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

Arthur Charguéraud (Inria), 2020

Addentum to Part 1: Semantics of exceptions

Published: 2020-05-25

I have been convinced by arguments that proper support for exceptions would be needed for the first implementation of the DAG-calculus. I sketch below what could be an extension of the DAG-calculus with customizable support for exception handling.

Unlike the rest of the material, I am here describing "ideas of the week-end", as opposed to fully worked-out, peer-reviewed, research results. Inevitably, there will remain a few rough edges.

Quick recap on the DAG-calculus without exceptions

In the DAG-calculus, a node describes a piece of computation. Initially, the DAG consists of a single node, which describes the entry point of the program. During the execution, new nodes are created dynamically. The original entry point then becomes the "sink" node, at the bottom of the DAG, and describes the final exit point of the program. Typically, a fork-join operation introduces to fresh nodes, and updates the current node to represent the continuation, i.e. the computation that comes after the fork-join.

An edge from A to B indicates that node A must complete before node B is allowed to start. When a node completes, it is removed from the DAG, together will all its associated edges. After the removal of these edges, other nodes that previously had dependencies (incoming edges) may become "ready" (zero incomming edges).

Every node features an "instrategy", which controls the representation of the incoming edges, and an "outstrategy", which controls the representation of the outgoing edges. When a node A completes, its outstrategy takes care of notifying the target of the outgoing edges of the removal of the edges. For example, for an edge from node A to node B, the instrategy of node B has is notified. Note that, depending on the in- and out- arity of the nodes, the methods associated with in- and out-strategies may need to handle queries concurrently.

Extension of the DAG-calculus with exceptions, first without futures

In the original presentation of the DAG-calculus, the only information carried out by the edges is whether a node has terminated. The result value produced by a computation can be transfered via the shared memory. In a language like ML, furthermore extended with exceptions, it makes sense for edges to carry the result produced by a node: either a value, or an exception.

The API for in- and out-strategies (which I haven't detailed so far), could be refined for handling return values and exceptions. More precisely, when an edge from A to B is removed, the instrategy of the node B receives the result of node A. If this result is an exception, the instrategy can follow different policies for how to handle the exception. This policy may be chosen on a per-node basis.

I see 3 useful patterns:

1. The policy that combines exceptions: if one edge or more carries an exception, then, when all incoming edges are removed (i.e. all branches have completed), propagate an exception that consists of the list of the exceptions gathered.
2. The policy that respects sequential execution order: collect results and exceptions from the incoming edges; if the resulting tuple is of the form `(v1, ..., vi, exn1, _, _, _)`, where the underscore indicate either a value, or an exception, or no result obtained yet, then propagates the exception `exn1`.
3. The policy that allows for eager propagation of exceptions: as soon as one edge carries an exception, propagate this exception. This policy is non-deterministic, but allows for debugging without waiting. If non-determinacy is an issue for debugging, it is always possible for the programmer to switch to another policy, presumably by means of setting the right flag in his code.

Note that, with policy 3, if a node raises an exception and that this exception is not caught, then the program would immediately terminate on that exception, without noticeable delay (just the time required to walk down a chain of edges).

Let me first clarify what it means to "propagate an exception". If the instrategy of a node decides to propagate an exception `e`, it essentially means that the contents of the computation associated with that node will be patched with a leading `raise e`. In other words, the exception will be raised in the stack associated with the computation of that node. This allows for semantically-enclosing exceptions handler to catch the exception.

It now remains to explain what happens to the branches that are cancelled, i.e. that have not yet terminated when an exception is propagated from another branch (for policies 2 and 3).

Cancelling the execution of disconnected nodes

If the instrategy associated with a node decides to propagate an exception before waiting for completion of all incoming edges (branches), its action over the DAG consists of effectively removing the remaining incoming edges into the node. Thereby, a sub-DAG gets disconnected from the rest of the DAG, that is, nodes no longer have a path reaching the sink. The results associated with the disconnected nodes are no longer needed.

If computations were pure, all these nodes could be discarded at once, and those currently running could be interrupted abruptly. However, in a world of impure computations, cancelling computations at arbitrary points is certainly not a good idea. The post on Trio makes that point too.

For cancelling computations, we need to somehow notify the nodes that they are no longer needed, and let them handle this notification on their own.

- If a node has not yet started, the scheduler may test before starting its execution whether the node is marked as cancelled. If so, it may invoke a custom `finalize` method that the programmer may have registered. For finalize methods to be executed in the right order, it seems necessary to follow the DAG order, that is, to invoke `finalize` only on ready tasks.
- If a node is currently running, it seems safe to raise the `Cancel` exception only at checkpoints that the programmer has marked explicitly in its code, via the `checkpoint` instruction.

For many parallel algorithms, the node from the DAG are set up in such a way that their execution time is never too long before they complete or invoke `yield`. For such nodes, it is perfectly acceptable to not attempt cancelling the execution of the node while it is executing. In other words, it suffices for cancellation to be checked before starting a node, and the `checkpoint` instructions do not need to appear anywhere explicitly in the code.

On the contrary, for programs that may feature nodes that involve long sequential execution, placing the `checkpoint` instructions can be quite tricky, especially if the code calls into existing sequential library. Dealing with this situation is certainly going to be challenging. A number of libraries could be identified as "safe to interrupt any time". In that case, the polling on cancellation could be handled by the runtime system. Not all libraries, however, would satisfy this property.

Support for futures

The problem that I discussed with futures is that it is not always clear where to propagate the exceptions that they may raise. Another important problem, at least for languages without a garbage collector, is that it is not clear at which

point a future is no longer needed, and thus can be removed from the DAG (in the model) and deallocated (in the implementation).

We can address both of these problems through the introduction of "shadow edges". A shadow edge relates a node A that describes a future to a node B, typically a node associated with a "finish" block (or a "nursery" in Trio's vocabulary). The interpretation of such an edge is double:

1. If the execution of the future raises an exception, then this exception is delivered to the instrategy of node B. This instrategy should have a policy able to handle such exceptions, and to prioritize its propagation with respect to that of other exceptions within the same scope.
2. If all proper incoming edges to node B have completed (i.e. if only incoming shadow edges remains on B), then the shadow edges incoming into B are removed. At this point, the node B becomes ready to execute. The future A gets disconnected from the DAG. This future will never be forced in the future, and it can be removed from the DAG. If it is running, its execution may be cancelled as explained above.

Note that there are a number of well-scoping rules (to be made precise) that the code should satisfy for things to work out smoothly. In particular, a future must not be forced outside of its scope.

Overall, the extension of the DAG-calculus that I'm sketching maintains the key property that the semantics can be explained independently of the scheduler. In particular, it is not specified how fast nodes are cancelled. This leaves room for many possible scheduler implementations.

Implementation of cancellation

For implementation the DAG-calculus without support for exceptions, it was sufficient to store, in out-strategies, pointers on instrategies. To support cancellable nodes, it seems to me that for an efficient implementation one would need backward pointers, following the edges backwards. Cancellation of branches would be implemented by a reverse DFS traversal of the graph. An implementation faces a number of complications due to potential races between the forward traversal of outgoing edges after nodes complete, and the backward traversal of edges for cancellation. Interesting research work ahead!

Tempting but tricky: exceptions for early algorithmic termination

Consider the function `array_any_index_of x a`, which searches for the index of any occurence of `x` in the array `a`. The code is implemented using the `parallel_for` construct, using a function that raises an exception as soon

as one occurence is found. The annotation `exn:eager` specifies the "eager propatation of exception" (policy 3, above).

```
exception Occurence of int

let array_any_index_of x a =
   let n = Array.length a in
   try
      parallel_for ~exn:eager 0 (n - 1) (fun i ->
         if a.(i) = x then raise (Occurence i));
      None
   with Occurence i -> Some i
```

There are three important aspects that the programmer must be aware of:

1. Unlike for sequential code, the occurence reported isn't necessarily the first one. For that, one would need to follow another policy for handling exceptions, e.g. writing `~exn:sequential`. But doing so means that in many cases, a large fraction of the array needs to be traversed even when an occurence is found early.
2. Unlike for sequential code, there is no guarantee that the function terminates faster than O(n), even if the array contains `x` in a cell near the front of the array. The randomness in the scheduler may very well lead to this segment of the array being processed last. In fact, depending on the scheduler, even if an occurence is found early, the delay associated with task cancellation may result in the entire array being traversed nevertheless.
3. Last, and most importantly, such an introduction of "speculative parallelism" like in the example considered is generally counterproductive for performance. Of course, if all the cores are free, then the above `parallel_for` is the only mean of providing these cores with some work. However, assume this code to be executed in a context that already spawned many parallel subtasks, and assume that the array contains an occurence, say at index "n/2 - 1". In a work stealing execution, if another core acquire some of the work involved, it is likely that all the cells of the array will be traversed. Yet, a sequential execution would have traversed only half on the array. Thus, the parallel execution will be crippled by the fact that it performs unnecessary work.

What the third point suggests is that speculative parallelism is not always a win. One would need some means of expressing priority: exploit parallelism from this specific parallel-for construct only when the scheduler has no other ready node to work on. Yet, this kind of rules generally is hard to exploit for at least two reasons:

- It is not easy for decentralized schedulers to handle this kind of priorities, because they do not have a global view of what nodes remain to be executed.
- Any policy based on tests such as "if there is no other ready node available at that point in time" can be defeated when another running task suddently

starts spawning a large number of parallel nodes (with higher priority).