Dag-Calculus: A Calculus for Parallel Computation

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

Arthur Charguéraud
Inria
arthur.chargueraud@inria.fr

Mike Rainey
Inria
mike.rainey@inria.fr

Filip Sieczkowski
Inria
filip.sieczkowski@inria.fr

Abstract

Hardware advances on multicore computers in the last decade has brought parallel computing to the mainstream, leading to the development of many programming languages for writing parallel programs. These parallel languages support several different abstractions/paradigms for parallelism, such as fork-join, async-finish, futures. While they may seem similar, these abstractions lead to different semantics, language design and implementation decisions, and can significantly impact the performance behavior of end-user applications.

In this paper, we consider the question of whether it would be possible to unify various different paradigms of parallel computing. To this end, we propose a calculus, called *dag calculus* that can encode fork-join, async-finish, and futures, and possibly others. We first describe the dag calculus and its semantics, establish translations from the aforementioned paradigms into dag calculus. These translations establish that dag calculus is sufficiently powerful for encoding programs written in prevailing paradigms of parallelism. We then present concurrent algorithms and data structures for realizing dag calculus in practice and prove that the proposed techniques are indeed consistent with the semantics. Finally, we present an actual implementation of the calculus and present an empirical evaluation by comparing its performance to highly optimized code from prior work. The result show that the calculus is not only very general but also that it competes well with the state of the art, which benefits from many years of engineering effort, and sometimes even beats it.

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

1. Introduction

Hardware advances in the past decade has dramatically broadened the availability of parallel computers, based usually on multicore chips. This development led to 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 [18], Fork/Join Java [30], Habanero Java [25], NESL [9], parallel Haskell [13, 27, 32], parallel ML [17, 44], TPL [31], and X10 [14].

These programming languages, typically developed as an extension to an existing language (such as C, Haskell, Java, and ML), provide support for parallelism by using at least several different language primitives, including for example fork-join, async-finish, and futures. The fork-join and async-finish primitives, which can be used to express nested-parallel computations, are similar but async-finish is more flexible. Futures, which originated from work on functional programming languages (e.g., [20]), is perhaps the most functional of all these primitives, allowing parallel computations to be treated as first-class values in a modern functional programming language. These parallelism primitives are broadly used in many languages today and have had significant influence on the practice of parallel computing (e.g.,[9, 13, 14, 17, 18, 25, 27, 30–32, 44]).

While various parallelism primitives may all appear similar at a superficial level, there are deep differences between them. These differences affect the design and the implementation of parallel programming languages. Perhaps for this reason, programming languages tend to support only some of these primitives. For example, the Cilk language is primarily based on fork-join parallelism. The X10 and the Habanero Java languages support both async-finish and futures. The Parallel Haskell and Parallel ML languages support both futures and fork-join parallelism. While they originated from functional programming research, futures can now be found in many languages and libraries including C [1] and Java.

The particular primitives used to express parallelism can also have significant impact on the application. For example, fork-join primitives tend to work well for balanced divide-and-conquer algorithms but may behave poorly in highly irregular algorithms, such as those for graph problems, which can be expressed more efficiently using async-finish (e.g., [4, 15]) Fork-join programs can preserve important locality properties of a parallel program during execution but futures may not [2]; though more restricted versions of futures may [22]. Futures can improve performance of some applications asymptotically by enabling pipelining [8]. Furthermore, some applications can benefit even more relaxed forms of parallelism where programs can by structured more freely [39].

The relative diversity of parallelism abstractions and their deep impact on the design and implementation of programming languages application behavior raises several important questions.

- Is there a unifying model or calculus that can be used to express and study the different forms of parallelism?
- Can such a calculus be realized efficiently in practice as a programming language that can serve, for example, as a target for compiling different forms of parallelism?

In this paper, we answer these questions affirmatively. We first 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 from parallel programming languages with these constructs to dag calculus. We show that dag calculus is realizable in practice by describing precisely an imple-

mentation suitable for modern hardware-shared-memory computers, such as multicores, by presenting the concurrent data structures and algorithms needed (Section 5). Since the implementation must handle difficult concurrency conditions that are not exposed in the calculus, we prove that the implementation is faithful to the semantics (Theorem 4). Finally, we present an actual implementation and present an experimental evaluation (Section 7).

The starting point for dag calculus is a relatively well-known idea: many parallel computations can be represented as directed-acyclic graphs, written "dags", where vertices represent sequential pieces of computation, variously called threads, tasks, strands, fibers etc., and edges represent the dependencies between them. This technique is broadly exploited in algorithms research to express algorithms [26] and also to analyze crucial scheduling algorithms used for parallel programming languages (e.g., [2, 5, 11, 19, 22]). In the algorithms world, however, dags are treated as known, static structures, used primarily for modeling and analysis.

In this paper, we propose dag calculus as a system where evaluation dynamically constructs a dag and where evaluation itself is controlled by the structure of the dag. This feature gives the programmer full control over the structure of the computation and thus over crucial aspects of parallelism. In this approach, vertices of the dag are labeled with expressions representing 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 them (their expressions) in some non-deterministic order. When evaluated, an expression performs some computation and also possibly creates new vertices and edges. Evaluation in dag calculus can thus be viewed as a concurrent interleaving of conventional computation and updates to the structure of the dag.

We formalize this informal description of the dag calculus by specifying its syntax and semantics (Section 3). Dag calculus includes expressions for creating threads and dependencies or edges between them, as well as another primitive called `yield` that can be used to suspend the execution of the running thread, and perform a context switch to go back to the scheduler. The `yield` operation is useful for starting parallel sub-computations and resuming the execution of a thread after completion of these sub-computations. The practical realization of `yield` may depend on the control operators and mechanisms available in the target platform.

The semantics of the dag calculus abstracts away from several important details that a realistic language implementation must deal with. The challenges stem from concurrency and scheduling. In the specification of dag calculus, we model parallelism, as is standard, by using an interleaving semantics. An actual implementation must run concurrently and thus must use efficient concurrent data structures and algorithms to represent the state of the system. Another challenge relating to concurrency involves the specification of several key operations in dag calculus. These operations, such as the updates to the vertices and edges of the dag maintained in the calculus, occur sequentially and atomically in a single step. For efficiency reasons, an actual implementation must perform such operations concurrently in parallel. The second class of challenges stem from scheduling, which is treated as a non-deterministic choice in the semantics. In contrast, an implementation must provide a scheduler that executes on the same machine as the program and must communicate with the program. We address a number of these challenges by specifying at quite a precise level the algorithms and data structures for realizing the calculus in a modern hardware-shared memory parallel computer (Section 5). The concurrent algorithms and data structures that we propose for realizing dag calculus in practice are quite complex. We therefore prove that they implement the semantics faithfully (Theorem 4).

Based on the algorithms and data structures proposed, we give an actual implementation of DAG calculus as a library (Section 7). Since the implementation must deal with many lower level issues, such as representation of memory, control, and concurrency, it is written in C++11. Nevertheless, our implementation retains a very functional flavor thanks to its heavy use of functional features of C++11, such as lambda functions. We also 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. Our results show that the performance of our implementation of dag calculus is compatible with highly optimized implementations from prior work (e.g., Cilk), and in some cases performs even better than them, sometimes thanks to its ability to represent more interesting dependency relations than is possible using parallelism primitives, such as fork-join, async-finish, and futures.

For the formalisms and the proofs in the paper calculus, we assume a sequentially consistent memory model. In our implementation and experiments, however, we assume x86-TSO semantics [35], and we use the necessary synchronization facilities to ensure correctness. We expect that the semantics and the proofs could be generalized for weak memory models.

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.
- Algorithms for implementing dag calculus in practice and a proof establishing that the implementation is correct.
- An optimized C++ implementation and its evaluation.

Appendix. Due to space constraints, we have omitted the proof details from the paper. We have uploaded a technical appendix that contains the proofs onto the conference management system.

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
```

The key restriction of fork-join parallelism is that only two parallel computations can be spawned each time. To spawn larger number of parallel computations, fork-join calls can be nested. Cilk's "spawn" and "sync" primitives provide a bit more flexibility by allowing any (statically) fixed number of parallel computations one by one. The async-finish construct, described below, provides a more direct, more flexible, and possibly more efficient way of spawning arbitrarily many subcomputations.

Async-finish. The async-finish construct 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. 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.

```
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 `e` 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. The difference between fork-join and async-finish is the ability to spawn an arbitrary number of computations that can join at the same point. With fork-join, we can only spawn a fixed number of such computations. With async-finish, we can spawn as many as desired. For this reason, async-finish can offer a powerful mechanism for expressing parallel computations, especially those with irregular synchronization behavior.

The comparison between async-finish and futures is less clear cut. Neither appears superior to the other. In fact, they serve as duals of each other in a certain sense: futures enable synchronization via data dependencies whereas the async-finish construct enables synchronization via control dependencies. More specifically, the **force** primitive of futures synchronize between a computation with a future at a data dependency, leaving control dependencies between them implicit. Dually, the **finish** construct of async-finish synchronizes a number of **async**'ed parallel computations by enforcing a control dependency, leaving data dependencies between them implicitly. 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, called dag-calculus. The calculus extends an imperative λ-calculus with primitives for dynamically creating a Directed Acyclic Graph, or a dag, that represents the computation. We refer to such a dag as a *computation dag*, or simply as a *dag*. In this work we focus on the dynamic aspects of parallel computation using dags. Thus, the programs in the dag calculus can fail at runtime. We model failure by the reductions getting stuck, and do not provide any static checks that would prevent programs from doing so.

Abstract Syntax. Figure 1 shows the syntax of dag calculus. 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. For operating on the dag, we introduce a new language construct, `newTd`, and fur primitive functions: `release`, `newEdge`, `self` and `yield`. When presenting examples, we use some straightforward syntactic sugar, e.g., semicolons for sequencing.

The expression `newTd` e creates a new thread for evaluating the expression e, and returns the corresponding thread (identifier). Note that this construct does not evaluate its argument, as its rationale

$$
\begin{aligned}
e &::= v \mid e \oplus e \mid e \otimes e \mid (e,e) \mid \pi_i\ e \mid\ e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \\
&\quad \texttt{alloc()} \mid e := e \mid\ !e \mid\ \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \\
&\quad \texttt{newTd}\ e \mid \texttt{release}\ e \mid \texttt{newEdge}\ (e,e) \mid \texttt{self()} \mid \texttt{yield()} \\
v &::= x \mid \ell \mid \texttt{t} \mid \texttt{n} \mid \texttt{()} \mid (v,v) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \\
K &::= \bullet \mid \texttt{let}\ x = K\ \texttt{in}\ e \mid K := e \mid v := K \mid\ !K \mid \texttt{release}\ K \\
&\quad \mid \texttt{newEdge}\ Ke \mid \texttt{newEdge}\ vK \mid \ldots
\end{aligned}
$$

Figure 1. Abstract syntax for DAG-Calculus.

is to spawn e as a separate computation thread. When created, a thread is not ready for execution but becomes available for addition of edges to and from other threads. Such edges between threads represent dependencies between them. When the depedencies are set up, the expression `release` e, where e evaluates to a thread t, is used to "release" the thread for evaluation. When released a thread may be scheduled for evaluation, but only after all the threads that it depends on complete their evaluation.

As a thread evaluates, it may choose to add additional edges between itself and other existing threads. The construct `self()` returns the thread that evaluates it.

The expression `newEdge` (e_1, e_2) inserts an edge from thread e_1 to thread e_2 as long as the edge does not violate important invariants, specified by the dynamic semantics, that essentially every parallel program must observe.

The `yield()` construct *suspends* the thread that evaluates it by descheduling it and returning the control to the scheduler. This control transfer permits a thread to suspend and wait for newly discovered dependencies to complete, allowing on-the-fly updates to the structure of the computation. Thus, we are able to exploit parallelism regardless of the context in which a given parallel construct is encountered. This ability is particularly important for nested parallelism, where the thread that yields may itself have outgoing edges, which are thus preserved, as observed in Section 4.

An example. Although the dag calculus is not meant to be a user-level programming language, we consider here an example, both to illustrate how parallel programs may be expressed in dag calculus and to present a high-level overview of the main ideas behind the dynamic semantics of dag calculus. We note that this example is relatively high level and does not give a precise accounting of all the details. The precise semantics is specified later in the section following the example.

Figure 3 shows the dag-calculus code for our running example, a parallel Fibonacci function, based on the classic recursive definition of Fibonacci numbers. As the function `fib_dc` evaluates, it creates threads via `newTd` and synchronization edges via `newEdge`. A *synchronization edge* or simply an *edge* from a thread t to t' requires thread t to complete before t' can start executing. The synchronization edges between threads control how the evaluation proceeds. At any point, those threads that are released and have no incoming edges are said to be *ready* and can be executed in parallel.

Consider a call to `fib_dc` with input n, and let t be the thread executing the call. If n is less than `2`, then the thread returns n. Otherwise, the thread t can compute in parallel the Fibonacci of $(n-1)$ and $(n-2)$. To this end, the thread t allocates two locations, *la* and *lb*, for the results of the two parallel computations. It then creates two threads *ta* and *tb* for computing Fibonacci of $(n-1)$ and $(n-2)$. These threads complete by writing their results to *la* and *lb* respectively, making them available to the caller thread t. After creating the new threads, thread t creates new edges from *ta* and *tb* to itself, which is available via `self`. This operation creates dependencies from the newly created threads to t itself. Thread t can now release *ta* and *tb*, via `release`, allowing them to be evaluated, and yields control by calling `yield`, which suspends t, stopping its

evaluation. By creating two new threads and yielding control to them, the main thread t essentially nests the computations to be performed inside itself.

Since *ta* and *tb* have no incoming edges, they can be evaluated at any time after they are released. Since t has incoming edges from *ta* and *tb*, its evaluation should be paused until *ta* and *tb* complete. For this reason, it is important for thread t to yield after creating dependencies from other threads to itself. In general, when adding dependencies (edges), the programmer needs to make sure not to create a cycle and not to add an edge into an evaluating thread, except when creating a thread's subcomputations. The rules of the dynamic semantics of dag calculus include premises that enforce these constraints.

When a thread completes, it notifies the threads that have an edge from it. Thus, when *ta* and *tb* complete, they notify t, which becomes ready and may be scheduled for evaluation. When evaluated, the thread t resumes from where it yielded, computes the sum $!la+!lb$, then returns the result.

We can illustrate a complete evaluation of `fib_dc` by using a dag. In this commonly used representation, each vertex represents a continuous (not including yields and resumptions) section of the evaluation of a thread. To distinguish them from threads, such vertices are sometimes called as tasks, strands, or sparks. Figure 2 shows the dag for an evaluation of `fib_dc(4)` used for computing the 4^{th} Fibonacci number. In the dag, the subscripts denote the input argument for the corresponding call to `fib_dc`. The root thread t_4 spawns two threads t_2 and t_3, for each recursive call. In the dag, the spawn action is illustrated by *spawn edges* from t_4 to t_2 and t_3 The threads t_2 and t_3, in turn, synchronize with the vertex representing the continuation of t_4, labeled as t'_4, via edges from their continuation into t'_4.

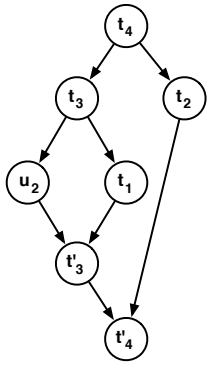

Figure 2. The computation dag for `fib_dc(4)`.

The dynamic semantics for dag calculus tracks the dynamic state of the parallel computation also by maintaining a dag. But the dags that are maintained by the calculus are significantly different than the dag shown in the example. The example dag should therefore be taken for intuition but should not be interpreted as an accurate representation of the dags maintained by the semantics. For example, the example dag in Figure 2 illustrates a complete evaluation. In contrast, dag calculus must construct the dag on the fly during evaluation and also use it to guide evaluation by scheduling its vertices. To this end, the dynamic semantics of the calculus maintains the dag as a set of vertices representing the currently live set of threads and a set of edges representing the currently live set of synchronization dependencies. The semantics ensures that a thread is never executed before its dependencies are completely satisfied. In addition, the semantics does not represented spawn edges explicitly. Intuitively speaking, the spawn edges are implicitly represented by creating a thread exactly when it is called for and staging for scheduling after it is released. The rest of this section presents a precise specification of the dynamics semantics of dag calculus.

Dynamic Semantics. We present the dynamic semantics for the dag calculus as a contextual semantics, with a reduction relation over computation dags. Formally, a computation dag is a pair (V, E) consisting of vertices V and edges E. Each vertex represents a thread. The edges E consist of pairs of vertices corresponding to synchronization dependencies between threads. Vertices are specified by the map V, which maps a thread to an expression and to a thread *status*. A thread status, denoted by s, is one of

```
1  function fib_dc n is
2    if n < 2 then n else begin
3      let la = alloc () in
4      let lb = alloc () in
5      let ta = newTd (la := fib_dc (n - 1)) in
6      let tb = newTd (lb := fib_dc (n - 2)) in
7      newEdge (ta, self ());
8      newEdge (tb, self ());
9      release ta;
10     release tb;
11     yield ();
12     !la + !lb
13   end
```

Figure 3. Dag-calculus code for a parallel Fibonacci function.

new, *released*, *executing*, and *finished*, written as N, R, X, and F, respectively. Threads status encode the current point in the life-cycle of a thread: a thread is always created in the N (new) state, and when released it transitions to the R (released) state. After a thread t is released and all other threads that it depends on are finished, t can be scheduled for evaluation. When scheduled for evaluation, the status of a thread changes to X (executing). When the evaluation of thread completes, its status changes to F (finished). As it evaluates, a thread may also yield the control, by calling `yield`, causing the thread to be suspended and having its status changed back to R (released).

The dynamic semantics for dag calculus, shown in Figure 4, involves two judgments: one for *thread-local reductions* and one for *dag reductions*. The first judgment, written $\sigma, e \to e', \sigma'$, relates an input store and an expression with an output expression 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.

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 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 of 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. Values, however, can be communicated to other threads through the store.

The rule NewTd creates a fresh 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 inserts a dependency edge between two given threads, called t_1 and t_2. It is the programmer's responsibility not to introduce cyclic dependencies, otherwise the program can get stuck. The NewEdge further requires that the target vertex t_2 has not already started executing, unless t_2 is equal to the thread that is trying to insert the edge.

The rule Yield encodes a very restricted form of control, particularly designed for parallelism. The effect of `yield` with respect to our dynamic semantics is simple: the status of the thread is changed from X (executing) to R (released), meaning that the thread is longer running. The consequences, however, are far-reaching, due to its interaction with NewEdge. In particular, since NewEdge allows the calling thread, t, to serve as the target of the edge, t_2, it is possible to insert an edge ending at a thread t that is currencly executing. If, an edge (t_1, t) is inserted and thread t yields, its evaluation will stop and

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

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

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

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

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

$$\frac{V(t) = (K[\texttt{self}\ ()], \mathsf{X})}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[t], \mathsf{X})], E, \sigma}\ \textsc{Self} \qquad \frac{V(t) = (K[\texttt{yield}\ ()], \mathsf{X})}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[()], \mathsf{R})], E, \sigma}\ \textsc{Yield}$$

Figure 4. Dynamic semantics for dag calculus. 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 $V(t)$.

t cannot be scheduled again until t_1 completes its evaluation and the edge (t_1, t) is deleted. As shown in the `fib_dc` example discussed earlier in this section, this ability to insert an edge enables a thread to "turn over control" to other threads that it creates and to wait for them to complete. This in turn, allows nesting, parallel computations, a crucial ability for expressing a broad range of parallel programs. In order to facilitate such nesting by allowing threads to insert edges targeting themselves, the dynamic semantics provides the rule Self, which returns the thread that fires the rule. Note that if a thread yields and has no incoming edges, it can be immediately rescheduled, since the rule Start applies. As we will see in the next section, combining `yield` with dynamic manipulation of the dag allows us to encode interesting computation patterns.

4. Parallelism in the Dag Calculus

In this section, we show how to translate three classic, high-level parallel constructs into our dag calculus. The source calculi considered in this section are minimal, and hence do not include features like conditional expressions or unit values where these are not needed. These and other sequential features are orthogonal to the task of translating the parallel constructs, and the translations we present can be trivially extended to handle them.

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 `forkjoin` (e_1, 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.

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

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

Translation The translation from this source language to our dag calculus is written $[\![e]\!]$ for expressions and $[\![v]\!]$ for values. 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 [[forkjoin (e1, e2)]] =
2   let l1 = alloc ()
3       l2 = alloc ()
4       t1 = newTd (l1 := [[e1]])
5       t2 = newTd (l2 := [[e2]])
6    in newEdge (t1, self ()); newEdge (t2, self ());
7       release t1; release t2; yield ();
8       (!l1, !l2)
```

We allocate two memory cells, `l1` and `l2`, for storing the results of the branches (line 2–3). We then create two threads, `t1` and `t2`, for the two branches (lines 3–4), assigning the results of the two computations to the corresponding locations. The results are subsequently read from these locations and put into a pair (line 8), but before that point, some interesting dag manipulation happens.

Firstly, we create edges from `t1` and `t2` to the thread that spawned them. Once the setup of the edges is complete, the threads are released (line 7). After releasing the subcomputations, the main thread needs to wait for the completion of `t1` and `t2` before reading the result values, and thus calls `yield` to suspend its execution. At the point when both `t1` and `t2` have finished, the main thread can be rescheduled—and at this point the results of the subcomputations can be safely read. Note that due to nondeterminism the subcomputations can finish even before the main thread yields: in this case, the main thread is allowed to restart the computation immediately.

Note that this pattern very closely resembles the code of the recursive branch of the parallel Fibonacci function shown in Figure 3. In fact, if we applied our translation to the fork-join version of Fibonacci from Section 2, we would obtain code that is virtually identical to the one from Figure 3.

Theorem 1 (Correctness of the translation of fork-join)**.** *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:*

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

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.

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

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 within 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.

$$\texttt{finish}(\{L[\texttt{async}(e)]\} \uplus S) \rightarrow \texttt{finish}(\{L[()]\} \uplus \{e\} \uplus S)$$
$$\texttt{finish}(\{()\} \uplus S) \rightarrow \texttt{finish}(S)$$
$$\texttt{finish}(\emptyset) \rightarrow ()$$

Translation The translation from this source language to our dag calculus is written $[\![e]\!]_t$ for expressions, and $[\![v]\!]_t$ for values. It takes, in addition to the expression e (or value v), 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:

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

In order to translate a complete program e, we introduce a dummy thread identifier t_0, and compute $[\![e]\!]_{t_0}$. 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 [[async(e)]]_t =
2   let t' = newTd [[e]]_t 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 [[finish({e})]]_t =
2   let t2 = self ()
3       t1 = newTd ([[e]]_t2)
4   in newEdge (t1, t2); release t1; yield ()
```

Similarly to the translation of fork-join, we need to create a thread for the subcomputation, `t1`, and synchronize it with the main thread. However, since the translation of e requires a name of the join node, we capture the name of the main thread by binding the result of `self`. Thus, any `async` calls within e will have the appropriate thread identifier passed to them. Since the finish block is a computation and always returns a unit value, we do not need to perform any additional work in the main thread after yielding: after all the `async` calls finish, the main thread can simply continue its work.

Theorem 2 (Correctness of the translation of async-finish)**.** *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.* status $(V(t')) = \mathsf{F}$*), we have (for some source-language heap h):*

$$[t \mapsto (l := [\![e]\!]_{t_0}, \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^{*} V, \emptyset, \sigma \Rightarrow \emptyset, e \rightarrow^{*} n, h$$
$$[t \mapsto (l := [\![e]\!]_{t_0}, \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^{\infty} \qquad \Rightarrow \emptyset, e \rightarrow^{\infty} .$$

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 $e \bigcirc\!\!\rightarrow e'$, captures the reduction relation for all expressions that are neither a `future` nor a `force`. Its rules are standard. The second judgment, written $M \rightarrow M'$, 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 $M \rightarrow M'$. 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 $[\![e]\!]$ for expressions, and $[\![v]\!]$ for values. 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 allocate a location, `l`, and a thread `t` which executes the body of the future. The thread is then released, so that the future can begin evaluation, and the pair built of `t` and `l` is returned.

```
1 [[future(e)]] =
2   let l = alloc ()
3       t = newTd (l := [[e]])
4   in 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)`. The translation creates an edge from `t` to the calling thread and in order to ensure proper synchronization, it yields. After the main thread is restarted, i.e., after the future has finished its computation, the final value can be recovered by reading the location `l`. In case the future has finished before it is forced, the edge-creation results in a no-op, and the thread can be restarted the moment it yields control.

```
1 [[force(v)]] =
2   let (t, l) = [[v]]
3   in newEdge (t, self ()); yield (); !l
```

Remark: the translation above can be optimized by testing whether the future has already completed; in this case, `force`(v) may immediately return the value `!l`. This could lead to reduced scheduler overheads caused by yielding.

Theorem 3 (Correctness of the translation of futures)**.** *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

In Section 3, we gave a declarative presentation of dag calculus. Here, we demonstrate that dag calculus can be given a realistic implementation. There are three main challenges.

- Concurrency: the declarative presentation assumes the operations for adding and removing edges to be atomic, whereas in fact many of these operations might execute concurrently.
- Control flow: the yield operation that suspends the execution of a thread requires a context switching operation. In particular, it involves storing continuations in vertices, with the additional complication that these continuations may be resumed on a different processor.
- Efficiency: the representation of the dag should allow for operations on it to be implemented efficiently. In particular, we need to maintain the pool of ready vertices in a distributed fashion, and to introduce a data structure for representing edges. The edge data structure needs to avoid contention when frequently accessed by different processors.

To describe how we address these challenges, we present a reference implementation of the dag calculus. In particular, our pseudo-code makes explicit 1) the main loop executed by the scheduler instance running on each of the processors, and scheduling execution of vertices obtained from the work queues; 2) the control flow, which alternates in a coroutine fashion between the execution of the main loop of the scheduler and the execution of the code associated with the dag vertices; and 3) the interaction with the concurrent data structures implementing incounters and outsets.

Before we describe this pseudo-code, we first motivate and present the context-switching operations on which we rely, and also describe our runtime representation of vertices and edges.

Suspending and resuming computations. An important design constraint for the implementation is the need to suspend and resume computations, which stems from the semantics of `yield`. In our approach, each instance of the scheduler executes in a main loop and switches to running the computation assigned to a dag vertex. This design allows us to implement a resumption mechanism that we use closely resembles coroutines: the main loop of the scheduler and the computations assigned to the dag vertices switch control to each other, resuming the partially executed thread when needed. The pattern is as follows. To begin working on a fresh vertex, the scheduler suspends itself and switches control to the computation associated with the vertex. When the execution of the vertex finishes, the control is switched back to the scheduler. Then the scheduler resumes its work and can continue executing other vertices. Alternatively, during the execution of the vertex, the `yield` command may be encountered. In this case, the execution of the vertex is suspended, and the control is passed back to the scheduler. The suspended computation, also referred to as a *continuation* in the following, is attached to the vertex. The execution of the suspended vertex may be subsequently resumed, when the vertex gets scheduled again.

In our pseudo-code, we assume an abstract data type called `cont` for representing continuations. We may create, save and load continuations using the three functions described next. First, the function `new_cont(f)` creates a new continuation that corresponds to the state at the beginning of the evaluation of `f()`. We use this function to create an initial state of each vertex the first time it is scheduled for evaluation. Second, the function `jump_cont(c)` restores the continuation `c` and continues evaluation there, discarding the

continuation (context) in which the command was encountered. We use this function to finish the evaluation of a vertex and return control to the scheduler. Third, the function `swap_cont(c1,c2)` saves the current continuation into `c1` and then restores the continuation `c2`. (Remark: `jump_cont` is an optimized version of `swap_cont` that does not require saving the current continuation.) We use the function `swap_cont` both when starting the execution of a vertex and when a vertex yields. Together, these three functions allow us to precisely describe the coroutine manipulation in our pseudocode.

Representation of vertices and edges. We next describe the runtime representation of vertices and edges. The corresponding signatures appear in Figure 5.

Each vertex is represented as a record with five fields. The `run` field contains a pointer to the code of the computation associated with the initial execution of the dag vertex, that is, before the vertex performs any yield operation. If a dag vertex has been suspended as a result of a yield operation, then the corresponding continuation is stored in the `cont` field; otherwise the field contains a null value. The record also contains three fields for representing edges. The `in` field describes the vertex *incounter* data structure, which keeps track of the (number of) edges incoming to the vertex. Symmetrically, the `out` field describes the vertex *outset* data structure, which keeps track of the (targets of) edges outgoing from the vertex. Last, the `releaseHandle` field is used to keep track of an artificial in-edge that we introduce to ensure that the vertex cannot be considered for execution when it has status `N` (new) or `X` (executing). We discuss these data structures and explain the purpose of `releaseHandle` in the following.

An incounter data structure is responsible for counting the number of incoming edges into a particular vertex, and detecting when this number reaches zero. When reaching zero, the incounter pushes the vertex into the work queue. There are two methods for manipulating an incounter: `increment` and `decrement`. These operations can occur concurrently in any order, as long as the following 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. To enable key optimizations in the implementation of the incounter data structure (and in particular to allow the use of concurrent tree data structures that avoid contention), the operation `increment` returns a *handle*, which points somewhere into the structure of the incounter. Any call to `decrement` is required to match a prior call to `increment` by providing the corresponding handle.

The outset is responsible for representing a set of vertex identities, corresponding to the targets of the outgoing edges of a vertex. 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 to 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 the `decrement` operation on all the handles stored, i.e. those for which the add operation returned `true`. This parallel notification operation is allowed to be distributed on several processors. Parallelism is useful for efficiently processing very large out-degree vertices. (Note that we do not specify here by which mechanism the notification operation may get distributed.)

We next explain the role of the `releaseHandle` field associated with every vertex. This field is used to introduce artificial in-edges whose purpose is to prevent from being added to the work queue threads that are either not yet released or already executing. To understand the (rather subtle) potential issue, consider the following example—the description of this example may safely be skipped in first reading. Assume a vertex t (with no incoming edges) to be executing the following operations: create a vertex t' (intuitively, a sub-task of t), create an edge from t' to t, execute a few additional computations, then yield (with the intention of resuming only after completion of t'). While t is executing the additional computations and has not yielded, the thread t' might get migrated to a second processor, and may run to completion before t reaches its yield point. In such a situation, the termination of t' leads to the removal of the outgoing edges of t', that is, to the removal of the edge from t' to t. As a consequence of the removal of the only incoming edge into t, the vertex t would get added to the work queue of the second processor (by virtue of the `decrement` operation), and may thus may get scheduled immediately afterwards for execution on the second processor. At this point, the vertex t would thus be executing both on the first and on the second processor, a situation that is clearly undesirable.

As illustrated by the above example, there is an issue if a vertex might get its last incoming edge removed while it is still executing. To prevent such situation, we add, for every executing vertex, an artificial incoming edge into the vertex. We store the handle associated with this edge in the `releaseHandle` field of the vertex, so as to be able to remove the artificial in-edge when the thread completes. We use exactly the same mechanism for dealing with freshly-created vertices, preventing them from being scheduled before they are released. (Remark: a naive approach of using a field of the vertex structure in order to explicitly keep track of the status of each vertex leads to potentially harmful races; by instead reusing the concurrent incounter data structure, we avoid such races.)

Structure of the scheduler. The declarations at the top of Figure 6 describes the per-processor variables. For each processor, we use three variables: the work queue of the processor, the vertex that is currently executing (if any), the continuation used to switch the control back to the main loop of the scheduler (valid when a vertex is executing). Note that these processor-local variables are always accessed at the processor that is executing the code, even when a continuation gets migrated form a processor to another.

The function `schedulerLoop` near the top of Figure 6 describes the main loop executed by each of the processors. At each iteration of the loop, we check whether its work queue is empty. If it is empty, then we enter a load-balancing scheme (not detailed here, e.g. relying on the work-stealing scheme) in order to attempt to populate its work queue. When the work queue is not empty, we pick a vertex out of it, store it into the `current` variable, and process the vertex as described next.

Before executing the `current` vertex, we first need to prepare it in two ways. First, we add an artificial in-edge onto the vertex (for reasons explained earlier), setting up the `releaseHandle` field of the vertex. Second, if the vertex has never been executed before (i.e., its `cont` field is null), then we set its `cont` field, using the `new_cont` function applied to the `enter` function. The function `enter` performs a sequence of three actions: first, it executes the `run` method associated with the current vertex, second it marks the vertex as finished, and third it returns the control to the scheduler.

Once the current vertex is prepared, we begin its execution by switching the control to the continuation stored in the vertex, using the `swap_cont` operation. Eventually, either as a result of completion of the vertex or upon a call to the `yield` operation, the control is returned to the scheduler, at line 15. At this point, we test whether the vertex has completed or not. If the vertex has completed, then we call the `parallelNotify` operation in order to call the `decrement` operations on the incounters of the vertices associated with all the outgoing edges of the vertex. If, however, the vertex has only yielded, then we just remove the artificial in-edge whose handle was stored in the `releaseHandle` field of the vertex. At this point, the scheduler is ready for the next iteration of its main loop.

```
struct vertex
  void run() // computation body of the vertex
  cont* cont // suspended continutation (null if finished or never yielded)
  incounter* in // counter of the number of incoming edges
  outset* out
    // set of handles, each handle being associated with the
    // incounter of the target vertex of one of the outgoing edges
  incounterHandle* releaseHandle
    // a handle on an artificial in−edge used to prevent new and executing
    // vertices from being added to the work queue

struct incounter // abstract
struct incounterHandle // abstract, depends on the incounter type
incounter* new_incounter(vertex* v)
 // creates a counter with value zero
incounterHandle* increment(incounter* i)
 // increments the counter by one, and return a handle to be
 // provided for performing the matching decrement operation
void decrement(incounterHandle* h)
 // when reaching zero, decrement calls workQueue.push
 // on the vertex v that was provided at construction

struct outset // abstract
outset* new_outset()
 // creates an empty set of incounter handles
bool add(outset* o, incounterHandle* h)
 // add a handle to the set; may return false if parallelNotify
 // has been called already; if add returns true, then the
 // handle is guaranteed to be processed by parallelNotify
void parallelNotify(outset* o)
 // calls the decrement operation on all handles that were added to the set
```

Figure 5. Representation of dag vertices and edges. Some implementation details are left abstract.

Implementation of primitive dag operations. Finally, we discuss the primitive dag-manipulation operations. The implementation is shown in the bottom half of Figure 6.

The `newTd` primitive, which applies to an expression e, is implemented using the function `createThread`, which allocates a vertex and assigns: the computation body to e, a fresh incounter, and a fresh outset. We also set up an artificial in-edge into the vertex, storing the corresponding handle into the `releaseHandle` field. This artificial in-edge is removed when the vertex gets released.

The operation `release`, when called on a vertex, calls the `decrement` operation on the handle stored in the `releaseHandle` field of the vertex. This removes the artificial in-edge that was set at the creation of the vertex. If no other edges have been added in the meantime, then the vertex gets pushed into the local 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 return false if the source vertex has already completed. In this case, the increment operation needs to be undone, by decrementing the incounter.

The operation `yield` suspends an executing vertex and switches control back to the scheduler, using `swap_cont` to save the current continuation into the `cont` field of the vertex and restore the continuation of the scheduler.

Finally, the operation `self` returns the current vertex.

Correctness of the implementation with respect to the semantics. In what follows, we argue that the algorithm from Figure 6 does indeed correspond to the dag calculus from Figure 1, i.e., that the evaluation of the scheduler loops on a multiprocessor machine matches a valid execution of the dag calculus semantics.

The algorithm only executes safely for well-behaved programs that respect the rules of the dag calculus, such as the rule of only releasing vertices have status new or adding edges to vertices that are

```
processor_local queue<vertex*> workQueue // bag of vertices
processor_local vertex* current // running vertex
processor_local cont* proc_cont // continuation of the scheduler

void schedulerLoop() // executed by each processor
  while true // termination details omited
    if workQueue.empty()
      // implementation−dependent load balancing
      acquireWork() // blocking call
    current = workQueue.pop()
    current→releaseHandle = increment(current→in)
    if(current→cont == null) // initialize the continuation
      current→cont = new_cont(&enter)
    swap_cont(proc_cont, current→cont) // execute the vertex
    if(current→cont == null) // vertex has finished
      parallelNotify(current→out)
    else // vertex has yielded
      decrement(current→releaseHandle)

void enter() // execute the current vertex, assuming it has never yielded
  current→run()
  current→cont = null // mark vertex finished
  jump_cont(proc_cont)

// ''newTd e'' is short for ''createThread(fun () ⇒ e)''
vertex* createThread(runMethod)
  vertex* v = new vertex
  v→run = runMethod
  v→in = new_incounter(v)
  v→out = new_outset()
  v→releaseHandle = increment(v→in)
  return v

void release(vertex* v)
  decrement(v→releaseHandle)

void newEdge(vertex* v1, vertex* v2)
  incounterHandle* h = increment(v2→in)
  bool success = add(v1→out, h)
  if not success // vertex v1 has already completed
    decrement(h) // roll back on edge creation

void yield()
  swap_cont(current→cont, proc_cont)

vertex* self()
  return current
```

Figure 6. Realization of the scheduler loop and primitive operations. Details of load balancing and termination detection are omited.

not executing or ready. Thus, our correctness theorem connecting the behavior of the algorithm with the rules from the dynamic semantics needs to depend on an assumption characterizing well-behaved programs. To formalize this notion, we extend the dag semantics with two error rules (stated in the appendix), one for `release` and one for `newEdge`. We then assert that a program is well-behaved if none of its possible executions may reach an error state.

The correctness theorem relates a dag configuration (V, E, σ) with a machine state, written (M, S), where M denotes the memory and S describes the state of the processors. The state S_p of a processor p consists of the processor-local variables: work queue, current vertex, and current scheduler continuation; as well as the code pointer and the execution context of the processor.

Theorem 4. *Let e_0 to be a dag calculus expression, t_0 to be a thread identifier, l_0 a location for the final result. Assume that:*

- *r_0 the code pointer to the compiled code for e_0, i.e. in the sense that* Compile(e_0, r_0) *holds (*Compile *is defined in the appendix),*
- *$V_0 = [t_0 \mapsto (e_0, \mathsf{R})]$, which describes the initial vertex map, storing the initial vertex t_0 with body e_0 and released status,*

- $E_0 = \emptyset$*, which describes the initial set of edges,*
- $\sigma_0 = [l_0 \mapsto ()]$*, which describes the initial heap,*
- $M_0 = [l_0 \mapsto (), t_0 \mapsto \text{InitVertex}(r_0)]$*, which describes the initial memory state, with a memory cell at location l_0, and a representation of the initial vertex with run method r_0, fresh incounter and outset, and null continuation and releaseHandle (as described by the auxiliary* InitVertex *operator),*
- S_0*, which describes the initial state of the processors, by asserting that they are entering* `schedulerLoop` *(i.e., $\forall p.\ L_p = 6$, where L_p denotes the line of processor p), and asserting that all work queues are empty except one that contains exactly t_0 (i.e., $\exists p.\ Q_p = \{t_0\} \wedge (\forall p'.\ p \neq p' \Rightarrow Q_{p'} = \emptyset)$, where Q_p denotes the work queue of processor p).*
- *the source program is well-behaved is the sense that none of its execution may reach an error state: $\neg\ (V_0, E_0, \sigma_0) \twoheadrightarrow^* \bot$,*
- *the machine evaluates to a terminal state, in the sense that $(M_0, S_0) \rightarrow^* (M', S')$, for some (M', S') where the state S' is such that all work queues are empty ($\forall p.\ Q'_p = \emptyset$) and all processors are idle ($\forall p.\ L'_p = 9$).*

Then, there exists a configuration (V', E', σ') such that:

- *we have a corresponding reduction in the dag semantics: $(V_0, E_0, \sigma_0) \twoheadrightarrow^* (V', E', \sigma')$,*
- *(V', E', σ') is a terminal configuration: in the sense that vertices are finished ($\forall t \in \text{dom}(V').\ \text{status}(V'(t)) = \mathsf{F}$), and no edges remain ($E' = \emptyset$),*
- *the value $M'(l_0)$ stored at location l_0 by the machine execution matches the value $\sigma'(l_0)$ stored at that same location in the dag semantics, written* $\text{CompileVal}(\sigma'(l_0), M'(l_0))$ *in the appendix.*

We prove the theorem using a global invariant relating machine configurations and dag configurations. We refer to the appendix for details. We believe that the theorem could be extended to capture preservation of divergence; we leave this extension to future work.

6. Implementation

One implementation scheme that has proved effective for task-parallel systems is the following. Initially, the client of the system has in hand a program that is encoded as a DAG term. To start the computation, the client issues to the task-parallel system the initial DAG term and then instructs the task-parallel system to start. The system launches one pthread for each processor in the machine and keeps these pthreads running at least as long as the client program runs. Each of the pthreads begins by entering a scheduling loop, in which the pthread greedily executes DAG nodes that are ready. After the pthreads start running their scheduling loops, one of the pthreads gets assigned the root task of the DAG. The root task represents the entrypoint of the client program (i.e., `main` function). From thereon, all pthreads cooperate on executing tasks until no ready task remains to be executed.

What is challenging about implementing such as system on a multicore platform? We identify three key issues that relate specifically to our DAG calculus. One issue relates to the ability of nodes to add and remove edges in the DAG on the fly. We addressed this issue in the previous section.

Another challenge concerns the granularity-control problem, which is a classic problem in parallel computing. The granularity-control problem relates to overheads that are generated by DAG management, such as DAG-node creation, edge resolution, etc. If these overheads are too large, then the benefits of parallel execution can easily be negated, leading to poor performance in terms of time and energy, in particular. Solving this problem is crucial, but fortunately there is a large body of work addressing this problem that can be readily applied to an implementation of our DAG calculus.

The last challenge concerns the problem of load balancing. This problem is important because, on the one hand, a program in the DAG calculus can express arbitrarily many opportunities for parallelism, yet, on the other, any real-world system provides a fixed number of processors. The load balancing problem concerns how the parallelism is mapped from the computation DAG to the processors in the system over time. Fortunately, the body of work in this field is immense, and there are off-the-shelf solutions that address scheduling for our DAG calculus.

7. Implementation and Experiments

As established in Section 4, the dag calculus can serve as an intermediate language and different forms of parallelism, such as async/finish and futures, can be compiled to the intermediate language. What remains unanswered thus far is efficiency: can a library implementation of the dag calculus efficiently represent the different forms of parallelism? To test the implementation empirically, we ported the implementation to C++ to make a small, self-contained dag-programming library. In this section, we present an evaluation of the dag calculus. The results show that, even when compared to highly engineered parallel-programming systems, the dag calculus performs well. Furthermore, in some cases, thanks to its flexibility in allowing non-standard forms of synchronization to be expressed, outperforms such systems. We also show that in some cases, such as in futures, certain details of the underlying dag representation can affect performance, suggesting that a compiler that translates a high-level parallel language to dag calculus can make important optimizations.

Implementation. Our C++ implementation follows closely from the pseudocode shown in the previous sections. We implement `yield` using a shallow context-switch mechanism that saves the state of the registers (but not the signal mask). More generally, in a parallel runtime system that provides general-purpose control operators, e.g., `call-cc`, they can be used to implement `yield`.

One notable optimization that we perform is to avoid creating artificial in-edges for vertices involved in the translation of fork-join. This optimization is safe, because we are careful not to migrate freshly created vertices before the executing vertex yields.

For load balancing, we reused an implementation of a work-stealing scheduler from another study [3]. In the scheduler, each processor is assigned a private work queue in which to store the threads (i.e., dag vertices) that are ready to be executed. To realize load balancing, processors issue work request to a randomly selected processor, through shared memory. To ensure that each work request is eventually handled, each processor polls on a regular basis for work request.

For the implementation of incounters and outsets, our implementation includes a flexible mechanism by which the programmer can decide, for each vertex created at runtime, how its incounter and its outset should be represented. The choice of the representation essentially depends on the arity of the vertex. On the one hand, we have simple implementations with little overhead but that may suffer contention when the arity of a vertex is large. On the other hand, we have more complex tree data structures that can scale to large arity. In what follows, we give more detail.

Regarding incounters, there are essentially three useful implementations. The first one is specific to the case of vertices with at most one incoming edge, in which case no synchronization is needed. The second implementation relies on an atomic counter, updated using a fetch-and-add operation. It is particularly well-suited for small in-degree vertices, such as vertices involved in the encoding of fork-join. The third implementation is designed for the case of vertices with potentially many incoming edges. It consists of a (modified version of) scalable non-zero indicators [16]. In short,

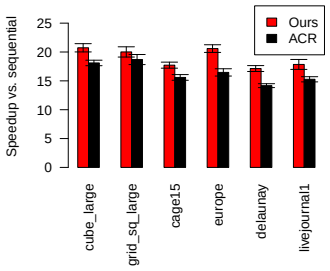

Figure 7. Parallel DFS using 40 cores, for various input graphs.

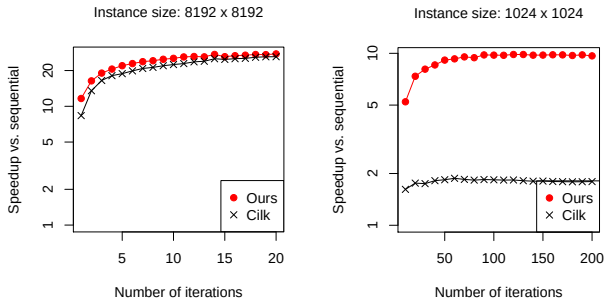

Figure 8. Gauss-Seidel application, using 40 cores.

such structures represented a counter as a tree data structure, where incounter handles correspond to nodes in the tree; the tree can be concurrently modified and is able to detect the moment at which zero is reached, with limited contention.

Regarding outsets, there are also three useful implementations. The first one accommodates only the case of vertices with at most one outgoing edge (as, e.g. forked vertices in fork-join), in which case no synchronization is needed. The second implementation relies on a Treiber stack, i.e. a locked queue, which is well-suited for smalll outdegree vertices. The third implementation is, here again, based on a growable tree data structure, where outgoing edges can be added concurrently with limited contention—this tree structure is essentially a variant of its incounter counterpart.

Experimental setup. We considered a collection of benchmark programs, described next, executing them on up to 40 cores.[1] For each data point, we report the average running time over 30 runs.

Parallel graph-traversal benchmark. Our first experiment considers the performance of a graph-traversal algorithm in which parallelism is expressed in the style of async/finish. What this experiment shows is that 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 [4]. 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 processors. The ACR code uses their own implementation of termination detection, whereas ours uses an incounter data structure. In this case, for the implementation of the incounter, we used the solution based on the scalable non-zero indicator [16]. Figure 7 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.

Parallel stencil-computation benchmark. 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 code unmodified, from the authors of another study [39]. 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 performance that is the same or slightly better than 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 independently for each algorithm.

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 does not fit directly into the regimes of async/finish or parallel futures.

One additional optimization that is employed by our algorithm is the use of *lazy binary splitting* [46, 47]. In particular, our algorithm creates one initial dag vertex to start the stencil computation. In this vertex, we store in a queue the indices of all the blocks that are ready to be processed. The initial dag vertex proceeds to process blocks from its queue in a sequential fashion until a work request is received from an idle processor. Once received, the work request triggers the dag vertex to fork off a copy of itself, such that the original dag vertex keeps half of the queue and the new one the other half. The idle processor subsequently starts working on the new dag vertex, while the original dag vertex continues to be processed on its original processor. This splitting process continues until there is no work left to do. The name for this technique is lazy binary splitting because the work queues are split only when there is an idle processor that is ready to receive the work. The benefit of this optimization is that the algorithm can express parallelism between blocks without having to pay the cost of creating a dag vertex for each and every block.

The plot in Figure 8 shows the results from this experiment. Owing to space limitation, we show the plots for only the grid sizes 1k and 8k, which we selected to match those reported in another study [39]—we obtained consistent results from other settings of the grid size. Results show that, for a large grid of size 8k, the Cilk version and ours achieve fairly similar speedups, with ours being slightly faster. But for a smaller grid size of 1k, the Cilk version achieves no more than 2x speedup, whereas our version achieves 10x. We explain this difference by the fact that the Cilk implementation can exploit parallelism only within one wavefront, whereas ours is limited only by the inter-block dependencies.

[1] We compiled the code using GCC `-O2 -march=native` (version 5.2). Our machine is running Ubuntu Linux kernel v3.13.0-66-generic. It 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 80 hardware threads. However, to avoid complications with hyperthreading, we did not use more than 40 threads.

Futures benchmark. Our next experiment uses a benchmark to evaluate our implementation of futures. Futures have been studied extensively since the early days of parallel computing [20]. Yet, there are very few, if any, practical, comparative studies involving futures. It might be because specific uses of futures can in many cases be replaced with another construct, such as async-finish or fork-join. For example, PPBS benchmarking suite [10], which is perhaps the most comprehensive parallel benchmarking suite that targets shared memory computers, such as multicore machines, has no benchmarks that are futures-specific. Facebook's Folly library uses futures in a more practical fashion to mask latencies imposed by IO calls [1]. But Folly applications, being IO bound, are too coarse grained for benchmarking the effectiveness of threading primitives.

Instead, we consider here a synthetic benchmark that is inspired by earlier work on algorithms for implementing futures on the PRAM machine [19]. The function `mixed`, shown below, creates a future f and forces it in a parallel loop for a specified number n times. The `parallel_for` loop is implemented with async-finish. What makes this benchmark challenging is that future f has a potentially very high number of dependencies that must be woken up and scheduled. In our evaluation, we take n to be 10 million.

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

More specifically, 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, $10n$ operations are performed in total.

The results are shown in Figure 9. To evaluate our implementation, we considered two different implementations of our concurrent dag structure. The first one uses for incounter the atomic counter cell and for outset a Treiber stack. The second one uses for incounter and outset the tree-based structures described in the beginning of the section. Results show that the shared incounter and Treiber stack perform best only when the number of processors is strictly fewer than ten. The reason is that these two structures are relatively simple and perform better under light or no contention, but quickly deteriorate when there is sufficient contention among the participating processors. Overall, this benchmark demonstrates the ability of our implementation to scale gracefully with large numbers of concurrent operations. Furthermore, the results suggest that, under low load, a simple implementation of the incounter and outset performs best, whereas the scalable implementation performs best otherwise. As we see in the next benchmarks, often it is easy to identify cases where the simple implementation suffices. In other cases, one can use the scalable implementations, paying what seems to be a reasonable overhead when the scalability of the structures is not needed.

Fork-join benchmarks. The final part of this study shows results from three benchmarks taken from the Problem-Based Benchmark Suite (PBBS) [10]. 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-calculus implementation and compared results with the original codes in Cilk. For each incounter, we used a counter cell that is updated by atomic fetch and add, and for each outset, we used a single non-atomic cell to store the pointer to the next dag vertex. These encodings are a good match for these fork-join applications because, in these applications, each thread has zero, one, or two incoming edges and zero or one outgoing edges. Results are shown in Figure 10. The results show that, even though the Cilk system has benefited from significant, long-term engineering effort, the encodings in our dag-calculus implementation perform nearly as well as the Cilk-based counterparts.

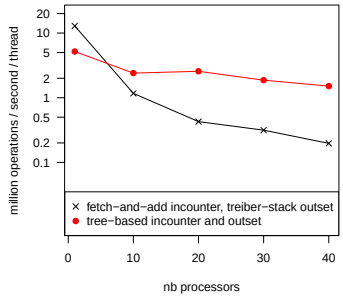

Figure 9. Performance, measured as number of dag operations per second per thread, of the call to `mixed`(10^7), for two combinations of incounter/outset implementations, showing that tree-based structures have higher overheads but better scalability.

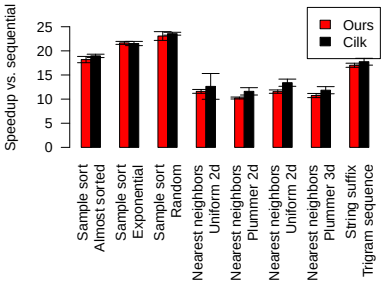

Figure 10. Results for fork-join programs: sample sort, string suffix, and nearest neighbors, all using 40 cores.

8. Related Work

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

Semantics of concurrency has been studied extensively and many concurrent calculi has been proposed, including CSP [24], π-calculus [33, 34] and actors [23]. 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. [6, 21]), and several programming languages such as Concurrent ML [40] and Pict [38] have been designed based on these calculi. Operational semantics for concurrent versions of ML have also been developed (e.g. [7, 37]).

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 [36], 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 [5, 11, 19, 26]), dags are sometimes used for presenting a cost-semantics [41–43] for parallel programs. For example, Greiner and Blelloch [19] present a cost semantics for a language with futures based on dags. They also present algorithms for the PRAM model. Spoonhower et

al. [45] use a similar technique for presenting a cost semantics to account for space use for a language with parallel tuples.

There have 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 [39]. 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 [12] resembles futures, but provides a mechanism for deterministic programming with shared mutable state. LVars [28] are one alternative approach that have recently been extended to support primitives with the same power as both async/finish and futures [29].

The semantics and the proofs presented in this paper all assume a sequentially consistent memory model. Our C++ implementation, however, operates on an Intel X86 machine with a weak memory model, and employs the necessary atomic read-modify-write operations and memory fences to ensure correct behavior. We expect, that the semantics and the proofs could be generalized to accommodate weaker memory models. For example, for x86-TSO, the store σ in the formulation of our semantics (Section 3) would include the write buffers associated with each of the processors [35].

9. Conclusion

Calculi for concurrent programming abound. In contrast, there is no sufficiently general calculi for parallel computing. In this paper, we present such a calculus and show that it can be implemented on modern parallel multicore hardware. Our experiments show that the proposed calculi and its practical realization, which is a relatively young implementation, performs well compared to the state of the art parallel systems that have benefited from many years of engineering. Natural future directions of research include development of a parallel functional-programming language that uses the dag calculus as an intermediate language, extensions of our model to interact with shared memory with relaxed semantics, and development of a type system for enforcing safety of dag calculus programs.

References

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

[2] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. *Theory of Computing Systems (TOCS)*, 35(3): 321–347, 2002.

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

[4] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. A work-efficient algorithm for parallel unordered depth-first search. In *ACM/IEEE Conference on High Performance Computing (SC)*, page 1, 2015.

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

[6] J. C. M. Baeten. A brief history of process algebra. *Theory of Computing Science*, 335(2-3):131–146, May 2005. ISSN 0304-3975.

[7] 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.

[8] Guy Blelloch and Margaret Reid-Miller. Pipelining with futures. *Theory of Computing Systems*, 32(3):213–239, 1999. ISSN 1433-0490. . URL http://dx.doi.org/10.1007/s002240000117.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] Guojing Cong, Sreedhar B. Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay A. Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, pages 536–545, 2008.

[16] 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.

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

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

[19] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Transactions on Programming Languages and Systems*, 21(2):240–285, March 1999. ISSN 0164-0925.

[20] 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.

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

[22] Maurice Herlihy and Zhiyu Liu. Well-structured futures and cache locality. *ACM Transactions on Parallel Computing*, 2(4):22:1–22:20, February 2016. ISSN 2329-4949. . URL http://doi.acm.org/10.1145/2858650.

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

[29] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: Quasi-deterministic parallel programming with lvars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 257–270, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. . URL http://doi.acm.org/10.1145/2535838.2535842.

[30] 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.

[31] 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.

[32] 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.

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

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

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

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

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

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

[43] 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.

[44] 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.

[45] 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.

[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.

[47] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *TOPLAS*, 36(3):10:1–10:51, September 2014. ISSN 0164-0925. . URL http://doi.acm.org/10.1145/2629643.