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)

# About This Document

This document describes new language features designed to allow APL developers to take advantage of multiple processors to execute code in parallel. The language features are labelled “experimental” in v14.0, and partly implemented using APL models. However, Dyalog believes that the design is very close to being complete, and urges all users with applications containing potential parallelism to use the experimental implementation during 2014, to help confirm the design and reveal any weaknesses that need to be worked on in the final implementation.

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

### Audience

It is assumed that the reader has a basic understanding of Dyalog APL language.

# 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.

### Isolates

An *Isolate* is a namespace which is semantically equivalent to a normal namespace created using ⎕NS, except that it is managed by a separate process (which is running a separate copy of the Dyalog APL interpreter). 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 in most cases there is no difference between namespaces and isolates. There are noticeable differences, some of which are due to “fundamental” issues, plus a few which are artefacts of the fact that isolates are currently implemented using a model which is partly written in APL.

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.

Differences which are due to limits of the current implementation, and which may disappear or change as the implementation improves and eventually becomes completely primitive, include:

* Some expressions which would not produce output when executed in a namespace, will produce output when executed in an isolate. 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. At present, there is no other way to manufacture futures in Dyalog APL, but other mechanisms may appear in the future. 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.

# Getting Started

The following examples are intended to illustrate the most important features of futures and isolates. All examples assume that the isolate workspace has been loaded, so that the function ø (isolate.New) and the operator IÏ (parallel each) are defined.

### …

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)

# Advanced Topics

Load Balancing

Remote Servers

# 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 (&) and the ⎕TGET/⎕TPUT/⎕TSYNC family of system functions. [↑](#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)