store our incounters directly in a matrix data structure, our algorithm fits does not fit directly into the regimes of async/finish or parallel futures.

The plot in Figure 5 shows the results from this experiment. In our study, we collected data from multiple settings of the grid size and number of iterations. Owing to space limitation, we show the plots for only the grid sizes 1k and 8k here, which we selected to match those reported in another study [38]. The results show the advantage of using relaxed synchronization: the Cilk implementation can exploit parallelism only within one wavefront, whereas ours is limited only by the inter-block dependencies. The maximum speedup is nearly the same for both codes. However, the Cilk version gets close to maximum only once the problem size is large enough to feed all cores. This threshold occurs, for instance, when the grid size is 8k and the number of iterations 10.

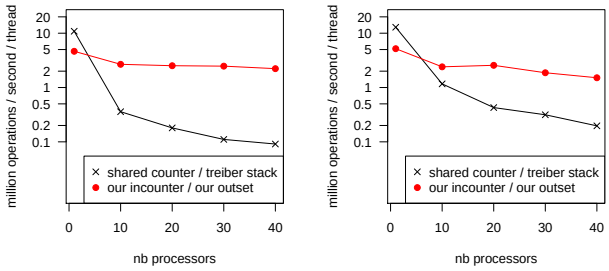
Comparison with a graph-traversal algorithm. We present the following experiment to show that an implementation of our DAG calculus can support state-of-the-art graph-traversal and deliver the same or even slightly better results than a state-of-the-art implementation. The algorithm we consider is the Pseudo DFS algorithm of Acar et al [**?**]. This algorithm performs reachability analysis for a given in-memory graph. We obtained the authors' code, labeled ACR, to use as baseline for our comparison. Both our implementation and that of ACR use work stealing to balance load among cores. The ACR code uses their own implementation of termination detection, whereas ours uses our generic incounter data structure. Figure 4(b) shows the results for a subset of the real-world graphs that are reported in the ACR paper. We selected the graphs to represent small-world and high-diameter cases. The reason that our version is always faster is that ours is always faster on a single core, thereby improving parallel performance.

Scalability study. This section compares scalability between various concurrent data structures that are candidates for incounter and outset. We begin with incounter. Figure 6(a) shows throughput (per core) achieved when all cores in the machine independently alternate between incrementing and decrementing on the same incounter. When several processors are involved, our resizable SNZI-based incounter significantly outperforms the single-cell atomic fetch-and-add counter (labeled as "shared counter"). We configured this experiment to resemble the one performed in the original SNZI paper [13], and our results are consistent with theirs. Note, however, that the original SNZI paper does not consider our version and considers a different platform.

The plot in Figure 6(b) shows results concerning the scalability of our outset structure. The plot shows throughput achieved for repeatedly calling the `add` operation, on all cores in parallel. For this experiment, we compare with a Treiber stack and two other data structures. The per-proc-buffers version represents a data structure consisting of a collection of core-private buffers that are allocated by the cores on demand. Each core-private buffer is updated in a single-threaded fashion, using no atomic operation. Each



(a) incounter study (b) outset study



(c) async/finish study (d) mixed study

Figure 6. Scalability study.

buffer is represented by a contiguous block of 4k cells. The shared-buffer structure is just a single size-4k block that is shared among all cores. When it overflows, one of the cores allocates a new block and replaces the old one. On the one hand, the results show that using a single, shared buffer, whether it is a Treiber stack or a single, shared buffer, scales poorly under heavy load, and on the other, they show that our outset algorithm achieves close to what is likely the maximum capable on our machine, which is represented by the per-proc buffers.

Now, to evaluate our encoding of async/finish, we have constructed a microbenchmark that performs 10 million async calls against the same finish block. Figure 6(c) shows the results. Like before, the fetch-and-add incounter scales poorly, whereas our incounter scales well. We see that, at scale, the incounter algorithm makes a significant difference, even though the incounter operations are spread out between other operations, such as spawning DAG threads.

Our final benchmark uses the sample code from Figure 7, with `mixed` passed the value 10 million. The results are shown in Figure 6(d). A call to `mixed(`n`)` leads to the creation of one future and `n` touches of it. This future recursively calls `mixed(n/2)`, and so on. In total $\log n$ futures are allocated and $2n$ touch operations are performed. We count five basic operations per touch (`newTd`, `increment`, `decrement`, `add`, and `parallelNotify`). Therefore, $10n$ oeprations are performed in total. This benchmark demonstrates the ability of our data structure to handle large outdegree (many touch operations on the same future).

```
1 let rec mixed n =
2   if n > 1 then
3     let f = future(mixed(n/2)) in
4     parallel_for i = 1 to n do force f
```

Figure 7. Sample code generating high in- and out-degree DAG nodes (performs no useful work). The `parallel_for` loop can be implemented with async-finish.

7. Related Work

We discussed most closely related work in the rest of the paper; we present a broader acccont here.

Semantics of concurrency has been studied extensively and many concurrent calculi has been proposed, including CSP [23], π-calculus [32, 33] and actors [22]. These calculi model concurrent computations involving interaction between many processes via some communication medium, typically called channels. Many different calculi have been studied (several surveys exist, e.g. [4, 19]), and several programming languages such as Concurrent ML [39] and Pict [37] have been designed based on these calculi. Operational semantics for concurrent versions of ML have also been developed (e.g. [5, 36]).

This abundance of formal semantics of concurrency contrasts starkly with its paucity for parallelism: relatively little exists beyond the well-known work on futures and fork-join parallelism. One recent exception is the work on Featherweight X10 by Lee and Palsberg [35], who give formal semantics of a language with async-finish parallelism. The core of their work is a type system for may-happen-in-parallel analysis. In contrast, we focus on the dynamics of parallel computations in general, which includes async-finish and also other parallelism abstractions.

In addition to their use in parallel-algorithm design [3, 9, 17, 25]), DAG's are sometimes used for presenting a cost-semantics [40–42] for parallel programs. For example, Greiner and Blelloch [17] present a cost semantics for a language with futures based on DAG's. They also present algorithms for the PRAM model but their algorithms do not try to control contention as ours do. Spoonhower et al. [44] use a similar technique for presenting a cost semantics to account for space use for a language with parallel tuples.

There has been several other proposals for structured/implicit parallel programming in addition to fork-join, async-finish, and futures approaches. OpenStream is a data-flow system that offers relaxed synchronization for parallel applications [38]. Our experiment with Gauss Seidel repeats theirs using our approach. We were, however, unable to compare to their work, because of difficulties in reproducing their results. We have been working with the authors to address this issue. Concurrent Revisions [10] resembles futures, but provides a mechanism for deterministic programming with shared mutable state. LVars [27] are one alternative approach that have recently been extended to support primitives with the same power as both async/finish and futures [28].

In this paper, we have presented concurrent, non-blocking data structures for implementing DAG's efficiently in parallel. There is a huge literature on concurrent data structures (e.g., [21]). We based our incounter on the SNZI data structure [13]. For outset, we are aware of no existing data structure that provides the features that are required. In particular, fast concurrent stacks, queues, and bags provide insert and remove, whereas our outset has a two-phase behavior. In the first phase there are only insert operations and in the second there is a parallel-map operation that broadcasts completion to all DAG threads that are listening on the outset.

References

[1] Folly: Facebook open-source library, 2015. https://github.com/facebook/folly.

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *PPoPP '13*, 2013.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.

[4] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, May 2005. ISSN 0304-3975.

[5] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 119–129, 1992.

[6] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990. ISBN 0-262-02313-X.

[7] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*, pages 213–225. ACM, 1996.

[8] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP '12*, pages 181–192, 2012. ISBN 978-1-4503-1160-1.

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46: 720–748, September 1999. ISSN 0004-5411.

[10] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 691–707, 2010. ISBN 978-1-4503-0203-6.

[11] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: a status report. In *Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, 2007. ISBN 978-1-59593-690-5.

[12] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005. ISBN 1-59593-031-0.

[13] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: Scalable nonzero indicators. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 13–22, 2007. ISBN 978-1-59593-616-5.

[14] Matthias Felleisen and Daniel P. Friedman. Control operators, the secd-machine, and the lambda-calculus. Technical Report TR197, Department of Computer Science, Indiana University, June 1986.

[15] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011.

[16] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[17] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Trans. Program. Lang. Syst.*, 21(2):240–285, March 1999. ISSN 0164-0925.

[18] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 9–17. ACM, 1984. ISBN 0-89791-142-3.

[19] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. ISBN 1107029570, 9781107029576.

[20] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, 2004. ISBN 1-58113-840-7.

[21] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. ISBN 0123705916.

[22] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, 1973.

[23] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. ISSN 0001-0782.

[24] Shams Mahmood Imam and Vivek Sarkay. Habanero-java library: a java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, pages 75–86, 2014.

[25] Joseph Jaja. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Company, 1992.

[26] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, 2010. ISBN 978-1-60558-794-3.

[27] Lindsey Kuper and Ryan R Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84. ACM, 2013.

[28] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: Quasi-deterministic parallel programming with lvars. *SIGPLAN Not.*, 49(1):257–270, January 2014. ISSN 0362-1340.

[29] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000. ISBN 1-58113-288-3.

[30] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, 2009. ISBN 978-1-60558-766-0.

[31] Simon Marlow. Parallel and concurrent programming in haskell. In *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, pages 339–401, 2011.

[32] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992. ISSN 0890-5401.

[33] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Inf. Comput.*, 100(1):41–77, September 1992. ISSN 0890-5401.

[34] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-03358-2.

[35] Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, page 1, 2012.

[36] Prakash Panangaden and John H. Reppy. *ML with Concurrency*, chapter The Essence of Concurrent ML, pages 5–29. CRC Press, 2005.

[37] Benjamin C. Pierce and David N. Turner. Proof, language, and interaction. chapter Pict: A Programming Language Based on the Pi-Calculus, pages 455–494. 2000. ISBN 0-262-16188-5.

[38] Antoniu Pop and Albert Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *TACO'13*, 9(4):53:1–53:25, January 2013. ISSN 1544-3566.

[39] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-48089-2.

[40] Mads Rosendahl. Automatic complexity analysis. In *FPCA '89: Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, 1989.

[41] David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, University of London, Imperial College, September 1990.

[42] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Principles of Programming Languages*, pages 355–366, 1995.

[43] K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. Multimlton: A multicore-aware runtime for standard ml. *Journal of Functional Programming*, FirstView:1–62, 6 2014. ISSN 1469-7653.

[44] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. *Journal of Functional Programming*, 20:417–461, 2010. ISSN 1469-7653.

[45] Hρakan Sundell, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 335–344, 2011. ISBN 978-1-4503-0743-7.

[46] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP '10*, pages 179–190, 2010.