# Asynchronous computing of irregular applications using the SVPN model and S-Net coordination

Alex Shafarenko
University of Hertfordshire

# Some context

- Exascale computing presents new challenges
- The user community: conservative and inertial
- Need a way forward
  - change the "logistics" of computing
  - preserve the "mathematics"
- This will not be possible within subject niches!
- Separation of concerns

# Solution: component technology

- Mathematical components
- Coordination language for concurrency engineers
- Two-level specification
- Important: the issue of scale!

# Medium grain mathematics

- lightweight components, no internal *persistent* state, no access to the environment's state
- any programming language (... for conservative users ...)
- can re-compute a component without breaking semantics
- can clone and move
- "instructions of an asynchronous dataflow machine"

# Large scale logistics

- Coordination language taking care of:
  - messages delivering arguments to functional components
  - messages produced by the functional components
  - aggregating and disaggregating messages (parameter lists) and providing a hierarchical abstraction OOP style
  - (synchronisation) storage

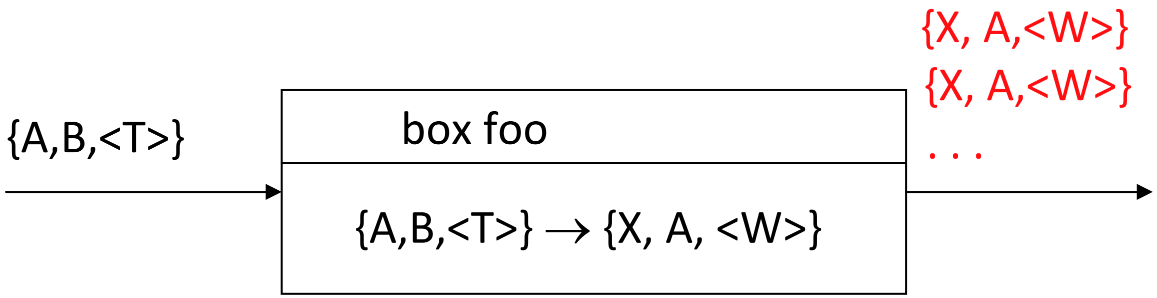
# Remove coordination spaghetti from application code

- Components only needing ONE library function: for message OUTPUT
  - Due to the lack of access to the environment's state, all input is available at the start
- Open question: to what extent the separation is possible?
- Dataflow agenda
  - has the same problem: data driven computation vs hierarchical data, inheritance, incapsulation, etc.

# Technology: S-Net at a glance

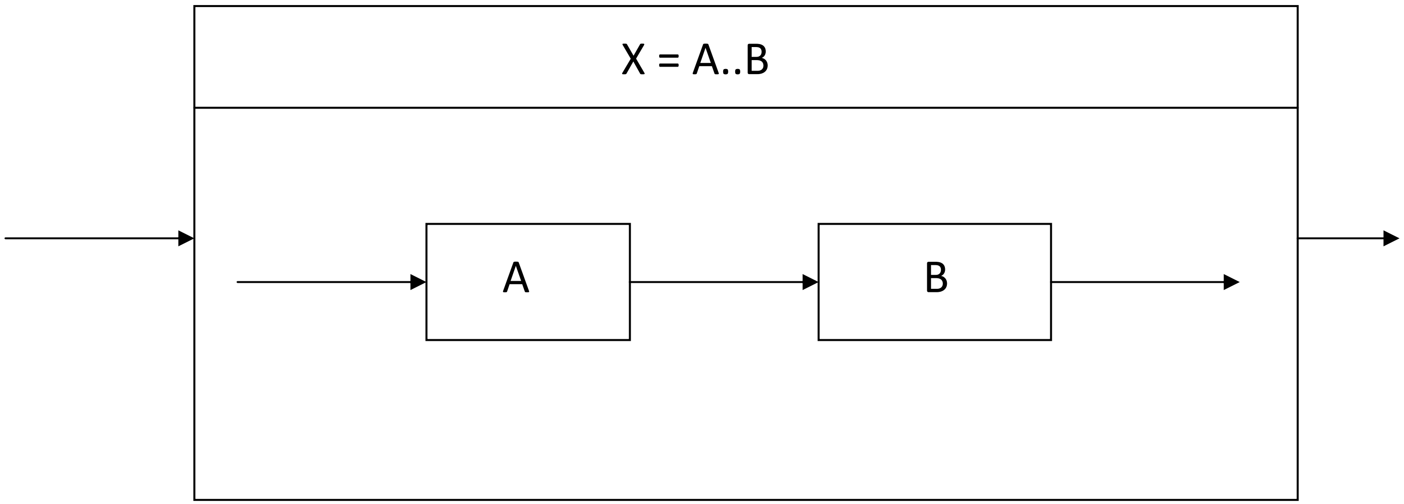
- SISO boxes
- connected by
  - Single Input stream
  - Single Output stream
- Streams transport records: **<u>sets</u>** of named entities
  - Opaq entities (value unavaliable to S-Net): fields
  - Transparent entities (vlues: integer): tags

- Box behaviour abstracted behind a type signature
- Boxes coded in a box language, not S-Net
- A box maps a single input record onto a stream (zero, one or more) of output records.



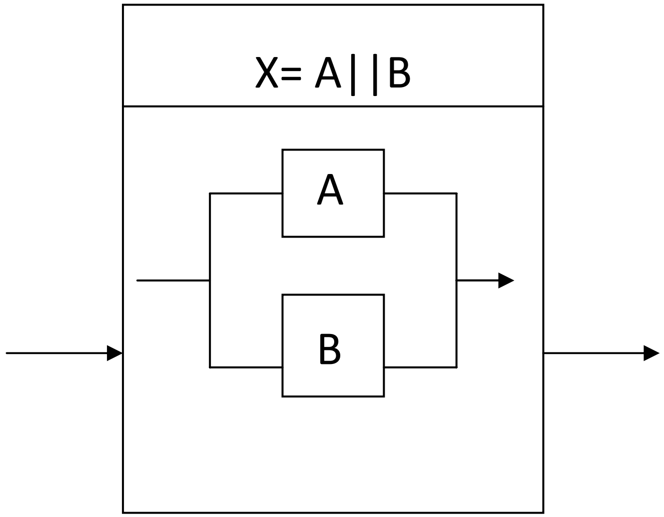
# Network combinators: Serial

- A and B operate concurrently
- Input-record $\rightarrow$ A; out(A) $\rightarrow$ B

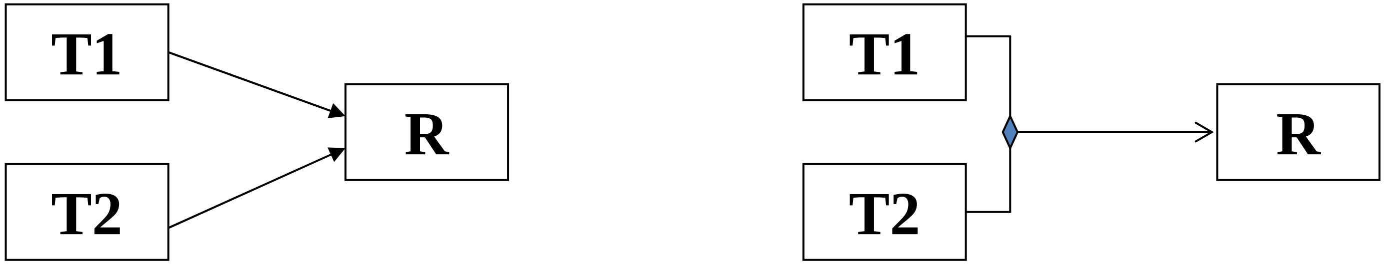


# Network Combinators: Choice

- Input records are matched with the input types of A and B
- If matches A, goes to A, else if matches B goes to B
- If matches both, goes to the best match
- If both matches are the same strength (e.g. {X,Y,Z} vs{X,Y} and {Y,Z}), then the choice is **nondeteministic**



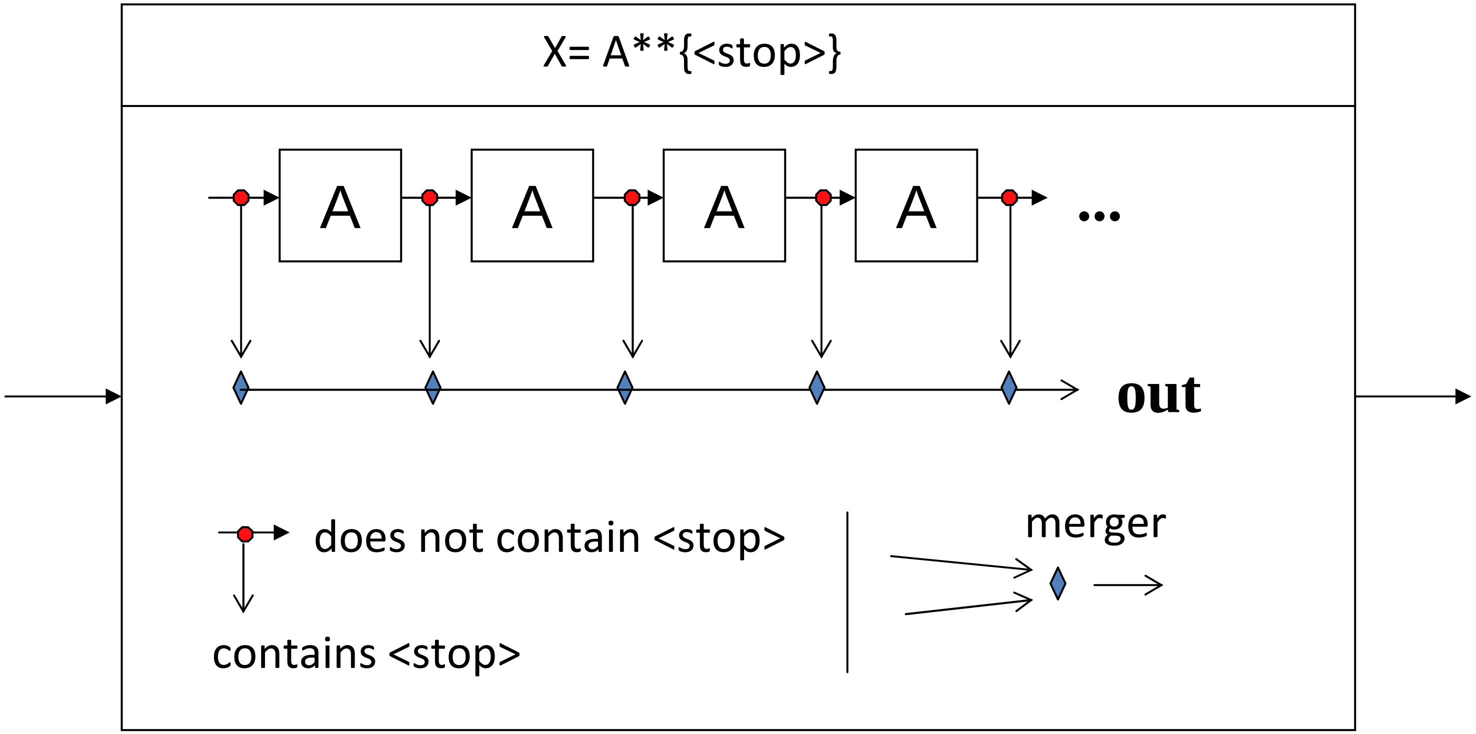
# Single stream concurrency




- Left: MIMO
  - Each channel blockable independently, state transitions inside
- Right: SISO
  - Merger nondeterministic, R gets an arbitrarily interleaved stream
  - No state transitions, must accept either kind of record
  - Even so, either substream can block the other one
  - Resources may currently be available for one substream
  - (implementation) Merger is a **demand-driven reordering buffer**

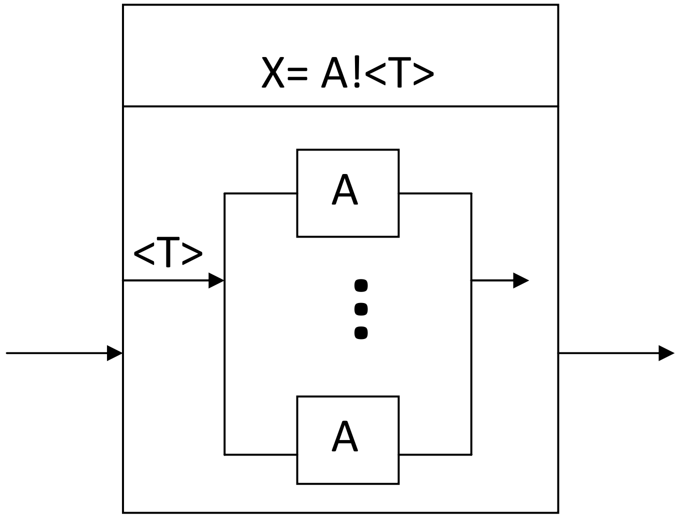
# Network Combinator: Serial Replication

- An unfolding chain of serially-connected replicas of A
- Unfolding ends when all outputs from A match <stop>



# Index Split

- All input records are required to have tag <T>
- The tag value determines which replica of A the record is to be sent to
- <T> has arbitrary integer values
- A may or may not require the knowledge of <T>

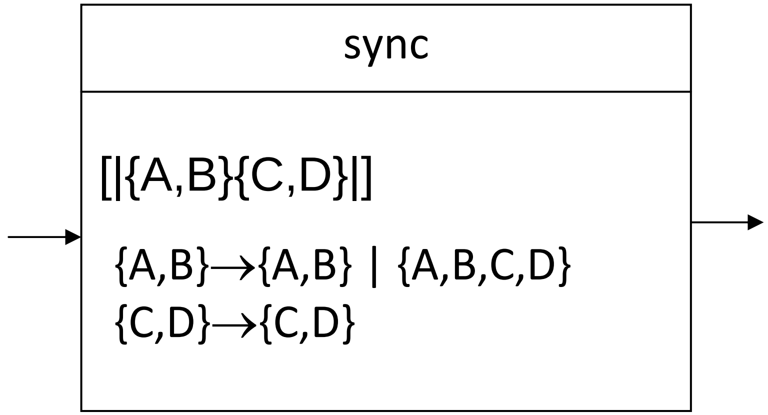



University of Hertfordshire

# Determinism

- All combinators (except ..) are supported in two versions
- Nondeterministic
  - |, !, *
  - The merger joins the streams out of order
- Deterministic
  - ||, !!, **, ..
  - The merger joins the streams in order
- The .. combinator does not contain a merger, hence one version
- We are discussing the introduction of '.' such that A.B allows reordering of records between A and B.

# Special Boxes


- Housekeeping
  - Eliminate record fields
  - Duplicate record fields
  - Rename records fields
  - Add tags
  - Manipulate tag values

- Synchronisation
  - Store record that comes first
  - Wait for the other kind of record, while waiting pass records of the first kind through
  - Then join the two records and die.
  - A dead synchrocell is a pass-through.

# Type Concept

- Messages are **<u>sets</u>** of fields/tags
- Subtyping as supersetting: {A,B} is good for {A}->{C}, since $\{A\} \subseteq \{A,B\}$
- Type signatures are set of rules:

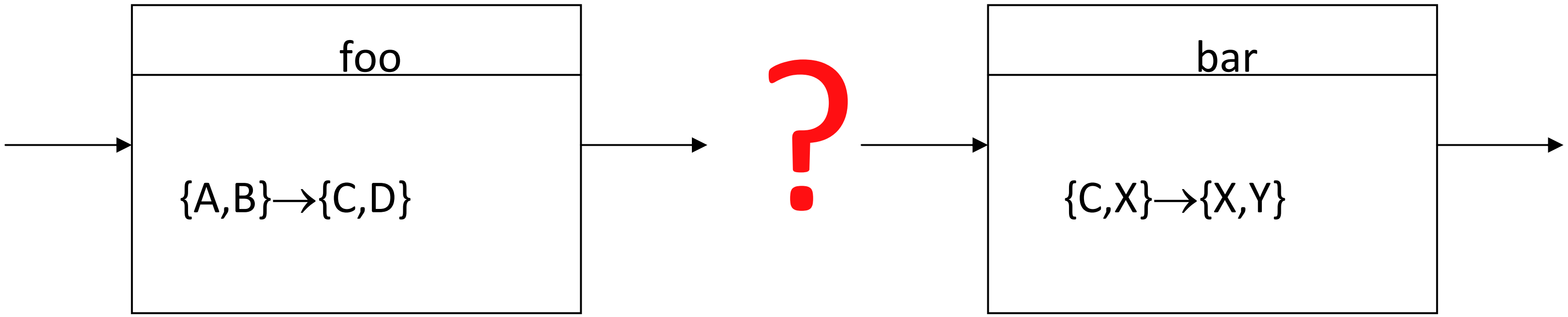
  {X,Y} -> {Y,Z}

  {V} -> {A,B,C}

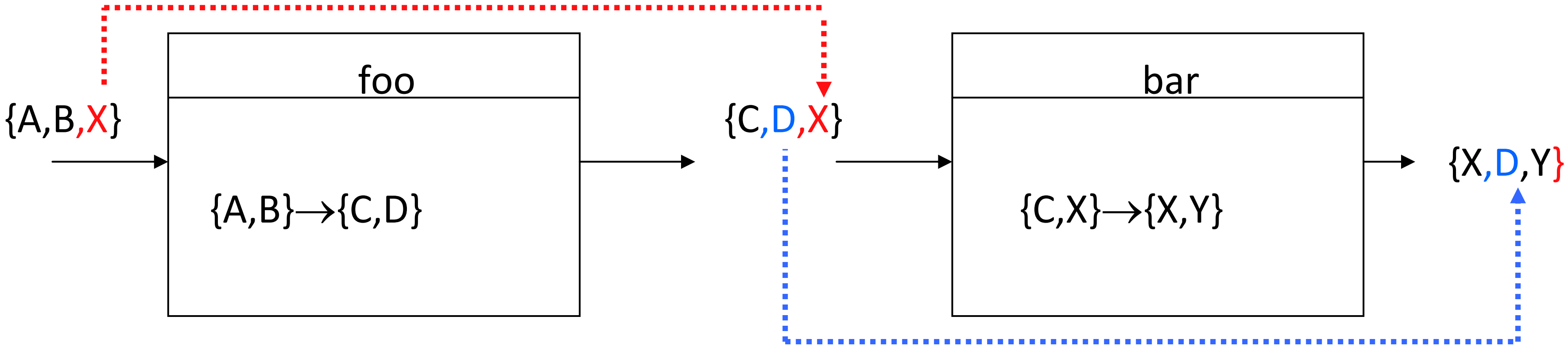
  here a superset is a subtype (can always add rules)

# Inheritance

- Box code typically designed in isolation
- Interfaces only partially overlap
  - Boxes may need extra parameters,etc...
- Network composition without redesign?

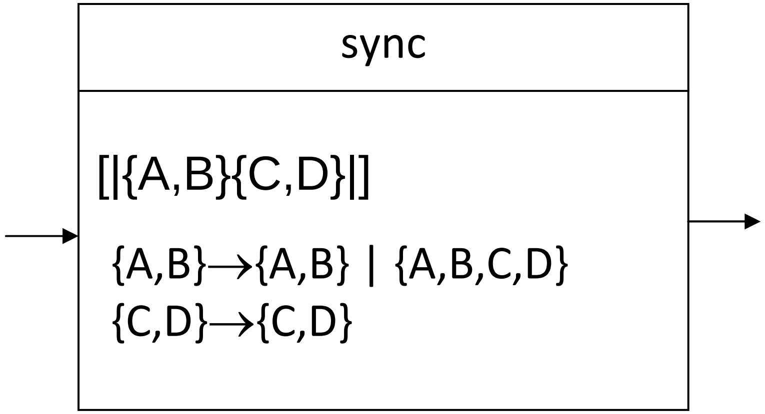


# Flow inheritance




- Observe that {A,B,X} is a subtype of {A,B}, hence aceptable as input of foo.
- *Instead of ignoring X, save it and attach it to all outputs of foo (thus lowering their type - which is valid). This is called "flow inheritance"*
- Similar with bar.
- The resulting signature is {A,B,X}→{X,D,Y}

# Inheritance for synchrocells




- Operational behaviour is symmetric
- Type signature is not
- Ignores the case when the synchrocell is *intended* not to synchronise
- Single inheritance (via the first pattern)
- The other patterns not inheriting, pure subtyping

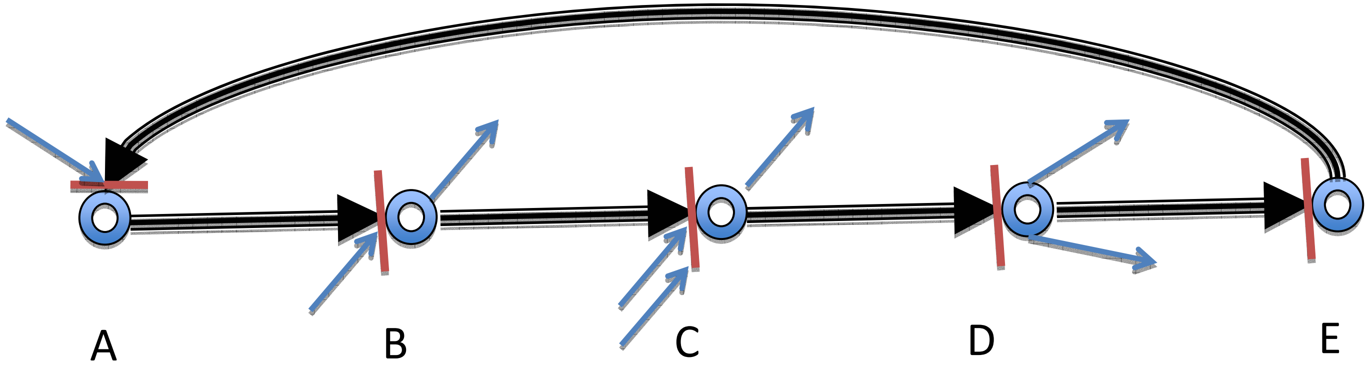
# Code example: parallel DES

```
net des ( {Key, Pt} -> {Ct}) {
  box xor( (Op1, Op2) -> (Result));
  box InitialP( (Pt) -> (L, R));
  box genSubKeys( (Key) -> (KeySet));
  box KeyInvert( (KeySet) -> (KeySet));
  box FinalP( (L, R) -> (Ct));
  net desRound {
    net feistel {
      net ExpandAndKeySelect {
        box BitExpand( (R) -> (Rx));
        box SubKey((KeySet, <C>)->(KeySet,NextKey,<C>));
      }
      connect
      [{R,KeySet,<C>}->{R};{KeySet,<C>}] ..
             (
               BitExpand
               |
               SubKey
             ) ..
      [|{KeySet,NextKey,<C>},{Rx}|]
                                *{Rx,KeySet,NextKey,<C>};
      net KeyMix
      connect
        [{NextKey, Rx} -> {Op1=NextKey, Op2=Rx}] ..
              xor .. [{Result} -> {BitStr=Result}];

      box Substitute( (BitStr) -> (SStr));
      box Pbox( (SStr) -> (Rf));
    }
    connect
    ExpandAndKeySelect .. KeyMix ..
    Substitute .. Pbox;
```

```
      net XorHalfBlocks
      connect
      [{L, Rf} -> {Op1=L, Op2=Rf}]
         .. xor .. [{Result} -> {R=Result}];
    }
    connect
     [{L,R,KeySet,<C>}
          ->{L,R,KeySet,<C=C+1>};{Rn=R}] ..
     (
       [{Rn} -> {L=Rn}]
             |
       (
       [{L,R,KeySet,<C>} -> {L}; {R, KeySet, <C>}]
        .. (
            [{L}->{L}]
             |
            feistel
           ) ..
        [|{L},{KeySet,Rf,<C>}|]*{L,KeySet,Rf,<C>} ..
          XorHalfBlocks
       )
     ) .. [|{L}, {R,KeySet,<C>}|]*{L,R,KeySet,<C>};
  }
  connect
   genSubKeys ..
        ( [] | ([{<Decipher>}->{}]..KeyInvert) ) ..
        InitialP ..
        [{L,R,KeySet} ->{L,R,KeySet,<C=0>}] ..
        desRound*{<C>} if <C==16> ..
        FinalP .. [{KeySet, <C>} -> {}];
```

# Threading by inheritance




vars X, Y, Z:

for(...) {

receive ...
segA **mod Y** ... send ...
send ...

receive ...
seg B ... send ...
send ...

receive ...
seg C ... send ...
send ...

receive ...
seg D ... send ...
send ...

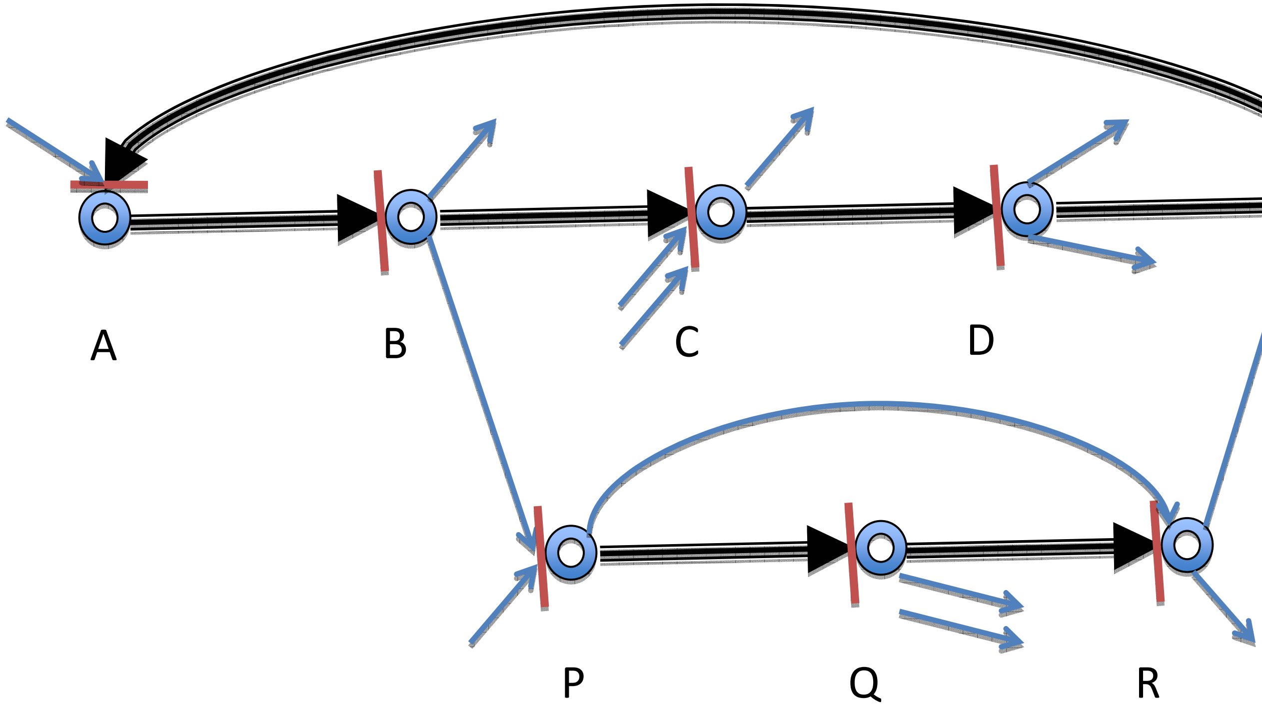
receive ...
seg E ... send ...
send ...

}

REPLACE control flow by data flow

encapsulate local vars in functional segments

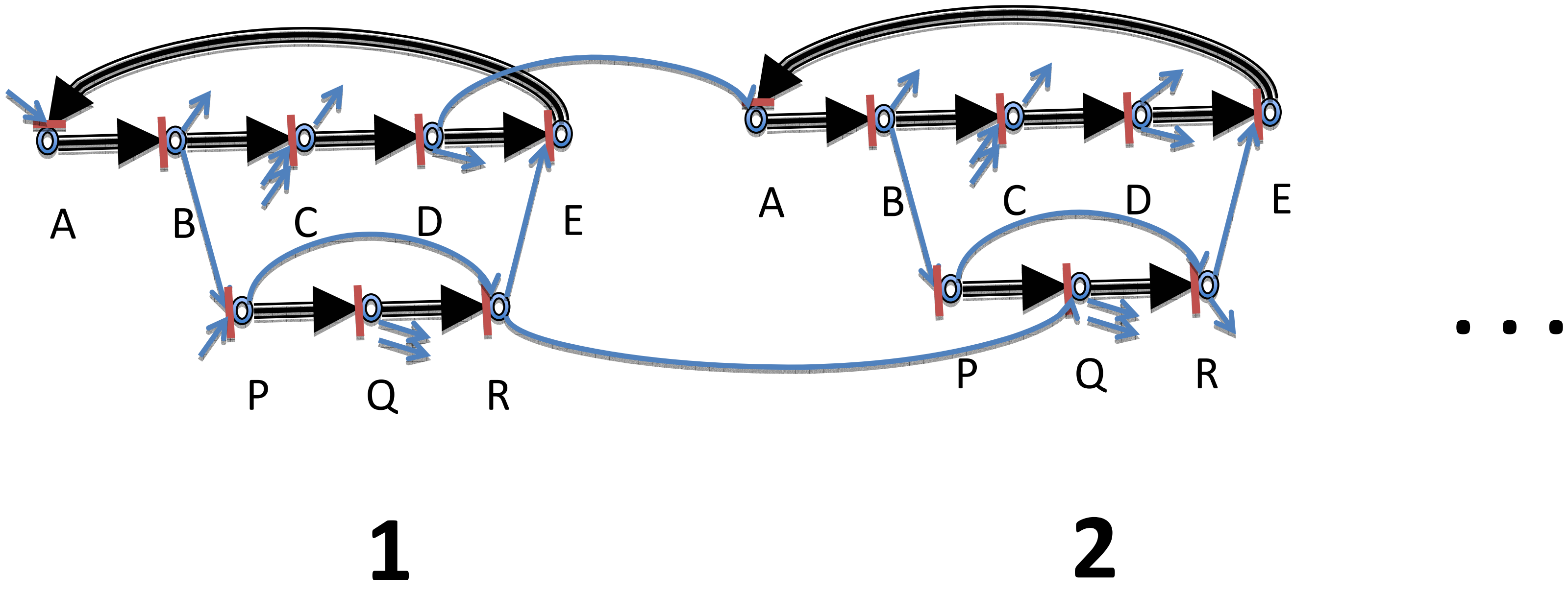
# Heterogeneity




Can spawn a new inheritance thread

encapsulate local vars in functional segments

# Add SPMD =>
# Spinal Vector Petri Net (SVPN)



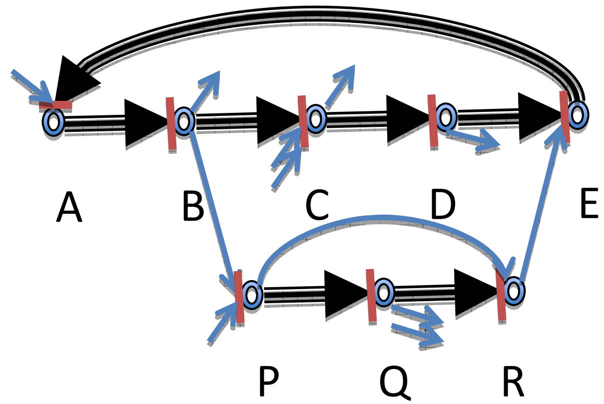

all messages carry a “virtual processor tag”

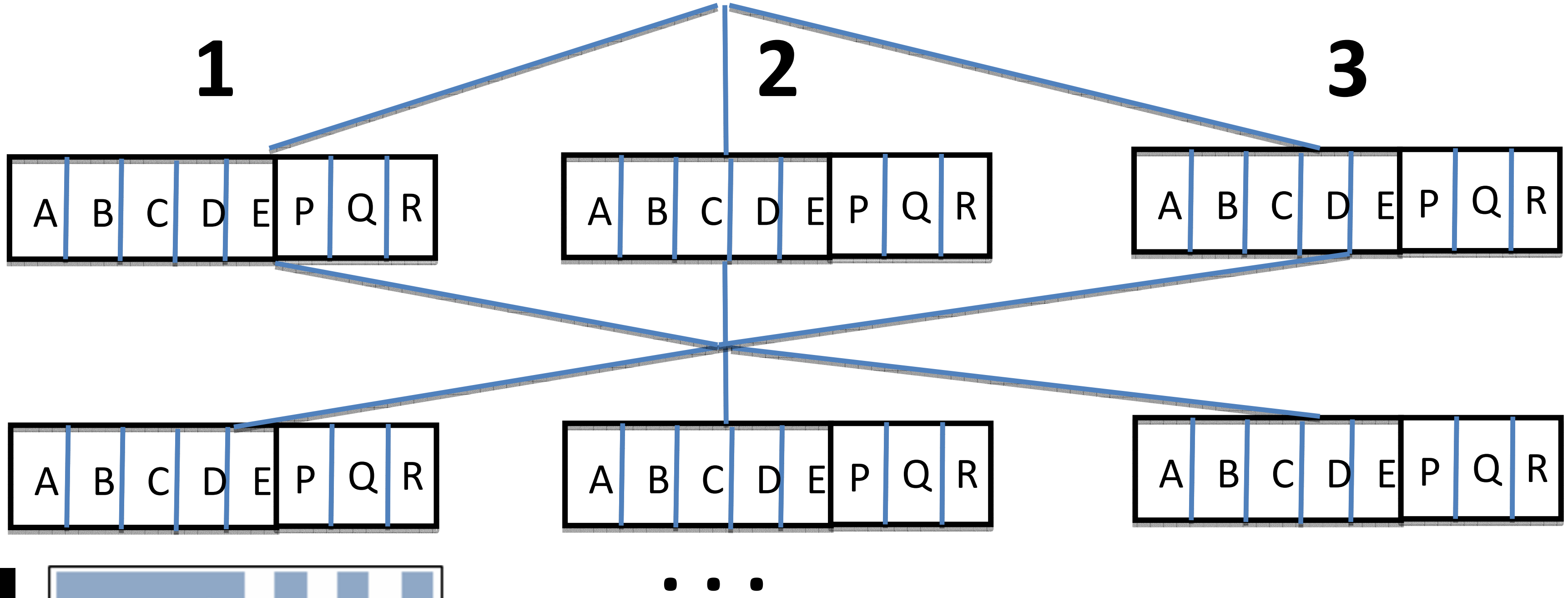
# Direct translation to S-Net

(Solver!<p>)*<out>

Solver = A|B|C|D|E|P|Q|R

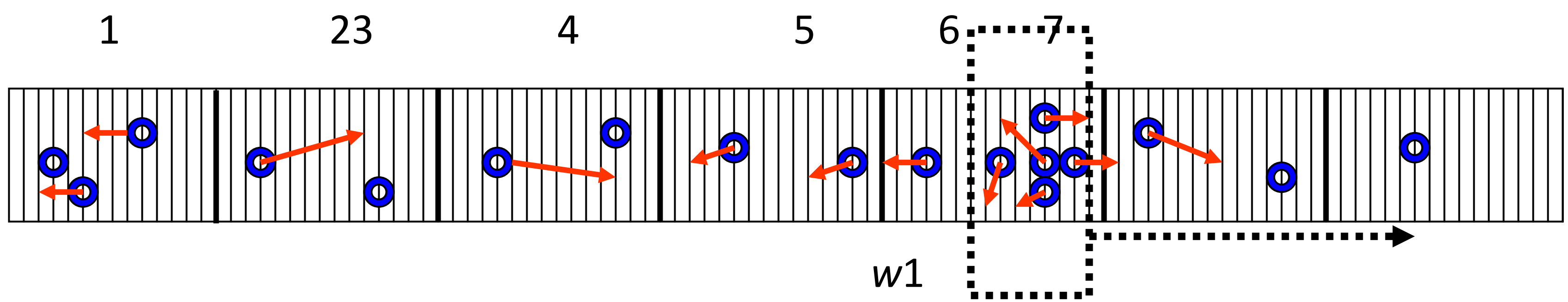
A = [|{}{message4A}|]..FuncA



# Particles in Cells (PIC)

- Simulation of plasma particles interacting with each other via electromagnetic field
- Consider 1d for simplicity, and 1 sort of particles

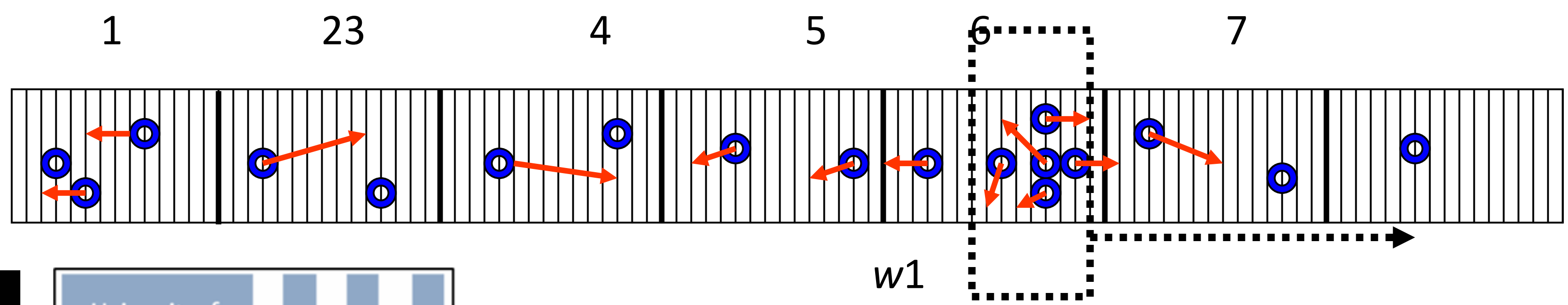



- Field split evenly, perfectly balanced, particles imbalanced.
- “Windows” are introduced representing work to be delegated to other processors.

University of Hertfordshire

# Basic concepts

- Particles are charged, each carrying a unit of charge.
- The field-grid nodes are assigned the charge of the particles near them, by interpolation.
- The field solver computes the field values due to the charges assigned to the nodes
- The particle pusher, applies EM forces to the particles due to the field values interpolated from the neighbouring nodes. The particles move accordingly.

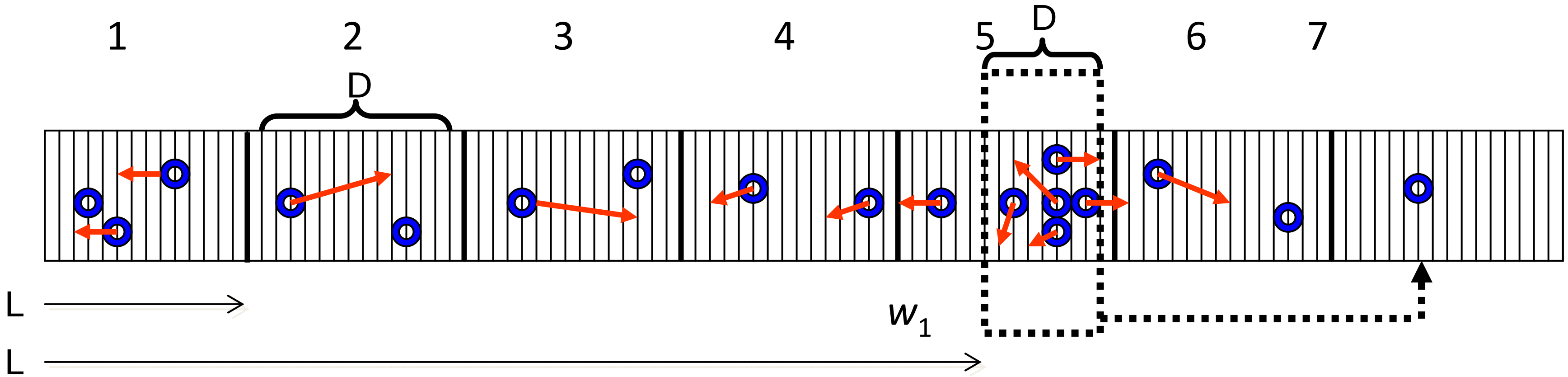


# Basic data structures

- The home record of a cell: particles pushed by the home base

  {<p>,<A>, ***Phi***, LD,<nw>, ***x*,*v***}
- The window record: particles pushed by a deputy

  {<p>,<A>,LD, ***x*, *v***, <return>,<id>}
- Processor tag <p>, stage <A>



# The net

<p>, <A>, ***Phi***, LD,<nw>,***x*,*v***

<p>, <A>,LD, ***x*, *v***, <return>

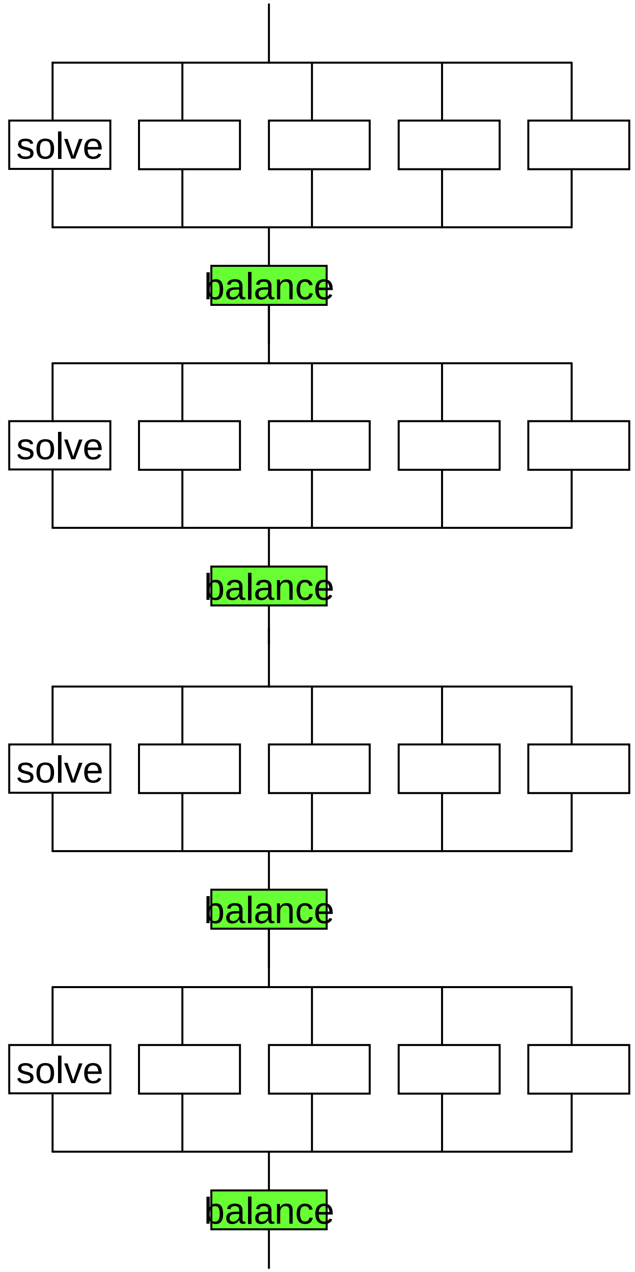
net solution {

net solve

connect stageA|| stageB|| stageC|| .. ;

}

connect (solve!<p>.. balance) * <out>



# Stages A,B,C

<p>,<A>,*Phi*, LD,<nw>, *x*,*v*
<p>,<A>,LD, *x*, *v*, <return>

net solution {
 net solve
connect stageA| stageB| stageC| .. ;
}
connect (solve!!<p>.. balance) * <out>

comments

**stageA:** {<p>,<A>, Phi, LD,<nw>, x}→{<p>,<B>,LD,rho},{<p>, <C>,rho} ,
{<p>,<C>,Phi,LD,<nw>,x}, as is
{<p>,<C>,Right}, p= p–1
{<p>,<C>,Left} p= p+1

{<p>,<A>,LD,x,v,<return>} → {{<p>,<B>,LD1,rho1} p= return
{<p>,<F>,<return>,LD, x,v} as is
{<p>,<D>,<loc>,LD1}} LD1=LD,loc=p,p=return

stageB=[| {<p>,<B>,LD,rho} {<B>,LD1,rho1} |]..cagr
cagr: {<B>,LD,LD1,rho,rho1} → {<C>,rho}
stageC=[| {<C>,Phi} {<p>,<C>,rho} {<C>,Right}{<C>,Left} |] ..fsolve..[{<D>} →{<D>};{<F>}]
fsolve: {<C>,rho,Phi,LD,Right,Left} →{<D>,Phi,LD}

# Stage A in detail: A1

stageA: {<p>,<A>,Phi,LD,<nw>,x}→{<p>,<B>,LD,rho}, {<p>,<C>,rho}, comments
{<p>,<C>,Phi,LD,<nw>,x}, as is
{<p>,<C>,Right}, p= p–1
{<p>,<C>,Left} p= p+1
{<p><A> LD, x, v, <return>} → {<p>,<B>,LD1,rho1}, p= return
{<p>,<F>,<return>,LD, x, v}, as is
{<p>,<D>,<loc>,LD1} LD1=LD, loc=p,p=return

stageA= stageA1 |stageA2

stageA1 = [{<A>,<nw>} if <nw==1>->{<B>,LD}; {<C>,<nw>}
->{<C>}; {<C>,<nw>}]
.. (
interpolate |
[{<nw>} ->{<nw>,<i>};{<nw>,<ii>}]..
( getEndPoints | [{<ii>}->] )
)

interpolate : {x, LD} → {rho, LD}

getEndPoints: {<p>,Phi, LD} → {<p>,Right}, {<p>,Left}

# Stage A in detail: A2

stageA: {<p>,<A>,Phi, LD,<nw>,x}→ {{<p>,<B>LD,<nw>,rho} comments

{<p>,<C>,Phi,LD,<nw>,x} as is

{<p>,<C>,Right} p= p–1

{<p>,<C>,Left}} p= p+1

{<p><A> LD, x, v<return>} → {<p>,<B>,LD1,rho1} p= return

{<p>,<F>,<return>,LD,x,v}, as is

{<p>,<D>,<loc>,LD1} LD1=LD,loc=p,p=return

stageA2 = [{<A>,<return>} → {<B>,<return>};{<F>,<return>};{<D>}]..

(

[{<B>,<return>} → {<B>,<p>=<return>}]..interpolate |

[{<D>,<return>,<p>} → {<D>,<loc>=<p>,<p>=<return>}] |

[ ]

)

interpolate : {x, LD} → {rho, LD}

University of Hertfordshire

# Conclusions

- SNet suggests a top down design style
- Records contain the state of computation, float between boxes
- Topology induced by tagging and type match
- Define signatures and insert synchronisers first
- Then refine signatures down to networks
- Finally the lowest level boxes should be stateless and generic

University of Hertfordshire UH

Universiteit van Amsterdam

VTT

University of Twente.

Imperial College London

Seventh Framework Programme

University of Hertfordshire UH