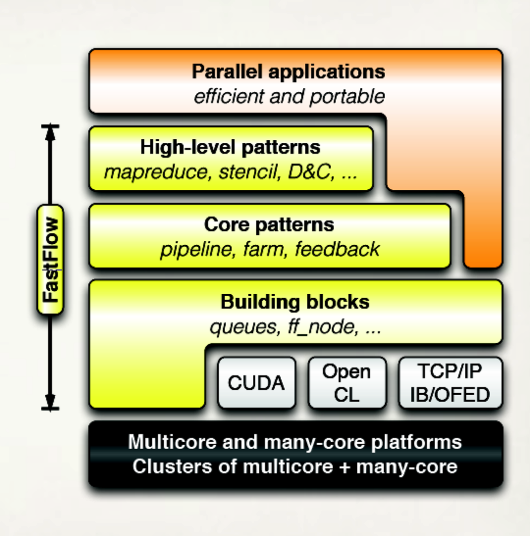
Extending FastFlow with a DSL : Why and how ? A look at some alternative approaches

Guy Tremblay
Professeur
Département d'informatique

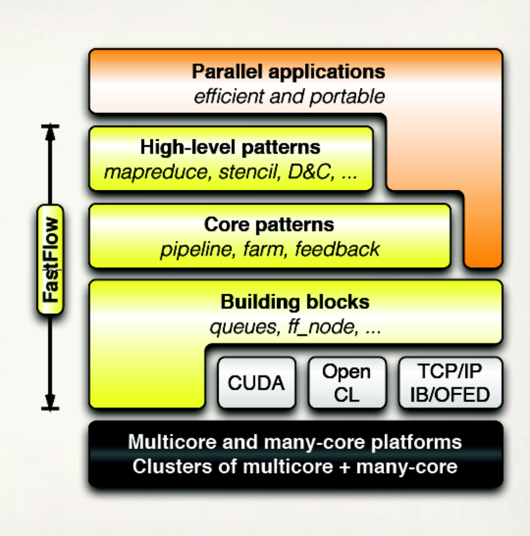
UQAM
http://www.labunix.uqam.ca/~tremblay

30 mars 2015

First element of context : FastFlow



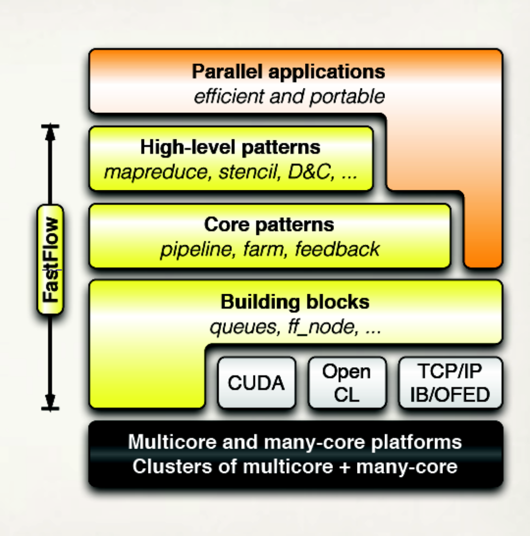
First element of context : FastFlow



Facts :

- The core patterns **can be** expressed in Ruby in a clean and simple way

First element of context : FastFlow



Facts :

- The core patterns **can be** expressed in Ruby in a clean and simple way
- The high-level patterns **could be** expressed in Ruby in a clean and simple way

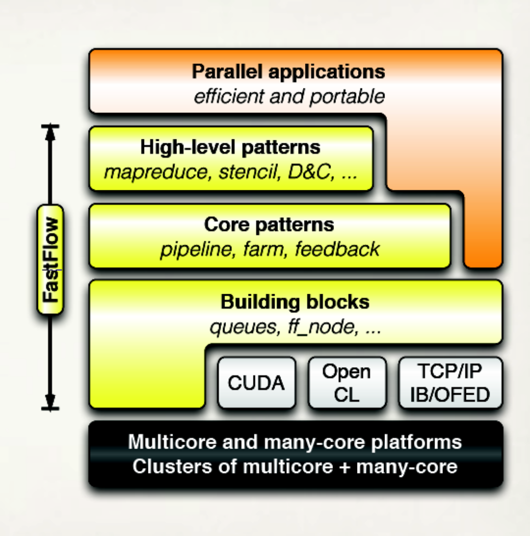
Second element of context : DSL

Domain-specific language :

A computer programming language of orangelimited expressiveness focused on a particular domain.

Source*: M. Fowler, 2011*

Two elements of context : FastFlow + DSL = ?



Key questions :

- What kinds of parallel applications can FastFlow currently deal with ?
- What other kinds of parallel applications could an extended Fastflow deal with ?

What kinds of parallel applications ?

High performance computing

- [The] use of super computers and parallel processing techniques for solving complex computational problems [...] through computer modeling, simulation and analysis.

Source: `http://www.techopedia.com/definition/4595/high-performance-computing-hpc`

Scientific workflows

- A means by which scientists can model, design, execute, debug, re-configure and re-run their analysis and visualization pipelines.

Source: `http://en.wikipedia.org/wiki/Scientific_workflow_system`

HPC applications vs. scientific workflows

Similarities

- Large number of *partially independent* tasks

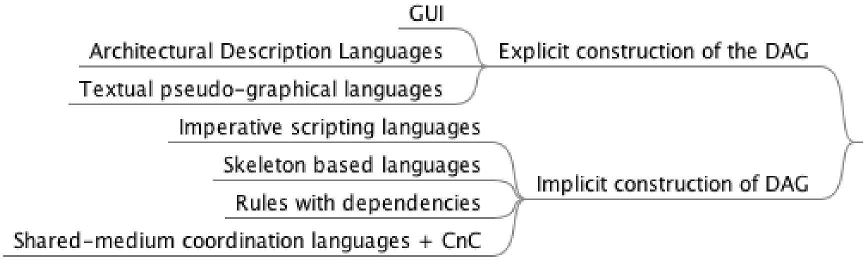
⇒ Need some form of coordination

⇒ Both often seen as DAG of tasks

Differences

- Tasks in workflows can be “very large”
  - A task can be a whole (HPC) application
  - A task may deal with files or databases, (remote) data analysis/mining services, Web services, etc.

Presentation outline



Presentation outline

- I remain neutral with respect to the kind of application — emphasis on similarities

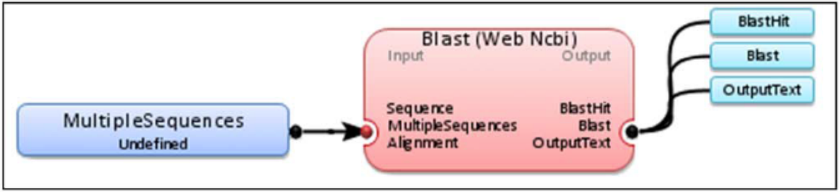
- I examine different approaches from two domains
  - Coordination languages for parallel programming
  - Scientific workflows

- Key goal = discussion and brainstorming
  - How do FastFlow differ from these approaches ?
  - Can some approaches be interesting in the FastFlow context ?

Explicit construction of DAG

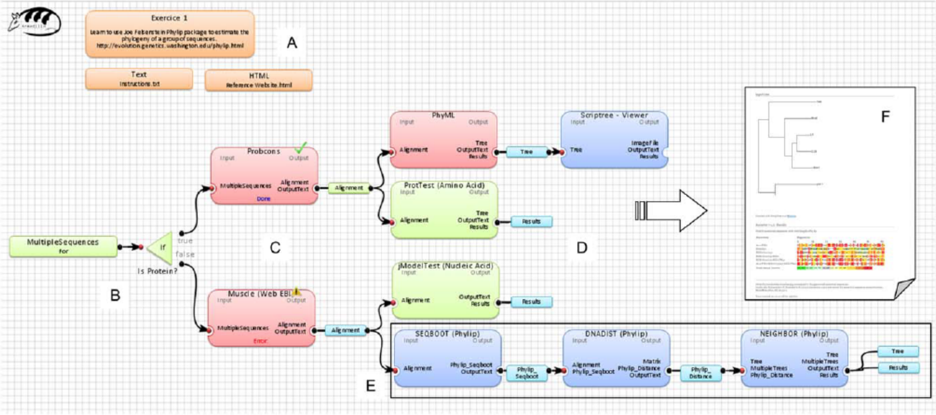
GUI : Graphical User Interfaces

Armadillo : a workflow engine for bio-pipelines



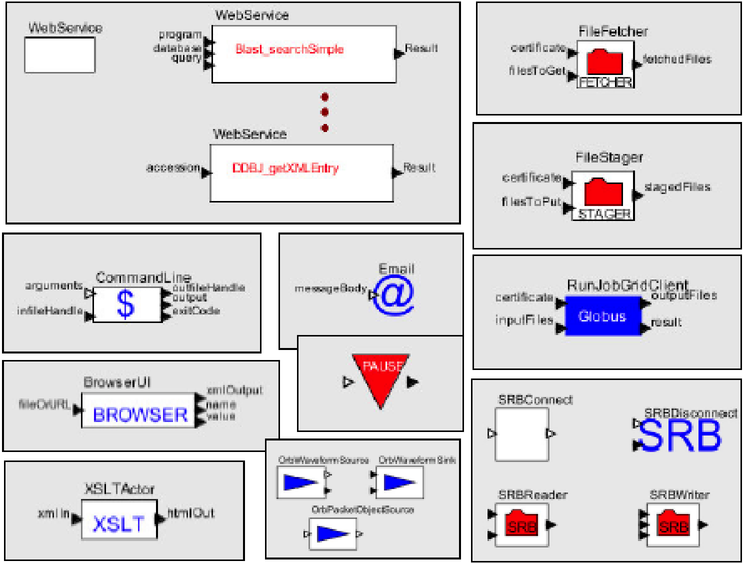
Source: "Armadillo 1.1 : An Original Workflow Platform for Designing and Conducting Phylogenetic Analysis and Simulations", Lord, Leclerc, Boc, Diallo & Makarenkov, PLOS one, 2012

Armadillo : a workflow engine for bio-pipelines (bis)



Source: "Armadillo 1.1 : An Original Workflow Platform for Designing and Conducting Phylogenetic Analysis and Simulations", Lord, Leclerc, Boc, Diallo & Makarenkov, PLOS one, 2012

Kepler : An engine for scientific workflows that provides various kinds of (complex) tasks



Kepler : An example with an SDF Director

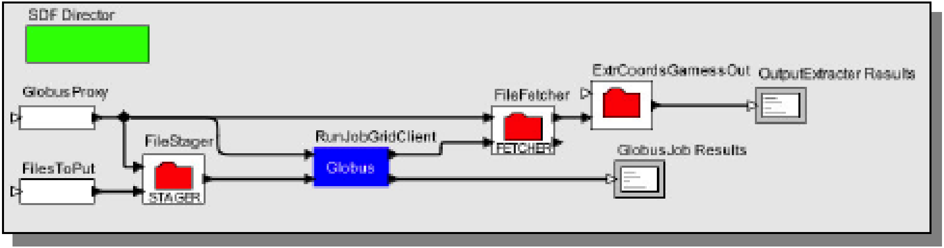
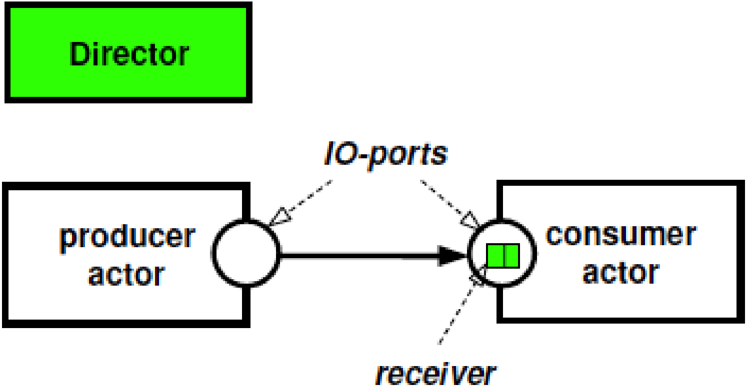


Figure 6: Grid actors and other KEPLER extensions.

Source: “Scientific Workflow Management and the Kepler System”, Ludascher & al., 2005

Kepler : The semantics of the workflow computation is customizable using directors



The director defines how actors are executed and how they communicate with one another.

Source: “Scientific Workflow Management and the Kepler System”, Ludascher & al., 2005

Kepler : Directors define and implement Models of Computation

Model of Computation (MoC)

A model of computation (MoC) is a formal abstraction of execution in a computer. [. . .] Directors are responsible for implementing particular MoCs, and thus define “orchestration semantics” for workflows.

Source: “Heterogeneous composition of models of computation, Goderis & al., 2009

Kepler provides various (pre-defined) MoCs, but the user can define new ones

- Process Network
- Static or Dynamic Dataflow
- Continuous Time
- Discrete Events
- Synchronous/Reactive
- Finite State Machines

Kepler : Abstract actor semantics

Action methods that must be implemented by actors :

`preinitialize`	
`initialize`	
`prefire`	check for firing readiness
`fire`	read/write tokens should not change state
`postfire`	can update state
`wrapup`	

Protocol :

execution ⟶ **preinitialize**, *type-check*, *run**, **wrapup**
run ⟶ **initialize**, *iteration**
iteration ⟶ **prefire**, **fire***, **postfire**

Source: “Scientific Workflow Management and the Kepler System”, Ludascher *et al.*, 2005

ADL : Architecture Description Languages

ADL = Architecture Description Language

An ADL is used to specify the structure of a system separately from its algorithmic aspects.

Source*: http ://c2.com/cgi/wiki ?ArchitectureDescriptionLanguage*

An ADL should allow a description of a software architecture in terms of components, connectors and configurations.

Source*: http ://www.igi-global.com/dictionary/architecture-description-language-adl/1423*

A lot of ADLs have been proposed : 28 pages for a list of currently known ADLs with short descriptions !

Architecturallanguages-rawdata.pdf

Ouvrir 1 / 28 86,4% Outils Remplir et signer Commentaire

Architectural languages	Objective (excerpts from papers or websites)	Links	Tool supported	Open source Tools	Commercial tools	Notes
AADL	AADL (Architecture Analysis & Design Language) is an ADL aimed to embedded real-time systems	aadl.info AADL publications	✓	OSATE ocarina	STOOD	
ABC/ADL	ADL supporting component composition. Besides the capability of architecting software systems, it provides support to the automated application generation based on SA model via mapping rules and customizable connectors	ICFEM 2002 paper	✓	ABCtool		
Acme	Acme is a simple, generic software ADL that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools.	home-page	✓	AcmeStudio		
ABACUS	ABACUS (Architecture-Based Analysis of Complex Systems) is a software package that can create multiple solution alternatives ("architectures") and then run various simulations or calculations against each alternative for metrics such as Cost, Agility, Performance and Reliability. Hierarchical 3D visualization provides an intuitive means for conceiving and communicating complex architectures.	Vendor's YouTube channel ECBS 2005 paper	✓		ABACUS	
AC2-ADL	new Aspect-Oriented ADL. AC2-ADL aims to provide a formal basis for representation of the tangling and scattering concerns and establish the software architecture with higher dependability.	ASEA 2008 paper IIS 2009 paper				
ACDL	An ADL to represent the centralized-mode architectural connection in which all	ECSA 2010 paper				

Coordination

Coordination is concerned with managing the communication which is necessary due to the distributed nature of a system [. . .] as well as with all aspects of the composition of concurrent systems.

Source: *"Coordination models and languages for parallel programming", Ciancarini & Kielmann, 1999*

Coordination is the process of building programs by gluing together active pieces.

Source: *"Coordination languages and their significance", Carriero & Gelernter, 1992*

Architecture description vs. coordination languages : It depends a lot on the viewpoint or background

Configuration and architectural description languages share the same principles with coordination languages. They view a system as comprising components and interconnections, and aim at separating structural description of components from component behaviour.

Source*: "Coordination models and languages", Papadopoulos & Arbab, 1998.*

An example in Darwin (process-oriented style) : The architecture, i.e., the structure

```
component supervisor( int w ) {
  provide result <port, double>;
  require worker <component, int, int, int>;
}

component worker( int id, int nw, int intervals ) {
  require <port, double>;
}

component calc_pi( int nw ) {
  inst#antiate
    supervisor( nw );
  bind
    worker.result -- S.result;
    S.worker -- dyn worker;
}
```

Source: “Coordination models and languages”, Papadopoulos & Arbab, 1998.

An example in Darwin (cont.) : The components, i.e., the processes

```
worker( int id, int nw, int intervals ) {
  ... Compute local value in area ...
  result.send( area );
}

supervisor( int nw ) {
  for( int i = 0; i < nw; i++ ) {
    worker.inst( i, nb, intervals );
  }

  double area = 0.0;
  for( int i = 0; i < nw; i++ ) {
    double tmp;
    result.in( tmp );
    area += tmp;
  }

  printf( "pi = %f\n", area );
}
```

An example in Rapide (process-oriented style) : The architecture

```
architecture ProdCons() return SomeType is
  Prod: Producer(100);
  Cons: Consumer;
connect
   (?n in Integer)
   Prod.Send(?n) => Cons.Receive(?n);
   Cons.Ack(?n) => Prof.Reply(?n);
end architecture ProdCons
```

Source: “Coordination models and languages”, Papadopoulos & Arbab, 1998.

An example in Rapide (cont.) : The components

```
type Producer( max: Positive ) is interface
  action out Send( n: Integer );
  action in  Reply( n: Integer );
behavior
  Start => Send(0);
  (?x in Integer) Reply(?x) where ?x < max => Send(?x+1);
end Producer;


type Consumer is interface
  action out Receive( n: Integer );
  action in  Ack( n: Integer );
behavior
  (?x in Integer) Receive(?x) => Ack(?x);
end Consumer;
```

Source: “Coordination models and languages”, Papadopoulos & Arbab, 1998.

An example in Manifold : Processes deal with streams. . . but also with events

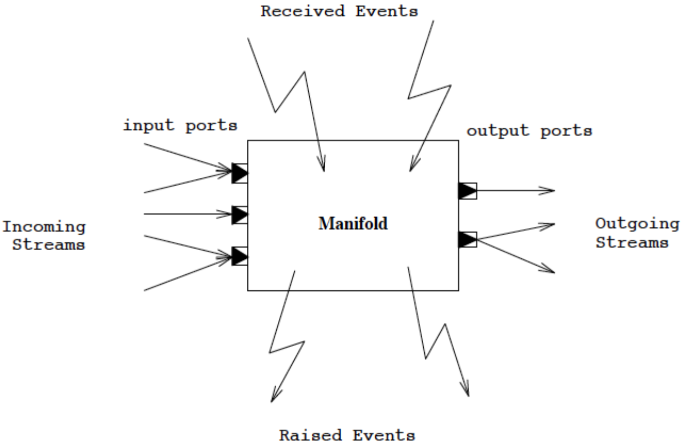


Figure 1: The model of a process in Manifold.

Source: “An overview of Manifold and its implementation, Arbab, Herman & Spilling, 1993.

An example in Manifold : Events are used for dynamic reconfiguration

```
port in input;
port out output;
{
 process A is A_Type;
 process B is B_Type;
 process C is C_Type;

 start: (activate A, activate B, activate C); do begin.

 begin: (A → B, output → C, input → output).

 e1: (B → input, B → C, C → A, A → B, output → a,
      input → output).

 e2: (C → B).
}
```

Source: “An overview of Manifold and its implementation, Arbab, Herman & Spilling, 1993.

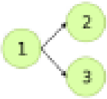
Textual pseudo-graphical languages

Leaf : A bio-pipeline workflow language that uses a textual DSL for graphically expressing DAG

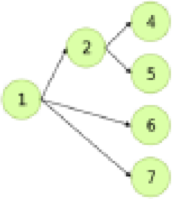
```
1 -> 2 -> 3;
```


```
1 -> 2 -> @1     ;
```


```
 /2
1<
 \3;
```

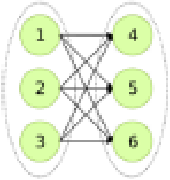



```
    /4
   2<
  /  \5
1<
  \  /6
   .<
    \7          ;
```

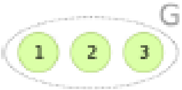


Leaf : A bio-pipeline workflow language that uses a textual DSL for graphically expressing DAG

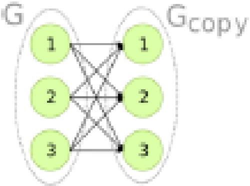
1, 2, 3 -> 4, 5, 6;



G: 1, 2, 3;

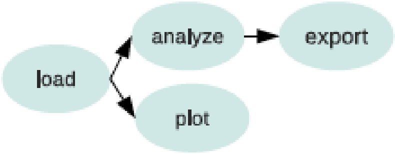


G: 1, 2, 3;
@G -> G;



An example in Leaf

```
lglgraph = r"""
      /analyze -> export
load <
      \plot
"""
```



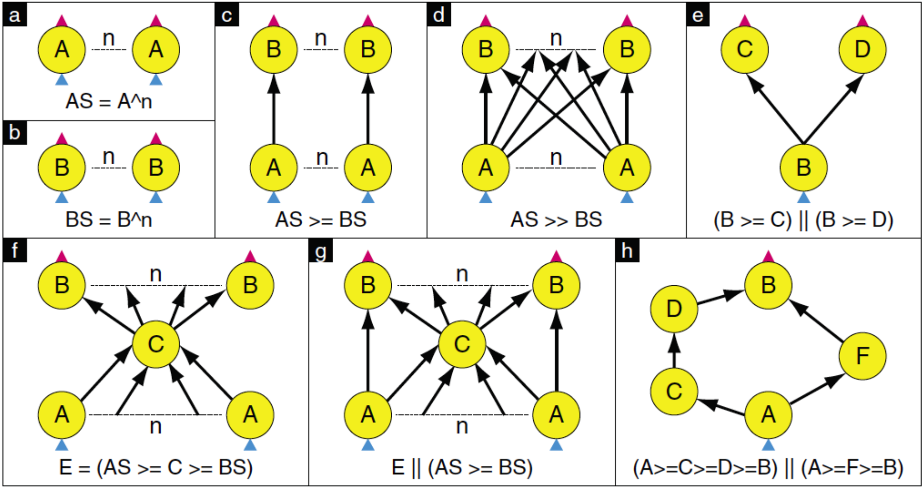
Source: “Bioinformatics pipelines in Python with Leaf”, Napolitano, Mariani-Costantini & Tagliaferri, 2013

Another example in Leaf

```
LogR_BAF_FileName[F] -> prepareInput[F]
;
                  /getSampleNames
                 |     -> exportCNVDiffMat [F],
                 |     clustergram [F],
                 |     distMatGfx [F],
                 |     CNVDiffMat,
                 |     intersectTBRegs
 sampleSheet[F] <
                  \               /genoTypeCheck
                   @prepareInput <
                                  \                          /makeBed [F]
                                   PennCNV[F]               |
                                      -> joinPennCNVout[F] <
                                                            \
                                                             addGeneInfo[F]
                                                                -> readFile
                                                                  -> manualClean
;
               /getGeneNames
              |  -> @clustergram,
              |     @CNVDiffMat,
              |     @exportCNVDiffMat
@manualClean <
               \
                |                    /exportMergedRegsTB [F]
                |   @intersectTBRegs<
                | /                  \mergeFragments
                | |                    -> exportFigExtRegs [F]
                .<
                  \
                   |         /exportPerRegion_clean [F]
                   reformat <
                             \
                              |                         /@exportCNVDiffMat,
                              |                         |  @clustergram
                              |             @CNVDiffMat <
                              |            /            \ computeFisher
                              |           |              |  -> @clustergram,
                              |           |                    @exportCNVDiffMat
                              geneCentric <
                                           \@exportCNVDiffMat
;
```

Source: “Bioinformatics pipelines in Python with Leaf”, Napolitano, Mariani-Costantini & Tagliaferri, 2013

Dryad : Another workflow language with textual DSL for expressing DAG (algebra-like operations)



Source: "Dryad : Distributed data-parallel programs from sequential building blocks", Isard *et al.*, 2007

Implicit construction of DAG

Imperative scripting languages

Swift : A language for scientific workflows consisting of two elements : specification of datasets + processing

XDTM

XML description of the (often complex) datasets

SwiftScript

Imperative scripting language that builds on XDTM

An example in SwiftScript : The calls to external applications are made explicit (with appropriate files)

```
(Run resliced) reslice_wf ( Run r ) {
  Run yR = reorientRun( r , "y", "n" );
  Run roR = reorientRun( yR , "x", "n" );
  ...
}

(Run or) reorientRun( Run ir, string dirct, string ovw ) {
   foreach Volume iv, i in ir.v {
     or.v[i] = reorient( iv, dirct, ovw );
   }
}

(Volume ov) reorient( Volume iv, string dirct, string ovw ) {
  app { reorient @filename(iv.hdr)
                 @filename(ov.hdr)
                 dirct
                 ovw;  }
}
```

Source: “Swift : Fast, Reliable, Loosely Coupled Parallel Computation”, Zhao *et al.*, 2007

Skeleton-based languages

SuperPAS (Parallel Architectural Skeletons) : Allows user-defined skeletons

Assertion (fact ?)

"*Most existing [skeleton frameworks] support a limited and fixed set of patterns that are hard-coded into those systems.*"

Source: "A model for designing and implementing parallel applications using extensible architectural skeletons", Akon, Goswami & Li, 2005

SuperPAS proposes a Skeleton Description Language (SDL)

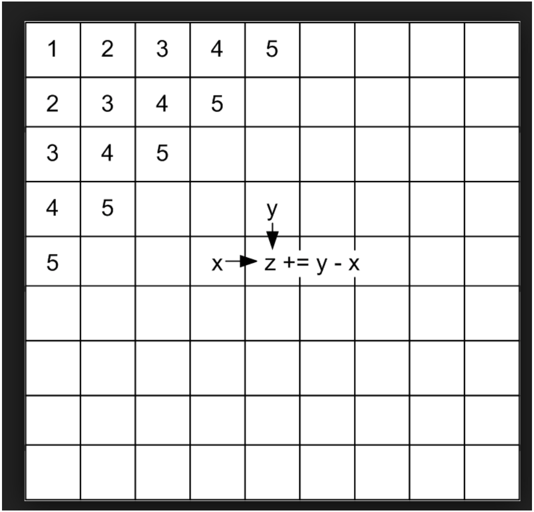
"Using the SDL, a skeleton designer can design and implement a new skeleton without understanding the low level details of the system and its implementation."

Source: *Ibid.*

Key characteristics of SuperPAS SDL

- Provides a set of multidimensional grids
  - Each node of a grid is a virtual processor
  - Each multidimensional virtual processor grid is equipped with its own communication primitives (peer-to-peer, collective, synchronization-only, etc.)
- The topology of an abstract skeleton is embedded in an appropriate multidimensional grid, possibly with *null processors*

Example : Wavefront computation, for example, used in dynamic programming algorithm



Source: `http://parallel.vub.ac.be/education/parsys/practicum.html`

Wavefront skeleton example in SuperPAS

```
integer size;
skeleton Wavefront(2) {

  LOCAL = {
    void init() {
      for( int i = 0; i < GetDimension(); i++ )
        SetDimensionLimit(i, size);
    }

    bool non_null( const Location &loc ) {
      return loc[1] <= loc[0]; // col. num. <= row num.
    }
  }
```

Wavefront skeleton example in SuperPAS (cont.)

```
PUBLIC = {
  void SendRight( Msg &m ) {
    Location loc = GetLocation();
    loc[1] = loc[1] + 1;
    SendPeer( loc, m );
  }

  void RecvRight( Msg &m ) {
    ...
  }

  bool IsAtDiagonal() {
    return loc[0] == loc[1];
  }

  ...
}
```

Rules and dependencies

Make is used for compilation and file manipulation tasks

The **global** ordering of tasks is implicit : it is expressed through rules that describe the required **(local) dependencies**

```
$ cat hello.c
#include <stdio.h>
int main() {
  printf( "Hello, World!\n" );
}

$ cat Makefile
hello: hello.c
        gcc -o hello hello.c

clean:
        rm -f hello hello.o
```

```
$ make
gcc -o hello hello.c

$ ./hello
Hello, World!

$ make clean
rm -f hello hello.o
```

Source: `http://hyperpolyglot.org/build`

Rake is similar to make, but is defined as an internal Ruby DSL

```
$ cat Rakefile
task :default => "hello"

file "hello" => ["hello.c"] do
  sh "gcc -o hello hello.c"
end

task :clean do
  rm_f "hello hello.o"
end
```

```
$ rake
gcc -o hello hello.c

$ ./hello
Hello, World!

$ rake clean
rm -f hello hello.o
```

Source: http://hyperpolyglot.org/build

A limitation of make : It cannot easily handle dynamic workflow definition

[With makefiles], it is difficult to describe the "multiple instances with a priori runtime knowledge" pattern [i.e., when] the number of instances is unknown before the workflow is started, but becomes known at some stage during runtime.

Source*: "Agile parallel bioinformatics workflow management using Pwrake", Mishima* et al.*, 2011.*

A limitation of Rake : It can be run in parallel, but with little control over the parallelism

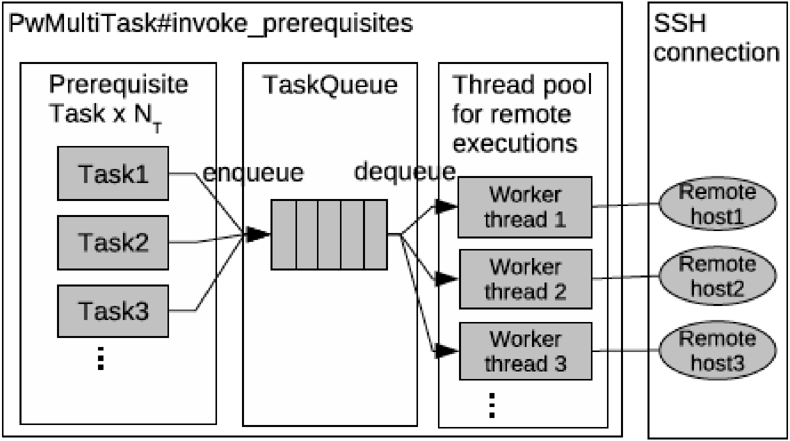
The original Rake has the `MultiTask` *class for parallel execution of prerequisite tasks in Ruby threads. [However,] Rake has no mechanism for controlling the number of threads nor thread pooling, [nor] for invoking processes on remote hosts.*

Source*: "Pwrake : A parallel and distributed flexible workflow management tool for wide-area data intensive computing", Tanaka & Tatebe, 2010*

Furthermore :

Rake uses Ruby threads, which are not really parallel in MRI Ruby (uses a GIL = Global Interpreter Lock) — but threads are really parallel in JRuby (JVM threads)

Pwrake : A distributed parallel workflow extension of Rake



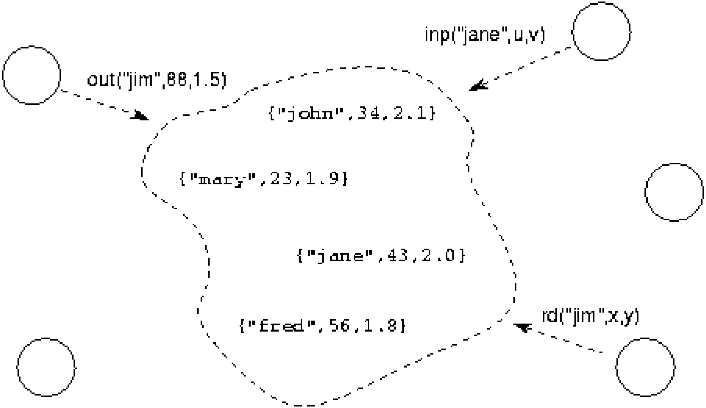
Source: “Pwrake : A parallel and distributed flexible workflow management tool for wide-area data intensive computing”, Tanaka & Tatebe, 2010

An example in Pwrake : A task can (and sometimes must) be explicitly invoked

```
SRCFITS = FileList["#{INPUT_DIR}/*.fits"]

file( "pimages.tbl" ) do
  OUTFITS = SRCFITS.map do |img|
    out = img.sub /^(.*?)([^\/]+).fits/, 'p/\2.p.fits'
    file( out => [img, HDR] ) do |t|
      t.rsh "mProjectPP #{img} #{out} #{HDR}"
    end
    out
  end
  pw_multitask( "Proj" => OUTFITS ).invoke
  sh "mImgtbl p pimages.tbl"
end
```

Shared-medium coordination languages

Linda : The first approach to explicitly introduce the idea of coordination language



Source: `http://www.mcs.anl.gov/~itf/dbpp/text/node44.html`

Provides a unique global **tuple space** with flat tuples

An example in Linda : A small number of coordination operations are provided within a standard language

```
int main( int argc, char* argv[] ) {
  int nbWorkers = atoi( argv[1] );

  for( int j=0; j < nbWorkers; j++ )
    eval( "worker", hello(j) );

   for( int j=0; j < nbWorkers; j++ )
     in( "done" );
}

int hello( int i ) {
  printf( "Hello world from %d.\n", i );
  out( "done" );
}
```

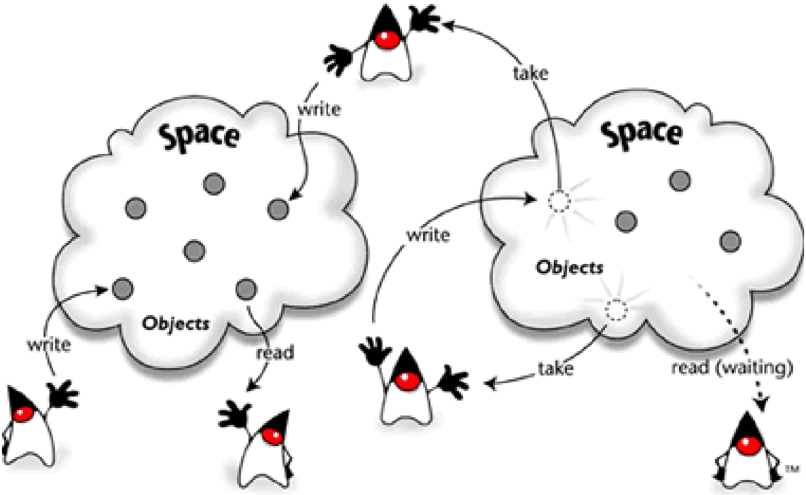
Source: http://web.archive.org/web/20090925185219/http://phi.sinica.edu.tw/instruct/workshop/html/linda/linda.html

An example in Linda : Conditional communication can be performed through pattern-matching of tuples

```
int nbWorkers = ...

for( int j=0; j < nbWorkers; j++ )
  eval( "worker", worker );

out( "barrier", 0 );
...
```

```
int worker {
  ... do something ...

  // Barrier-synchronization.
  int nb;
  in ( "barrier",?nb  );
  out( "barrier", nb+1 );
  rd ( "barrier", nbWorkers );
  ...
}
```

JavaSpaces : A Linda implementation in Java with multiple structured-tuple spaces



Source: `http://fr.slideshare.net/shengt/linda-andtuplespace-changed`

Concurrent Collections (CnC) : Origins and key concepts

Origins of CnC

- Dataflow architectures ⇒ implicit parallelism (1975)

  ⋮

- Linda ⇒ tuples and tuple space (1985)

  ⋮

- TStreams ⇒ tagged streams (2004)

⇒ Intel Concurrent Collections (2009)

Concurrent Collections (CnC) : Origins and key concepts

Origins of CnC

- TStreams ⇒ tagged streams (2004)

⇒ Intel Concurrent Collections (2009)

TStream's tagged streams were renamed collections

- A stream describes a collection of data objects produced by one computation and used by another (typical !)
- No FIFO ordering on the stream's values – the values are tagged ≈ key–value access (≈ tuple space)
- A stream is monotonic : An item, once inserted, is never removed (atypical !)

CnC's basic premise = Domain experts should not worry about parallelism constructs

Domain experts can identify the intrinsic data dependences and control dependences in an application, without worrying about what parallel constructs should be used so satisfy those dependences.

Source: "Dataflow Programming with Intel Concurrent Collections", V. Sarkar, 2011

Separation of Concerns between Domain Expert and Tuning Expert

The application problem

The work of the **domain expert**
- Semantic correctness
- Constraints required by the application

Concurrent Collections Spec

The work of the **tuning expert**
- Architecture
- Actual parallelism
- Locality
- Overhead
- Load balancing
- Distribution among processors
- Scheduling within a processor

Mapping to target platform

Goal:

serious **separation of concerns**:

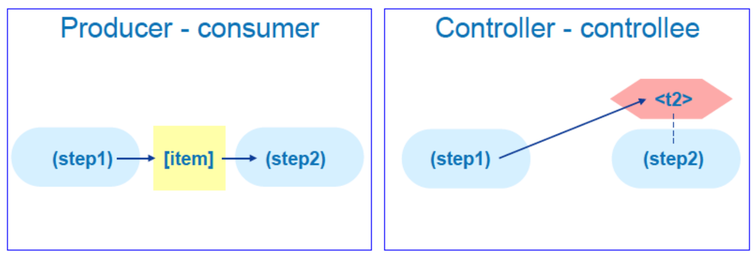
The **domain expert** does not need to know about **parallelism**

The **tuning expert** does not need to know about the **domain**.

Source: "The Concurrent Collections (CnC) Parallel Programming Model—Foundations and Implementation Challenges", Knobe & Sarkar, 2009

What the domain expert must do is express the semantic ordering constraints

Data dependences | Control dependences



Implicit parallelism

Parallelism is implicit, based on the resulting CnC graph.

An example to illustrate data collections and data dependencies : Filtering substrings

Input

Set of strings

Output

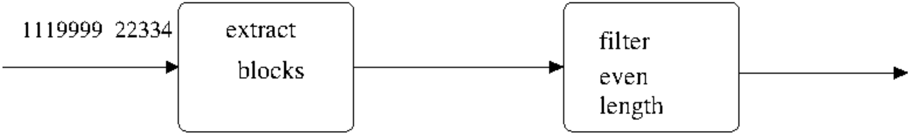
Set of substrings from input that...

- is a maximal block of identical characters
- is of even length

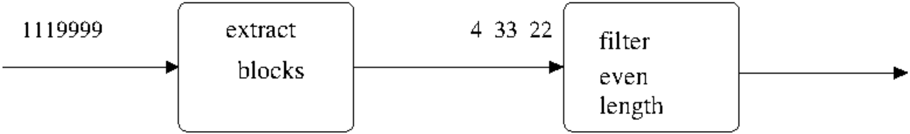
Example

Input	=	["22334", "1119999"]
Output	=	["22", "33", "9999"]

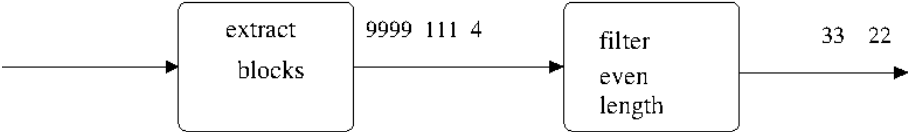
An execution in FastFlow : Initial state



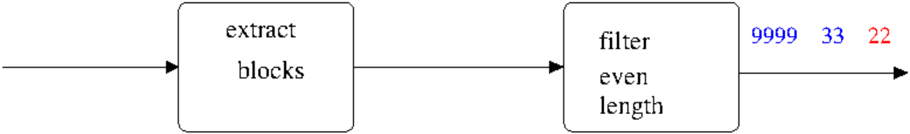
An execution in FastFlow : Some time later



An execution in FastFlow : Some time a bit later



An execution in FastFlow : Final state

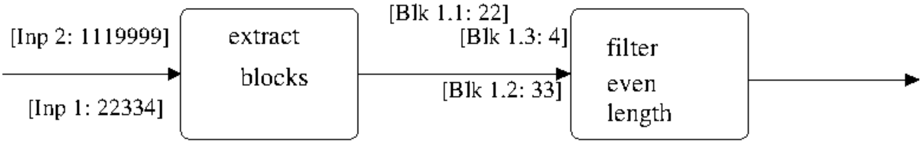


An execution in CnC (data collections only) : Initial state

Note :

- `[Foo tg: val]` = item in data collection `Foo` with tag (key) `tg` and associated value `val`.

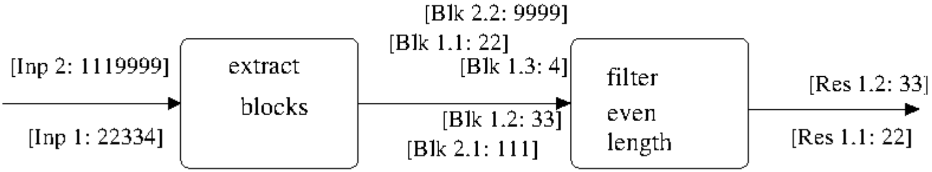
An execution in CnC (data collections only) : Some time later



Note :

- `[Foo tg: val]` = item in data collection `Foo` with tag (key) `tg` and associated value `val`.

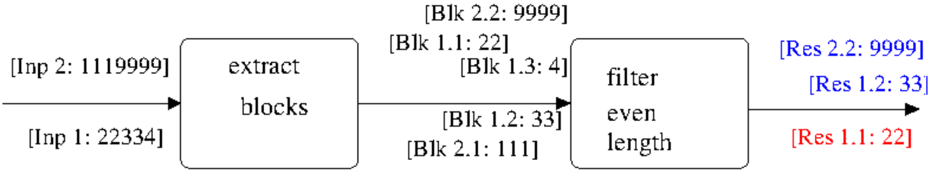
An execution in CnC (data collections only) : Some time a bit later



Note :

- `[Foo tg: val]` = item in data collection `Foo` with tag (key) `tg` and associated value `val`.

An execution in CnC (data collections only) : Final state

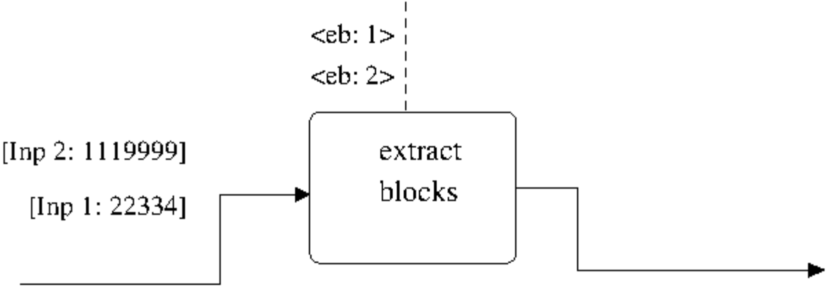


Note :

- `[Foo tg: val]` = item in data collection `Foo` with tag (key) `tg` and associated value `val`.

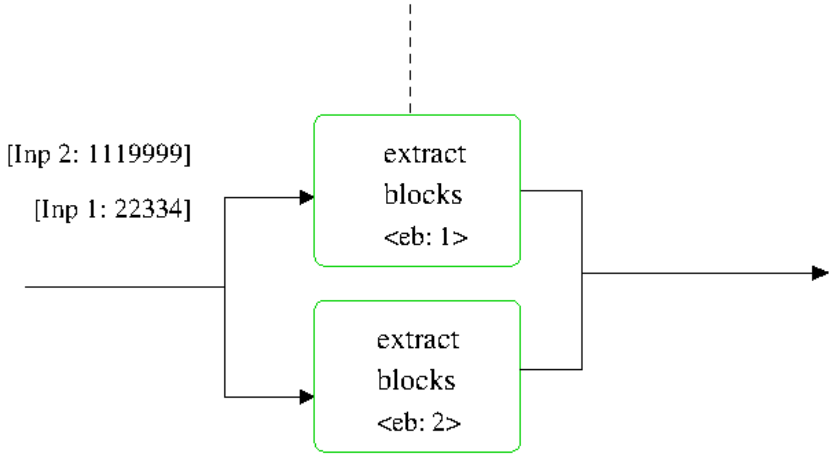
But, in CnC, a step requires an appropriate control tag to execute — to create instances of the step

Note : Steps are supposed to be purely functional ⇒ reexecution of a step is idempotent



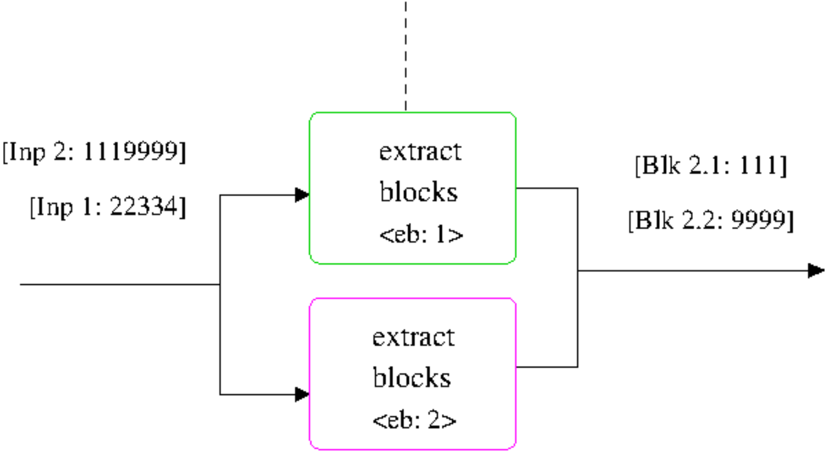
But, in CnC, a step requires an appropriate control tag to execute — to create instances of the step

Note : Steps are supposed to be purely functional ⇒ reexecution of a step is idempotent



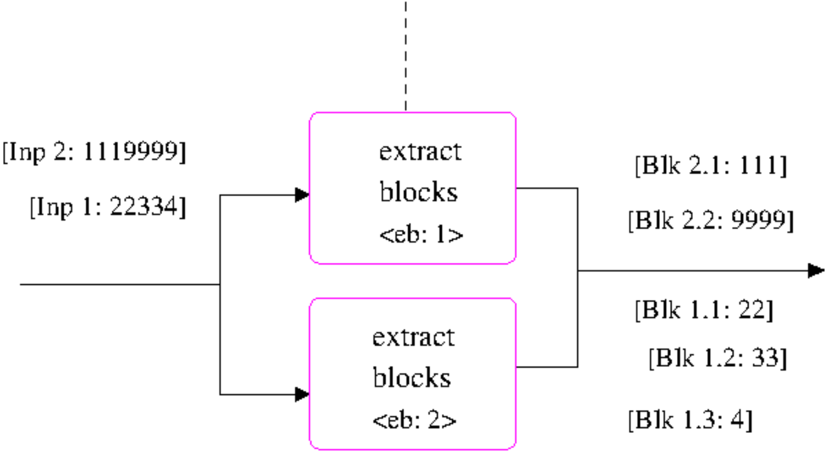
But, in CnC, a step requires an appropriate control tag to execute — to create instances of the step

Note : Steps are supposed to be purely functional ⇒ reexecution of a step is idempotent



But, in CnC, a step requires an appropriate control tag to execute — to create instances of the step

Note : Steps are supposed to be purely functional ⇒ reexecution of a step is idempotent

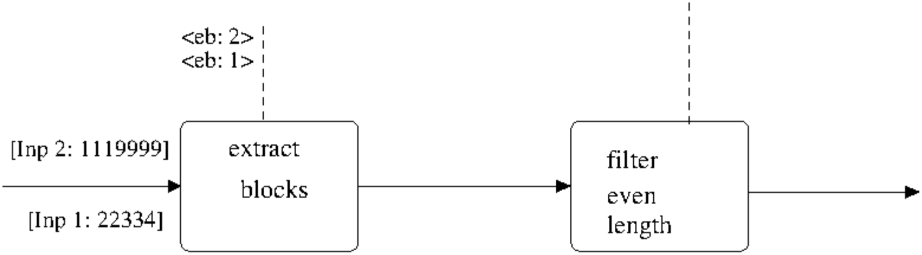


But, in CnC, a step requires an appropriate control tag to execute — to create instances of the step

Note : Steps are supposed to be purely functional $\Rightarrow$ reexecution of a step is idempotent

So :
control tag $\approx$ id of dynamic instance of a macro-dataflow node

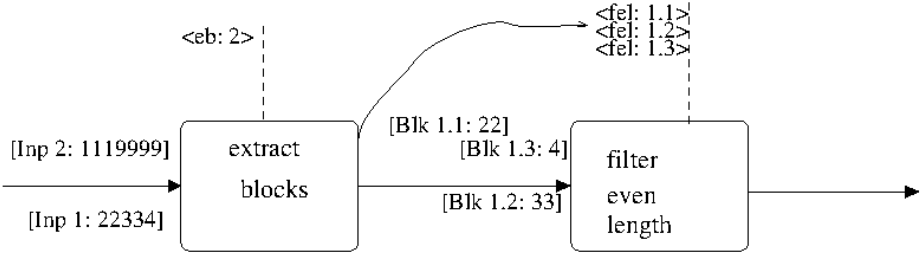
An execution in CnC (data and control collections) : Initial state



Note :

- [Foo tg: val] = item in data collection Foo with tag (key) tg and associated value val.
- <bar: tg> = item in tag (control) collection bar with tag tg.

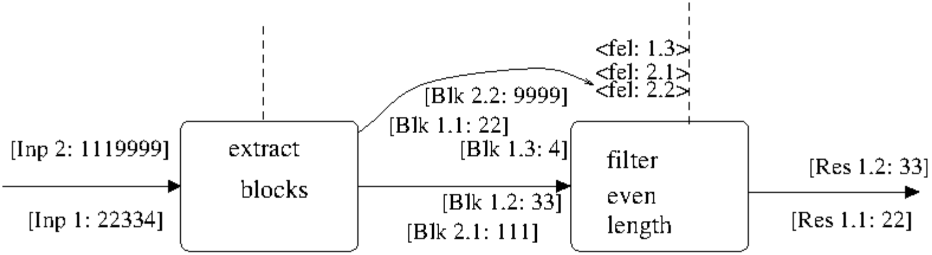
An execution in CnC (data and control collections) : Some time later



Note :

- [Foo tg: val] = item in data collection Foo with tag (key) tg and associated value val.
- <bar: tg> = item in tag (control) collection bar with tag tg.

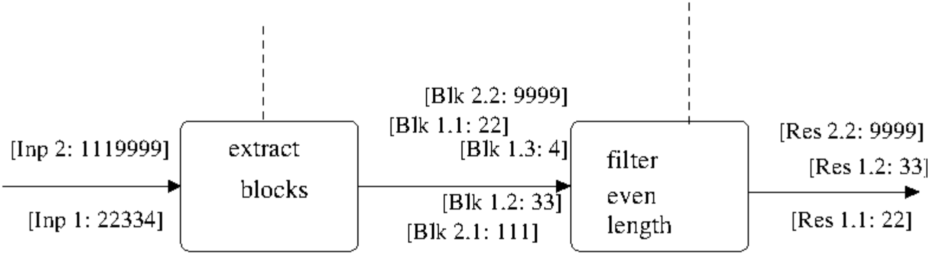
An execution in CnC (data and control collections) : Some time a bit later



Note :

- `[Foo tg: val]` = item in data collection `Foo` with tag (key) `tg` and associated value `val`.
- `<bar: tg>` = item in tag (control) collection `bar` with tag `tg`.

An execution in CnC (data and control collections) : Final state



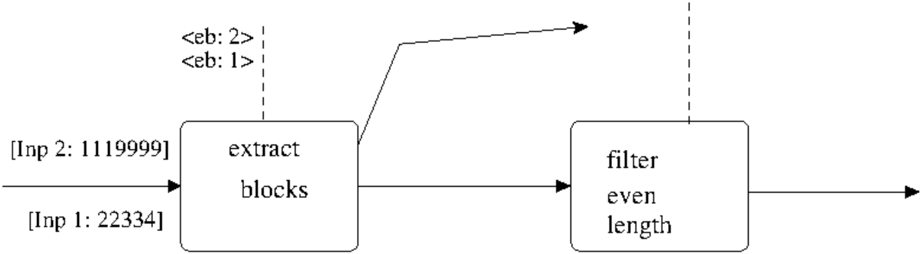
Note :

- `[Foo tg: val]` = item in data collection `Foo` with tag (key) `tg` and associated value `val`.
- `<bar: tg>` = item in tag (control) collection `bar` with tag `tg`.

FastFlow vs. CnC : For simple pipelines, they are similar and control tags seem superfluous

Assertion

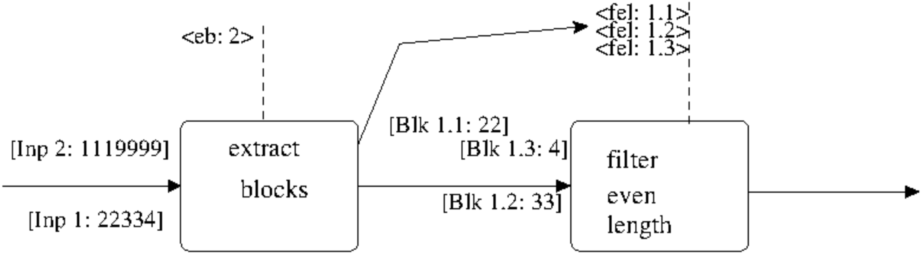
For simple pipelines, there is a one-to-one correspondence between tags and data items.



FastFlow vs. CnC : For simple pipelines, they are similar and control tags seem superfluous

Assertion

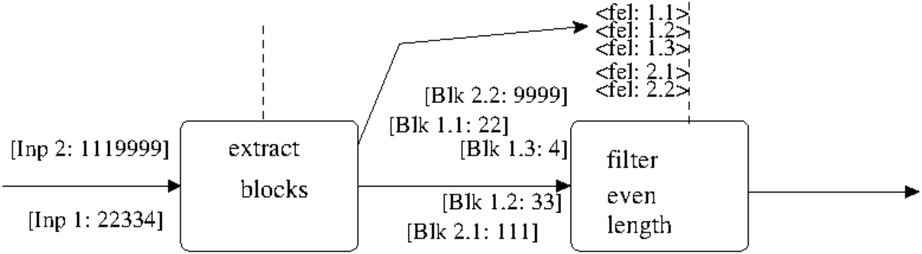
For simple pipelines, there is a one-to-one correspondence between tags and data items.



FastFlow vs. CnC : For simple pipelines, they are similar and control tags seem superfluous

Assertion

For simple pipelines, there is a one-to-one correspondence between tags and data items.



FastFlow vs. CnC : An example to illustrate the difference

Input

Set of strings

Output

Set of substrings from input that...

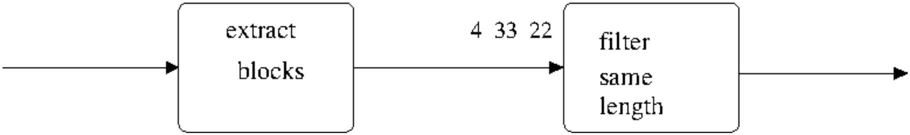
- is a maximal block of identical characters
- is the same length as preceding block in same string

Example

Input = ["22334", "1119999"]
Output = ["33"]

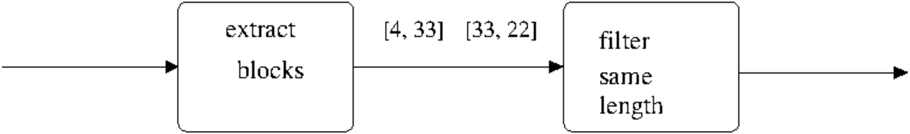
A possible FastFlow solution : `filter` process is not functional — it requires an internal state

Given `"22334"` as input…



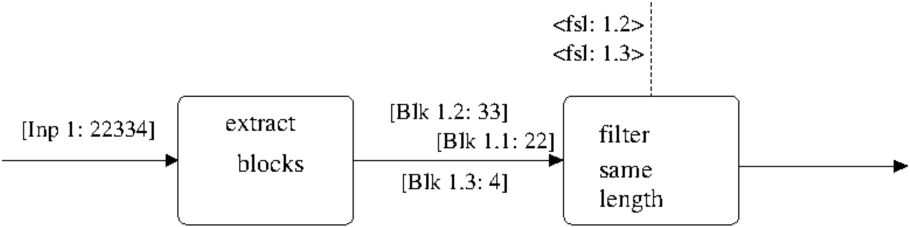
Another possible FastFlow solution : `filter` process is now functional, but tasks are more complex

Given `"22334"` as input. . .



A possible CnC solution : All steps are functional

Given "22334" as input. . .



def filter same length(<fsl : i.j>) =
  output [Blk i.j] if length [Blk i.j] == length [Blk i.j-1]

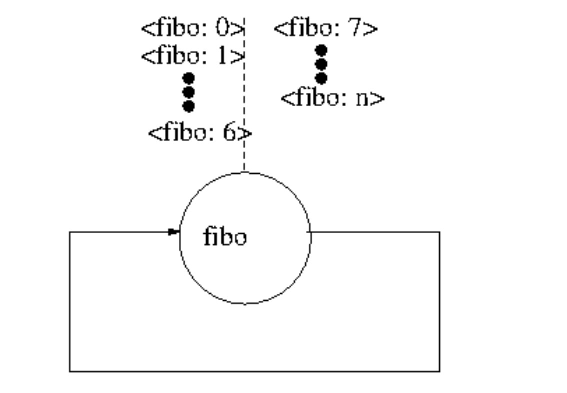
FastFlow vs. CnC : CnC can express memoized recursive non-strict definition

Example (Fibonacci in a non-strict functional language)

```
fibo( n ) = fibos[n]
  where
    fibos[0] = 1
    fibos[1] = 1
    fibos[i] = fibos[i-1] + fibos[i-2], 2 <= i <= n
```

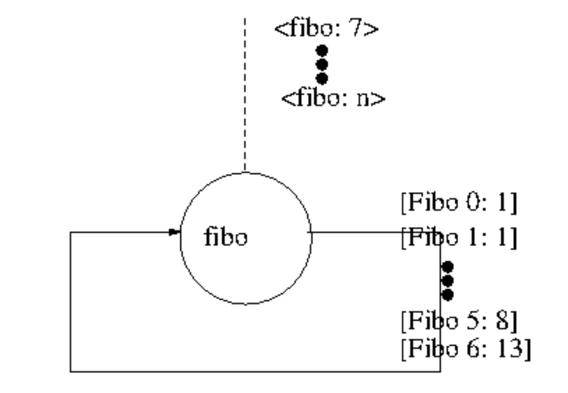
FastFlow vs. CnC : CnC can express memoized recursive non-strict definition

Example (Fibonacci in CnC)



FastFlow vs. CnC : CnC can express memoized recursive non-strict definition

Example (Fibonacci in CnC)

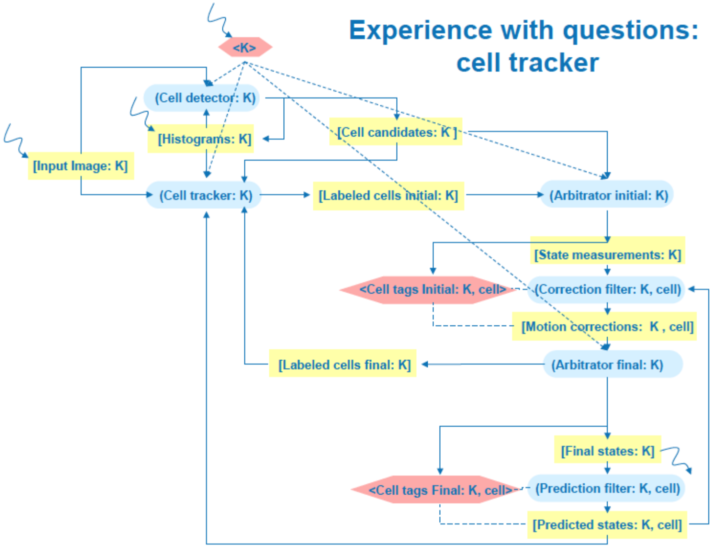


FastFlow vs. CnC : CnC can express memoized recursive non-strict definition

Example (Fibonacci step for CnC version, in pseudocode)

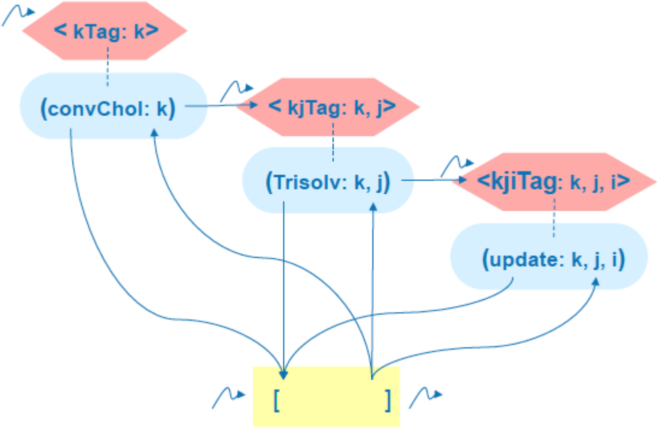
```
def fibo( <fibo: n> )
  case n
   when 0
     Fibo.get(0)
   when 1
     Fibo.get(1)
   else
     r = Fibo.get(n-1) + Fibo.get(n-2)
     Fibo.put(n, r)
  end
end
```

FastFlow vs. CnC : In general, there may be no direct correspondance between control tags and data items

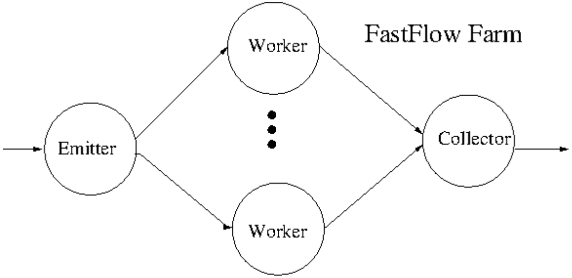


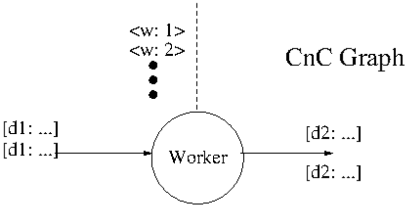
FastFlow vs. CnC : In general, there may be no direct correspondance between control tags and data items

Cholesky: graphical form



FastFlow vs. CnC : Fastflow core skeletons graphs deal with resources (threads), CnC graphs do not

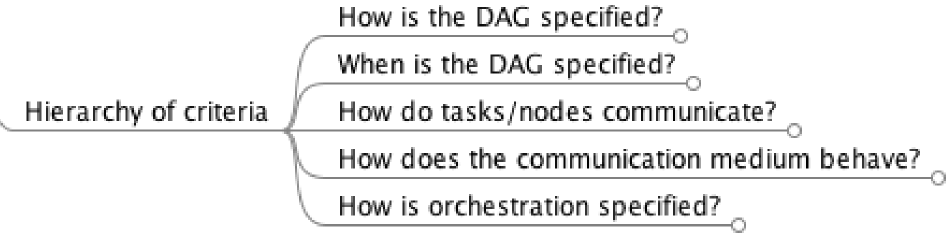




Comparison of some approaches

Some questions to compare the different approaches

A number of comparison criteria



How is the DAG specified ?

Explicitly

- Explicit links between nodes : GUI, ADL
- Graph algebra

Implicitly

- Imperative scripting languages
- Skeleton languages
- Rules and dependencies
- Shared-medium coordination language

A key feature of “recent” languages or approaches is that they support some form of dynamic tasks

Languages proposed by DARPA HPCS program

- Chapel
- Fortress
- X10

Older languages. . . or newer versions of existing languages

- Cilk
- OpenMP 3.0
- Habanero Java

Other languages

- CnC — Concurrent Collections

When is the DAG specified?

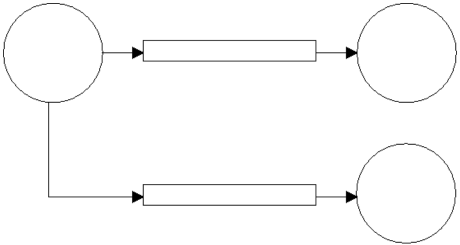
Statically

- GUI
- ADLs (some)
- Graph algebra
- Scripting languages (some)
- Skeleton languages
- Coordination languages (some)

Dynamically

- Scripting languages (some)
- Skeleton languages (some)
- Rules and dependencies
- Coordination languages and Concurrent collections

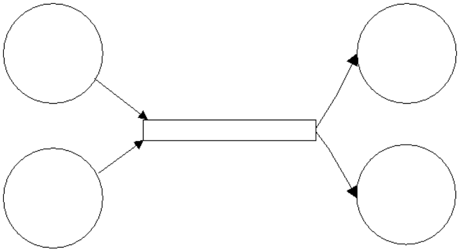
How do tasks/nodes communicate with one another ?



Privately

$\Rightarrow$

$1 \rightarrow 1$



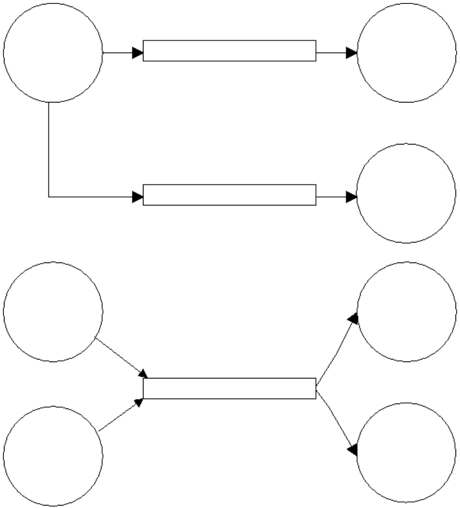
Publicly

$\Rightarrow$

$n \leftrightarrow m$

$n \rightarrow 1$

How do tasks/nodes communicate with one another ?



Privately

$\Rightarrow$

$1 \rightarrow 1$

Channel-based

Publicly

$\Rightarrow$

$n \leftrightarrow m$

$n \rightarrow 1$

Shared-medium

But… not all "channel"-based approaches lead to private communication

Channels in Go (Ruby-style)

```
c = channel!(Integer)

go! do
  0.upto(10) { |i| c << i }
  puts c.receive       # ∈ {1,2,10,20}
end

go! do
  0.upto(10) { |i| c << 10*i }
  puts c.receive       # ∈ {1,2,10,20}
end
```

The channel is explicit with a public name

⇒ available for use by any process (for reading or writing)

And some approaches that have channels (somewhere !) are part private/part public

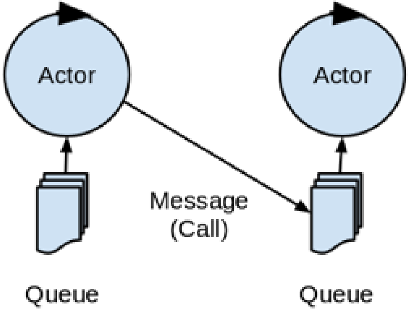
Scala actors

```
class PingActor extends Actor {
  def receive = {
    case Start(ponger) => { ponger ! Ping }
    case Pong => { println("Pong!"); sender ! Ping }
  }
}

class PongActor extends Actor {
  def receive = {
    case Ping => { println("Ping!"); sender ! Pong }
  }
}
...
pinger ! Start(ponger)
```

Actors

An Actor is like an object instance executed by a single thread. Instead of direct calls to methods, messages are put into the Actor's “mailbox” (queue). The actor single threaded reads and processes messages from the queue sequentially.

Source: `http://java-is-the-new-c.blogspot.it/2014/01/comparision-of-different-concurrency.html`



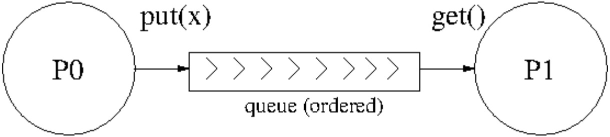
And some approaches that have channels (somewhere !) are part private/part public

Scala actors

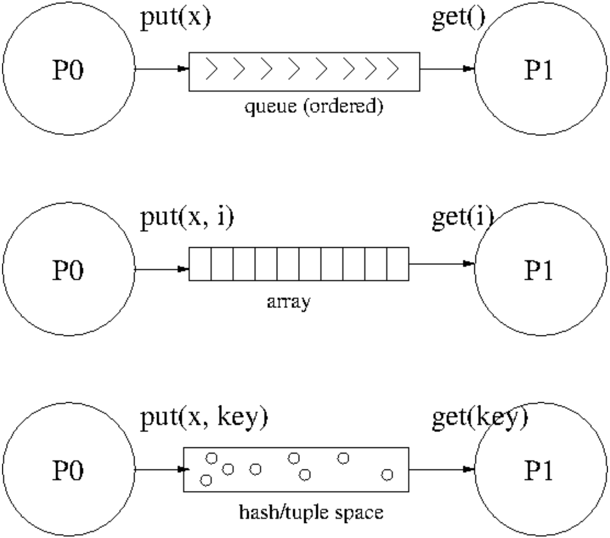
```
class PingActor extends Actor {
  def receive = {
    case Start(ponger) => { ponger ! Ping }
    case Pong => { println("Pong!"); sender ! Ping }
  }
}

class PongActor extends Actor {
  def receive = {
    case Ping => { println("Ping!"); sender ! Pong }
  }
}
...
pinger ! Start(ponger)
```

An actor owns a **mailbox** (with an implicit channel) : any process can send to it, but only the owner can read from it — the actor's name is explicit and public, not its channel !

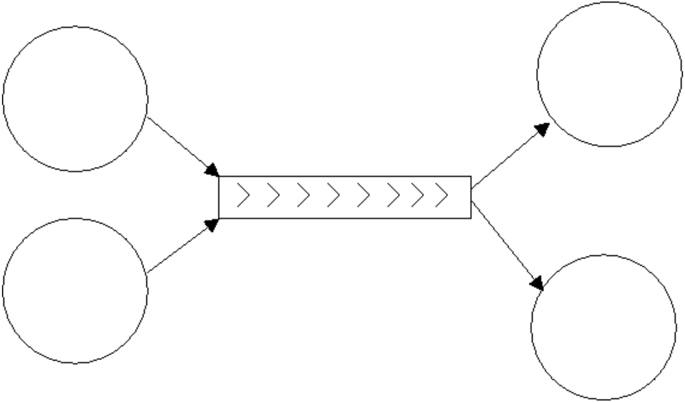
How does the communication medium behave : in the private case ?



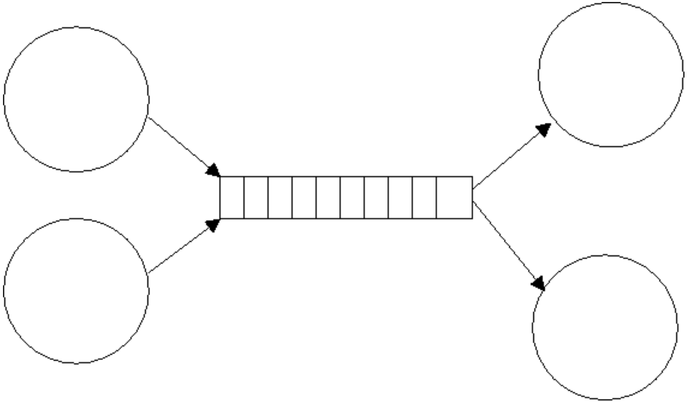
How does the communication medium behave : in the private case ?



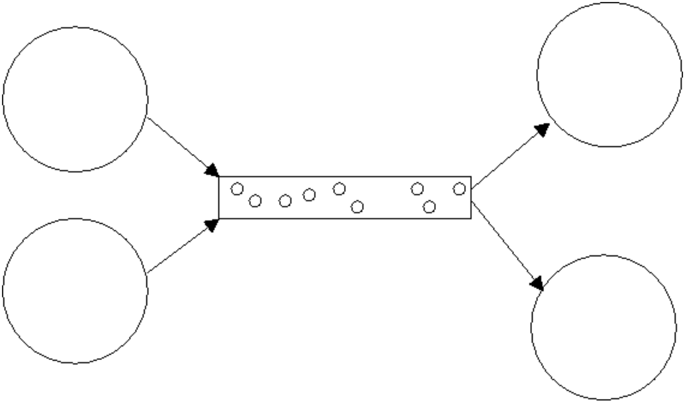
How does the communication medium behave : in the public case ?



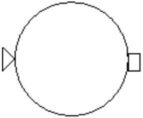
How does the communication medium behave : in the public case ?



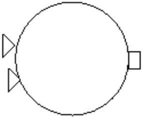
How does the communication medium behave : in the public case ?

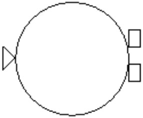


How many ports do nodes have and how are those ports identified ?

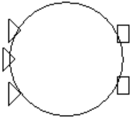


1/1



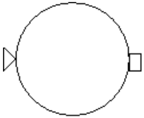


N/1 vs. 1/M

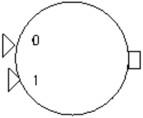


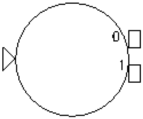
N/M

How many ports do nodes have and how are those ports identified ?

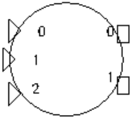


1/1



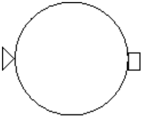


N/1 vs. 1/M

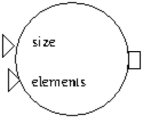


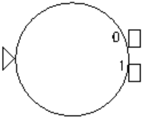
N/M

How many ports do nodes have and how are those ports identified ?

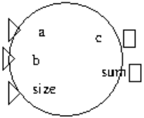


1/1





N/1 vs. 1/M



N/M

How is orchestration specified ?

Hard-coded in the framework

- Most of the approaches

Under user control

- Kepler

Some possible directions for FastFlow ?

Some possible directions ?

- Support for dynamicity ?
  - Dynamic (macro) dataflow ?
  - Concurrent collections ?
  - Task-based approach (might be similar to CnC) ?
  - User-defined evolving skeletons ?
  - User-defined orchestration ?
- Better support for explicit DAG ?
  - More general port interface ?
- More general communication medium ?
  - Higher level abstraction than FIFO queue ?