Parallel Computation Using Peach, Prank and Pouter

Morten Kromberg, Michael Hughes

Dyalog Ltd

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

mkrom@dyalog.com , michael@hughes.uk.com

Parallel programming with array processing languages

Abstract

One of the challenges currently facing software developers is to take advantage of the parallel hardware that is appearing, not only in large data centers, but also on every desktop. APL is an inherently parallel notation, which has the potential to make this relatively easy. At the lowest level, users should be able to expect the APL interpreter to distribute computations optimally across multiple cores, when evaluating expressions like:

```
1 2 3 + 4 5 6 × 7 8 9
```

… at least when the arrays are a bit larger. In fact, achieving this is not as simple as it might seem, as there are many bottlenecks in multi-processor machines which mean that efficient use of multiple cores by a "SIMD Interpreter" (so-called because each **S**ingle primitive **I**nstruction works on **M**ultiple items of **D**ata) is a significant challenge. Research also shows that the number of elements in arrays passed to primitive functions in commercial applications is typically very small [Bernecky1997], which means that parallelism at the level of individual primitive functions is unlikely to speed up typical applications very much.

Bernecky's proposal is to analyze the data flow, and compile highly efficient code which defeats the bottlenecks. In this paper, we describe an investigation into an alternative mechanism for improving parallel throughput on typical hardware available to users today, which relies on the user to explicitly identify the sections of application code that represent significant parallel opportunities, using a set of user-defined operators which could be implemented as primitives in a future APL interpreter.

These models of potential extensions to an APL interpreter are implemented by starting multiple processes which communicate using TCP/IP. They allow us to experiment with the performance characteristics of parallel execution using multiple cores in one or more co-operating machines, and the tuning parameters that may be required to optimize throughput on different hardware configurations. So far, experiments suggest that although they may require small changes to application code, the use of these operators has the potential to provide significantly more "bang for the buck" than the implementation of fine-grained parallelization in interpreters.

Introduction

In addition to the low-level parallelism embedded in most primitive functions, APL implementations offer a number of higher-level parallel constructs, which can be used to express parallel execution of derived or user-defined functions or expressions:

- The ***each*** operator (¨), which is available in virtually all modern systems
- The ***rank*** operator (⍤), implemented in SHARP APL and J (and often emulated using user-defined operators in other APL systems)

- The ***outer product*** operator (`∘.`), which is available in all systems
- The ***dot*** in Dyalog APL: when placed between an array of objects on the left and an expression on the right, dot applies the *expression* to its right to each of the objects on the left.

Each, rank, outer product and ***dot*** are different ways to express the application of primitive, derived or user-defined functions in parallel. In current APL systems, the multiple calls to user-defined functions expressed using code fragments like (`f¨data`), (`f⍤1⊢data`) or (`objects.f data`) are executed sequentially. We have experimented with the implementation of four experimental user-defined operators named `P.Each`, `P.Rank`, `P.OuterP` and `P.Dot`, each of which executes calls to user-defined functions in parallel.

Remote Namespaces

At the core of our model is the notion of a *remote namespace*; a namespace which is managed by a separate process, which might even be running on a separate machine, but which can be referenced to as if it were part of the current workspace. In a future version of APL, one could imagine a primitive called *fork* (`Ψ`), which forks the namespace passed as its right argument off as a separate process. The following hypothetical example shows how it might be possible to split a job into two separate processes in this future APL system:

```
wtdavg←{(data+.×⍵)÷⍴data} ⍝ Some hard work to do
ns1←Ψ #         ⍝ Fork the root (=entire current workspace)
ns2←Ψ #         ⍝ Fork it again
input←2 4⍴⍳8    ⍝ Two rows of data
(ns1 ns2).data←⊂[2]input  ⍝ Distribute a row to each space
(ns1 ns2).wtdavg 1 ¯1 0 2 ⍝ two parallel wtdavg calls
```

In this hypothetical interpreter, the APL system would handle the creation of the forked spaces, and the execution of expressions which "dot into" the remote spaces (using the semantics for dot applied to namespaces in current Dyalog APL). This would allow APL users to seamlessly access data and call functions in the remote namespaces. At present, we have a model implemented in APL, where a user-defined function called `Y` models the `Ψ` primitive, and a user-defined operator called `Dot` does its best to allow access to the remote spaces. In the model, the above example (without the definitions of `wtdavg` and `input`) could currently be written as follows (`Y` and `Dot` have been placed in a namespace called `P` for *parallel*):

```
nss←P.Y # #       ⍝ Fork two instances of the root
(nss P.Dot 'data←')⊂[2]input  ⍝ Assign to data in each space
(nss P.Dot 'wtdavg')⊂1 ¯1 0 2 ⍝ Call function
 1.75  3.75
```

The syntax of the proposed `Ψ` function can be modeled very closely, but it is not possible to do the magic required to get close to the "seamless access" that a real dot produces using a user-defined operator. As the example shows, parts of the code fragments to be executed need to be quoted, and extra care needs to be taken in transferring data between the namespaces. If the user makes mistakes, the resulting errors may be quite confusing. At the same time, although we believe that the remote dot *will* be a useful tool in its own right which it is available at the primitive level, parallel implementations of the primitive operators *each*, *outer product* and *rank* are probably more directly applicable to parallelizing existing user applications. For this reason, once we demonstrated that the fundamental idea of a "remote Dot" was workable, we have only used it as a building block for the other remote operators, rather than proposing it as a tool for end users in its current form.

General Strategy

The general strategy employed by the operators is to use `P.Y` to initialize an optimal[1] number of remote namespaces (also referred to as "slave tasks"). These namespaces are hidden away from the end user in an internal variable. Once the slaves are set up, the operators and other utility functions which will be discussed in the following are essentially cover-functions for `P.Dot`[2].

The operators `P.Each`, `P.Rank` and `P.OuterP` allow the user to express parallel operations on arbitrarily large arrays – theoretically mapping to an arbitrarily large number of parallel processes. In practice, the number of actual slave tasks will roughly correspond to the number of available "cores". The operators map the problem to the available processes by dividing the arguments up into suitable chunks and repeatedly asking the slaves to execute pieces of the problem until all the elements of the arguments have been processed and all elements of the result have been collected. Since the main purpose of the exercise is to maximise performance, we are keen to reduce the amount of data transmission and the overhead required to manage the slave tasks.

In order to reduce the communications overhead, we do not make one call per element. After an initial set of calls which allow us to "calibrate" the problem, we decide on a suitable partition size which keeps overhead low, but avoids having to wait too long for lagging slaves to complete work at the end of a process (the default target is to have partitions which run for no less than a second).

As each partial result is received, it is inserted into the correct elements of the overall result. If a slave takes "too long" to complete processing of a unit of work, then as other slaves become free towards the end of the request, the slower processes have their work reallocated and a race is allowed to complete the task in as short a time as is possible.

Any result already computed by another slave is ignored, and a quick check at the end ensures that all parts of the result have been populated. Any empty cells in the final result are resubmitted – although this case is unlikely, it is built in as a safe-guard.

Passing Arguments

It has been very helpful to us to have a handful of real end users who wanted to use our model to solve real problems! This helped us understand that two different partitioning strategies should be used, depending on which operator is being used:

1. **Linear partition**, used by `P.Each` (¨) and `P.Rank` (⍤)
2. **Indexed partition**, used by `P.OuterP` (∘.)

Linear partition is used when each element of the data is processed only once, as in:

```
1 2 3 4 5 Foo¨ 6 7 8 9 10
```

`Foo` only processes each number once as in:

```
1 Foo 6
2 Foo 7
```
Etc.

---

[1] In practice, there is little to be gained from starting more parallel tasks than the number of physical cores available. Our model allows the user to configure the number of processes to be started on each available machine.

[2] For reasons of efficiency, we don't actually use the general-purpose `P.Dot` operator internally, but the slave processes receive the same TCP messages as if we were doing so.

In this case it is sufficient to partition each argument into corresponding chunks for each call to a slave process.

Indexed partition is used where elements of the data are reused, which is the case for outer product:

```
1 2 3 4 5 ∘.Foo 6 7 8
```

Each element in the left argument is combined with each element in the right argument:

```
(1 Foo 6)(1 Foo 7)(1 Foo 8)
(2 Foo 6)(2 Foo 7)(2 Foo 8)
```
Etc.

In this case it turns out that it is generally much more efficient to send the *entire* left and right arguments to all slaves at the start of the process, and then send indices into each array to tell the slave which items to process in each partition.

Global Data and Code

If the user-defined functions which are called applied with `P.Each`, `P.OuterP` or `P.Rank` have no side-effects, it is often sufficient to fork the current workspace and use the operators to make function calls and collect the results. In practice, it may be undesirable to fork the entire workspace, which could be very large. There may also be items of global data (or code) which change between one invocation of a parallel operator and the next – where it would be unnecessarily expensive to discard the remote namespaces and fork them all over again. To cater for these situations, a utility function called `P.Set` makes it possible to pass global variables (and functions) to the slave tasks. Used monadically, `P.Set` takes a list of names and transfers copies of the named objects to every existing slave task, for example:

```
P.Set 'PRICES RATES PriceCalc'
```

`P.Set` can also be called with an array on the left which has as many elements as there are active slaves and a single name on the right. In this case, the variable with that name is set to a different value in each slave process. For example, the following expression would assign distinct IDs to the variable `SLAVEID` in the remote namespaces.

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

Using different values for variables is not recommended for "normal use" of the operators, as it has the potential to make results depend on the particular slave which is used for a particular call to the user-defined function, something which the user cannot control. However, it can be useful for specialized applications and for debugging.

If functions have side-effects, such as modifying global variables, the function `P.Get` can be used to retrieve the results. For example, if we defined a new function which modifies a global log and returned the current length of that log:

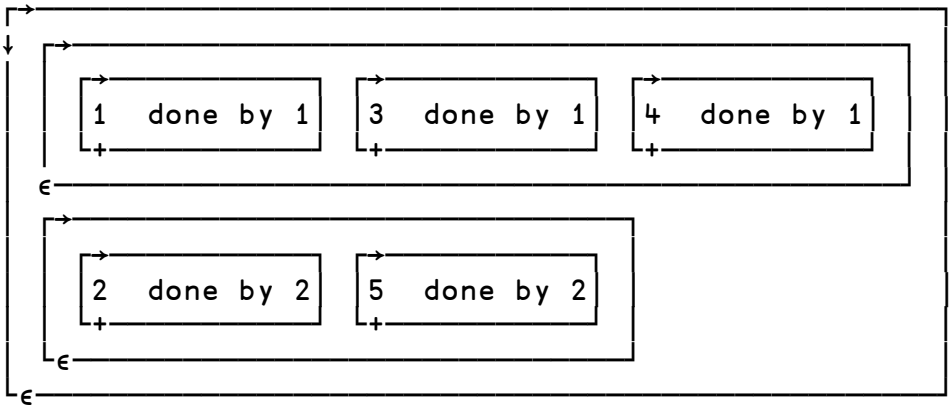
```
     ∇ r←Foo x
[1]    r←⍴LOG←LOG,⊂x,' processed by ',⍕SLAVEID
     ∇

      LOG←''           ⍝ Create empty log
      P.Set 'Foo LOG' ⍝ Transfer Foo & empty log
      (⍳P.SlaveCount) P.Set 'SLAVEID' ⍝ Set IDs
```

```
      'Foo' P.Each ⍳5 ⍝ Call Foo 5 times
 1  1  2  3  2
```

... then we can use `P.Get` to retrieve the contents of the logs:

```
      ]disp ⍪P.Get 'LOG'
```



`P.Get` returns one element per slave, containing the value of the named variable, and thus allows us to retrieve results which are stored in global variables. With a small amount of recoding, our experience is that it is fairly straightforward for an APL application developer to turn almost any loop in an existing application program into a suitable combination of fork (`P.Y`), `P.Set`, `P.Each` / `P.PouterP` / `P.Rank`, and `P.Get`.

Implementation Details

Under Microsoft Windows, slave processes on the local machine are started using the Microsoft.Net method `Process.Start` in the Microsoft.Net `System.Diagnostics` namespace. This has the advantage that we get hold of a .Net Process object, which allows us to monitor and terminate the slave tasks very easily. The Unix/Linux implementations will use shell commands, or in the case that the entire active workspace is forked, processes will be started using `4000⌶`, an I-Beam function which is available in Dyalog APL for Unix and Linux, which forks[3] the current process into two indentical images, using operating system calls which are unfortunately not available under Windows.

The right argument to the fork function `P.Y` can be a namespace reference, in which case the remote namespace will be initialized as a copy of that namespace. The most common namespaces will probably be `#` (the root – which means a complete copy of the current workspace), and `⎕NS''` (the empty space – which is subsequently populated using `P.Set`). Alternatively, the argument can be a character vector which names a workspace that should be copied into the space when the slave process starts.

Initializing a remote namespace from the root is special-cased on all platforms. Under Unix and Linux, it will use `4000⌶` (so far, all development and testing has been done under Windows). Under Windows, instead of transferring initial data via a socket, the active workspace is `⎕saved` into a folder which must be visible to all slaves, and copied by the slaves.

If multiple machines are to be used then a *relay server* task must have been started on each remote machine before they can participate. The only function of this relay server is to start the individual slave processes on its machine when requested by the single controlling task, using the me-

[3] 4000 = four k = fork, geddit?

chanisms described above. Once created, these remote slave processes communicate directly with the controlling task in exactly the same manner as the local slave processes that the controlling task has created on its own machine. Apart from starting processes, the relay server is only used to shut unresponsive slaves down again (this cannot be done by the controlling task, as it is on a different machine).

A class called `RNS` (for Remote NameSpace), implements the simulation of remote namespaces. Instances of this class are returned by the fork function `P.Y`, one per process created by fork. The namespaces created by fork are taken as an explicit argument to `P.Dot`, and as implicit arguments to the `P.Each`, `P.OuterP` and `P.Rank` operators.

Once the slave process is started, a TCP connection is opened using a tool called Conga [Conga20], which is shipped as part of any Dyalog installation. This package allows APL objects, including namespaces, to be transferred via TCP, using secure/encrypted connections if necessary. The slave receives an initialization package which tells it what to place into the remote namespace, and reports back that it is ready to do work.

Conga allows connections to multiple peers, and allows multiple requests to be queued on each connection. At present we only use one request in turn per connection but we do submit multiple simultaneous requests across several connections. The results from these tasks can return at different speeds, Conga handles all communications processing and APL is only involved when a complete packet has been received.

Most of the logic described above is encapsulated within a `Process` class, which contains all the logic required to start a process, information about the current state of the slave process, the details of the Conga connection to it, and the necessary process handles. A *destructor* function in the Process class ensures that, if the instance of the process should go out of scope (if the variable containing the list of remote namespaces is expunged or emptied), the remote tasks will automatically be shut down in a controlled fashion. Each instance of an `RNS` contains a corresponding instance of `Process`, which connects it to the actual remote namespace.

Challenges: Initialization and Unresponsive Slaves

The biggest hurdle that had to be overcome was the coordination of the different speeds in initialization and setup of the tasks. Within a single machine with multiple similar cores and very high speed intra-machine communication, initialization is straightforward. However, when machines with varying speeds are connected together - and in particular when some network connections are significantly slower than others - initialization of the slowest machines can cause a significant reduction in overall throughput.

The first implementation forced all tasks to wait until the slowest was initialized. This caused all the tasks to appear sluggish to start. Once started, although the slower tasks did not contribute as much as the faster processes, they did contribute enough to make their use worthwhile. However the slower tasks also contributed to a sluggish end as the perception was compounded by the need to wait until the last task had completed before the final result could be returned.

A better approach was to allow the faster tasks to go on ahead and allow the slower tasks to catch up. Once the slower tasks had caught up they could then contribute to the processing. Allowing the faster tasks that had completed to take over the unfinished work from the slower tasks allowed the processing to complete even faster. Towards the end of a parallel operation, this *can* lead to two slaves working on the same task; the system uses whichever result arrives first and ignores duplicates.

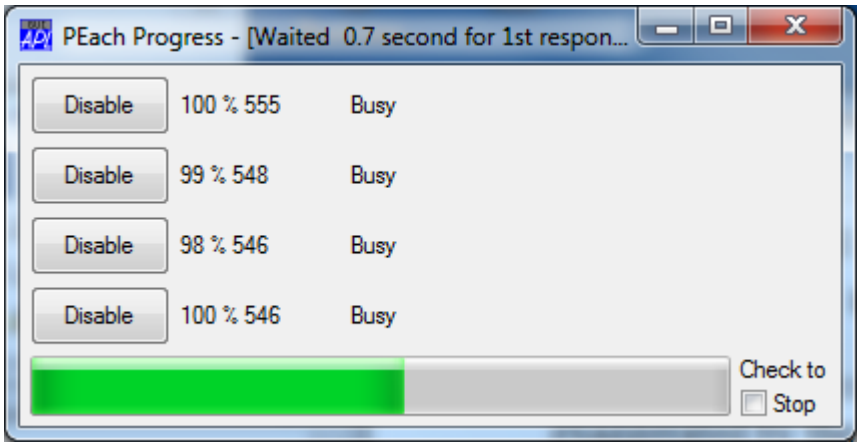
This initialization mechanism has been refined, and is still evolving. We have discovered that using shared files to transfer the same initial data to multiple slaves appears to be more efficient than multiple TCP/IP transmissions (typically containing identical data) from the controlling task.

The second big challenge is to handle failed slave processes elegantly; performance degradation is unavoidable, but the system should be resilient and not hang if the process can be completed with reasonable performance. A mechanism for enabling and disabling tasks is being prototyped. This allows a failed or apparently failed slave to be reset or to be taken out of the processing completely. The use of temporary component files as external shared storage allows (re-)initialisation of tasks to be done both faster and more easily.

When tasks run for more than a few seconds, a progress form can be displayed. This is still very functional rather than ergonomic but it suffices for the first set of trials. It is intended to be improved. There are as many rows as there are slave processes. The buttons cause the tasks to be disabled or enabled individually. The % figure gives the relative usage between tasks or the relative work done and the number following gives the actual number of packages processed.

The last part gives the current state of the task, normally this is “Free” or “Busy” but can range from “Disabled”, “Initializing”, “Setting”, etc.

The final check box to the right of the progress bar allows the entire process to be aborted. This results in an error signaled from the `P.` operator currently running.



Errors in User Code

Failures of the infrastructure, resulting in unresponsive slave tasks, obviously need to be handled as smoothly as possible. The use of remote namespaces also makes it more difficult for the developer to deal with errors in his or her *own* code, as functions may fail on a machine which the user has no physical access to. Even if there is access to the consoles on which slaves are running, a single error in user code will typically cause all slave processes to suspend, making the process of debugging and task resumption very difficult.

The model has a number of options for error handling, and these will undoubtedly need to be extended as we gain experience with the use of `P.Each` and the other operators. Currently, the controlling variable `Fork.OnError` has three possible settings:

`Stop` The default: causes the operator to stop as soon as any slave encounters an error, and signals the error to the calling environment.

`Continue` Marks all the result elements in the failed partition as invalid and continues processing. Variables `P.AllOK` and `P.Errors` make it possible to check whether the operation was completely successful, and retrieve error messages for failed elements.

`Repro` If an error occurs, the user is offered the option of reproducing the error on the client side. If the response is affirmative, the function call which failed is repeated with error traps disabled, in order to allow local debugging of the call that failed. An option to transfer patched code to all slaves before resumption of processing needs to be added.

Results

`P.Each` and `P.OuterP` have been used in a couple of real applications[4], and the results show that very significant speedups are possible using hardware which is easily available. On a dual-core laptop, results such as the following are typical:

```
]cputime {+/1=⍵∘.|⍵} ¨ ⍳10000
      CPU (avg):  6658
      Elapsed:    6708
]cputime {+/1=⍵∘.|⍵} P.Each ⍳10000
      CPU (avg):    32
      Elapsed:    3755
```

In this example, 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 real work is all being performed in the slave processes. The time reported here is the overhead of handling the communications with the slaves (in this case, roughly 1% of the total CPU time).

```
]cputime xx←(⍳1000)∘.{(+/⍳⍺)÷+/⍳⍵} (⍳1000)
      CPU (avg):  31231
      Elapsed:    31613

]cputime yy←(⍳1000){(+/⍳⍺)÷+/⍳⍵} P.OuterP (⍳1000)
      CPU (avg):  1747
      Elapsed:    4724

      xx≡yy
1
```

The overhead is a significantly higher in this example, probably because the individual tasks are very lightweight (a million operations consume less than 5 seconds), and partitioning is more complex for outer product. However, the example shows that significant speedups are possible, even for "cheap" function calls. In fact, the speedup is a bit too high for comfort in this case (6.5x on a dual core processor!), and the Dyalog team will need to take a look at the efficiency of the primitive outer product operator applied to user-defined functions (the slaves will have been using each on the indexed partitions that they receive).

The Memory Bottleneck

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

[4] `P.Rank` has been implemented for completeness, as Dyalog is considering adding the rank operator to APL in the not-too-distant future - and also because "peach and prank" looked good in the title of the paper ☺.

– 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 will have significantly different performance profiles when there are resource conflicts. You will need to experiment a little to find optimal settings for each task that you need to perform.

To illustrate, consider the following three functions, which are included as part of the test suite which is included with the distributed `parallel` workspace – in the namespace `QA` (for Quality Assurance):

```
     ∇ i←LoopTest i
[1]    :While 0<i←i-1 ⋄ :EndWhile ⍝ No memory, lots of CPU
     ∇

     ∇ r←Mixed n
[1]    r←n?n
[2]    r←+/+\+\⍦⍋⍦⍋r ⍝ Some work, but also memory scans
     ∇

     ∇ r←ThrashMemory n
[1]    r←+/⍳n ⍝ Lots of generated data, almost no "work"
     ∇
```

These functions illustrate different points on the "parallelizability scale". The function `QA.TestGeneral` runs the above functions on a right argument of `(⍳500)` using both `¨` and `P.Each` and displays a little table which records the speedup that was achieved. You should expect some variation from one run to the next, but the following numbers are typical on modern dual core machines:

```
     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 a factor of 2 for the job which consumes a lot of CPU and uses little memory[5] – but the function which spends most of its time writing integers to memory and then adding them up only speeds up very slightly. If you monitor the system, both cores will probably 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.

[5] The figure of exactly two which occurred in this particular test run is not going to happen every time.

An example of a "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.

Conclusions

The experimental user-defined operators described in this paper have shown that it is practical to put multi-core computing "at the fingertips" of APL users. Some work remains in "hardening" the tools so that users do not need to understand anything about the plumbing, but we are close to having an "industrial strength" implementation that could be used not only by anyone with a dual- or multi-core system, but also small clusters of "compute servers" on a local area network - or even via the internet.

A test suite has been kept up-to-date as changes that are made to the model, and this has been an invaluable tool during the entire project. An example of the use of every feature is included in this suite and unintended side-effects introduced by changes are relatively easily detected using the `QA.TestAll` function. In addition to allowing us to make changes with confidence that silly errors will be detected, the scripts act as a good source of documentation for how the operators can be used.

We still have some work to do to refine the failure modes, especially when machines are connected via slow networks, and when a group of machines with very different CPU and network speeds are connected together.

In terms of implementing the operators as primitives, the efficiency of the APL model seems to be excellent (overhead roughly 1%), so there is little incentive to rewrite the code in C at this point.

The `parallel` workspace will be shipped as a standard component of Dyalog version 13.0, and is available for version 12.1 at no cost, on request from support@dyalog.com.

Acknowlegements

Many thanks to Yvo Vermeylen and Brecht Dekeyser at CONAC in Brussels for providing the original impulse to get started on this work, and to Yigal Jhirad and Blay Tarnoff at Cohen and Steers in New York for helping us understand the need for a fast `P.OuterP`.

References

[Bernecky1997] Bernecky, Robert: APEX, The APL Parallel Executor, pp 14-15.

[Conga20] Conga v2.0 User Guide, http://www.dyalog.com/documentation/12.1/index.htm

Appendix A: Syntax Reference

In the following, `{lv}` indicates an optional left argument.

Current usage syntax is:

`P.Init  n`	Initializes `n` tasks ready for use. If n is an empty vector, initialized as many tasks as there are cores available on local and remote machines.
`P.Set 'ab cd'`	Transfers `ab` and `cd` to each of the tasks, they can also be function names
`lv P.Set 'data'`	Transfers one element of `lv` into `data` to each slave task.
`Data←{lv} (nss P.Dot expr) rv`	Simulates the Dyalog Dot, running the equivalent of `{lv} expr rv` in remote namespaces `nss`.
`Data←{lv} (foo P.Each) rv`	Runs function foo as if `{lv} foo¨var`
`Data←lv (foo P.Outer) rv`	Runs outer product as if `lv ∘.foo rv`
`Data←{lv} (foo P.Rank argrk) rv`	Runs rank as if `{lv}(foo ⍤argrk) rv` where argrk is the argument rank of how foo will be applied to lv and rv
`(ab cd)←P.Get 'ab cd'`	Retrieves data for `ab` and `cd` from each slave.