The tool of thought for expert programming

|  |
| --- |
| pdfgreyline |

Futures and Isolates  
  
or  
  
Experimental Parallel Language Features in Dyalog APL Version 14.0

Dyalog Limited

Minchens Court, Minchens Lane  
Bramley, Hampshire  
RG26 5BH  
United Kingdom

tel: +44(0)1256 830030

fax: +44 (0)1256 830031  
email: [support@dyalog.com](mailto:support@dyalog.com)  
http://www.dyalog.com

Dyalog is a trademark of Dyalog Limited  
Copyright © 1982-2014  


Dyalog is a trademark of Dyalog Limited  
Copyright © 1982 - 2014 by Dyalog Limited.  
All rights reserved.

Version 14.0

Revision: 20140311\_140

No part of this publication may be reproduced in any form by any means without the prior written permission of Dyalog Limited, Minchens Court, Minchens Lane, Bramley, Hampshire, RG26 5BH, United Kingdom.

Dyalog Limited makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Dyalog Limited reserves the right to revise this publication without notification.

UNIX is a registered trademark of The Open Group.  
  
All other trademarks and copyrights are acknowledged.

Contents

[1 About This Document 1](#_Toc385366870)

[1.1 Audience 1](#_Toc385366871)

[2 Philosophical Preamble 2](#_Toc385366872)

[2.1 Parallel Language Features in Dyalog APL 2](#_Toc385366873)

[2.2 Automatic Parallelization 2](#_Toc385366874)

[2.3 Explicit Parallelization 3](#_Toc385366875)

[3 Introducing Futures and Isolates 4](#_Toc385366876)

[3.1 Isolates 4](#_Toc385366877)

[3.2 Futures 5](#_Toc385366878)

[4 Getting Started 6](#_Toc385366879)

[4.1 … 6](#_Toc385366880)

[5 Troubleshooting 7](#_Toc385366881)

[5.1 Differences between Isolates and Namespaces 7](#_Toc385366882)

[5.1.1 Trapped Errors 7](#_Toc385366883)

[5.1.2 Unexpected Output 7](#_Toc385366884)

[5.1.3 File Ties Not Shared with Isolates 7](#_Toc385366885)

[6 Debugging Isolates 8](#_Toc385366886)

[Appendix Z: Implementation Details 9](#_Toc385366887)

# Introduction

Dyalog believes that the isolate workspace and language support for futures are now close to complete, but this document was still very much “work in progress” at the time that Beta 4 was shipped. If you are interested in testing these features, write to [beta140@dyalog.com](mailto:beta140@dyalog.com) to register your interest in receiving updated documentation as it becomes available.

This document describes proposals for two new primitives, and a new type of array, designed to allow APL developers to take advantage of multiple processors to execute code in parallel:

* The function “isolate” (**¤**), which creates a special kind of namespace known as an *isolate*, inside which all APL language statements are executed in parallel with the main APL process.
* The monadic operator “parallel” (**∥**), which derives a function that will execute the left operand in an empty isolate.
* A new type of array known as a “future”, which is a pointer to the future result of an expression currently being executed in an isolate. Futures can be passed around as function arguments and as items of nested arrays, but will “block” any expression which actually needs the value pointed to by the future.

In version 14.0, futures are implemented as a new primitive feature supported by the interpreter. The isolate function and the parallel operator are provided as models which are partly implemented in APL, provided in the distributed workspace ws\isolate.dws.

**Important:** Although version 14.0 only provides a model, Dyalog believes that the design is very close to being complete, and urges all users with applications containing potential parallelism to make use of the experimental implementation, to help confirm the design and reveal any weaknesses that need to be worked on in the final implementation as part of the core language.

The current plan (subject to change without notice) is to produce a revised model based on user feedback for version 14.1, and a fully “primitive” implementation in version 15.0.

# Philosophical Preamble

This section contains a discussion of issues surrounding the parallelisation of APL code. If you are looking to quickly get started with using futures and isolates, skip to section 3.

### Parallel Language Features in Dyalog APL

Ever since the first implementation, Dyalog APL has included many language features which are parallel in nature, although the implementations did not perform actual parallel execution:

* Most primitive functions work on arrays, performing operations on all the elements of argument(s).
* Many classical primitive operators from APL2 such as each (f¨), inner product (f.g), outer product (∘.g), and some reductions (f/) and scans (f\), express repeated and thus potentially parallel execution of one or more functions.
* New operators added to Dyalog APL more recently, such as rank (f⍤n), key (f⌸), and even some uses of the power operator (f⍣g), also express potentially parallel operations.
* In Dyalog APL, the use of the dot to execute an expression within the scope of an array of objects or namespaces, expresses potentially parallel execution (namespaces.foo).

Until recently, none of the above language constructs caused actual parallel execution. Starting with version 14.0, Dyalog intends to give high priority to implementing actual parallel execution of APL language statements.

There are two main tracks to pursue:

* Automatic parallelization, where the interpreter automatically infers that a ”traditional” APL language statement can be parallelised, and automatically decides to execute code in parallel when this will improve the performance of an application.
* Explicit parallelization, where the language provides new mechanisms that allow the programmer to declare that certain parts of the application could or should be executed in parallel.

### Automatic Parallelization

Given the number of parallel language construct in the APL language, it might appear that automatic parallelisation should be easy to implement. In fact, parallel execution of a number of scalar dyadic functions on floating-point arguments was introduced in version 12.1 of Dyalog APL, and some user applications did realise noticeable speed-ups. However, automatic parallelisation of traditional APL expressions faces a number of significant challenges:

1. Most arguments are small: Parallel execution has a set-up time to start parallel threads or processes, and a synchronization cost when the results computed by separate processes are coalesced into a single array result. In typical real applications, the arguments to most primitive functions only have a very small number of elements. Any potential speed-up from parallel execution will be lost in the setup
2. Cost of memory access: Although it is common for modern hardware to have multiple cores, the memory cannot support many cores reading or writing significant volumes of data simultaneously: a queue will form. Thus, operations which perform small quantities of processing compared to data movement (such as adding or multiplying large arrays), cannot be parallelized efficiently on current hardware.
3. Side-effects: When operator expressions are applied to user-defined functions, those functions can have side-effects. Many existing applications rely on the current order of execution of operator expressions and would fail or produce incorrect results if user-defined functions started executing in parallel.

Successful automatic parallelization on existing hardware will require and APL language system which is able to coalesce multiple primitive APL operations into larger units which can be effectively executed in parallel – plus the ability to detect that user-defined functions are free from side-effects. Dyalog is conducting and funding research into these areas and does expects to increase the amount of automatic parallelization that Dyalog APL supports.

However, in the short term, explicit parallelization though futures and isolates promise to provide more “bang for the buck” for the application developer.

### Explicit Parallelization

As mentioned in the previous section, it can be difficult for the APL engine to determine whether parallelization is both safe and worthwhile. However, in many cases the programmer will know where the potential for significant parallel execution lies in an application, and be able to identify sections of code that are free of side-effects (or able to “manage” any side-effects that might exist) - and represent enough work for the setup and synchronization costs to be worthwhile.

In the past, many languages have provided language extensions[[1]](#footnote-1) aimed at allowing parallel processing. These have typically depended on the programmer to make use of explicit synchronization features such as “semaphores”, which allow threads to mutually exclude each other from interfering with each other as they modify shared data, notify each other of progress, and wait for the one step to complete before a dependent task can be started. These features are notoriously difficult to use and often give rise to deadlocks and other difficult-to-detect defects in code, even when used carefully.

The challenge is to come up with new language features which make it easy for APL users to introduce parallelism, without reducing the readability of the code – and without making the application fragile and prone to synchronization of timing errors.

This has been the main design goal for futures and isolates.

# Introducing Futures and Isolates

*Futures* and *Isolates* are extensions to the APL language which are intended to provide the APL programmer with mechanisms for expressing the existence of potentially parallel sections of code in a *deterministic* fashion. Briefly:

* An *isolate* is a namespace which is semantically equivalent to a normal namespace, except that expressions which are executed inside an isolate can run in parallel (in a separate process) to the main application process. The current proposal is to create isolates using a new primitive function ¤ (isolate).
* A *future* is an array of unknown rank, shape and content, which is returned as the result of any expression that is executed inside an isolate. Futures can be passed as arguments to user-defined functions and arrays containing futures can be subject to structural transformations such as reshape or partitioned enclose, without blocking. Blocking (until the isolate manufactures a result) occurs if a future is passed to a function which needs to know the actual value (for example, a mathematical primitive function).

Between them, futures and isolates make it easy to divide application code into sections which can run in parallel: If one function required the result of code which is still executing in parallel, the “dependent” function will automatically wait until the data that it needs is available.

### Isolates

An isolate created using ¤ is almost identical to a namespace created using ⎕NS. The contents of an isolate can be referenced using the same “dot notation” that is used to refer to the contents of a regular namespace – and it is possible for code running inside an isolate to refer back to the parent space using the ## (Parent) symbol. However, isolates reside in separate processes and do not share a common workspace with the main process, and this does introduce some restrictions.

The most important differences between isolates and namespaces are the features that make isolates a tool for simple implementation of parallel execution within an application:

* Any evaluation which occurs within an isolate is handled in a separate process. If more than one processor is available, expressions inside isolates will run in parallel with the main process. If more than one isolate exists, execution proceeds independently within each isolate.
* An expression executed within an isolate immediately returns a *future*, without waiting for the expression to finish execution. A future is a placeholder for an as yet un-computed value (see section 3.2 for more information). The future automatically turns into a real value when the expression produces a result.

A number of differences are due to the fact that the isolate is actually inside a separate process. Although the contents – both code and data – are “only a dot away”, there are some restrictions on making references between isolates (or they would not deserve the name).

* Because the isolate is running in a separate process, it does not share file ties or any process-related handles or resources with its parent process[[2]](#footnote-2).
* Inside an isolate, the special symbol ## (aka “Parent”) can be used to refer to the root of the parent processes workspace. In a namespace, it would refer to the space inside which the isolate was created.
* Code running inside an isolate can ***only*** refer to spaces that are not contained within itself ***via*** the root of the parent workspace (referenced using ##, as mentioned above). You cannot pass a reference to something contained within one isolate to another isolate.

A handful of differences are due to the current implementation. These may disappear or change as we collect feedback and the implementation improves and eventually becomes completely integrated with the interpreter. Examples of these differences include:

* Expressions which produce a shy result when executed in a namespace may no longer be shy when executed in an isolate, because a future was returned and was “materialised” in a subsequent step – for session display. For example: nsref.(X←42).
* By default, errors occurring within an isolate will be trapped and reported to the calling environment. See the section 4 for information on how to disable error trapping and debug errors inside isolates.

### Futures

A future is an array of unknown rank, shape and content, which is returned as the result of any expression that is executed inside an isolate. Futures have the following characteristics:

* A future can be assigned to a variable, passed as the argument to a function or operator, stranded together with other elements to form an array, or be inserted into a nested array, while remaining a future (with unknown content).
* Arrays *containing* nested (enclosed) futures can be passed as arguments to user-defined functions or operators, or subjected to primitive structural functions such as reshape or compress, so long as no function which needs to know the shape or content of the future is encountered.
* If a primitive or system function which needs to reference the contents of the future is encountered, an attempt is made to display the future in the session, or pass it to an external program such as a Microsoft.Net method or a shared library function, the current thread will suspend until the expression which produced the future completes, and an actual value is assigned.

In other words, future results of expressions that are currently being executed inside one or more isolates (which are running in separate processes) can be passed around an application until the actual values are needed for the next computational step, at which point the code (the thread) that needs it will automatically block until the value is available.

Together, isolates and futures are designed to make it straightforward to write applications which contain sections that can run in parallel. No semaphores or special mechanisms are required to synchronize the independent expressions: When a result that is being computed asynchronously is required, the “consume” of this result will simply wait until it can proceed.

### Parallel Operator

Parallel (∥) is a monadic operator which returns a derived function which will immediately return a future, and execute the operand function within an empty isolate: It provides a mechanism for parallel execution of a function without the overhead of explicitly creating an isolate.

The parallel operator is typically combined with each (¨), in which case the operand function is invoked multiple times. For example, the elapsed time for the following expression should be slightly more than 3 seconds (without the parallel operator, it would be roughly 12 seconds):

⎕DL ∥¨3 3 3 3  
3.003 3.004 3.003 3.003

The temporary isolate is empty when the function starts running: the operand function may create global names for its own use, but they will be discarded together with the temporary isolate.

# Getting Started

While the ability to deal with the new type of array known as a future is a primitive feature of Dyalog APL version 14.0, the mechanisms for the creation of isolates and the execution of expressions within them are modelled in APL with the help of some undocumented I-Beams.

The model can be found in the distributed workspace ws\isolate.dws, in the form of a namespace called isolate, a function called ø and an operator named IÏ. The following functions and operators expose models of future primitive functions or operators:

|  |  |  |
| --- | --- | --- |
| **Name (within #.isolate)** | **Primitive Equivalent** | **Description** |
| New ''|*ns* | ¤ arg | Creates a new isolate – empty or cloned from ns |
| f ll | f∥ | Parallel operator |
| f llEach | f∥¨ | Equivalent of Parallel Each |
| f llOuter | ∘.(f∥) | Parallel outer product |
| f llRank m | (f∥)⍤m | Parallel rank operator[[3]](#footnote-3) |
| f llKey | (f∥)⌸⌸ | Parallel key operator3 |

The isolate namespace exposes the functions Config, AddServer and StartServer, which allow the setting of configuration options and the ability to start and use “remote” isolate servers. For more on these, see section 5.

Finally, the following global names are defined as shortcuts to the two most commonly used names in the isolate namespace, in order to provide closer visual fidelity with the final expressions that will use primitives.

|  |  |
| --- | --- |
| Name | Description |
| ø | Model of ¤ function (cover for isolate.New) |
| IÏ | Model of ∥¨ (cover for isolate.llEach) |

In the rest of this document, most parallel examples will make use of the above shortcuts: ø is used as a placeholder for the future ¤ and IÏ in place of ∥¨. Let us take a closer look at them:

### Parallel Each

A very simple example, which illustrates the potential of isolates very well, is the behaviour of parallel invocations of the system function ⎕DL, which consumes almost no system resources, but takes a fixed amount of elapsed time. In “normal” Dyalog APL:

T←⎕AI ⋄ ⎕←⎕DL¨8⍴3 ⋄ ⎕←⎕AI-T  
3.008 3.004 3.005 3.004 3.007 3.009 3.007 3.007  
0 7125 24054 0

The second line of output shows that the elapsed time for this expression is very slightly more than 24 seconds, which should be no surprise: the each operator applies the ⎕DL function sequentially to each of the 8 elements in the right argument.

In v14.0, we are able to run the 8 delays in parallel using the operator named IÏ, which is a cover for isolate.llEach - and represents the future primitive operator pair ∥¨:

T←⎕AI ⋄ ⎕←⎕DL IÏ 8⍴3 ⋄ ⎕←⎕AI-T   
3.001 3.002 3.003 3.003 3.002 3.003 3.002 3.003  
0 3734 3872 0 ⍝ Timings need to be redone

**Note:** The first time isolates are used after )LOAD of the workspace containing the isolate namespace, there may be an additional delay while the isolate host processes are started.

In the current (v14.0 Beta 4 implementation), there is also a significant overhead (~0.1 seconds) for the creation and destruction of each temporary isolate created by IÏ: we expect to have reduced this by an order of magnitude or more before 14.0 is released.

Of course, parallel invocations of ⎕DL are not THAT useful. Let us examine what happens if we use a (very slightly) more interesting example. Consider the following function which loops a given number of times, does some fairly hard work, and returns the number of elapsed milliseconds as its result:

∇ r←loop n;m  
[1] m←n  
[2] r←3⊃⎕AI  
[3] :Repeat ⋄ n←n-1  
[4] work←≢∪10000?10000  
[5] :Until n≤0  
[6] r←(3⊃⎕AI)-r  
 ∇  
  
Unlike ⎕DL, this function consumes a large amount of CPU time compared – it is in a tight CPU loop. Let us repeat an experiment similar to the one above:

T←⎕AI ⋄ loop¨ 4⍴1E4 ⋄ ⎕AI-T  
3134 3149 3097 3233  
0 12609 12617 0

T←⎕AI ⋄ +loop IÏ 4⍴1E4 ⋄ ⎕AI-T  
7363 7499 7470 7522  
0 6281 7665 0 ⍝ retest with new version

We can see that the parallel invocations come fairly close to running twice as fast, completing the four calls to loop in 60% of the time (7.6 vs 12.6 seconds). This is not a bad result on the machine where this was run, a fairly typical laptop of the current era – with a 1.6Ghz Intel Core i5-4200 with 2 cores (split into 4 logical processors). Although this kind of machine has two cores, they share the same memory sub-system, so the cores will compete for memory accesses and slow each other down – so a factor of two is unachievable.

### Explicit Isolates

When a function is invoked under parallel each, an empty isolate is created for it to run in, and the isolate is destroyed when execution completes. In some situations, it makes sense to create a number of isolates ahead of time, and use them to execute more than one expression. Isolates are created using the function ø, which represents the future primitive function ¤. The right argument can be a namespace to clone, or an empty vector to create an empty namespace. For example:

is1←ø '' ⍝ Create an empty isolate  
 is2←ø '' ⍝ Another one  
 is1.X←1 2 3 4 ⍝ Give the name X a value in 1st isolate  
 is2.X←5 6 7 8 9 ⍝ A different value in the 2nd isolate  
 isolates←is1 is2 ⍝ References to isolates can form an array  
 isolates.(+/X) ⍝ Execute (+/X) in 2 isolates in parallel  
10 35  
 T←⎕AI ⋄ isolates.(⎕DL 3) ⋄ ⎕AI-T  
3.003 3.003  
0 3000 3010 0

The last example shows us that, compared to parallel each, the overhead is much lower when pre-existing isolates are used (about 4ms in this case). We can repeat the example with our loop function:

isolates←isolates,ø¨'' ''⍝ Add another 2 isolates  
 isolates.⎕FX ⊂⎕CR 'loop' ⍝ Fix loop function in each isolate  
 loop loop loop loop   
 T←⎕AI ⋄ isolates.loop 1E4 ⋄ ⎕AI-T  
6569 6535 6491 6566  
0 5453 6989 0

As expected, the result is a bit better than what we saw using IÏ, because we avoid the cost of managing the temporary isolates.

### A More Realistic Example

∇ r←GetPages addresses;⎕USING;srcUri;client;r;address;i  
[1] ⎕USING←'System,mscorlib.dll' 'System.Net,System.dll'  
[2]  
[3] r←(⍴addresses)⍴⊂''  
[4] client←⎕NEW WebClient ⍬  
[5]  
[6] :For i :In ⍳⍴addresses  
[7] srcUri←⎕NEW Uri(addresses[i])  
[8] r[i]←⊂client.DownloadString srcUri  
[9] :EndFor  
 ∇

text←⊃GetPages,⊂'http://www.usnpl.com/manews.php'  
z←('(<a href=")(.\*?)(.com/">)'⎕S'\2.com/') text

On Wednesday I will write some nice examples to go here, showing:

* Parallel execution
* The “peek” I-Beam
* Reaching back into the main process to read/modify globals
* Making “callbacks” into the main process (SQL query)
* The old “parallel” workspace

# Advanced Topics

Queueing: Although isolates and the main process can both be using multiple APL threads (created using the spawn operator &), all calls INTO an isolate are placed in a queue and only a single call into an isolate will be running at any one time. Even if a future has been passed back as a “result” to the caller, the isolate will not start processing the next expression until the previous one has finished. The same is true for calls back to the main process from isolates: Only one such call back will be active at any one time.

Load Balancing

Remote Servers

Running as a Windows Service

Integrating with your application: LX, workspace to load, CONGA/DRC

Runtime vs Devt: Startup speed

Unix?

# Troubleshooting

Hints and tips… Common error messages and things that go wrong.

Remember to “resume all threads”!

### Differences between Isolates and Namespaces

The following example is intended to demonstrate some of the differences between isolates and namespaces:

### Trapped Errors

(see section 6 on debugging isolates for more information)

### Unexpected Output

ns←⎕NS '' ⍝ Create an empty isolate  
 is←isolate.New '' ⍝ Create an empty isolate  
 ns.(X←10?10) ⍝ Run an expression in ns  
 is.(X←10?10) ⍝ With an isolate, this produces output  
6 9 3 4 8 2 7 1 5 10

### File Ties Not Shared with Isolates

Unlike namespaces, isolates are separate processes, so they do not share any resources that belong to the current process, such as file ties, loaded libraries, TCP sockets, etc.

'file1' ⎕fstie 0 ⍝ Tie a file in main process  
1  
 'file2' ns.⎕fstie 0 ⍝ Tie a file “in” a namespace  
2  
 'file1' is.⎕fstie 0 ⍝ Tie first file in the isolate  
1  
   
 ⎕fnames,⍪⎕fnums ⍝ root and ns are in the same process  
file1 1  
file2 2

is.(⎕fnames,⍪⎕fnums) ⍝ isolate is a separate process  
file1 1

# Debugging Isolates

# Appendix Z: Implementation Details

xxx

1. Including Dyalog APL, which provides a spawn operator (&) the ⎕TGET/⎕TPUT/⎕TSYNC family of system functions, and the :Hold … :EndHold control structure. [↑](#footnote-ref-1)
2. In fact, as we shall see later, an isolate *could* be running on a completely different machine, which might not even be running the same operating system or be able to see the same file system as the parent process. [↑](#footnote-ref-2)
3. Might change before release – or the operators might be renamed: In the current model, to avoid blocking in order to “mix” the results into the complete result matrix, llRank and llKey return nested arrays of futures, which need a final “mix” to produce the result that the primitive operator would return. [↑](#footnote-ref-3)