Isolates, Futures and the Parallel Operator

Dyalog'12 Version

Introduction

This document proposes three new features which aim to make it easy for APL users to express the existence of potential parallelism within individual APL expressions or larger blocks of code. The main focus is on "coarse-grained" or "task" parallelism, where an application contains expressions which will consume enough CPU time for it to be worthwhile to spend a small amount of time "spinning up" separate processes to run them, and subsequently collecting a set of results. The issue of "fine-grained" or "data" parallelism at the level of the APL primitives themselves is a separate problem which requires a different set of solutions (most likely involving compilation).

The Problem

Current Dyalog APL is unable to take advantage of the inherent parallelism in operators like *each* or *outer product*, which invoke the operand function more than once. Dyalog APL does provide a "spawn" operator (&), which launches a function in a new "APL thread". However, the APL interpreter itself only executes APL statements using a single C thread, awarding time slices to the APL threads in turn. This "pseudo" multi-threading is an important capability, allowing applications (especially servers) to remain responsive while executing long-running tasks. However, except where APL threads make asynchronous calls to external DLLs, including COM or Microsoft.NET assemblies, the current APL threading does not allow an application to easily make use of parallel cores.

The "obvious" solution to this problem is to enhance the APL interpreter and memory manager to make them "thread safe", allowing APL users access to true multi-threading. However, adding multi-threading to code which was not designed for it is not an attractive proposition; the resulting bug curve would probably be unacceptable and the functionality of the interpreter with respect to existing threading support might change in subtle ways.

A complete rewrite of the interpreter might be a better choice than adding threading to what we have today, but it would probably take a very long time before we have an interpreter which has the same performance and reliability - and behavioral changes would be even harder to avoid. Given the memory usage pattern of an APL interpreter, it seems likely that the single-threaded performance would suffer, if all memory accesses require some kind of locking due to several threads operating on the same heap.

Isolates (previously known as "Asynchronous Namespaces")

Some years ago, the idea emerged that we could implement "asynchronous namespaces", each managed by a separate interpreter process but connected together in ways which would allow APL expressions to make calls across process boundaries using the existing "dot" notation. The existing "PEACH" (Parallel Each) tool is based on a model of this, implemented in APL using TCP sockets for the remote procedure calls.

For years, we thought of this as "the best we could do" in terms of enabling parallel programming in APL. The good news is that a number of (relatively) recent arrivals on the programming language scene have

elected to take a similar approach – presumably without having the same challenges caused by a 30-year-old core implementation. Several language designers (and users) are now making the claim that concurrent applications are more easily and reliably implemented based on message passing between "actors", which operate in completely separate scopes. The name "actor" is used by the Erlang and Scala communities; Google's new programming language (DART) uses the name "isolates" to refer to similar constructs, and since this name feels like a better fit for the proposed functionality, we will use it to refer to "asynchronous namespaces" in the rest of this document.

Except when marshaling calls between each other, or working on shared resources like locked files, Isolates operate completely independently of each other, executing on separate operating system threads.

Futures

Although actors and isolates are fundamentally asynchronous, the languages which provide them as native constructs - and a few other languages including C# - also tend to make it possible for a request to result in a "future". A future is a reference to a result which will be manufactured later. Futures provide synchronization in a very simple way; a thread which actually needs the computed value only has to refer to it in order to be suspended until the value becomes available.

In Dyalog APL, it is proposed that any item of an array can be a future. This idea is inspired by "concurrent collections" and was brought to our attention by Aaron Hsu of Indiana University, who is doing PhD work on this subject and has been looking at using APL as a model for describing parallel processing – partly funded by Dyalog Ltd. "Structural" operations such as reshape, and other functions that do not actually use the "value" of an item, can operate on arrays containing futures without being suspended. Mathematical functions like + or × will block if the argument values are not available.

Parallel Operator

A mechanism to create isolates, coupled with the ability to issue function calls and receive futures, is sufficient to support "heavyweight" isolates, which might be expected to persist for some time and process more than one "message". For "lightweight" isolates, which are only required in order to provide an execution environment for a single function call, it will be useful to have an operator which is similar to the existing spawn, but which manufactures isolates as required and immediately returns an array of futures.

Language Extensions and Examples

Creating Isolates

A monadic function "isolate" is proposed, denoted by the symbol ⌑ (which is supposed to suggest an isolated world). The monadic definition of ⌑ is similar to monadic ⎕NS, except that the result is an isolate rather than a normal namespace (in other words, the right argument can be a list of names or a namespace to be cloned).

Dyadic ⌑ could be defined as `{⎕NEW ⍺ ⍵}`, allowing the creation of an isolated instance of a class. If so, monadic ⌑ can be viewed as having a default left argument, which is the built-in class "dynamic" (in modern terminology, Dyalog namespaces are "dynamic classes") – with the right argument being the constructor argument for this class.

As with ⎕NS and ⎕NEW, the result of ⍠ is a reference, which allows the use of dot notation to refer to names within the space – and also to execute expressions in the context of the space (resulting in futures).

Calling Back from an Isolate

As is the case for existing namespaces, the name **##** can be used within an isolate to refer to the parent space. This will also hold when one isolate creates another. **#** will refer to the mother of all isolates in the current process. These prefixes can be used to refer to names - and execute expressions - in parent spaces.

Isolate Synchronization

For Isolates that don't actually want to be completely isolated, and have side-effects that they do need to co-ordinate with other threads, it would be good if the existing `:Hold` mechanism could be extended so that it works across all the isolates which have the same original ancestor. If this is not possible without impacting the existing uses of `:Hold` too much, a new synchronization mechanism will be required.

A related question is whether futures can be passed across Isolate boundaries. We may need to insist that an attempt to pass a future from one isolate to the next requires blocking to wait for its value – at least in the first release(s).

The Parallel Operator

The symbol ⫴ is proposed for the parallel operator. Parallel is similar to the existing spawn (&) operator, with the following differences:

- ⫴ creates isolates as required, and initiates asynchronous function calls within them (the operand function and argument values are copied into the isolate).
- Where functions derived using spawn (&) return a thread number, functions derived using parallel (⫴) return futures (in both cases, the result is returned immediately).

Imagine that we wanted to distribute the work involved in computing `+/{+/⍳⍵}¨⍳100` across multiple cores. The parallel operator allows us to distribute the 100 invocations of the inner function across a number of separate threads[1] as follows:

```
      +/{+/⍳⍵}⫴¨⍳100
```

Of course, the above expression will wait until all 100 results of the function invocations have been returned before it can compute the final `+/`. However, we ***could*** use the array of 100 futures returned by the parallel derived function to request further asynchronous distribution of the work, so that the next level of aggregation can start as groups of results become available:

```
      sums←{+/⍳⍵}⫴¨⍳100              ⍝ returns array containing 100 futures
      partitions←(100⍴25↑1)⊂sums     ⍝ 4 partitions of 25 futures
      +/+/⫴¨partitions               ⍝ Sum each group in separate thread
171700
```

[1] It is expected that the implementation will use a configurable pool of threads, and not actually create 100 O/S threads in this example. Efficient use of this pool will be one of the biggest challenges in the implementation.

This works because the partitioned enclose (⊂) in the 2nd expression above, being a "structural" function, does not need the result values; it can operate on the structure immediately. However, the four +/ that are applied to each partition will need the actual values; each one of these four threads will be suspended until the 25 values that it needs have been produced. The final +/ receives four futures and waits until all four partitions have been aggregated. The three expressions above potentially make use of (100+4+1) parallel or overlapping threads to compute the final result.

If we were to write the above using explicit locks, we would need to add statements to create, trigger and synchronize on the locks. When using arrays of futures returned by a parallel operator, minimal changes are required to the code. We can easily continue to reason about an algorithm which has parallel sections, because it has the same definition if we simply remove all uses of ‖. The number of threads that the interpreter actually uses and the degree of concurrency that is achieved only affects the speed – not the number of times that any line of APL code is executed, or the final result. Of course, the interpreter will be managing locks for us, but this is hidden from the users view.

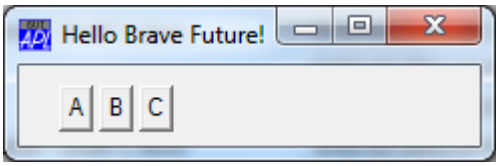
It will be possible to use the parallel operator and futures to dispatch functions which DO have side-effects (and hopefully use holds / locks to manage them). Futures will still provide simplification, where they are used.

Explicitly Created Futures[2]

Although it is expected that the vast majority of futures will be created by the parallel operator, futures are a generally useful synchronization mechanism for applications. An application should be able to create an array of futures, launch tasks which will asynchronously populate the array, and suspend until all the results are available.

The symbol ⍥ - symbolizing a target into which a result will subsequently be inserted – is proposed for the function "futures". The right argument could be a numeric vector giving the shape of the resulting array of futures (similar to the left argument of dyadic ⍴).The following example illustrates how an array of futures can be used to allow a couple of "old style" APL threads to wait for selected combinations of GUI events):

```
PRESSED←⍥ 3                          ⍝ 3 buttons to press
ButtonPress←{PRESSED[⍺]←1}
'F' ⎕WC'Form' 'Hello Brave Future!'
                    ('Size' (40 230))('Coord' 'Pixel')
{i←⎕Aɩω ⋄ ('F.',⍵) ⎕WC 'Button' ⍵
          (10,20×i) ('Event' 'Select' 'ButtonPress' i)}¨'ABC'
```




```
{Z←∧/PRESSED[1] ⋄ ⎕←'A WAS PRESSED'}&⍬
{Z←∧/PRESSED ⋄ ⎕←'ALL BUTTONS HAVE BEEN PRESSED'}&⍬
```

[2] Although sometimes explicitly created, the futures discussed here are all "implicit" futures in the terminology normally used when discussing futures: http://en.wikipedia.org/wiki/Future_%28programming%29

The callback function on each button replaces the corresponding future by a real result. If we now click on buttons A, B, and C in turn, we will see the message “A WAS PRESSED” displayed in the session as soon as A has been clicked, and the “ALL BUTTONS” message when all have been clicked.

Futures are references: Futures act as references until values are assigned. In other words, if `A` is an array containing a future, then `B←A` will mean that the use of (the values contained within) either `A` or `B` will result in suspension until the value has been produced. When a value is assigned, it will affect all arrays containing a reference to that future.

Selective assignment required: The assignment of values to elements of future arrays must be performed using indexed assignment or some other form of selective specification. If a name containing future elements is re-assigned in entirety, the references to any futures contained within the array will simply be discarded (which is the normal behavior for any non-selective assignment). A scalar future will need to be updated using an expression like (`A[⊂⍬]←…`) or (`(1/A)←…`).

“Managing” futures: Although the design of futures is specifically intended to make them behave like “magic” elements of arrays that can be passed around in code which is distributing the responsibility for waiting, and block when a value is required (“implicit” futures), there will be situations where code will want to know which values are now available, or cancel work that has been started. One could imagine an I-Beam or system function (similar to the old `⎕SVC/⎕SVQ` functions for managing shared variables) which, when passed an array containing futures could query or modify the state of the futures contained within. Alternatively, a function could be provided which wrapped an implicit future in an object (a so-called “explicit” future), which has properties and methods which provide control.

Linking Futures and Isolates

It may be useful to be able to register a link between a future and the APL thread or isolate which is expected to produce the result. The parallel operator will naturally make this kind of association automatically. Dyadic ⍝ with a right argument containing futures and a left argument containing corresponding thread IDs or Isolate references could be used to make an explicit association, either on or following the creation of the futures.

The existence of such links would allow the interpreter to signal an error to a waiting process, if a thread or isolate that it is waiting for is terminated. Likewise, an Isolate working on a result which nobody is waiting for any longer could be notified of this fact. In the latter case, it should possibly be up to the code in the Isolate to register the desire to be notified (via a trappable interrupt or other mechanisms), if there is nobody listening. In the case where an Isolate is terminated, it is possibly most user-friendly to signal an error in the listening code by default.

Parallel Dot

In Dyalog APL, an expression of the form (namespacearray.expression), where an array of references to namespaces appears to the left of a “dot”, executes the expression in the context of each namespace, producing a result with the same shape as the array of namespaces.

If any elements of the array on the left are isolates, the corresponding expressions will run asynchronously; the corresponding elements of the result will initially be futures, which will be filled in as the expressions complete.

```
      NSS←⍎¨100⍴⊂⍵    ⍝ Create 100 isolates
      NSS.N←⍳100      ⍝ Assign N to 1 in the first … 100 in the last
      +/NSS.(+/⍳N)    ⍝ The same computation as before
171700
```

The explicit creation of isolates and the use of the dot is appropriate when isolates are expected to maintain state and be used for repeated calls. Above, each isolate maintains its own value of the variable N, which is used in subsequent expressions.

Summary

Isolates, Futures and the Parallel operator provide a set of features that make it simple for APL users to take advantage of multiple cores. In particular, the parallel operator provides a simple mechanism for the APL users to provide information which is difficult for the interpreter to infer: the points in an application where 1) parallel execution is safe (there are no side-effects – or if there are, suitable steps have been taken to perform synchronization) and 2) the overhead of executing sub-expressions in different threads is justified.

Most importantly, the "futures" returned by the parallel operator allow the parallelization of algorithms without requiring the insertion of explicit synchronization code using a locking mechanism. This means that parallelization is not only simple, it is safe: There is no risk of introducing bugs through parallelization (so long as the functions executed in parallel do not have unmanaged side-effects).

The features are similar in philosophy to those recommended by designers of modern languages who have multi-processing as a particular focus, such as Erlang, Scala and DART.

As mentioned in the introduction, the issue of detecting fine-grained parallelism at the level of APL primitives, and distributing the work of chains of primitives across highly parallel hardware, is a separate topic. The solution to this problem probably requires sophisticated data flow analysis and compiler technology.

References

http://en.wikipedia.org/wiki/Actor_model#Programming_with_Actors
http://en.wikipedia.org/wiki/Erlang_%28programming_language%29
http://en.wikipedia.org/wiki/Future_%28programming%29
http://ruben.savanne.be/articles/concurrency-in-erlang-scala
http://www.grobmeier.de/dart-isolates-08112011.html
http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/