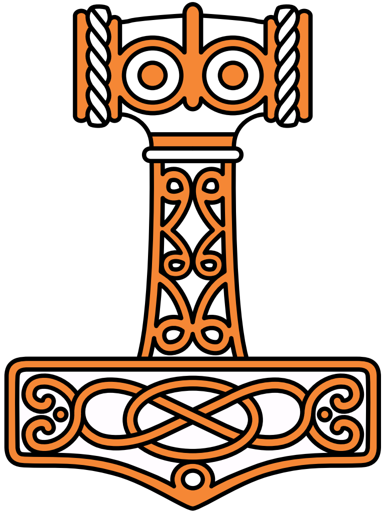
Olhão 2022

Futures and Isolates

(TP2)

Morten Kromberg



Goals

- Give an overview of Futures and Isolates
- Discuss the implementation and configuration options
- Demonstrate how to troubleshoot and debug applications which use isolates
- Discuss how to determine whether a given application is likely to speed up…
- If we have time, experiment with parallelising own code (did anyone bring something to test)?

NB: Mostly Repeat of Dyalog'14

- Nothing fundamental has changed
- There is no fundamental change in functionality
  - Significant usability enhancements and better utilities
- It now works quite reliably in all supported versions of APL
  - This was not the case in 2014

The Plan

Six sessions of 10-minute intro + 20 minutes experimentation

13:30-14:30

- Introduction: What are futures and isolates?
- Errors, Tracking progress, Interrupts

14:45-15:45

- Operator models
- Configuration Options

16:00-17:00

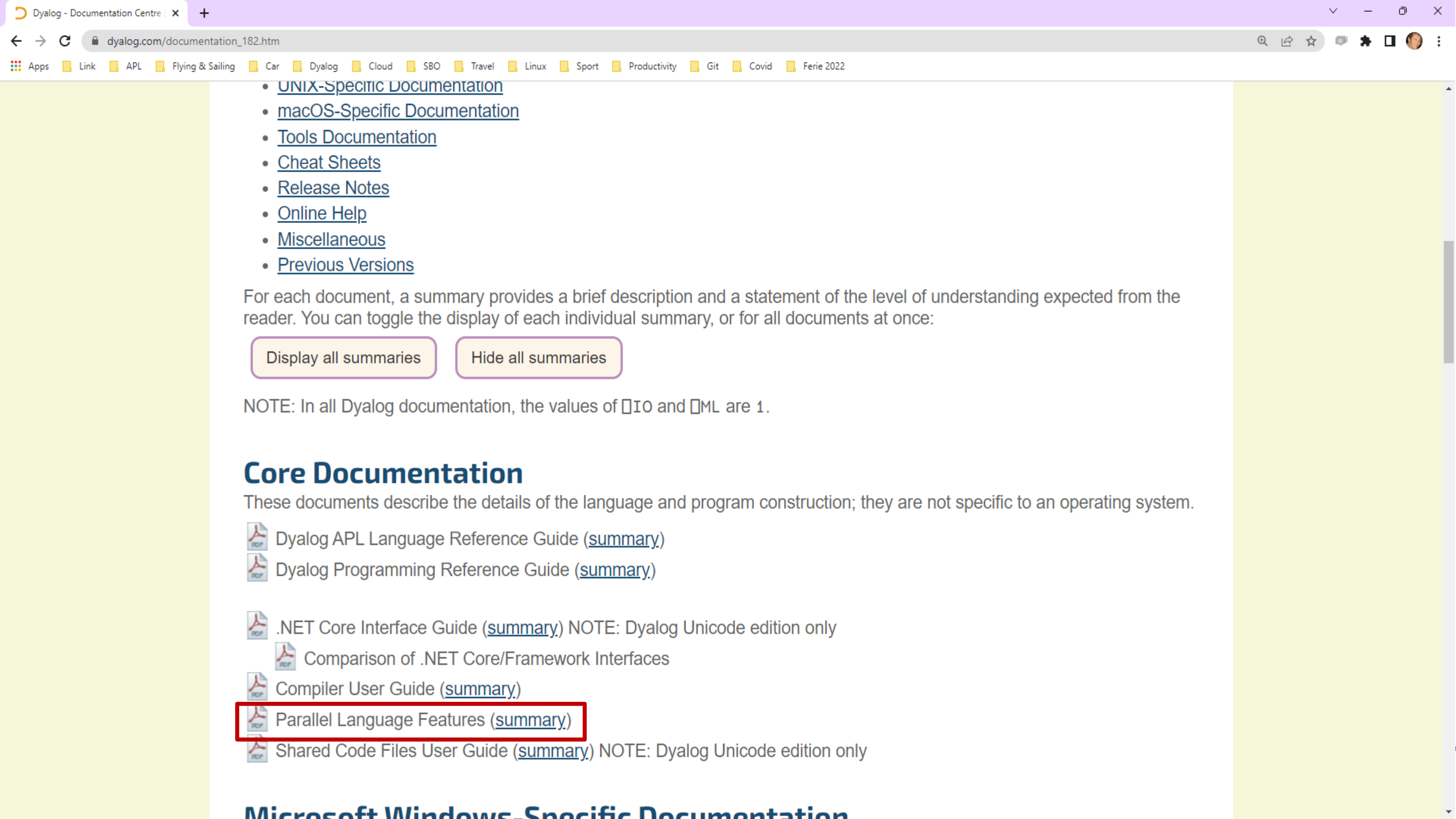
- Debugging & Troubleshooting
- Performance - when and how to use isolates in practice

Materials

Materials used can be found in

https://github.com/dyalog-training/2022-TP2

- Unzip the latest release, or
- Copy the folder 2022-TP2 from the USB drive
- Also open a tab on https://docs.dyalog.com/latest/Parallel%20Language%20Features.pdf



- UNIX-Specific Documentation
- macOS-Specific Documentation
- Tools Documentation
- Cheat Sheets
- Release Notes
- Online Help
- Miscellaneous
- Previous Versions

For each document, a summary provides a brief description and a statement of the level of understanding expected from the reader. You can toggle the display of each individual summary, or for all documents at once:

Display all summaries   Hide all summaries

NOTE: In all Dyalog documentation, the values of ⎕IO and ⎕ML are 1.

Core Documentation

These documents describe the details of the language and program construction; they are not specific to an operating system.

Dyalog APL Language Reference Guide (summary)

Dyalog Programming Reference Guide (summary)

.NET Core Interface Guide (summary) NOTE: Dyalog Unicode edition only

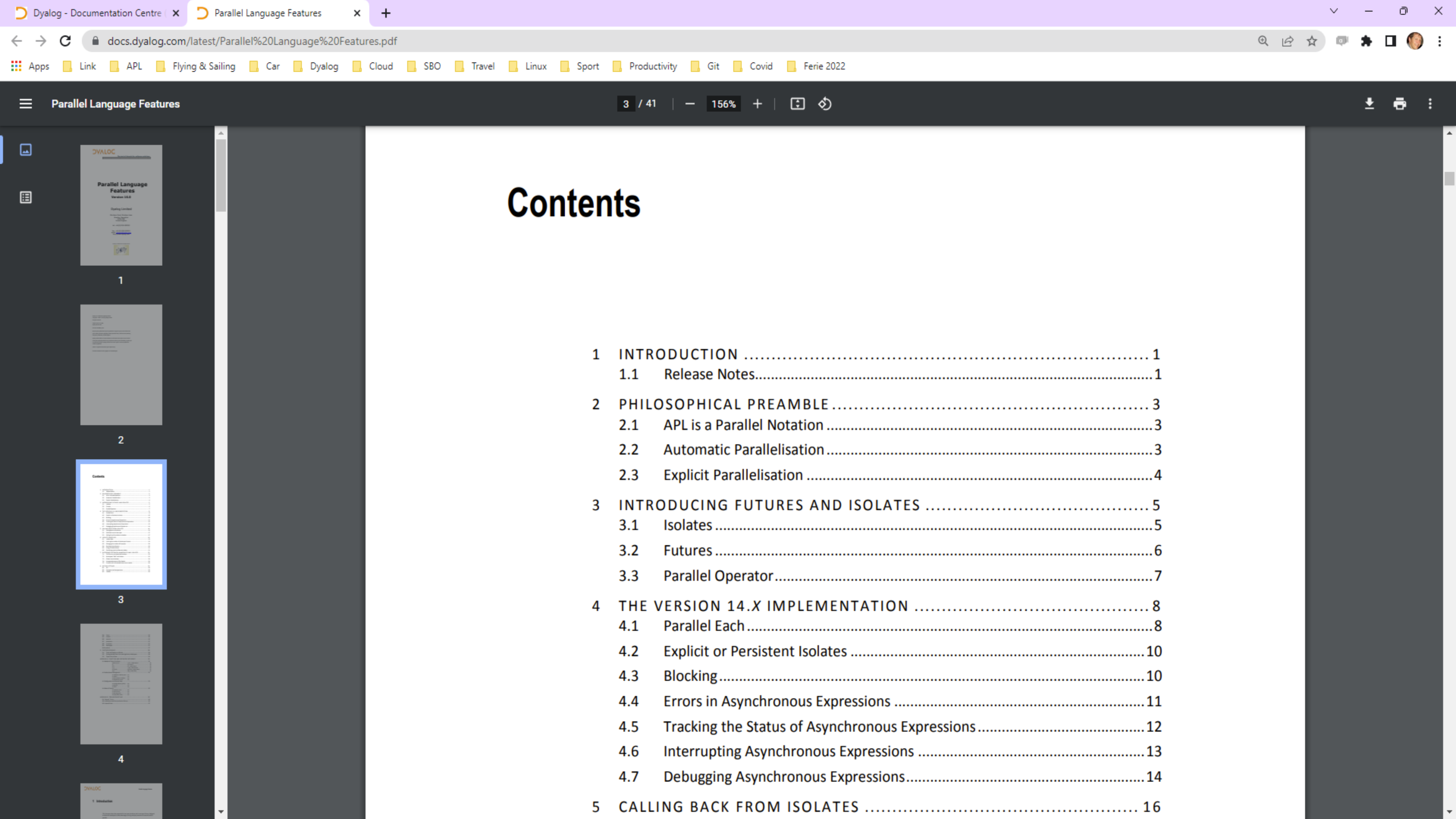
Comparison of .NET Core/Framework Interfaces

Compiler User Guide (summary)

Parallel Language Features (summary)

Shared Code Files User Guide (summary) NOTE: Dyalog Unicode edition only

Microsoft Windows-Specific Documentation



Dyalog - Documentation Centre
Parallel Language Features
docs.dyalog.com/latest/Parallel%20Language%20Features.pdf
Apps
Link
APL
Flying & Sailing
Car
Dyalog
Cloud
SBO
Travel
Linux
Sport
Productivity
Git
Covid
Ferie 2022
Parallel Language Features
3 / 41
156%
1
2
3
4

Contents

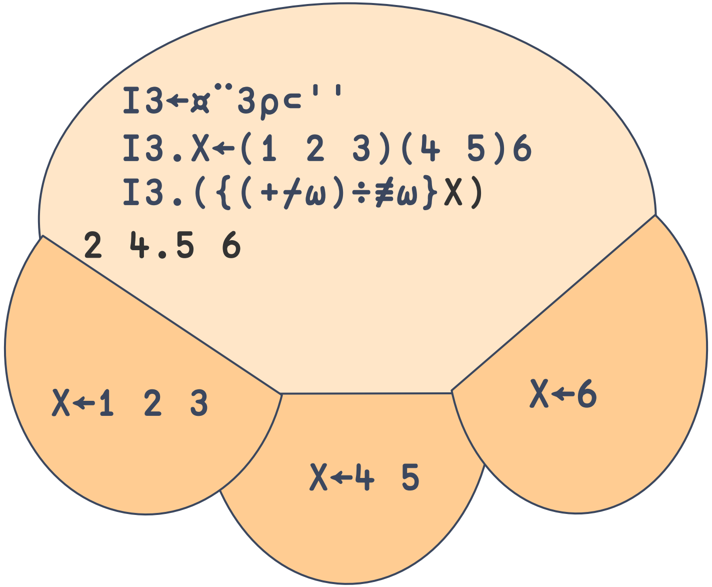
Futures and Isolates

- Goal: Allow the APL user to explicitly express parallelism in a natural way
- In the interpreter, futures and isolates enable coarse-grained *task* parallelism
  - Tasks with a duration of at least 100ms
- In a compiler, futures can be used to express fine-grained *data* parallelism

Isolates

- An ***Isolate*** tastes, smells, looks like a Dyalog namespace, except that...
- Expressions executed ***in the isolate*** run in a separate process from the main interpreter thread (”in parallel”)

Isolates in Action



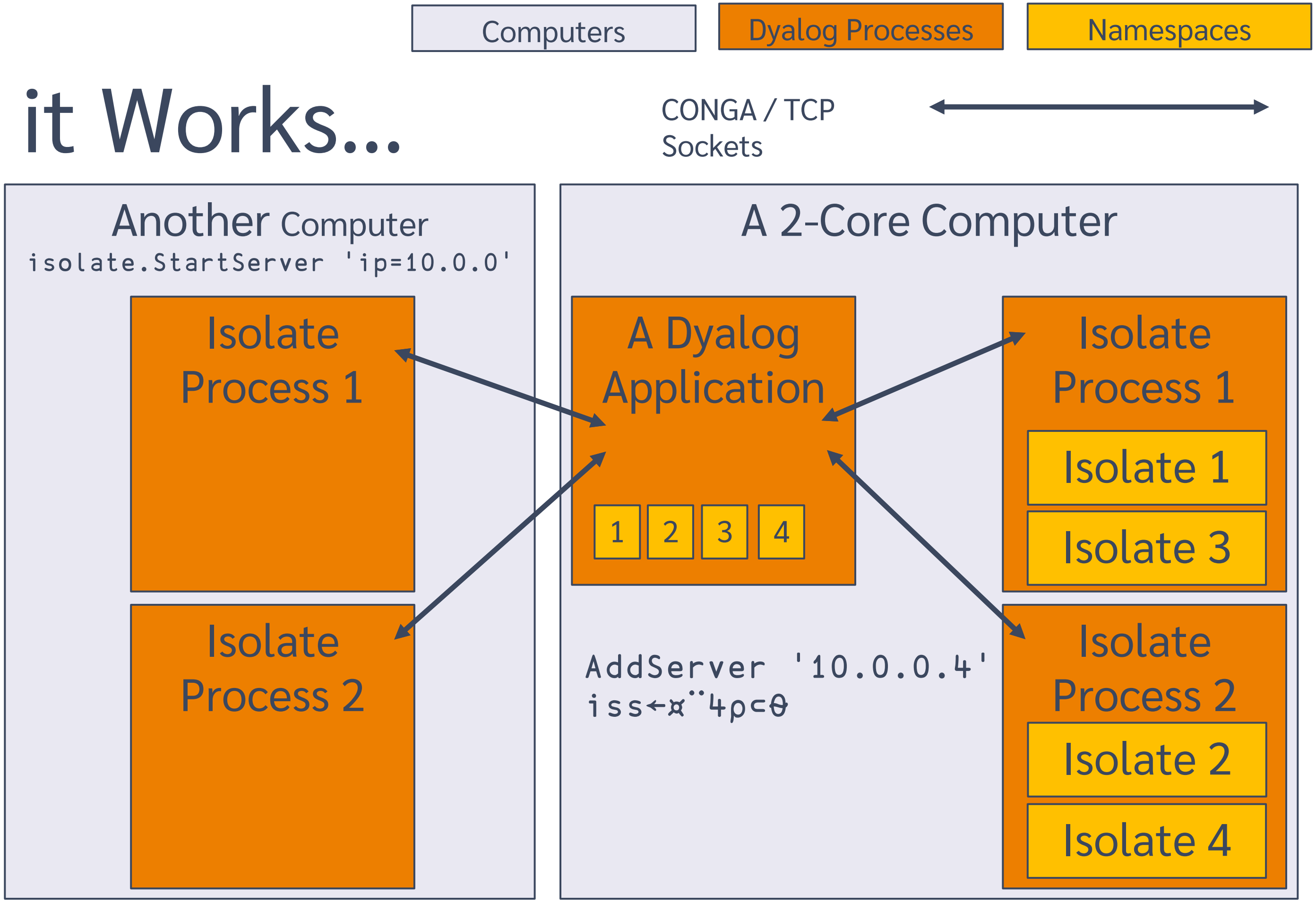
```
  I3←¤¨3⍴⊂''
  I3.X←(1 2 3)(4 5)6
  I3.({(+⌿⍵)÷≢⍵}X)
2 4.5 6
```

```
X←1 2 3
```

```
X←4 5
```

```
X←6
```

How it Works...



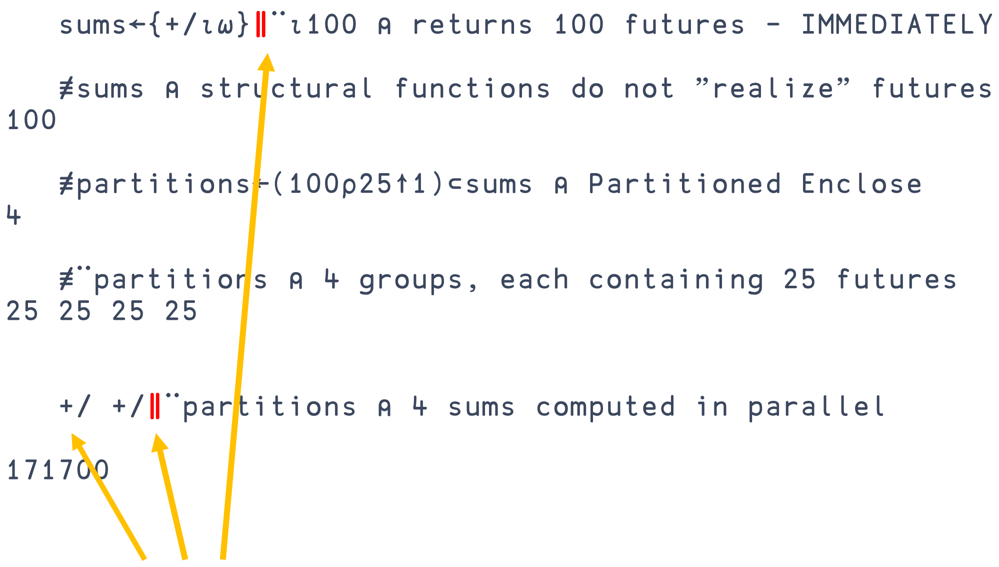
```
isolate.StartServer 'ip=10.0.0'
```

```
AddServer '10.0.0.4'
iss←¤¨4⍴⊂⍬
```

Futures

- The result of an expression executed in an Isolate is a ***Future***
- Futures can be passed as arguments to functions without blocking
- Structural functions can work on arrays containing futures without blocking
- Primitives which need to reference the ***value*** will block

The Parallel Operator ‖



```
   sums←{+/⍳⍵}‖¨⍳100 ⍝ returns 100 futures - IMMEDIATELY

   ≢sums ⍝ structural functions do not ”realize” futures
100

   ≢partitions←(100⍴25↑1)⊂sums ⍝ Partitioned Enclose
4

   ≢¨partitions ⍝ 4 groups, each containing 25 futures
25 25 25 25


   +/ +/‖¨partitions ⍝ 4 sums computed in parallel

171700
```

(We used 1+4+100 parallel threads to compute the end result)

Monadic operator *parallel* (‖) derives a function which:

- creates an empty isolate
- executes the operand inside the isolate
- returns a future (and discards the isolate)

Deterministic Parallelism

Inserting or removing Parallel operators does not change the meaning of the code. Thus, parallelism does not interfere with the notation.

```
   sums←{+/⍳⍵}∥¨⍳100
   partitions←(100⍴25↑1)⊂sums
   +/+/∥¨partitions
171700
```

(as long as your functions have no side effects)

(… and there are no errors)

Session 1 Summary

- Isolates can be created using `isolate.New`
  … or `ø` for (really) short.
- The right argument can be
  - A vector of vectors of names to be copied
  - A namespace reference to be cloned
  - A simple vector containing a workspace name to `⎕CY`
- An isolate looks, tastes, feels and smells a lot like a namespace

ø will one day become primitive ¤

Some Restrictions

- An expression executed in an isolate MUST return a result
  - The result may not be a Function or a Class.
  - If you pass namespace *refs* (either way), the spaces will be ***copied***. Actual *refs* between processes are not possible.
  - Shy results are emboldened by being futures.

Spelling

Proposed Primitive	APL model in isolate.dws	Alternative Long Form	
¤	ø	`isolate.New`	
‖	`II`	`isolate.ll`	
‖¨	`IÏ`	`isolate.llEach`	
‖⌸	`IIĐ`	`Isolate.llKey`	Note 1
‖⍤	`IIö`	`Isolate.llRank`	Note 1
∘.f‖	`o_II`	`Isolate.llOuter`	

Note 1: the models of Key and Rank omit the implicit "mix", as this would force futures to be materialised

Parallel or Async?

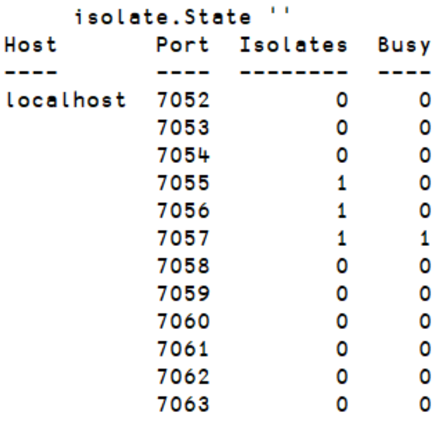
- You don't have to be doing lots of ***identical*** things in parallel
- You could be doing quite different things asynchronously

Session 1 Exercises

- Verify that you can create an isolate using `ø` or `isolate.New`
- Create a vector of isolates, distribute data across the elements. Compute something in parallel.
- Practice initialising isolates from various sources:
  - a namespace
  - a workspace (eg dfns.dws)
  - a list of names

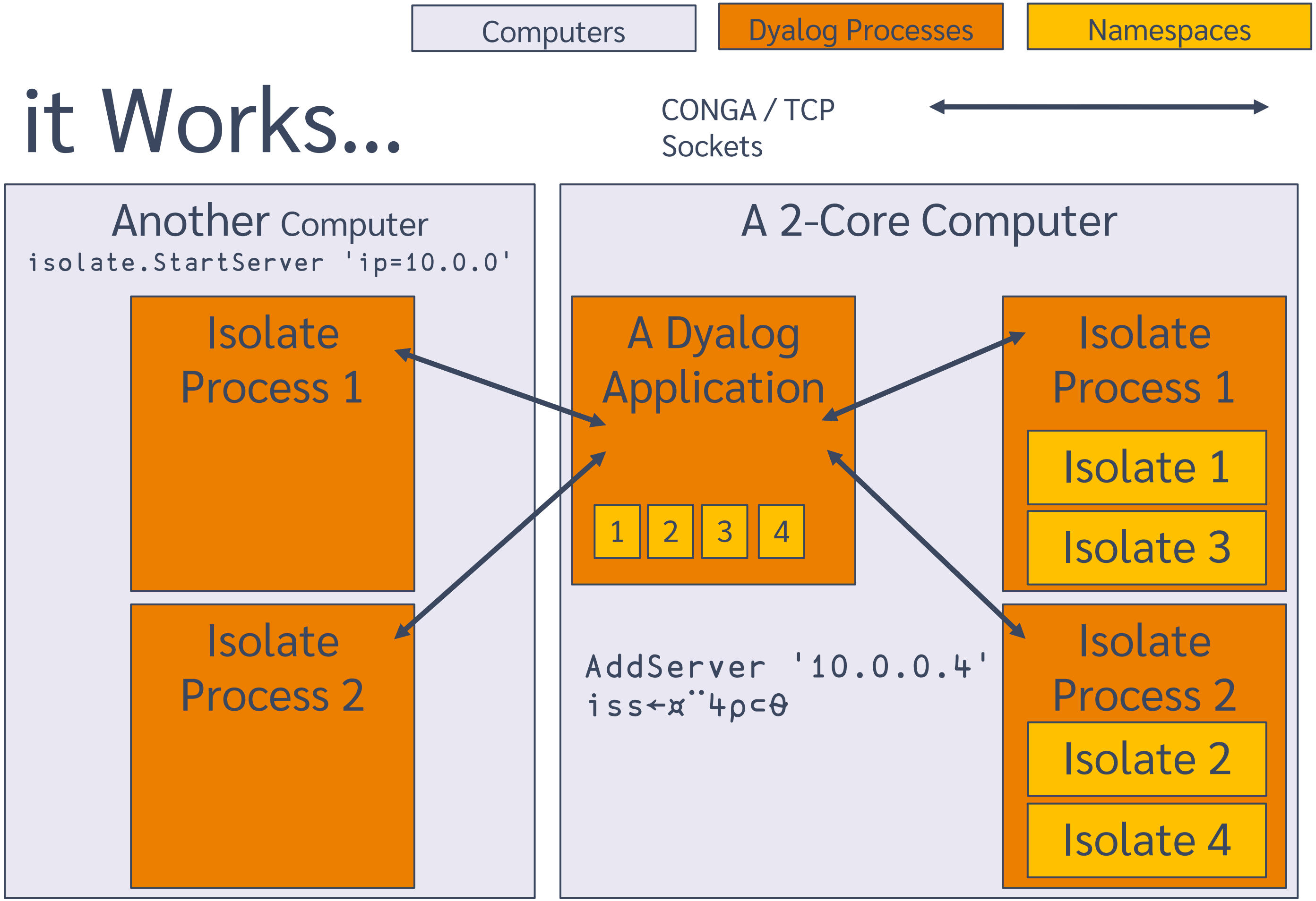
Session 2 Summary (1/2)

- **The Result** of ANY expression executed in an isolate is a ***future***
  - The interpreter will block on a future when it needs to know the value and it is not yet available
  - Structural functions can manipulate array of futures without blocking (no need to know values)
- **Errors** are signalled when an attempt is made to USE data, not when the error occurs
  - If you don't look at the data, errors may go completely undetected
- **Interrupting** returns control to the client, but does NOT stop the function call
  - A new call to an isolate which has not finished processing the previous request will be queued, even if you are not waiting for the result
  - However: Calls to a different isolate hosted by the same process will run in a separate thread
- **Isolate.State** can be used to check the state of all processes, how many isolates each is hosting, and how many of them are currently busy.



```
      isolate.State ''
Host           Port  Isolates  Busy
----           ----  --------  ----
localhost      7052         0     0
               7053         0     0
               7054         0     0
               7055         1     0
               7056         1     0
               7057         1     1
               7058         0     0
               7059         0     0
               7060         0     0
               7061         0     0
               7062         0     0
               7063         0     0
```

How it Works...



```
isolate.StartServer 'ip=10.0.0'
```

```
AddServer '10.0.0.4'
iss←¤¨4⍴⊂⍬
```

Session 2 Summary (2/2)

- The state of an array containing futures can be inspected using functions in isolate namespace, each of which returns a result the same size as the named array:

`Values`	Available values, with unfulfilled futures replaced with the value given as the left argument (`⎕NULL` by default)
`Available`	A Boolean array with 1 marking values which are computed.
`Failed`	A Boolean array with 1 marking futures which have encountered errors (and will not be computed).
`Running`	1s identify futures where the isolate is still running.

Session 2 Exercises

- Experiment until comfortable with the use of

  `Values Running Failed Available`

  to inspect the results of asynchronous calls. For example:

```
     isos←isolate.New¨'' '' ''
     delays←isos.⎕DL 5 10 15
     isolate.Values 'delays'
5.093  [Null]  [Null]
```

- Hint: see section 4.5 of the documentation:
  https://docs.dyalog.com/latest/Parallel%20Language%20Features.pdf
  Tracking the Status of Asynchronous Expressions

First Coffee Break

- Except coffee probably isn't available yet

Session 3 Summary

- `isolate.ll` (or `II`) is a model of the parallel operator ‖
- `isolate.llEach` (or `IÏ`) is a model of what will be ‖¨
- The parallel operator(s)
  - Create one or more empty isolates

    (in the processes which have the smallest number of pre-existing isolates)
  - Inserts a copy of the operand function into each isolate
  - Invokes the function in each isolate
  - Discards the isolates

Proposed Primitive	APL model in isolate.dws	Alternative Long Form
¤	ø	`isolate.New`
‖	`II`	`isolate.ll`
‖¨	`IÏ`	`isolate.llEach`
‖⌸	`IIÐ`	`Isolate.llKey`
‖⍤	`IIö`	`Isolate.llRank`
∘.f‖	`o_II`	`Isolate.llOuter`

- ”Classical” Dyalog threading can be used to launch a thread which will wait on an asynchronous computation while the main application thread continues

Session 3 Exercises

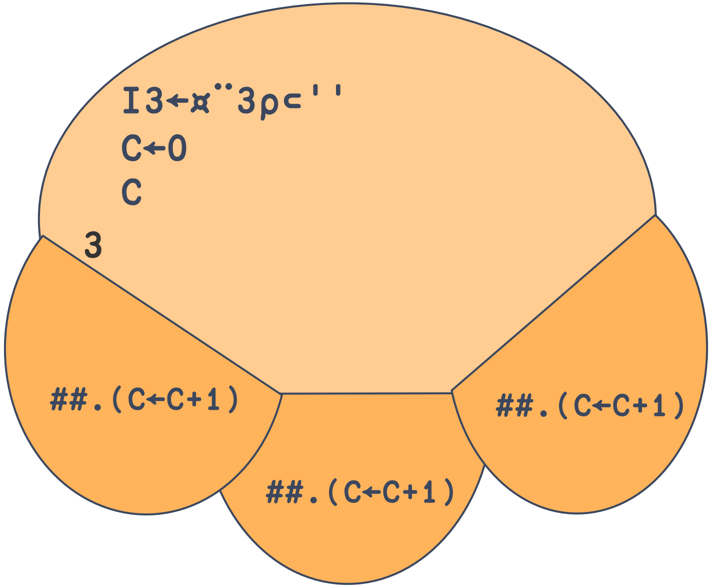
- Experiment with functions derived from `isolate.llEach` or `IÏ`. For example:

```
fooAsynch←foo isolate.llEach
```

- Unfortunately I have found that in recent versions of APL, `isolate.ll` and `II` block on monadic operands, due to a bug in ⊢ (it blocks on futures).
- Write a function which:
  - Starts an asynchronous calculation
  - Does something else
  - Displays output in the session (or if you prefer, a GUI object), when data becomes available.

Proposed Primitive	APL model in isolate.dws	Alternative Long Form
¤	ø	isolate.New
‖	II	isolate.ll
‖¨	IÏ	isolate.llEach
‖⌸	IIĐ	Isolate.llKey
‖⍤	IIö	Isolate.llRank
∘.f‖	o_II	Isolate.llOuter

Callbacks to Main Workspace



```
      I3←¤¨3⍴⊂''
      C←0
      C
3
```

```
##.(C←C+1)
```

```
##.(C←C+1)
```

```
##.(C←C+1)
```

Session 4 Summary (1/2)

- Configuration settings can be listed using
  ```
      isolate.Config ''
  ```
  - Don't enable it unless you need it, as it adds noticeable overhead to the isolate mechanism
- Callbacks from isolates to the main process are enabled using
  ```
      isolate.Config 'listen' 1
  ```
- Following a configuration change which affects how processes are started or connected, it is recommended to do a
  ```
      isolate.Reset 0
  ```
  - The right argument is currently ignored, but please use 0
    (if you care about whether application keeps working in the future).

Session 4 Summary (2/2)

- From an isolate, ## is a reference to the root of the main (client) process workspace.
- Thus, `##.XYZ` corresponds to `#.XYZ` in the main workspace (shared by all isolates)
- Calls into any isolate, including calls to ##, are serialised: Only one call is executed at a time
  - This allows function calls to perform atomic updates without adding synchronisation mechanisms

Troubleshooting #1

- `Unable to create isolate processes`
  - If not using the default isolate workspace location: Check the setting of the “workspace” option: remember that runtime interpreters may have no WSPATH
  - Switch to (`'runtime' 0`) and see whether you can spot any hints in the session output.
- `Everything is hung…`
  - Try restarting all threads. It is recommended not to use “pause threads on error” when using isolates (should not be a problem in recent versions of Dyalog APL).
  - If you have had an error or interrupt deep inside the isolate model, you may have a thread pool issue. Try (`isolate.Reset 0`).

The 2nd bullet point was important in 2014

Should hopefully not be relevant today

Session 4 Exercises

- Call an expression in an isolate which makes a callback to the root, e.g.

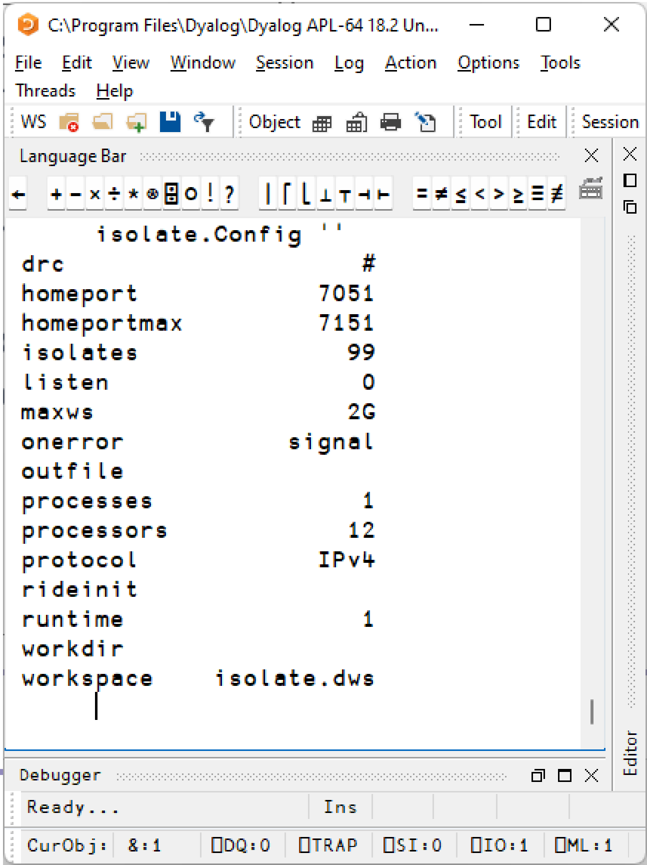
  ```
      myIS.(##.foo)
  ```

  ```
  (Hint: set 'listen' to 1)
  ```

- Repeat the call from more than one isolate in parallel

  ```
      isolates.(##.foo)
  ```

  Verify that the calls to foo are serialised.



```
      isolate.Config ''
drc                    #
homeport            7051
homeportmax         7151
isolates              99
listen                 0
maxws                 2G
onerror           signal
outfile
processes              1
processors            12
protocol            IPv4
rideinit
runtime                1
workdir
workspace    isolate.dws
```

2nd Coffee Break

Session 5 Summary

- Enable debugging with:
```
    isolate.Config 'onerror' 'debug'
```
- This will automatically select the development interpreter, rather than a runtime (regardless of the runtime configuration setting).
- Switch back with
```
    isolate.Config 'onerror' 'signal'
```
- Under Windows, the window caption of a suspended isolate process is modified to help you find it

Session 5 Summary

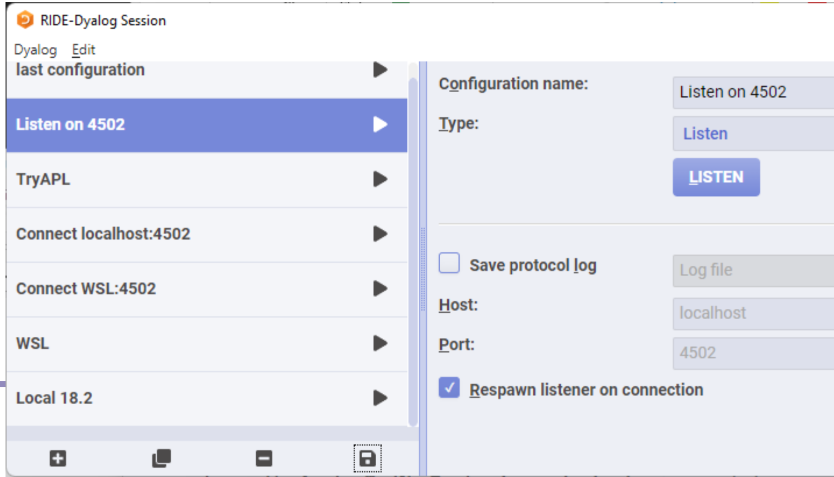
- To debug with RIDE, set:

```
    isolate.Config 'rideinit' 'POLL:address:port'
```

- A RIDE window will be opened for each isolate process, so you probably want to pretend you only have 2 processors

```
    isolate.Config 'processors' 2
```

- Set RIDE up to listen on the selected port
  - NB: Select "Respawn listener..."



Configuration name: Listen on 4502

Type: Listen

□ Save protocol log Log file

Host: localhost

Port: 4502

☑ Respawn listener on connection

Session 5 Exercises

- Put a bug in your code, and fix it inside an isolate
- If you have RIDE installed, see if you can get debugging working

Configuration Options

Option Name	Default	Description
drc	#	Location of CONGA namespace to use
homeport	7051	The lowest port number that will be used
homeportmax	7151	The highest port number to try listening on
isolates	99	Number or isolates allowed per process
listen	0	1 to allow isolates to issue callbacks to parent process
maxws	'64000'	By default, uses the same setting as the current APL session
rideinit	''	Ride configuration, typically CONNECT:ip-address:port number
Onerror	'signal'	Signal errors to the line waiting for results
processes	1	The number of processes to start per processor
processors	4	Number of processors (default determined automatically)
runtime	1	Whether to run isolates using the runtime engine
workspace	'isolate'	Workspace to load when starting new isolates

More limitations / Gotchas

- Beware of isolates sharing a process
- Don’t create excessive numbers of isolates:

```
ISOLATE ERROR: All processes are in use
      {+/⍳⍵}IÏ ⍳500
     ^
```

- Remember *refs* cannot cross process borders
  - Namespaces will always be COPIED
    e.g. `ref←is1.ns`

Troubleshooting #2

- `Warning: Ports in use…`
  - Either you have two APL sessions both using isolates
  - Or you have “zombie” isolate processes, typically created if you exit from your APL process without running the destructors
  - Currently, there is no way to kill them other than using TaskMgr

Re-using Isolates

- If you have a large number of parallel calls to make, one isolate per call may not give the highest throughput
  - You may end up with "too much for your hardware"
  - If the calls do not all take the same amount of time, some of the isolates will be idle part of the time
- Instead, it may be better to create a "reasonable" number of isolates and reuse them
- The namespace `ll` in the distributed `isolate` workspace contains operators `Each` and `EachX`, which help with this

Re-using isolates, continued...

- `ll.Each` is a monadic operator utility which creates one isolate for each processor, makes one function call to each isolate, and then re-uses them as they become available:

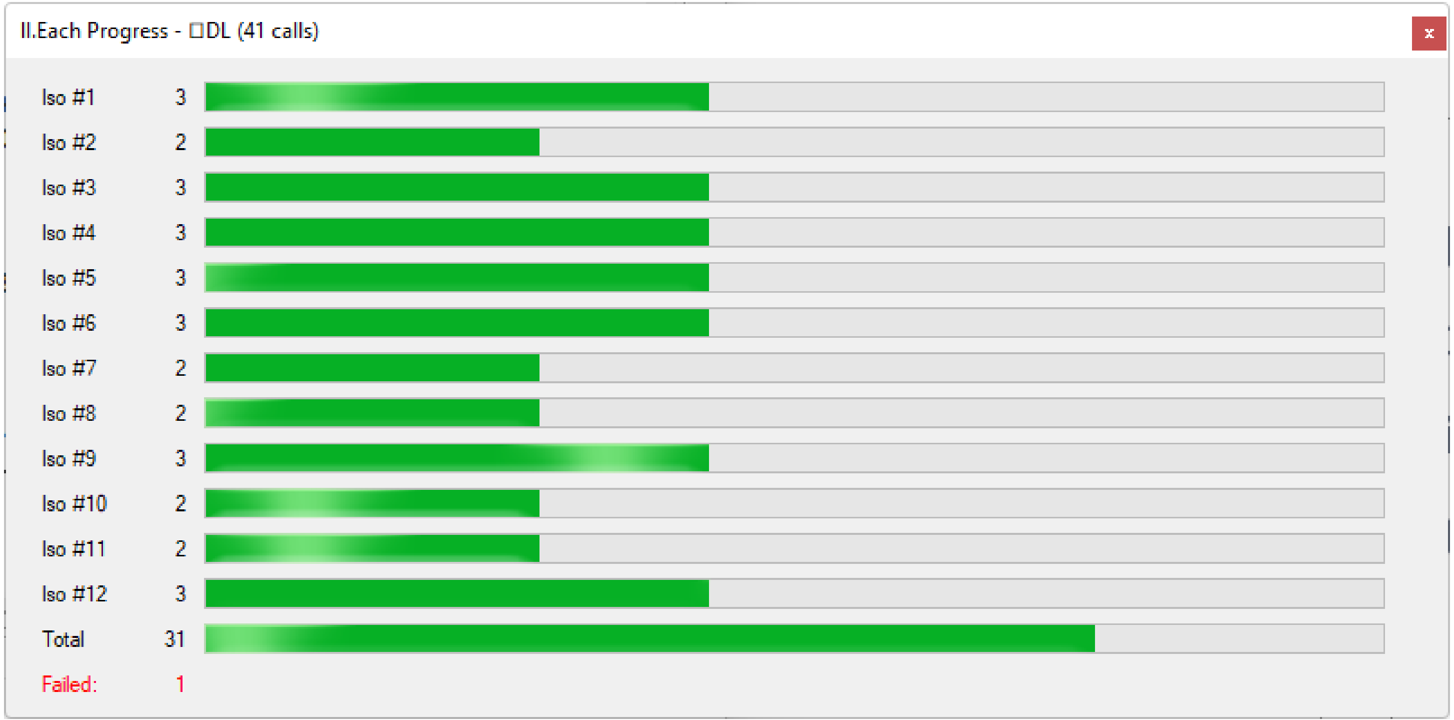
```
(⎕Dl ll.Each) 20⌽¯1,?40⍴10
```

- `ll.EachX` gives you more control: The right operand is an array of references to the isolates that you want to use, and the left argument allows you to specify a callback function to be invoked each time a result is returned, and some user-defined data.

```
iss←ø¨6⍴⊂'myws' ⍝ 6 isolates made from myws
('MyCalc' 'MyCallBackFN' 'Running MyCalc') ll.EachX iss) ⍳100
```

- If you do not provide a callback function, `EachX` will pop up a progress form... If the user closes this form, the operation will be abandoned.

EachX Progress Form



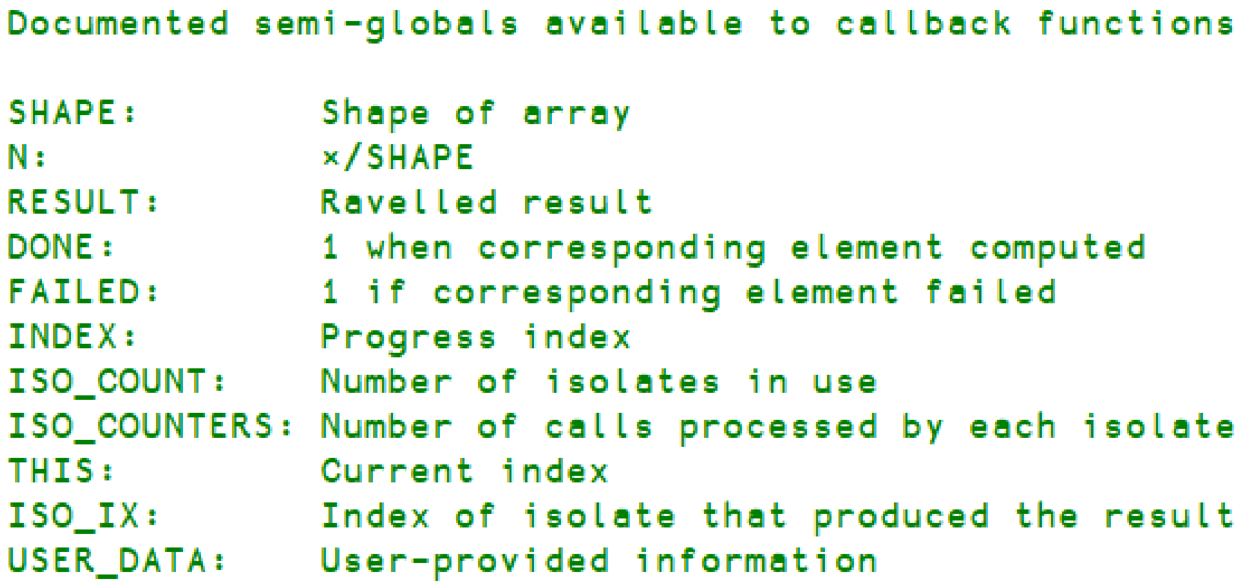
II.Each Progress - □DL (41 calls)

```
Iso #1      3
Iso #2      2
Iso #3      3
Iso #4      3
Iso #5      3
Iso #6      3
Iso #7      2
Iso #8      2
Iso #9      3
Iso #10     2
Iso #11     2
Iso #12     3
Total      31
Failed:     1
```

Unfortunately…

- The distributed version of `ll` is a quick hack, which can be improved
- The folder `Examples` in the distributed materials contains a much improved version of `ll`
- It will be in the v19.0 isolate workspace & documentation

[new] ll.EachX semi-globals



```
Documented semi-globals available to callback functions

SHAPE:         Shape of array
N:             ×/SHAPE
RESULT:        Ravelled result
DONE:          1 when corresponding element computed
FAILED:        1 if corresponding element failed
INDEX:         Progress index
ISO_COUNT:     Number of isolates in use
ISO_COUNTERS: Number of calls processed by each isolate
THIS:          Current index
ISO_IX:        Index of isolate that produced the result
USER_DATA:     User-provided information
```

Session 6 Summary (2/2)

- The left operand of `EachX` can be a two or three element vector:
  `(fn callbackfn user_data)`
- `callbackfn` is called each time a function call is completed, with a dummy right argument; it can inspect documented semi-globals and produce output
- The callback function must return 0 to continue or 1 to cancel the calculation
- If you do not supply a callback fn, a form is displayed to track progress; closing this form aborts the operation
- Deciding how to parallelise your operations (if at all) is "complicated"

Session 6 Exercises

- Test `ll.Each` and `ll.EachX`

  For example:

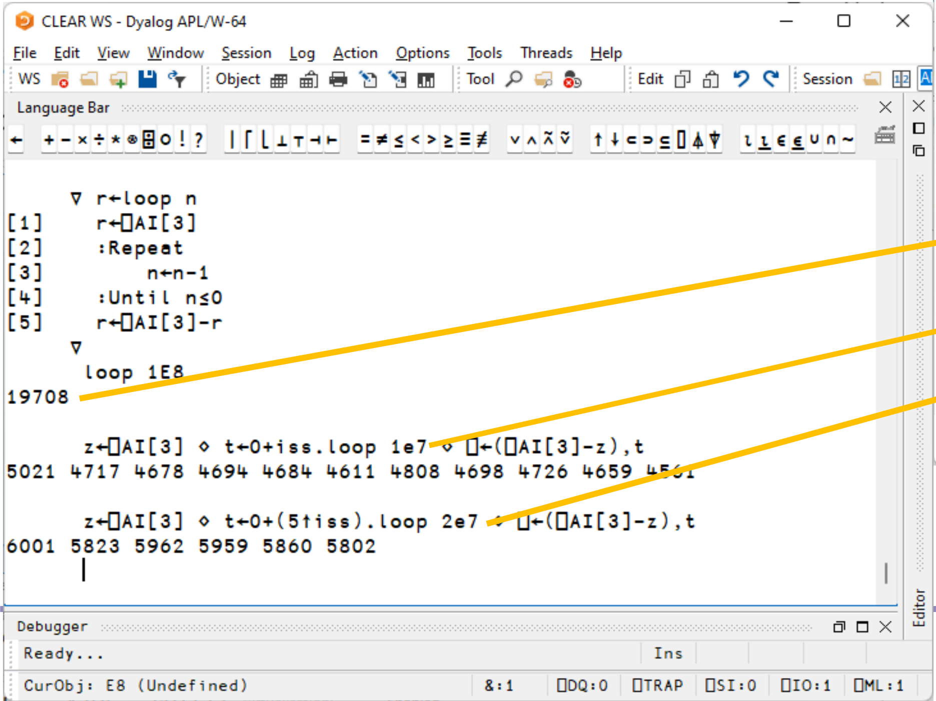
  ```
        ⎕DL ll.Each ?40⍴10
  ```

- Advanced: Write your own callback function. If you want to do this, first:

  ```
  ]link.import # [TP2]/Mandelbrot/ll.apln
  ```

Session 7 – Performance

- Let's take a closer look at what kinds of things we can actually speed up…



CLEAR WS - Dyalog APL/W-64
File Edit View Window Session Log Action Options Tools Threads Help
WS Object Tool Edit Session
Language Bar
Debugger
Editor

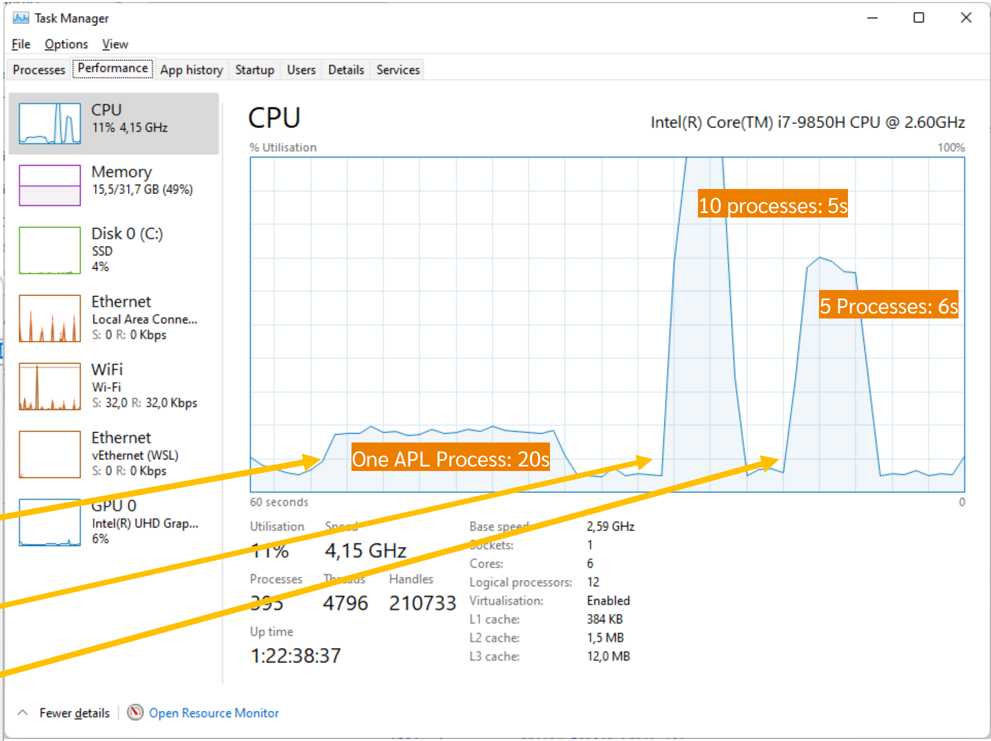
```
      ∇ r←loop n
[1]     r←⎕AI[3]
[2]     :Repeat
[3]         n←n-1
[4]     :Until n≤0
[5]     r←⎕AI[3]-r
      ∇
     loop 1E8
19708

     z←⎕AI[3] ⋄ t←0+iss.loop 1e7 ⋄ ⎕←(⎕AI[3]-z),t
5021 4717 4678 4694 4684 4611 4808 4698 4726 4659 4561

     z←⎕AI[3] ⋄ t←0+(5↑iss).loop 2e7 ⋄ ⎕←(⎕AI[3]-z),t
6001 5823 5962 5959 5860 5802
```

Ready... Ins

CurObj: E8 (Undefined) &:1 ⎕DQ:0 ⎕TRAP ⎕SI:0 ⎕IO:1 ⎕ML:1



10 processes: 5s

5 Processes: 6s

One APL Process: 20s

Utilisation 11%

Speed 4,15 GHz

Processes 395

Threads 4796

Handles 210733

Up time 1:22:38:37

Base speed: 2,59 GHz

Sockets: 1

Cores: 6

Logical processors: 12

Virtualisation: Enabled

L1 cache: 384 KB

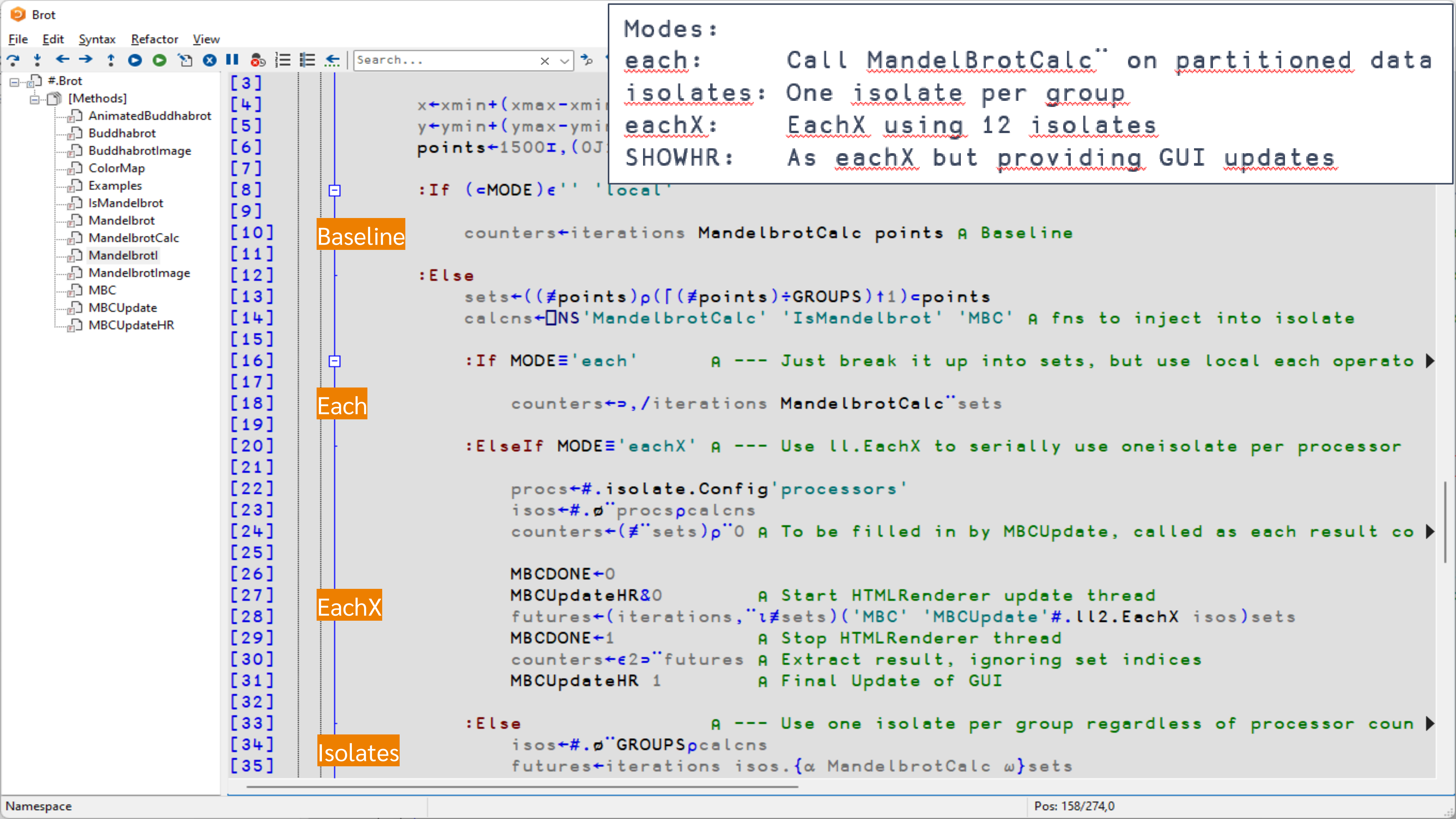
L2 cache: 1,5 MB

L3 cache: 12,0 MB

Isolates

```
Called with iterations=1000 and (⍴set)=4 million
# or elements in cur, inx starts at 4 million and reduces as points escape

         ∇ count←iterations MandelbrotCalc set;inx;cur;i;esc
    [1]       ⍝ Inner loop of Mandelbrot
    [2]       ⍝ iterations => Max nummer of iterations
    [3]       ⍝ set => complex numbers to calculate iterations for.
    [4]    cur←set
    [5]    inx←⍳≢count←(≢set)⍴iterations    ⍝ points that don't escape get maximum value
    [6]    (cur inx)←(~IsMandelbrot set)∘/¨(cur inx)  ⍝ trim points that are known not to escape
    [7]    :For i :In ⍳iterations
43% [8]        esc←4<cur×+cur                ⍝ these will never come back
 1% [9]        count[esc/inx]←i              ⍝ store iteration number at which they escaped
10% [10]       (cur inx)←(~esc)∘/¨(cur inx)  ⍝ stop computation for escaped points
    [11]       :If 0∊⍴inx ⋄ :Leave ⋄ :EndIf  ⍝ all have escaped ⋄ done
45% [12]       cur←set[inx]+×⍨cur            ⍝ Mandelbrot step : z←c+z*2
    [13]   :EndFor
    [14]
         ∇
```



```
Modes:
each:      Call MandelBrotCalc¨ on partitioned data
isolates: One isolate per group
eachX:     EachX using 12 isolates
SHOWHR:    As eachX but providing GUI updates
```

```

 x←xmin+(xmax-xmi
 y←ymin+(ymax-ymi
 points←1500ɪ,(0J

 :If (⊂MODE)∊'' 'local'

     counters←iterations MandelbrotCalc points ⍝ Baseline

 :Else
     sets←((≢points)⍴(⌈(≢points)÷GROUPS)↑1)⊂points
     calcns←⎕NS'MandelbrotCalc' 'IsMandelbrot' 'MBC' ⍝ fns to inject into isolate

     :If MODE≡'each'       ⍝ --- Just break it up into sets, but use local each operato

         counters←⊃,/iterations MandelbrotCalc¨sets

     :ElseIf MODE≡'eachX' ⍝ --- Use ll.EachX to serially use oneisolate per processor

         procs←#.isolate.Config'processors'
         isos←#.ø¨procs⍴calcns
         counters←(≢¨sets)⍴¨0 ⍝ To be filled in by MBCUpdate, called as each result co

         MBCDONE←0
         MBCUpdateHR&0          ⍝ Start HTMLRenderer update thread
         futures←(iterations,¨⍳≢sets)('MBC' 'MBCUpdate'#.ll2.EachX isos)sets
         MBCDONE←1              ⍝ Stop HTMLRenderer thread
         counters←∊2⊃¨futures ⍝ Extract result, ignoring set indices
         MBCUpdateHR 1          ⍝ Final Update of GUI

     :Else                 ⍝ --- Use one isolate per group regardless of processor coun
         isos←#.ø¨GROUPS⍴calcns
         futures←iterations isos.{⍺ MandelbrotCalc ⍵}sets
```

MBTest results

```
     MBReport z

Baseline 21.5 seconds. Speedup factors:

 Mode \ Blocks        4      12     100    1000
 each               1.1     1.3     1.5     1.4
 isolates           1.2     1.5     1.8     1.8
 eachX              1.4     1.5     1.9     2.4
 SHOWHR                                     1.4

Modes:
each:     Call MandelBrotCalc¨ on partitioned data
isolates: One isolate per group
eachX:    EachX using 12 isolates
SHOWHR:   As eachX but providing GUI updates
```

Potential Future Work - Discussion

Fundamental

- Replace model with primitives
  - Perhaps primitives only run "in process" isolates
  - Launching [remote] processes and other things that "require configuration" remains as APL code
  - See next slide
- Add ability to return functions or classes

Pragmatic

- Start-up logging
- Ability to terminate an asynchronous call
- Fault tolerance: `ll.EachX` to transfer work to remaining isolates on network failure etc
- Management mechanism for “batches” of work

Morten's Proposal for Dyadic ‖

Syntax	Name	Current Equivalent	Description
`f‖0`	Thread	`703⌶foo&`	Run foo in current ws with threads. `f` could be a .NET method.
`f‖1`	Fork	`⎕SAVE` and create isolates from ws. Similar to `⎕RUN` in SHARP APL. *	Invoke foo in forks of the current ws, **in the same process.**
`f‖⍬`	Parallel Each	`fÏ`	Current isolate model does this: invoke `f` in empty isolates
`f‖iss`	Isolate	`iss.⎕FX ⊂⎕CR f ⋄ iss.f`	Run in existing isolates

All of the above return futures

Also extend ⎕NA so ‖ (in place of &) gives a future-returning function

Futures and Isolates

- Goal: Allow the APL user to explicitly express parallelism in a ”natural” way

- How close are we?

Extra Topics

If we have time …

- Isolate servers
- Using your application workspace as the isolate host

Using Remote Servers

- Start isolate processes using StartServer:

  ```
      isolate.StartServer 'ip=192.168.0'
  ```

- This uses all the usual Config settings to decide how many processes to start, whether to use runtime, allow debugging, etc.

- As a client, you can add and remote servers using:

  ```
      isolate.AddServer 'address' ports
      isolate.RemoveServer 'address'
  ```

- Use `isolate.State ''` to monitor status.

- You can "easily" launch isolate servers in the cloud using the dyalog/dyalog docker container.
  - We will produce a dyalog/isolate contained which is suitable for launching in a scaled environment.

Using your own WS as "host"

- By default, isolate processes start by loading `ws/isolate.dws`
- We have seen how you can create isolates (namespaces) by copying a workspace into a namespace. However, you may prefer to have your code in the root (#) , perhaps even running a thread to keep your application alive in each process.
- To use your own application workspace as the base for isolate processes:

```
)COPY conga DRC
)COPY isolate isolate
```

- Modify your latent expression to call isoStart before your own application boot. For example:

```
⎕LX←'#.isolate.ynys.isoStart ⍬ ⋄ Run'
```

- Your application boot function use `isolate.isSlave` to check for this case and no start the application in that case. For example:

```
→isolate.isSlave⍴0
```