Language abstractions and scheduling techniques for efficient execution of parallel algorithms on multicore hardware

Arthur Charguéraud (Inria), 2020

Foreword

The Multicore OCaml team has made significant progress in the recent years. There now seems to be interest in working on the high-level parallelism constructs. Such constructs are also tightly connected to the problem of controlling the granularity of parallel tasks.

I've been working on parallel constructs and granularity control from 2011 to 2019, together with Umut Acar and Mike Rainey. We published a number of papers, each of them coming with theoretical bounds, an implementation, and evaluation on state-of-the-art benchmark of parallel algorithms.

While we mainly focused on C++ code, I speculate that nearly all of our ideas could be easily applied to Multicore OCaml. Porting these ideas would deliver what seems to be currently missing in Multicore OCaml for efficiently implementing a large class of parallel algorithms.

Gabriel Scherer and François Pottier recently suggested to me that it appears timely to share these results with the OCaml community. I'll thus try to give an easily-accessible, OCaml-oriented introduction to the results that we have produced. Note, however, that most of the ideas presented would apply essentially to another other programming language that aims to support nested parallelism.

I plan to cover the semantics of high-level parallelism constructs, to describe and argue for work-stealing scheduling, to present a number of tricks that are critical for efficiency, and to advertise for our modular, provably-efficient approach to granularity control. I'll post these parts one after the other, as I write them.

Historical background

Umut, Mike and I pursued a long line of work on nested parallelism that started long before we even started our academic career. I'd like to point out the lines of work that certainly had direct impact on our work.

Work stealing is an old idea. The reference paper on the analysis of work stealing traces the idea back to Burton and Sleep's 1981 research on parallel execution of functional programs and Halstead's 1984 implementation of Multilisp. Here we are, 35 years later, discussing the implementation of Multicore OCaml.

Robert Blumofe and Charles Leiserson's theoretical bound for work stealing published in 1993 was a major breakthrough. Their theorem was generalized and its proof simplified over the years. This theoretical result was the starting point of the work on the the Cilk project. Cilk is an extension of C/C++ that provides support for parallel-for loops and the fork-join (spawn-sync) constructs. A reference paper on the implementation of Cilk is that on Cilk5 (PLDI'98).

In passing, I should also mention the X10 language, developed at IBM since 2004. X10 is class-based language, featuring the async-finish construct, and implemented using a work-stealing scheduler. Numerous papers have been published about X10, including on language semantics, optimimizations, and granularity control techniques.

Guy Blelloch has a rich line of work on parallelism. He worked on NESL in the 90's, then developed numerous parallel algorithms and techniques for multicore programming. Most of his recent work is implemented in Cilk/C++, and delivers the state-of-the-art results, on a state-of-the art collection of benchmarks for irregular parallelism called PBBS.

John Reppy worked on Concurrent ML (CML) in the early 90's, a concurrent extension of SML. In the year 2000s, the rise of multicore hardware motivated work on optimizing CML. This motivated the Manticore project, which included not only CML-style constructs, but also NESL-style parallel arrays.

Matthew Fluet is the main developer of MLton, a whole-program optimizing compiler for Standard ML, with support for CML. MLton is extend by MultiMLton, which provides various kinds of fine-grained parallelism.

Umut Acar published his first paper together with Guy Blelloch and Robert Blumofe in 2000, on the Data locality of work stealing, Umut also was a collegue of Matthew Fluet for a couple of years, and he was also familiar with the line of work on Manticore.

Mike Rainey worked for his master and his PhD on the Manticore project, together with John Reppy and Matthew Fluet and others. a reference paper on the implementation of Manticore is the JFP'11 paper.

I personnally had no background knowledge about parallel programming when I joined Umut and Mike at MPI-SWS in 2011. My main experience was on programming sequential algorithms, and in complexity analysis.

For the first few months, we hacked using Manticore, but then quickly switched to C++, which offered us better absolute performance, avoided interference with the GC while working on scheduling and granularity control, and enabled us to reuse existing benchmarks. We implemented all of our work in the form of a C++ library called PASL, which allowed us to not spend time hacking into the back-end of a specific compiler.

Whenever possible, we evaluated the performance of our scheduling techniques by comparing our port of the PBBS benchmarks against Blelloch et al's Cilk

programs. In 2012, our work continued at Inria and at CMU, funded in part by the ERC DeepSea project.

In the recent years, Umut Acar's group at CMU has developed extensions to the MLton compiler to support fork-join parallel programs. The implementation is called MPL (MaPLe), and it is used at CMU both for research and teaching purposes.

This historical background section is meant to give a broad picture of the line of works and implementations involved. I'll be citing some specific papers throughout the text. I do not plan, however, to discuss all the related work in depth—for that, I refer to the related work sections from the publications.

Part 1: Semantics of parallel constructs

Published: 2020-05-22

In this first part, I discuss fork-join, parallel-pairs, n-ary fork-join and parallel-tuples, async-finish, parallel-for loops, and futures. For each construct, I describe its semantics in a couple of sentences, explain in which kind of algorithms it is relevant, compare its expressive power with other constructs, and say just a few words regarding their benefits in terms of performance. Then, I'll present the DAG-calculus, which presents a simple, unified semantics that handles arbitrary nesting of all these constructs. Finally, I'll discuss the semantics of exceptions in parallel code.

Throughout the text, a "fiber" refers to a lightweight thread managed by the runtime of the language, as opposed to a a heavyweight thread managed directly by the operating system.

Fork-join

The semantics of:

```
t0;
fork_join (fun () -> t11) (fun () -> t12);
t2
```

is to execute `t0`, then execute the two "branches" `t11` and `t12` potentially in parallel, then move on to the execution of the "continuation" `t2` only after the two branches have completed.

With fork-join, the two branches must return unit. We'll discuss shortly afterwards the generalization to parallel-tuple, which allows collecting results from each of the branches.

The two branches may concurrently act over memory, according to the semantics of concurrency—i.e. the rules provided by the memory model for Multicore OCaml. One essential point is that all the write operations performed in either of the two branches must be visible to the continuation `t2`. Likewise, all the operations performed in `t0`, before the fork-join operation, must be visible to both branches.

Fork-join operations typically appear nested in recursive algorithms. For example, consider a function that increments in parallel all the cells in the range [i,j] from an array "a". Its code is as follows, where the base case includes granularity control: batches of fewer than 1000 cells are processed sequentially.

```
let rec parallel_incr a i j =
   if j - i <= 1000 then begin (* base case *)
      for k = i to j do
         a.(k) <- a.(k) + 1
      done
   end else begin
      let m = (j - i) / 2 in
      fork_join (fun () -> parallel_incr a i m)
                (fun () -> parallel_incr a (m+1) j)
   end
```

A surprisingly-large number of algorithmic problems can be handled using the exact same pattern as in the function above, and in many cases deliver the best-known speedups (when executed with a work-stealing scheduler).

Parallel-pairs

A parallel-pair is a variant of fork-join that allows gathering return values from the two branches. The semantics of:

```
t0;
let (x1,x2) = parallel_pair (fun () -> t11) (fun () -> t12) in
t2
```

is similar to that of fork-join, except that `x1` denotes the result of `t11` and `x2` denotes the result of `t12`. These two variables are bound in the continuation `t2`.

Remark: papers on parallel-ML used the syntax `(|t11,t12|)` for describing a parallel pair; but let's not discuss syntax in this document.

Fork-join can be trivially encoded using parallel pairs, by just ignoring the unit result values, as follows.

```
let fork_join f1 f2 =
   ignore (parallel_pair f1 f2)
```

Reciprocally, parallel-pair can be encoded using fork-join, using an array for storing the results from each of the two branches.

```
let parallel_pair f1 f2 =
   let r = [| None; None |] in
   fork_join (fun () -> r.(0) <- Some (f1()))
             (fun () -> r.(1) <- Some (f2()));
   (unsome r.(0), unsome r.(1))
```

Observe that I do not use an atomic array to store the results, and use conventional reads and writes to manipulate the array `r`. Doing so is correct because of the rule that operations performed before `fork_join` are visible by both branches, and operations performed in the two branches are visible in the continuation, after the `fork_join`. Anyway, a low-level implementation of parallel-pairs could coordinate with the GC to avoid the allocation of the options in the array `r`.

N-ary fork-join and parallel-tuples

So far, I have presented binary versions of fork-join and parallel-pairs.

A number of algorithms naturally require 4-way branching. For example, matrix-multiply starts by making 4 recursive calls in parallel, one of each quarter of the input matrix.

In theory, n-ary fork-join can be easily encoded using a (possibly incomplete) binary tree of binary fork-join constructs. Likewise for parallel-tuples, which generalize parallel-pairs.

For example, one can encode a 3-way and 4-way fork-join as follows:

```
let fork_join_3 f1 f2 f3 =
   fork_join (fun () -> fork_join f1 f2) f3

let fork_join_4 f1 f2 f3 f4 =
   fork_join (fun () -> fork_join f1 f2)
             (fun () -> fork_join f3 f4)
```

In practice, however, it may be possible to reduce the synchronization costs and delays by providing primitive support for bounded-arity n-ary fork-join and/or parallel-tuples. For example, one could synchronize 4 branches using a single concurrent counter, rather than involving 3 concurrent counters as the encoding would do. We'll get back in another section to these implementation details.

Note that it is not hard to implement a function called `fork_join_list` that is able to handle a list of branches, by applying a divide and conquer approach to build a binary fork-join tree.

In practice, it is handy for writing programs to have reasonable syntax for parallel-tuples and fork-join. But let me postpone discussion of syntax for now.

Async-finish

Async-finish is a construct that allows forking a variable number of branches with more flexibility than n-ary fork-join.

The finish construct introduces a semantic block in which the evaluation may spawn branches using the async construct. The async operation is non-blocking, it simply creates a parallel sub-computation. These async-ed sub-computations may themselves spawn additional branches using async, within the same finish block. When the execution reaches the last instruction from the finish block, the finish construct blocks until all the branches spawned (either directly or indirectly) in the scope of the finish block have completed.

The basic version of async-finish can be captured by a pair of two functions.

```
finish : (unit -> unit) -> unit
async  : (unit -> unit) -> unit
```

Let me illustrate their use on a program that counts the number of ways to solve a puzzle starting from an initial configuration. A single finish block is introduced. During the exhaustive, recursive search, each recursive call gives rise to an async. Those async-ed branch may themselves execute recursive calls and produce further async-ed branches. The program terminates with the synchronization of the finish block, that is, when all async-ed branches have completed.

```
let nb = Atomic.make 0

let _ =
   finish (fun () ->
      let rec search config =
         if problem_solved config then
            Atomic.incr nb
         else begin
            let parallel_search config2 =
               async (fun () -> search config2) in
            List.iter parallel_search (get_transitions config)
         end in
      search initial_config)
```

Observe that the number of "async-ed" branches that synchronize on the same finish can be huge. There exists specially-crafted concurrent data structures called SNZI (sum-non-zero-indicators) that are able to implement the "termination detection" (a.k.a. the "join resolution") efficiently (without suffering from contention). I'll get back to that structure later on.

Async-finish can trivially encode fork-join. The traditional encoding is:

```
let fork_join f1 f2 =
   finish (fun () ->
```

```
async f2;
f1 ())
```

Note that a single `async` suffices. It would be also correct to write `async f1` instead of just `f1()`, however this would be redundant, because the main body of the finish block already accounts for one thread. It would also be correct to async `f1` then execute `f2()`, however the above presentation ensures, with work-stealing schedulers, that `f1` gets executed locally, i.e. using the same core as the one that handled the function call to `fork_join f1 f2`.

Reciprocally, a program written using async-finish can be rewritten, using fork-join. Intuitively, every async can be implemented as a binary fork join between that async-ed branch and the continuation. Encoding async-finish using fork-join thus requires a continuation- passing style (CPS) translation, which is impractical to perform by hand.

Let me conclude this section by arguing that it makes sense for a surface language to include a fork-join construct in addition to async-finish.

- fork-join has a key advantage: its arity is known at compile time, thus the synchronization of its branches can be implemented more efficiently than if fork-join was encoded using async-finish.
- async-finish enables describing more complex spawn patterns, without the need for a CPS-translation of the code. Moreover, if many branches need to synchronize on a single finish, an encoding using fork-join would introduce an inefficient cascade of intermediate synchronizations.

Nested async-finish

When programming modularly, different finish blocks can be nested. Some languages restrict the async-finish construct by allowing threads to async only with respect to the nearest finish block. Nevertheless, it can be useful to async with respect to outer finish blocks—just like it can be useful to break out of a loop that is not the nearest one.

To support the general case, we need each finish block to introduce an identifier, so that each async operation may specify on which finish block it should "join". In that presentation, the constructs are typed as follows.

```
labelled_finish : (block_id -> unit) -> unit
labelled_async  : block_id -> (unit -> unit) -> unit
```

One question is whether OCaml should provide: - only the restricted operations `finish` and `async`, - only the general operations `labelled_finish` and `labelled_async`, - or both.

The general API is unnecessarily heavy for most practical usages. The restricted API is limited for some applications, and does not seem to be trivially derivable from the general API. Keeping track of the current finish block of each fiber

can be done at the runtime level, but is hard if at all possible to do at the user level. It seems to me, at this point, preferable to expose both the general and the restricted version of async-finish.

Parallel-for loops

Parallel-for is an extremely useful construct in practice. For example, to increment all the cells of an array in parallel, it suffices to write:

```
parallel_for 0 (n-1) (fun i ->
   a.(i) <- a.(i) + 1)
```

Parallel-for loops can be encoded using binary fork-join, following the same scheme as for the function `parallel_incr` presented earlier on.

```
let rec parallel_for i j f =
   if j - i <= threshold then begin
      for k = i to j do
         a.(i) <- a.(i) + 1)
   end else begin
      let m = i + (j-i)/2 in
      fork_join (fun () -> parallel_for i m f)
                (fun () -> parallel_for (m+1) j f)
   end
```

One absolutely essential question is what value of `threshold` one should use. If `f` denotes a trivial operation, `threshold` should be in the order of magnitude of 1000. If, however, `f` denotes a large computation, `f` should be just 1. A poor-man's solution would be for `parallel_for` to take the value of `threshold` as argument. This approach is very invasive and requires hard-coding constants in ways that make the performance of the code not portable. Later on, I'll present general solutions to this granularity control problem.

There also exists a way to avoid the granularity control problem specific to parallel-for loops (and to certain classes of tree computations). The idea is that parallel-for loops can also be scheduled using a "lazy splitting" technique. In that approach, the loop begin executing just like a sequential loop. During the execution of that loop, if another core queries for work, the range of remaining loop iterations gets split in two halves, and the upper half is sent to the other core. This lazy splitting scheme can be emulated using the async-finish construct, provided some support for polling for queries.

I'll cover the implementation details of lazy splitting in another section. For most applications, if we assume that granularity control can be properly handled, then the simple encoding of parallel-for based on fork-join suffices.

Priorities and evaluation order

The evaluation of a structured parallel construct such as fork-join, parallel-pair, or async-finish spawns branches. The scheduler has an important decisions to make: among the branches, which one should the current core (i.e., the core that executed the parallel construct) work on first?

For a binary fork-join, it probably makes most sense to execute the left branch before the right branch. This way, a single-core execution of `fork_join f1 f2` is equivalent to `f1(); f2()`, matching the left-to-right reading order.

For a parallel-pair, we may similarly want the execution of a parallel pair to match the evaluation order of a sequential pair. The evaluation of tuples in OCaml is famously unspecified. Even though the evaluation of parallel-order should presumably be just as unspecified, it would be useful in practice to have it implemented in the same order. Thus, for a parallel-pair, currently written `parallel_pair f1 f2`, to match the sequential evaluation order of `(f1(), f2())`, we should have the current core work first on `f2`, the second branch.

For a async-finish construct, there is also a choice to make. At each `async` point, the scheduler has a choice between executing the body of the `async` first, or executing the continuation (that is, the remaining of the finish block) first. An equivalent sequential program would naturally be obtained by simply removing the `async` calls, replacing `async f` with `f()`. To match the evaluation order, the scheduler should get the current core to work on the `async` first.

However, there are applications for which it may be desirable (more efficient) to get the current core to complete its work on the finish block first, and then only start working on the async-ed branches. Depending on the code, terminating the finish block may allow accessing data while it is still in the cache; this data may be gone from the cache after the evaluation of the async-ed branch. Terminating the finish block may also enable freeing auxiliary data earlier.

In summary, I believe that it may be desirable for an implementation of async-finish to allow the user specifying the relative priority between an async branch and its continuation.

Futures

Interface

A "future" captures a computation that can be performed concurrently to the rest of the program. The result of a future can be queried for at any point of the program after the creation of that future.

A future is thus similar to an object of type `Lazy.t`, except that its evaluation may take place concurrently to the execution of the rest of the program. We say that a future is "forced" (or "touched") when its result is queried.

There are several possible variants of futures, but they all share the same programming interface, essentially the same as for `Lazy` thunks.

```
future : (unit -> 'a) -> 'a future
force : 'a future -> 'a
```

There are many possible use cases for futures. Let me illustrate just one of them, for highlighting the notion of speculation. Consider the sequential program, where `t1` and `t2` denote two expensive computations, of similar duration.

```
if t1 then t2 else 0
```

Now, assume that the programmer knows that `t1` returns `true` most of the time. It is possible, using speculative parallelism, to get the program to run twice faster, by rewriting it in the following form.

```
let x = future (fun () -> t2) in
if t1 then force x else 0
```

In this code, the result of `t2` is created as a future, named `x`, which may be executed in parallel of `t1`. If `t1` returns true, the `force` operation either reads the result of the future, or waits for the completion of the future. If `t1` returns false, the result of `x` is discarded. The evaluation of `x` may even be cancelled (though with some care in case I/O computations are involved).

Semantics

The creation of a future is a constant-time, non-blocking operation. The force operation can encounter several cases:

- If the force operation sees that the future has already been evaluated, it may simply read the corresponding result, which was stored in the memory representation of the future.
- If the force operation sees that the evaluation of the future has never begun, it atomically flips a flag to indicate that the execution of the future has begun, and starts evaluating the future.
- If the force operation sees that another thread is currently processing the future, the currently-running thread needs to suspend its current execution. In that case, the scheduler would typically assign to the current core another available task. The suspended task needs to be registered somehow with the future, so that it receives a notification when the result of the future becomes ready.

The semantics of futures admits several variants, depending on the desired priority and degree of strictness in the evaluation of futures. As far as I know, there is no completely standard naming scheme for these variants. I will use the following names:

- A “lazy” future describes a computation whose evaluation must not start before it is first forced.

- A "strict" future describes a computation that will be performed at some point before the end of the program, even if the future is never forced.
- An "eager" future is a strict "future" that is moreover assigned a higher priority of evaluation than the continuation. In other words, as soon as a future is created, its evaluation begins. The continuation may be evaluated in parallel if another core is available to process it.
- A "speculative" future is one whose strictness/laziness is left unspecified: its evaluation may start as soon as the future is created, or it may never take place if the future is never forced.

Speculative futures have the main drawback that whether a future gets executed or not may depend on a random scheduling process. This property makes it a nightmare to debug or profile the program.

Lazy futures usually make a lot of senses in languages with laziness by default. However, in a strict language such as OCaml, using lazy futures only would mean that opportunities for parallelism will be missed. A typical situation would see, on the one hand, a lazy future ready to execute, and, on the other hand, cores sitting idle, just because it is not yet known that the result of the future will eventually be needed. That said, it may be easy for an implementation to provide the option to choose between a lazy future or a strict future at the time of creating a future.

Regarding strict futures, there are essentially two choices: whether they should be eager or not, i.e., whether the body of the future should be executed first by the local core, or whether the continuation should be executed first. As far as I understand, both variants may be interesting, depending on the situation. It really is a scheduling issue, which I'll come back to when discussing work stealing schedulers.

Preemption

The operation system scheduler may decide, at any time, to suspend a (system) thread, resume another thread, or migrate a thread to a different core. On the contrary, if we use our own scheduler for scheduling the execution of fibers, we have full control, and are able to impose different rules. (Typically, such a scheduler would be set up by allocating one (system) thread per available core, each of these threads running an instance of the scheduler that handles the execution of the fibers.)

An important question is whether the scheduler should be "preemptive" or not. The question of preemption may appear to be only related to the scheduler, and not to the semantics of the parallel language. However, it is related to the semantics, because if preemption is possible, then (1) a larger number of instruction interleavings are possible, and (2) the cost semantics can be affected, in particular the peak memory usage.

Depending on the class of programs that one has in mind, preemption can be tempting For example, in Racket, the speculative execution of a future may

begin but then be suspended and resumed later.

However, in general, I would strongly advise against any preemptive scheduler, because it leads to uncontrolled memory usage. Typically, a future may begin by allocating tons of data in its very first instructions. If several such futures are started and not executed to completion, the peak memory usage can be orders of magnitude larger than what could happen in the worst-case execution of a non-preemptive scheduler.

More generally, for programs that perform a lot of arithmetic computations and few memory operations, it does not matter much when computations are evaluated. However, for programs that do allocate or manipulate a significant amount of memory, it does matter very much.

Encodings based on futures

The "future" construct is the most expressive parallel construct. Indeed, futures can encode fork-join, async-finish, and parallel-for loops.

Futures can be used to encode fork-join.

```
let fork_join f1 f2 =
   let b = future f2 in
   f1;
   force f2
```

Futures can be used to encode async-finish. A block-id consists of the address of an atomic reference on a list of futures. Each of these futures in the list correspond to one async-ed branch. At the end of the finish block, we force all the futures from that list.

```
type block_id = ((unit future) list) Atomic.t

let finish f =
   let b : block_id = atomic_ref [] in
   f b; (* evaluate the contents of the finish block *)
   List.iter force !b

let async b f =
   atomic_push b (future f)
```

Other applications of futures

Futures are probably quite useful for programming programs performing I/O and depending on the external world. Besides, some researchers have speculated about the benefits of using future for pipelined computations such as graphics processing. Recent work also argues for the interest of futures of hiding the latency of I/O operations. On this topic, there is a SPAA'16 paper by Stefan Muller and Umut Acar (implemented in parallel SML) and, and work a SIAM'20 paper by Singer et al. (implemented in an extension of Cilk with support for futures).

I haven't personnally written many programs involving futures. It appears to me that most parallel algorithms can be efficiently implemented without futures, using only well-structured parallel constructs. Nevertheless, it looks to me like a good idea to include support for futures. If given sufficient control over how the futures are scheduled, the programmer might be able to put them to good use in specific application domains.

The work from Daniel Spoonhower's PhD thesis show that scheduling policies for futures can be devised, implemented, and proved to deliver good performance, at least under sufficiently-strong assumptions.

Unified semantics for all constructs: the DAG calculus

Motivation

Assume that Multicore OCaml comes with an API including fork-join, parallel-tuples, async-finish (with and/or without labels), and futures. What would be the simplest way to explain to the programmer the semantics of a program that can arbitrarily nest all these constructs?

Technically, the encoding of all constructs in terms of futures does provide a semantics. However, this answer is unsatisfactory for three reasons.

- First, it requires a deep understanding of futures, which we should not assume if we are not advertising the broad use of futures.
- Second, it requires the programmer to understand all these encodings.
- Third, while it provides a model for the semantics, it does not tell the right story with respect to the cost semantics, i.e. it would not suggest where the scheduling overheads come from. It would be more satisfying to present a model for the semantics that can also be used to describe the cost semantics.

With my colleagues, we published at ICFP'16 a paper that precisely addresses the question of providing a simple, unifying semantics. We named our semantics the "DAG calculus". We show that the state of the program during an execution can be represented as a dynamically-constructed graph, where nodes represent tasks/fibers/subcomputations, and where edges represent dependencies. The graph is directed and acyclic, thus is a DAG.

In what follows, I'll summarize at a high-level the main ideas of the DAG calculus. I'll discuss how it can be implemented later on.

The DAG calculus offers a low-level programming API, essentially with a function for dynamically adding nodes and edges. The nodes and edges get consumed as the execution proceeds: if a node completes its execution, it is removed from the DAG, together with its outgoing edges. If a node has zero incoming edges, it is "ready": its execution may begin.

The parallel constructs are encoded in a very simple manner using these basic operations. Our semantics specifies accurately when it is legitimate to add an edge, and when it is not.

Our semantics thus remains given by an encoding, but, compared with encoding based on futures, our presentation is much simpler and matches closely what a realistic, optimized runtime could implement. For example, our C++ library-based implementation of a parallel runtime faithfully follows our encodings.

Note also that the low-level programming API offered by the DAG-calculus may also be directly manipulated by the programmer, to describe more complex parallelism patterns than what can be encoded using any of the parallel constructs presented so far.

Interface

The DAG-calculus operations are:

- `newNode`, adds a node, which captures a piece of computation (represented as a "fiber"). When created, the node is not available for execution until the `release` operation is invoked on that node. This two-step process is necessary to leave the time to set up incoming edges on that node.
- `newEdge`, adds a dependency between two nodes. There are restrictions, for example no cycles must be created, and it is not allowed for the target edge to be a ready task, because its execution might get started concurrently.
- `release` should be invoked on a node to hand it over to the scheduler. After `release` is called, as soon as a node has no incoming edges left, it becomes ready for scheduling.
- `yield` is an operation that suspends the currently running thread (fiber), and updates the contents node associated with that thread with "what remains of the current computation". It is a specific form of a "capture continuation" operation.
- `self` returns a pointer on the currently running thread. This operation is useful to add incoming edges onto the current node, which will then describe a "continuation" after a `yield` operation.

These operations can be type-checked as follows.

```
type node
newNode : (unit -> unit) -> node
newEdge : node -> node -> unit
release : node -> unit
yield : unit -> unit
self : unit -> node
```

Implementation

To DAG calculus can serve as a basis for an efficient implementation. It suffices to extend the API to allow specifying, for each node, the strategy to be employed for representing the set (or number) of incoming edges and the set of outgoing

edges. Indeed, depending on the arity, and on whether the arity is statically known or not, more or less efficient strategies are available.

Moreover, an implementation based on the DAG calculus can accommodate very important optimisations specific to work-stealing executions, such as an optimisation for fork-join when a core discovers, after completing the left branch, that the right branch has not been migrated to another core.

Encoding of parallel constructs in the DAG-calculus

The fork-join construct is encoded as follows.

```
let fork_join f1 f2 =
   let t1 = newNode f1 in
   let t2 = newNode f2 in
   newEdge t1 (self());
   newEdge t2 (self());
   release t1;
   release t2;
   yield()
```

The n-ary fork-join construct follows exactly the same pattern as above, simply adjusting the number of branches.

The parallel-pair construct is similar, except that one has to deal with returned values. The DAG-calculus, which aims at simplicity, does not integrate support for nodes that deliver a return value. Instead, operations encoded in the DAG-calculus are responsible for allocating a cell to store the results of their branches.

The parallel-tuple construct can be encoded using n-ary fork-join, or can be encoded directly as follows (example for arity 2).

```
let parallel_tuple_2 f1 f2 =
   let r1 = ref None in
   let r2 = ref None in
   let t1 = newNode (fun () -> r1 := Some (f1())) in
   let t2 = newNode (fun () -> r2 := Some (f2())) in
   newEdge t1 (self());
   newEdge t2 (self());
   release t1;
   release t2;
   yield();
   (unsome !r1, unsome !r2)
```

The labelled async-finish construct is encoded as follows, where `b` denotes a finish block identifier.

```
let labelled_async b f =
   let t = newNode f in
   newEdge t b;
   release t
```

```
let labelled_finish f =
   let b = self() in
   let t = newNode (fun () -> f b) in
   newEdge t b;
   release t;
   yield()
```

The future construct is encoded as follows.

```
type 'a future = (node * 'a ref)

let future f =
   let r = ref None in
   let t = newNode (fun () -> r := Some (f())) in
   release t;
   (t, r)

let force (t,r) =
   newEdge t (self());
   yield();
   unsome !r
```

A concrete proposal

It seems to me that, to maximize backward and compatibility and accommodate for extensions, it would be best to present all constructions in the form of functions. Only the case of parallel tuples is a bit tricky, because we'd like this construct to work for all arities, with a reasonable syntax and a simple enough type-checking rule.

API for constructs

```
(* Parallel-for *)
parallel_for : int -> int -> (int -> unit) -> unit

(* Fork-join *)
type branch = unit -> unit
fork_join_2 : branch -> branch -> unit
fork_join_3 : branch -> branch -> branch -> unit
fork_join_4 : branch -> branch -> branch -> branch -> unit
fork_join_list : branch list -> unit

(* Parallel-pairs, without ad-hoc syntax;
   unclear how to typecheck higher arities. *)
parallel_pair : (unit -> 'a) -> (unit -> 'b) -> ('a, 'b)
```

```
(* Async-finish *)
finish : (unit -> unit) -> unit
async  : (unit -> unit) -> unit

(* Labelled async-finish *)
type block_id
labelled_finish : (block_id -> unit) -> unit
labelled_async  : block_id -> (unit -> unit) -> unit

(* Futures *)
future : (unit -> 'a) -> 'a future
force : 'a future -> 'a

(* DAG-calculus *)
type node
newNode : (unit -> unit) -> node
newEdge : node -> node -> unit
release : node -> unit
yield : unit -> unit
self : unit -> node
```

Syntax for parallel-tuples

Let me talk about syntax—breaking the golden rule, for just a few lines, to point out that it is not completely obvious what is the best way to deal with parallel tuples or arbitrary arities.

Solution 1: don't use any ad-hoc syntax, just extend somehow the typechecker (or use GADTs?) to deal with the list of arguments.

```
let (x,y,z) = Parallel.tuple [
                (fun () -> f a);
                (fun () -> g b);
                (fun () -> h c)]
```

The main drawback of this approach is that it is super heavy as syntax. It's basically as bad as what one needs to write in C++ (without macros).

```
parallel::tuple([&]{ f(a); };
                [&]{ f(b); };
                [&]{ f(c); };);
```

Solution 2: introduce a dedicated syntax. To indicate that `f a` and `g b` can be evaluated in parallel, one could write:

```
let (x,y,z) = (| f a, g b, h c |)
```

The first drawback of this approach is that it requires modifying the syntax in a nontrivial way. The second drawback of this approach is that it does not allow for providing optional arguments that are useful for specifying scheduling

parameters related, e.g., to the choice of a particular join-resolution strategy, or to granularity control.

Solution 3: use binding extensions. Gabriel Scherer pointed out the possibility to write:

```
let| x = f a
and| y = g b
and| z = h c
```

This solution has the same problem in terms of providing optional scheduling options to the parallel-tuple operation.

Solution 4: introduce `parallel` as a function to be put in front of the conventional tuple syntax, or in front of a list of arguments, or in front of curried arguments; and introduce in OCaml a way to make function arguments "lazy by default", that is, with an implicit `fun () -> ...`

```
let (x,y,z) = Parallel.tuple (f a, g b, h c)
let (x,y,z) = Parallel.tuple [f a; g b; h c]
let (x,y,z) = Parallel.tuple (f a) (g b) (h c)
```

One drawback of this approach is that it the implicit `fun () -> ..` might be found confusing.

Refinements to come

I'll later refine the interface by adding a few optional arguments to allow tuning implementation details.

For example, for nodes it is useful to specify which data structures should be used to represent the edges. The `instrategy` provides support for representing incoming edges, while the `outstrategy` provides support for representing outgoing edges.

```
newNode : ?instrategy:instrategy -> ?outstrategy:outstrategy ->
          (unit -> unit) -> node
```

As another example, it will be useful to equip parallel-for loop with information helpful for granularity control. Concretely, a function that gives the asymptotic cost of a range of iterations is useful.

```
parallel_for : ?complexity:(int -> int -> int) ->
               int -> int -> (int -> unit) -> unit
```

Semantics of exceptions for parallel constructs

Let's forget for now about unstructured concurrent programming involving futures, and consider first a strongly structured parallel programming pattern such as fork-join (or parallel-tuples), async-finish, and parallel-for. If one or

several branches throw exceptions, how should these exceptions be handled? How do they interact with the execution of the other branches?

Let's first take a step back. In sequential programming, there are (at least) two important applications for exceptions:

1. exceptions for tracing errors,
2. exceptions for terminating algorithms early.

Regarding the tracing of errors, the desirable property for debugging a parallel program are determinacy and reproducibility. If the randomness of scheduling decisions affect which exception comes back to you, then you are in deep trouble, for sure. To ensure reproducibility in the presence of randomized scheduling and branches performing arbitrary side-effects, it is necessary to execute all the branches until completion, even if one of the branches is detected to raise an exception. (Moreover, beyond reproducibility concerns, abruptly killing running threads is generally a bad idea for programs that interact with the outside world.)

There are two obvious possibilities for dealing with exceptions in parallel branching constructs:

1. Execute all branches until the end; if one or more branches throw an exception, combine the exceptions from all branches and return a single exception carrying the list of those exceptions (or, more generally, use some kind of custom function for combining exceptions into one).
2. Exploit the fact there exists an implicit order on the branches, and propagate only the exception raised by the "first" branch that raises an exception, with respect to that order. This allows, potentially, for killing (cancelling) certain branches early, if a prior branch already raised an exception.

Both behaviors seem acceptable to me.

The solution (1) is that implemented in the X10 language, which offers async-finish as main parallel construct (See the paragraph "The rooted exception model" from the introduction to X10.

There are, in my opinion, two key benefits to the solution (2). First, it does not require introducing additional tooling for handling "an exception that bundles a list of exceptions". Second, it means that an execution of the parallel program using several cores always raises exactly the same exception as an execution of that same program using a single core, and interpreting the parallel constructs as sequential composition—i.e. executing the "sequential elision" of the parallel program.

The property that a parallel program should deliver the same results as its sequential elision is particularly desirable because several approaches to granularity control rely on this assumption. We'll come back to that point.

Let's come back to the question of whether it is desirable or feasible to cancel the execution of a branch whose execution has already started and whose result will certainly not be needed. As far as I know, both the Cilk project and the

Manticore project provided some amount of support for cancellable async-ed tasks. However, the implementation details are quite complex, because the task to cancel may have spawned numerous subtasks, and these subtasks may be scattered around numerous cores. Moreover, note that a given subtask may get "cancelled" as a result of any of its ancestor fork-join operations having a left branch raising an exception. Thus, a basic polling mechanism may be insufficient for detecting cancellation.

If you are nevertheless interested in understanding what it takes to support cancellable tasks, you cna find details in the description of the implementations of cancellable exceptions in Manticore and of exceptions in JCilk. (JCilk was a prototype extension of Java with support for the Cilk constructs.)

Support for early termination in parallel algorithms

Let's take a step back. I wrote that exceptions have two important applications: for tracing errors, and for terminating algorithms early. For tracing errors, it is not important to cancel branches early, one can presumably afford to wait until completion of all the parallel branches. Regarding early termination of algorithms such as in branch-and-bound algorithms, I'll argue in what follows that it is not hard to implement task cancellation via polling.

Recall the program that performs a parallel search to count the number of solutions of a puzzle. Assume now that we are only interested in finding one solution, and want to interrupt the parallel search as soon as have found one solution.

This early termination pattern can be implemented easily by having all parallel branches of the search polling on a shared cell, whose contents indicate whether a solution has been found. Note that, although this shared cell is accessed concurrently by several threads, it is safe to read and update it using non-atomic operations.

```
let found = ref false

let _ =
   finish (fun () ->
      let rec search config =
         if !found then
            () (* branch is interrupted *)
         else if problem_solved config then
            found := true
         else begin
            let parallel_search config2 =
               async (fun () -> search config2) in
            List.iter parallel_search (get_transitions config)
```

```
      end in
    search initial_config)
```

This programming pattern can be captured through an API that generalizes the labelled async-finish as follows. The cell used for polling is handled by the library. A function `labelled_interrupt` updates this cell to mark the finish block as interrupted. A function `labelled_interrupted` should be used in the client code to check whether the branch should continue its execution. An additional function `labelled_check_interrupt` is included; it is explained further on.

```
(* Interruptible labelled async-finish *)
type block_id
labelled_finish      : (block_id -> unit) -> unit
labelled_async       : block_id -> (unit -> unit) -> unit
labelled_interrupt   : block_id -> unit
labelled_interrupted : block_id -> bool
labelled_check_interrupt : block_id -> unit
```

The fact that polling takes place only at specific places is both a weakness and a strength. It is a weakness because it might take some time for a branch to be interrupted if it called another time-consuming function. At the same time, it is a strength because the programmer controls the point of interruption, and is able, e.g., to cleanly close ongoing I/O operations.

Using the above interface, the code for the interruptible puzzle resolution can be rewritten as follows.

```
let _ =
  labelled_finish (fun b ->
    let rec search config =
      if labelled_interrupted b then
        () (* interrupt the branch *)
      else if problem_solved config then begin
        print config;
        labelled_interrupt b (* interrupt the finish *)
      end else begin
        let parallel_search config2 =
          labelled_async b (fun () -> search config2) in
        List.iter parallel_search (get_transitions config)
      end in
    search initial_config)
```

For more complex programs, it might be convenient for the user to kill the current branch by raising an exception when `labelled_interrupted` returns true. For that purpose, we extend the interface with a function call `labelled_check_interrupted`, whose behavior is to invoke `labelled_interrupted` and raise a specific exception if the result is true. This exception is caught and ignored by the finish block.

Our example can be rewritten by writing `labelled_check_interrupt b` at the entry point of the `search` function. Additionnally, if we would like to be able to interrupt the branch during the iteration over the neighbor configurations, we can add another instance of this polling operation inside the `parallel_search` function. The resulting code is:

```
let _ =
   labelled_finish (fun b ->
      let rec search config =
         labelled_check_interrupt b;
         if problem_solved config then begin
            print config;
            labelled_interrupt b (* interrupt the finish *)
         end else begin
            let parallel_search config2 =
               labelled_check_interrupt b;
               labelled_async b (fun () -> search config2) in
            List.iter parallel_search (get_transitions config)
         end in
      search initial_config)
```

Considering the parallel algorithms concerned by "early termination" that I know of, I would tend to think that it is sufficient to offer support for interruptible branches only for async-finish. For introducing interruptible branches in a code written using fork-join or parallel-pairs, there are two easy way out: either use the async-finish encoding for these constructs, or manually introduce a cell for polling on termination.

Overall, I speculate that it is not needed to bother tackling the challenge of implementing support for cancellable tasks in the parallel runtime, because for most applications this can be done easily in the user code. Maybe it will appear useful to support cancellable tasks in the future, but at the moment support for cancellable tasks does not appear to me as a top-priority.

Semantics of exceptions for futures

There are not so many choices for the semantics of exceptions in futures.

If we mimic the semantics of exceptions for lazy thunks, the rule would be: if the evaluation of the future raises an exception, then this same exception is raised upon every force operation on that future.

There is a big issue, however. What if a future is executed, its execution raises an exception, but the future is never forced? What should happen to that exception?

Raising the exception at top-level out of any context would be problematic. It means that scheduling decisions may affect whether a program raises an

exception or not. Such lack of determinacy makes debugging intractable.

Silently dropping the exception without any feedback to the user would also be quite problematic. Usually, it means that the code contains a bug, and that this bug was not revealed only because the result of the future turned out to not be needed to compute the final result for the particular input data considered, although it might have been needed for a slightly different input.

The only reasonable way out that comes to my mind is to ensure that this situation never happens. In other words, a future that is executed and that may raise an exception should be forced at least once.

If we follow this rule, then it leaves us with three acceptable programming patterns for futures.

1. Lazy futures are always acceptable, because they are never executed before they are forced.
2. Strict futures are acceptable if the programmer is certain that the future will be forced at least once.
3. Strict futures are acceptable if the programmer is certain that the future will never raise an exception (other than out-of-memory).

Of course, the notion of "being certain" is quite questionable. Many programmers are certain that their code is correct until proven otherwise! Could a static analysis ensure that a future is enforced at least once, for many useful programming patterns?

Exceptions in the presence of non-termination

One last observation I'd like to point out regarding exceptions is that the order of evaluation of the branches, and the property of whether branches can be cancelled while they execute, both affect whether a program reports an exception or whether it diverges. Consider for example the (minimalistic) example program:

```
finish (fun () ->
   let rec f () =
      async f;
      async (fun () -> raise Not_found);
      f()
      in
   f())
```

This program spawns infinitely-many branches, half of which diverge and half of which raise an exception. Depending on the details of the scheduler, this program would either diverge or raise an exception.

I believe that this situation is somehow inevitable. It seems hard to define a deterministic semantics for a parallel program that diverges, because doing so

would require some form of preemption to ensure fairness among all the threads, and I have argued earlier on for the drawbacks preemption.

Semantics of exceptions in the DAG calculus

The DAG-calculus paper left exceptions to future work. Whatever semantics we settle on for parallel constructs and for futures, it would be highly desirable in my opinion to describe it as a (possibly-trivial) extension of the DAG-calculus.

Acknowledgements

I'd like to thank François Pottier (Inria), Mike Rainey (CMU), Guillaume Ryder (Google), and Gabriel Scherer (Inria) for their feedback on drafts of this material.