

Parallel Each User Guide

Version 1.1

(companion to Dyalog APL v13.0)

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
<http://www.dyalog.com>

Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2011



Copyright © 2011 by Dyalog Limited.

All rights reserved.

Version 1.1.0

First Edition March 2011

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.

TRADEMARKS:

IBM is a registered trademark of International Business Machines Corporation.

Microsoft, MS and MS-DOS are registered trademarks of Microsoft Corporation

Unix is a trademark of X/Open Ltd.

Linux is a trademark of Linus Torvalds.

Windows, Windows NT, Visual Basic and Excel is a trademark of Microsoft Corporation.

Intel and Core are trademarks of Intel Corporation

All other trademarks and copyrights are acknowledged.

Contents

THE PARALLEL WORKSPACE	1
<i>Preamble</i>	1
<i>Introduction</i>	1
Formal Syntax	3
When to Use the Parallel operators	4
MANAGING SLAVE PROCESSES	6
<i>Initializing Global Data</i>	6
<i>Collecting Results</i>	7
<i>Creating Slave Processes</i>	9
<i>Terminating Slave Processes</i>	10
<i>If Everything Stops Working</i>	10
DEBUGGING	11
<i>Fork.OnError</i>	11
<i>Fork.Disabled</i>	13
<i>Server-Side Debugging</i>	13
REMOTE SLAVES	14
<i>Using P.Each on More than One Machine</i>	14
TUNING	17
<i>Setting the Block Size</i>	17
<i>Feedback</i>	17
<i>Transferring work</i>	19
OPTION REFERENCE	20
FUNCTION REFERENCE	22

CHAPTER 1

The Parallel workspace

Preamble

The `parallel` workspace contains tools which allow Dyalog application programmers to make use of multiple cores on one or more computers.

Throughout this document the use of the prefix `P` is synonymous with `Parallel`. Where generic attributes and behaviors are described then the reference to `P.Each` should be read as a reference to all the parallel operators `P.Each`, `P.OuterP` and `P.Rank`.

The `P.Each` (`⊆`), `P.OuterP` (`⊆.`) and `P.Rank` (`⊆⊆`) operators are APL models of the functionality that may be added to Dyalog APL as primitive operators, once the problem space is well understood and we have collected some experience from their use in real applications. The functionality provided by this workspace and these models should be expected to change in future releases (but since it is entirely coded in APL, users may continue to use old copies of the workspace at their discretion).

Unix and Linux are not supported in this version of the `parallel` workspace.

All examples in this document assume `(⊆ML ⊆IO)←0 1`.

Introduction

In APL, the primitive operator *each* (`⊆`) is used to execute primitive or user-defined functions repeatedly on a set of arguments. For example, the following expression computes the number of co-primes¹ of ω that are less than ω for each integer in the range 1 to 25:

```

      {+/1=ω∨ιω}⊆ι25
1 1 2 2 4 2 6 4 6 4 10 4 12 6 8 8 16 6 18 8 12 10 22 8 20

```

¹ Wikipedia: two integers a and b are said to be **coprime** or **relatively prime** if they have no common positive factor other than 1 or, equivalently, if their greatest common divisor is 1.

The primitive operator `¨` makes 25 sequential function calls. APL waits for each to complete before making the next call. `P.Each` is a user-defined operator which has the same syntax as the primitive operator but makes use of several APL processes to execute the calls in parallel. We can replace `¨` by `P.Each` in the above expression:

```
)load parallel
c:\...\parallel saved Mon Nov 30 13:17:15 2009

{+/1=ω∨ιω} P.Each ι25
1 1 2 2 4 2 6 4 6 4 10 4 12 6 8 8 16 6 18 8 12 10 22 8 20
```

The result is the same, but if your machine has multiple “cores”, `P.Each` will have split the function calls across a number of “slave processes” – one for each core. The `parallel` workspace also supports several machines working together to pool CPU resources – see the section titled *Remote Slaves* for more information.

There is some overhead in communicating with the slave tasks – and starting them, if they are not already running. As a result, the example above probably runs significantly slower using `P.Each`, because of the CPU time consumed by the actual function calls. However, if we do a significant amount of work, we should see an improvement (the following example executed on a laptop with an Intel Core 2 Duo CPU):

```
]cputime {+/1=ω∨ιω} ¨ ι10000
Running...
CPU (avg): 6958
Elapsed: 6708
]cputime {+/1=ω∨ιω} P.Each ι10000
Running...
CPU (avg): 32
Elapsed: 3755
```

The elapsed time is reduced by nearly 45%: execution is roughly 1.8 times faster using two cores when making these 10,000 function calls. Note that the reported CPU time drops to almost nothing, as the function calls are actually performed in slave processes. The time reported here is the overhead of handing the communications with the slaves (in this case, roughly 1% of the total CPU time).

```
]cputime xx←(ι1000)∘.{(+/ια)÷+/ιω} (ι1000)
Running...
CPU (avg): 59592
Elapsed: 60066
]cputime yy←(ι1000){(+/ια)÷+/ιω}P.OuterP (ι1000)
CPU (avg): 250
Elapsed: 1583
Running...
xx≡yy
1
```

Formal Syntax

The operators are defined as;

$$R \leftarrow \{ \alpha \} (\alpha \alpha \text{ P.Each}) \omega$$

models $R \leftarrow \{ \alpha \} (\alpha \alpha \text{ ``}) \omega$

α = left argument

ω = right argument

$\alpha \alpha$ = function, either as a dynamic function or the name of the function to be executed

Example: `R <- data {alpha+omega} P.Each data2`
`R <- data 'Foo' P.Each data2`

$$R \leftarrow \alpha (\omega \omega \text{ P.OuterP}) \omega$$

models $R \leftarrow \alpha (\circ . \omega \omega) \omega$

α = left argument

ω = right argument

$\omega \omega$ = function, either as a dynamic function or the name of the function to be executed

Example: `R <- data {alpha+omega} P.POuterP data2`
`R <- data 'Foo' P.POuterP data2`

$$R \leftarrow \alpha (\alpha \alpha \text{ P.Rank } \omega \omega) \omega$$

models $R \leftarrow \alpha (\alpha \alpha \text{ `` } \omega \omega) \omega$

α = left argument

ω = right argument

$\alpha \alpha$ = function

$\omega \omega$ = ranks, controls how the function $\alpha \alpha$ is applied to the cells of the data arguments

Example: `R <- data {alpha++/omega} P.Rank (1 2) data2`
`R <- data 'Foo' P.Rank (1 2) data2`

An additional and but as yet intended only as an internal operator

$$R \leftarrow \{ \alpha \} (\alpha \text{ Fork.dot } \omega \omega) \omega$$

models $R \leftarrow \{ \alpha \} (\alpha \alpha . \omega \omega) \omega$

α = left argument
 ω = right argument
 $\alpha\alpha$ = refs
 $\omega\omega$ = function

Example: `R←data ##.Fork.NSS Fork.Dot {α++/ω} data2`
`R←data ##.Fork.NSS Fork.Dot 'Foo' data2`

When to Use the Parallel operators

In theory, n cores should perform a job n times as fast as a single core – but in practice this is rarely the case. In addition to the overhead of managing slave processes and transmitting arguments and results, the cores need to share resources – in particular memory and disk storage – and network resources. The fact that disk and network bandwidth might be a bottleneck probably comes as no surprise, but the impact of sharing memory can also be significant. In a modern multi-core microprocessor, each core has some of its own high-speed cache “on chip”, but all the cores share the same main memory (“RAM”) – and some cache levels can also be shared. If the function being executed requires frequent access to off-cache data, the cores will compete for main memory access and all slow down. The bandwidth of main memory access is often only just enough to satisfy a single core, if that core is in a loop reading memory.

In some cases, adding processors to a task will actually slow it down. Machines have significantly different performance profiles when there are resource conflicts. You will need to experiment a little to find optimal settings for each task you need to perform.

To illustrate, consider the following three functions (which you can find in the QA namespace in the `parallel` workspace):

```

[1] ▽ i←LoopTest i
      :While 0<i+i-1 ◇ :EndWhile A No memory, lots of CPU
      ▽
      ▽ r←Mixed n
[1]   r←n?n
[2]   r←+/\+\▽Δ▽Δr A Some work, but also memory scans
      ▽
      ▽ r←ThrashMemory n
[1]   r←+/\in A Lots of generated data, almost no “work”
      ▽

```

These functions illustrate different points on the “parallelizability scale”. Running the function `QA.TestGeneral` on your machine runs the functions with a right argument of (1500) for both `Each` and `P.Each`. It displays a table showing the improvements. Some variation from one run to the next is to be expected but the perfect score of 2.00 (on a dual core machine) for `LoopTest` in the following is a freak accident:

2 Tasks / 2 Cores	Each	P.Each	Relative
{#.QA.LoopTest 10000}	2916	1458	2.00
{#.QA.Mixed 100000}	8737	5402	1.62
{#.QA.ThrashMemory 5E6}	7613	6829	1.11

As can be seen above, the speedup is close to 2 for the job which consumes a lot of CPU and uses little memory – but the function which spends most of its time writing integers to memory and then adding them up, only speeds up very slightly. In fact, both cores will be reported as 100% “busy” in all of the above cases – but when executing the last function, a very large amount of time is spent waiting for memory. In fact, if the system was trying to run any other tasks at the same time, overall system throughput will have decreased significantly – so throwing multiple cores at a task can in fact be counter-productive.

An example of a typical “successful” use of `P.Each` is a pension calculation application which computes pensions for hundreds of employees. The calculation for each employee is completely independent of the rest (except for reading a small amount of information from a database). Using 8 processes on an Intel machine with 2 “quad” processors (8 cores), `P.Each` speeded this application up by a factor of 5. You are unlikely to achieve higher speedups than this without using more than one physical machine.

CHAPTER 2

Managing Slave Processes

Initializing Global Data

Our first example used a function which did not require any data other than its arguments. In this case, the user-defined operator can replace the primitive without any other changes to the application. In the real world, functions often use global data and store results in global variables. `P.Each` can also support such functions – but a little more work is required to ensure that the environment is ready before each function call – and to collect any results that were saved as “side-effects” in the global variables.

By default, the slave processes are initialized with a copy of the active workspace at the time of the first use of `P.Each`. Any global data in the workspace is automatically available to the slave tasks. If global input data (or code) changes, we can transfer the necessary data to the slaves using the function `P.Set`.

Imagine that we have the following simple function, which computes statistics for a row in a global matrix, stores the results in a global variable. This isn’t a pretty function, and it probably doesn’t do enough work to speed up, but it can be used to illustrate the use of `P.Each` with “messy” functions.

```

▽ r←Stats i;data;avg;max;min;median
[1]   r←SLAVEID  # Return the ID of the slave
[2]
[3]   data←{ω[Δω]}DATA[i;]  # Extract row i and sort it
[4]   min←1↑data ♦ max←~1↑data
[5]   median←data[⌈0.5×pdata] ♦ avg←(+/data)÷pdata
[6]
[7]   MED_ABOVE_AVG+←median>avg
[8]   OUTPUT;←i,max,min,median,avg
▽

```

Before we call the function, we need to set `SLAVEID` and `DATA` in each process, and also initialize the result variables:

```

DATA←0.1×?1000 10ρ1000  # 1,000 rows of random data
MED_ABOVE_AVG←0 ♦ OUTPUT←0 5ρ0
P.Set 'Stats DATA OUTPUT MED_ABOVE_AVG'
(ιP.SlaveCount) P.Set 'SLAVEID'

```

The first (monadic) call to `P.Set` simply copies the current definition of the function (in case we changed it ☺) and the value of the three variables into each slave task. A dyadic call to `P.Set` requires one element for each active slave in the left argument, and sets the named variables to different values (so `SLAVEID+1` in the first slave and 2 in the second, etc). We can now call our function:

```
'Stats' P.Each 16
1 2 1 1 1 2
```

The result is the `SLAVEID` of the slave which processed each element. The function `P.CopyState` can be used to not only copy data to the slaves, but also replicate all the current native and/or component file ties in the slaves (obviously, the files must not be exclusively tied). `P.CopyState` takes a three-element vector on the right: A matrix of names to be moved to the slave workspaces, and two Boolean flags which specify whether to replicate the native and component file ties, respectively. For example:

```
P.CopyState (↑'LOG' 'SillySum') 0 1
```

The above copies two objects, and ties the same component files in each slave as are currently open in the active workspace.

Collecting Results

We can either use `P.Get` to retrieve it from each slave:

```
]display P.Get 'OUTPUT'
```

1	99.2	4.4	61.4	53.63
3	96.8	1	49.6	51.19
4	89	10.1	41.3	52.22
5	99.9	0.6	57.6	51.66

2	85.6	12.48	4	48.34
6	95.1	3.9	52.9	56.99

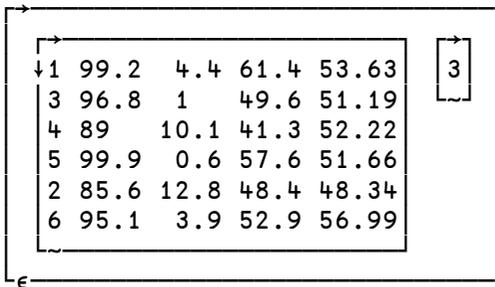
```
P.Get 'MED_ABOVE_AVG'
2 1
```

As we can see from the first column of `OUTPUT`, the first slave processed rows 1 3 4 5, and the 2nd slave rows 2 and 6 (which fortunately matches the result that we received earlier). The first slave encountered two rows where the median was higher than the average, and the second one.

The function `P.Gather` is designed to make it easy to aggregate global variables from all the slaves and combine them into variables as similar as possible to the result of running the function in a single process. The default function is `;` (catenation on the first dimension), but a matrix right argument allows specification of a different function for each variable. In our case, we want to catenate `OUTPUT` together, but add the “above average” counters together:

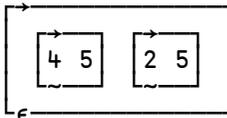
```
P.Gather 2 p'OUTPUT' ';' 'MED_ABOVE_AVG' '+'
```

```
]disp OUTPUT MED_ABOVE_AVG
```



In fact, the right argument to `P.Get` is simply executed in each slave process:

```
]disp P.Get 'pOUTPUT'
```



Creating Slave Processes

By default, the first invocation of `P.Each` will “fork” the current active workspace as many times as the machine has processors.

Under ‘nixes, forking will be done using the “fork” I-beam (`4000I`) to fork the active session (Unix support is still under development).

Normally, `P.Init` would be called at the start of the application (or a section that is intending to make use of `P.Each`). `P.Init` accepts the following arguments:

Argument	Meaning
' ' (empty vector)	Fork the active workspace as many times as the machine has processors (this is the same as calling <code>P.Each</code> without a call to <code>P.Init</code> first). Note that this cannot be done if there are threads or open windows (editor or trace), as <code>□SAVE</code> will fail.
n (integer)	Creates n clones of the current workspace.
ref1 ref2 (references to namespaces)	A vector of namespace references will initialize slaves from the namespaces.
wsname1 wsname2 (workspace names)	A vector of workspace names will initialize slaves using the named workspace.

Each slave process is a new APL task, started using the executable named in the variable `Fork.Exe` (normally `dyalogrt`) from the workspace in `Fork.WS` (normally `parallel`). If you want to create your own startup workspace, it has to have a latent expression calling the function `Fork.Start`. This function starts a Conga server on the port that it is asked to and waits for work orders and returns results via the Conga connection.

When a slave is initialized from a named name- or workspace, a namespace called `#.HASH` is created as a container for the application, and the namespace or workspace is materialized within this space. This is NOT done when the current workspace is cloned – in this case the slave workspace will have the same structure as the current workspace. If your application needs to be located in the root, you will need to either split the current workspace or write your own initialization code to load it into the slaves.

The function `P.State` displays the state of active processes:

```
P.State
Hosts:
Machine Port Cores
Local           2

Active processes:
# Host      Port  State
1 Local    14400 Idle
2 Local    14401 Idle
```

Terminating Slave Processes

The function `P.Reset` is used to end slave processes. Normally, it is called with a right argument of 0, causing a shutdown request to be sent to the slave and the closing of the Conga connection – and finally (after a delay), the termination of the slave process, if it does not shut down voluntarily. A right argument of 1 can be used if the workspace has lost track of the slave tasks. `P.Reset 1` will terminate all processes running the named executable (except the one that is running the `Reset`) – so it will also kill slaves or other runtime executables which might be performing completely different tasks. It should only be used as the very last resort²!

If Everything Stops Working

If everything comes to a halt, check whether you have the menu item *Threads/Pause on Error* checked. This is the default setting – and you probably want to keep it that way in order to allow orderly debugging when something fails (if you don't, some threads may continue to run and events may continue to fire as you attempt to debug a problem that has occurred).

When you finish debugging a problem, you need to select *Threads/Resume All Threads* so that all threads start running again. Not that you can bring background threads to a halt with an error which occurs in “immediate execution”. If you are using the tracer, you need to select the GREEN resume icon to get all threads running again – the black one will only resume the current thread.

² (`P.Reset 1`) has not been required by the author since the time when the infrastructure itself was being debugged – and in QA functions to test crash recovery.

CHAPTER 3

Debugging

Note: The behavior described in this section should be considered experimental and is likely to be improved in the first few releases of the `parallel` workspace.

Fork.OnError

If errors occur in a function being executed by `P.Each`, the default behavior is to abort `P.Each` as soon as all slaves terminate their current unit of work, and signal the error:

```
{7÷ω}P.Each 1 2 3 0 4 5 6
DOMAIN ERROR
  {7÷ω}P.Each 1 2 3 0 4 5 6
  ^
```

The variable `Fork.OnError` has a default value of `'Stop'`, which gives the behavior described above.

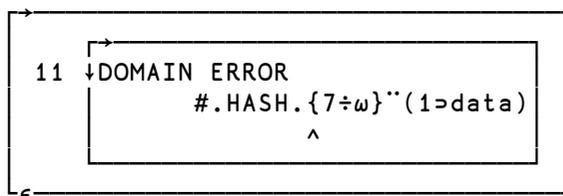
If `Fork.OnError←'Continue'`, units of work which cause errors will be marked as failed, and execution will continue. When working in this mode, you can use `P.ALLOC` and `P.Errors` to find out whether a call was completely successful:

```
{7÷ω} P.Each 1 2 3 0 4 5 6
7 3.5      1.4 1.166666667
```

```
P.ALLOC
0
```

```
⊃∘ρ P.Errors
0 0 2 2 2 0 0
```

```
]disp ↑`3>P.Errors
```



For each item which failed, `P.Errors` contains (`EN DM`) for the error.

A couple of things are worth noting:

1. Although only one of the elements in the right argument will cause a `DOMAIN ERROR`, slaves are often asked to process more than one element at a time (using the primitive `each` within the slave process). The entire block is flagged as failed. In the above example, the block size was 3. It may be convenient to force a block size of 1 in this mode, by setting `Fork.BlockSizeRange←1` 1.
2. On the server side, everything appears to happen inside the namespace `#.HASH`. In the event of a cloned workspace (as above), `#.HASH` is in fact a reference to `#`.

Finally, if `Fork.OnError←'Repro'`, then `P.Each` will offer to re-execute a failing block locally, so the failing call can be traced (assuming the necessary code and data is available). For example:

```

    ▽ r←a DIV b
[1]   r←a÷b
    ▽

    7 ('DIV' P.Each)1 2 3 0 4 5 6
PEach call failed, perform local repro? y
Right argument is in omega[ix], left in alpha[ix]
Set r[ix] to correct result and
→RESUME
DOMAIN ERROR
DIV[1] r←a÷b
    ^

```

The failing function is now suspended on the client side, so the environment can be inspected and appropriate steps taken to resume execution.

```

    a
7    b
0
    b←1
    →□lc
Resume execution? y
7 3.5 2.333333333 7 1.75 1.4 1.166666667

```

The remaining blocks will be processed by the slave tasks. Of course, if the function has side-effects and you were relying on `P.Gather`, the above process will not *quite* work – or at least the result of the `P.Gather` will be unreliable.

Fork.Disabled

If you are having problems with the remote execution of a function, you can set `Fork.Disabled←1` in order to temporarily suspend the use of the remote servers, and execute *all* function calls locally, even when your application calls `P.Each`.

Of course, if you have been using dyadic `P.Set` to distribute different data to slave processes, or are running code which has global side-effects, this may change the behavior of your system.

Server-Side Debugging

If you need to do actual debugging in the slave tasks in order to get to the bottom of a problem, or are having trouble with the parallel infrastructure itself, you should edit the `Fork` namespace, set `DEBUG←1`, and save the workspace so that this value is picked up when slave processes are started.

Set `Fork.Exe←'dya log'` in order to use the full development system rather than the runtime interpreter when starting run slave tasks. Localise and set `□TRAP` within your functions so that they will stop when they encounter an error – for example, `□TRAP←0 'S'`. With a little care, this should make it possible to at least inspect what is going on in the slave task. Remember to use “Resume All Threads” when continuing execution with the slave tasks, or the TCP server thread will remain paused and everything will grind to a halt.

CHAPTER 4

Remote Slaves

Using P.Each on More than One Machine

The `parallel` workspace makes it possible for several machines to work together and pool all their available cores, in order to create a “computational grid”. To do this, you need to start a “relay server” process on each machine (except the “client” machine, where `P.Init` is able to launch its own slaves). The relay servers are used to overcome the limitation of only being able to initiate tasks locally and so their sole purpose is to launch slave processes on the machine where they are running: once the slaves are started, a direct connection is opened between the client and the remote slave and the relay servers are then “out of the loop” until a slave needs to be shut down.

Each relay server must be started and running on their respective machines before `P.Init` is used on the client machine. The relay servers must load the `parallel` workspace, and be started with command line arguments which let the system know that they are a relay server and that they should offer relay services to a remote client. For example, you could start APL using the following command line:

```
dyalog.exe parallel.dws -ForkPort=5001 -ForkRelay=smeagol
```

To start a relay server, two command line parameters are required:

Name	Example	Meaning
<code>ForkRelay</code>	<code>Smeagol</code>	The ip address of the machine that should be allowed to request services
<code>ForkPort</code>	<code>5001</code>	The port that the server should listen on

In version 1.1, the existence of the `ForkRelay` parameter is used to determine that the process should act as a relay server – but the content is not used. In a future version, the idea is that incoming connections and work requests will only be accepted from the address named in this parameter. Mechanisms for running relay servers as “services” and using secure communications between machines are also envisaged for later releases.

You need to take the necessary steps to launch relay servers on the machines where they will be running. The existence of relay servers is declared to `P.Init` via the 4 column variable `Fork.RemoteServers`, of IP addresses and port numbers where the relay servers can be connected, the number of processes to start on each machine (0 to use all available cores), and optionally the name of the shared folder as seen by this machine, if

it needs to be accessed using a different path (or an empty vector to use the contents of `Fork.SharedFolder`). If you are going to clone the current workspace, you also need to set the variable `Fork.SharedFolder` to the name of a folder which can be seen by the client and all remotes; it will be used to save the current workspace so that the remotes can load it. The folder is also used to store all the variables transferred with `P.Set` since an audit trail is necessary to assist in the restarting of a disabled slave .

```

P.ProcessorCount
2
Fork.RemoteServers←1 4p'bree.dyalog.bramley' 5001 0 ''
Fork.SharedFolder←'\\fileserver\temp'
P.Init '' # Initialize slaves all available cores
P.SlaveCount
6
Parallel.State
Hosts:
Machine          Port  Cores
Local            5001   2
bree.dyalog.bramley 5001   4

Active processes:
# Host          Port  State
1 Local        14400 Idle
2 Local        14401 Idle
3 bree.dyalog.bramley 14400 Idle
4 bree.dyalog.bramley 14401 Idle
5 bree.dyalog.bramley 14402 Idle
6 bree.dyalog.bramley 14403 Idle

```

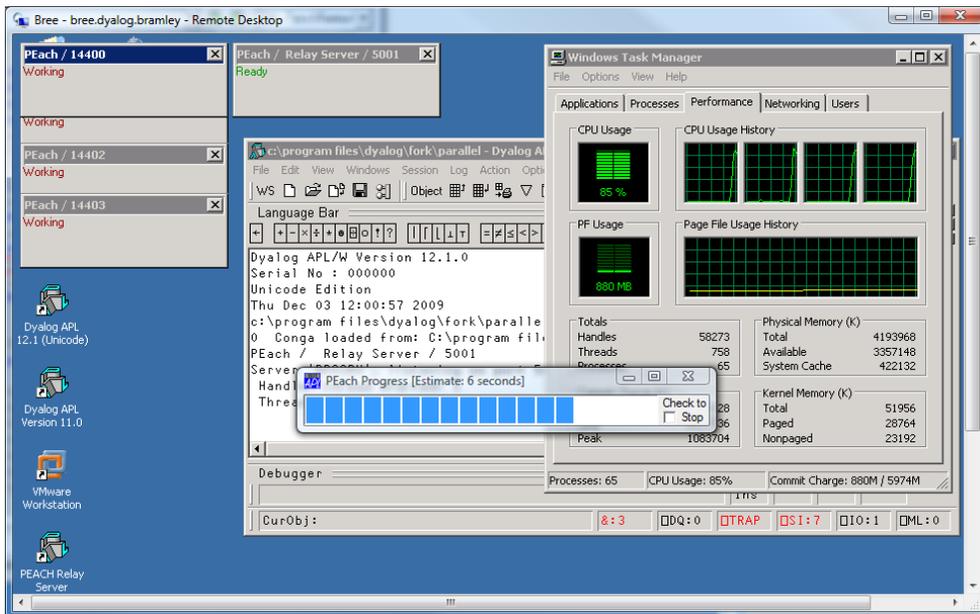
Six processors (2 local and 4 remote) shows improvement on the original “co-prime” function:

```

]cputime {+/1=ω∨ιω} ι20000
CPU (avg): 31138
Elapsed: 28353

]cputime {+/1=ω∨ιω} P.Each ι20000
CPU (avg): 31
Elapsed: 4896

```



This problem parallelizes very nicely; it manages to stay within the cache of each of the 6 cores.

CHAPTER 5

Tuning

Setting the Block Size

On its first use, `P.Each` attempts to compute an optimal “unit of work”, which will keep the communication overhead at a reasonable level, while ensuring that all the slaves are kept busy for roughly the same amount of time. Initially, each slave is given a single element of data to work on. When the first slave responds, `P.Each` uses the amount of time that this call took to compute the optimal block size. Two variables in the `Fork` namespace control this calculation:

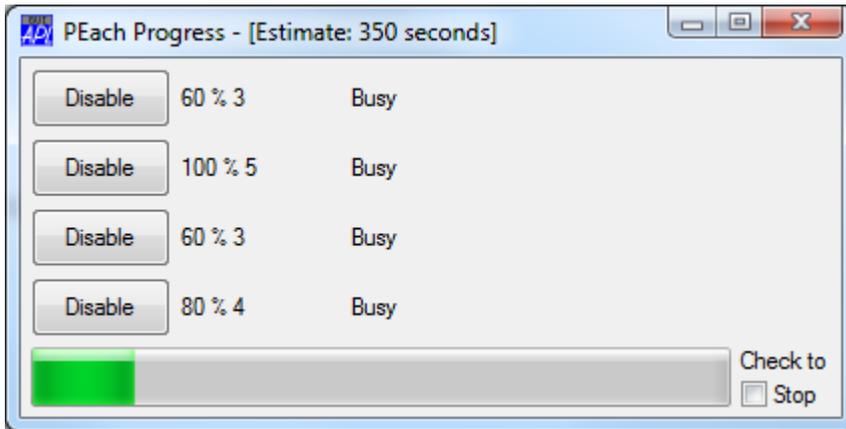
`Fork.BlockSizeTarget` is the number of milliseconds that is considered to be a reasonable unit of work. By default, this is set to `2000`.

The variable `Fork.BlockSizeRange` sets an upper and lower bound on the number of items to be computed in each unit of work.

Feedback

The value of the `ProgressAfter` property (default 3 seconds) determines when feedback is given back to the client. Although not fully functional in this release, it is intended that a later release will enhance it to give almost full control over the slaves while they are processing the data. At the moment the `Disable` (toggles to `Enable`) buttons are crude tools which should really only be used to reset a task before a complete new run is initiated. The reason is that the actions by the `WS` clone and the actions of the `P.Set` functions are not repeated once the task is restarted. The `P.Set` now records the data `Set` so it is intended to correct this as soon as possible. Once done it will potentially allow the stopping and starting mid process but this is not a feature yet.

For this release, the display should be viewed as a feedback panel rather than as a control panel.



The percentage following the buttons gives the relative efficiency of the slaves running. It is the number of work units processed by each slave as a percentage of the largest number processed. This means that slave 2 is working at full efficiency (100%), 4 next (80%) while 1 and 3 less so (60%).

The number following is the actual number of units of work completed by that slave – which explains the percentages. 5 is the max number of units processed by any slave. Each of the percentages gives each slave's performance indicator against the highest, (ie) $100 \times 3 \div 5 = 60$, $100 \times 5 \div 5 = 100$, $100 \times 3 \div 5$ and $100 \times 4 \div 5 = 80$. The second number keeps increasing as that slave processes more units of work and the sum is the total processed so far. The percentage gives a gauge of the efficiency and varies between 0 and 100%.

The free/busy word following the numbers indicates if the slave is busy processing data or whether it is free having completed some work and waiting for the next unit of work or whether the last available unit of work has been taken.

The green bar gives an approximation of how far the entire process has got. The bar now implies that about 15% of all the units of work have been processed.

The check box in the bottom right corner can be used to stop the processing completely.

Transferring work

The value of the `Transfer` property (default 0) determines what happens when a slave has completed all the work available to it. If `Transfer` is 0 then nothing is done and the client waits until all the slaves have completed their work before returning the result. If `Transfer` is set to 1, a setting only really appropriate when remote servers are used, the work being processed by one of the potentially slower tasks is copied to a free task. The first task is not stopped and a race to completion ensues. When one of the tasks completes, the result of the other is discarded. This attempts to ensure that the task is completed in the fastest time if there are mismatched processors running.

Warning: This attempt at speeding up the process is dangerous and unpredictable as the process generates a side effect such as a file update or global update to be used in conjunction with `P.Gather` since there is a danger of “double counting” some part of the work. This technique is only applicable in cases where there are no side effects or where the side effects are directly attributable and so the “double counted” elements can be eliminated from the final result.

APPENDIX I

Option Reference

The `Fork` namespace contains a number of variables which control the behavior of `P.Each`. The function `P.Defaults` sets a number of options within the `Fork` namespace which handles the forked processes. You can change these variables to change the behavior of the parallel workspace in various ways.

The available settings are:

Name & Default	Controlled Behaviour
<code>DRC←##.DRC</code>	Pointer to the Conga namespace which should be used
<code>IPVersion←'IPv4'</code>	IP protocol to use (alternative is <code>'IPv6'</code>)
<code>DEBUG←0</code>	Set to 0 to reduce the degree of error trapping when troubleshooting. See <i>Debugging</i> .
<code>Transfer←0</code>	States whether slow processes will transfer a copy of their work to faster processes once these processes are complete.
<code>WS←'parallel'</code>	Name of the workspace to load to create slave server tasks
<code>WSPATH←''</code>	Path to WS if different to calling workspace
<code>Exe←'dyalogrt'</code>	Name of the Dyalog executable to use to create slaves. See <i>Debugging</i> for more details.
<code>ExePath←''</code>	Alternate Path to Exe, if empty the environment variable <code>DYALOG</code> is used.
<code>PortRange←14400 14420</code>	The range of TCP/IP port numbers usable for slave servers
<code>nSlaves←-1</code>	Number of slave processes to start by default. <code>-1</code> means one for each physical processor in the machine.
<code>nThreads←0</code>	Max number of processes to actually use (should be less than or equal to the number of slaves started). <code>0</code> means all.
<code>Disabled←0</code>	For debugging purposes, set this to 1 to stop <code>P.Each</code> from using slaves; it will run all expressions “locally”
<code>OnError←'Stop'</code>	By default, <code>P.Each</code> abandons the entire execution on the first error (in the same way as the primitive). Alternatives are <code>'Continue'</code> and <code>'Repro'</code> . See <i>Debugging</i> for more information.

<code>ProgressAfter+3000</code>	How long to wait before producing a progress bar. A value of 0 switches the progress bar off.
<code>MaxWait+60000</code>	The maximum time to wait (in milliseconds) with no response from any slaves before a call to <code>P.Each</code> should be abandoned and the slaves “recycled”.
<code>BlockSizeRange+1 10000</code>	The minimum and maximum block size for <code>P.Each</code> operations. See <i>Tuning</i> for more information.
<code>BlockSizeTarget+2000</code>	The number of milliseconds considered to be a good “unit of work” when computing the block size. See <i>Tuning</i> .
<code>RemoteServers+0 4p 'server' 5001 ~1 ''</code>	An IP address or name and a port number, for each server (see the section on <i>Remote Servers</i>)
<code>SharedFolder+''</code>	The name of a folder which is shared with remote servers; used to save the current workspace when “cloning”

APPENDIX II

FUNCTION REFERENCE

This section lists the functions in the P namespace.

flag←ALLOK

Returns 1 if there were no errors encountered during the most recent call to **P.Each**.

Example: See **Errors**.

CopyState names [nfiles] [cfiles]

Copies the named variables, functions or operators to all slave processes. **names** must be a matrix of names. **nfiles** and **cfiles** are two optional Boolean flags; if set the control whether the current set of native and component file ties are replicated in each slave process.

Example:

```
P.CopyState(t'Foo' 'Data')1 0
```

Defaults

Re-initialises all parameters to default settings (see Appendix I).

errors←Errors

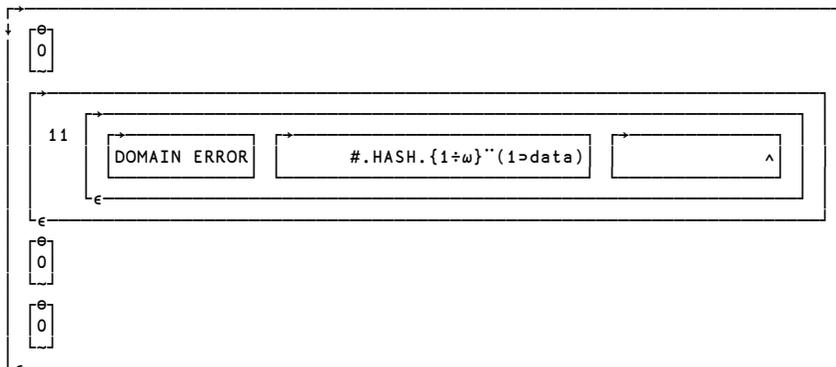
If `Fork.OnError←'Continue'`, this function returns the list of errors encountered during the last call to `Each`. For each element in the last call, `Errors` returns an empty vector of that element was processed successfully, or a 2-element vector containing (`EN` `DM`) for elements a slave task failed to process. See *Debugging* for more details.

Example:

```

    Fork.OnError←'Continue'
    {1÷ω}P.Each 1 0 3 4
1    0.3333333333 0.25
    Parallel.ALLOK
0
    ]disp ;Parallel.Errors

```



Gather names

Collects the global “results” from all the slave processes after using **Each**, saving the results in identically named global variables in the active workspace. If the right argument is a simple vector containing names separated by spaces, or a vector of vectors, the named variables from each slave workspace are concatenated together on the first dimension using the function `⌞`.

The right argument can also be a two-column matrix with variable names in the first column and aggregation functions in the second column. For example:

```
P.Gather 3 2p'Customers' '⌞', 'Products' '⌞', 'Sales' '+'
```

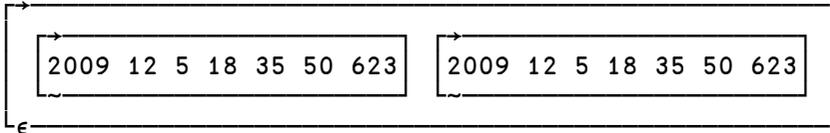
... would collect the values of three variables from each slave process: **Customers** and **Products** would be concatenated together on the 1st dimension, and **Sales** would be added together.

Get expr

Executes **expr** in each slave process, and returns a vector of results with as many elements as there are slaves. Any errors are trapped and signaled back.

For example:

```
Parallel.Get '1÷0'
DOMAIN ERROR
Parallel.Get'1÷0'
^
]disp Parallel.Get '⌞TS'
```



Init spaces

Terminates any existing slave processes, and starts a new set. A slave is started for each element of the right argument, which defines the “content” that the process will have: each element can either be a workspace name or a reference to a namespace.

If the right argument includes the namespace #, this is treated as a special case: The current workspace is saved to a temporary file and the slave(s) copy the workspace on startup, so they become “clones” of the current process. Other namespaces are passed to the slave in `□OR` format.

If the right argument is an empty vector, all available processors will be used and initialized using # (“cloning” the current workspace). If the right argument is an integer scalar, that number of processes will be started as clones.

Examples:

```
P.Init '' A Clone workspace as many times as possible
P.Init P.ProcessorCount A Clone using local cores only
P.Init 3p<□WSID A Make 3 copies of this WS as saved
```

n←ProcessorCount

Returns the number of processors available in the local machine.
For an example, see `P.State`.

Reset force

Terminates all current slave processes. If `force=0`, it asks each slave to shut itself down. If `force=1`, it simply terminates all existing processes (except the current process) which are based on the executable named in the variable `Fork.Exe`. Forcing a shutdown should only be done as a last resort.

[values] Set names

With no left argument, `P.Set` simply transfers named objects from the current workspace to all active slave processes. Example:

```
P.Set 'MyFn DATA'
```

When a left argument is provided, there must be as many elements in the left argument as there are slave processes – and only one variable name in the right argument. The variable is set to a different value in each slave process. Example:

```
(ιP.SlaveCount) P.Set 'SLAVEID'
```

The above statement would allow each slave to uniquely identify itself using the global variable `SLAVEID`.

n←SlaveCount

Returns the current number of active slave processes. For an example, see `State`.

State

Returns information about the available hosts and processors:

```
P.State
Hosts:
  Machine           Port  Cores
  Local             5001   2
  bree.dyalog.bramley 5001   4

Active processes:
#  Host           Port  State
1  Local          14400 Idle
2  Local          14401 Idle
3  bree.dyalog.bramley 14400 Idle
4  bree.dyalog.bramley 14401 Idle
5  bree.dyalog.bramley 14402 Idle
6  bree.dyalog.bramley 14403 Idle

P.ProcessorCount A # Local Processors
2

P.SlaveCount A # of Active Slaves
6
```