# The CodeTime Platform

# for Parallel Software

Presented By

Sean Halle

# Context

- What is the Problem?
  - Parallel software which is efficient is difficult to develop and maintain
  - Parallel software which is efficient must have its source modified for new hardware (and OS)
- What is my Proposed Solution?
  - A computation model which adds a coordination extension to Large Grain Dataflow
  - An all-inclusive platform built around that computation model

# *Outline of This Talk*

- Introduce the CodeTime platform
- Focus on aspects interesting to Application Developers
- Focus on detail for informed feedback from Compiler Researchers
- Describe the platform:
  - Show elements of the platform
  - Describe the function of each element
- Go in-depth on
  - Core idea == extending Large Grain Dataflow with coordination
  - A simple run-time system, the first working piece of the platform

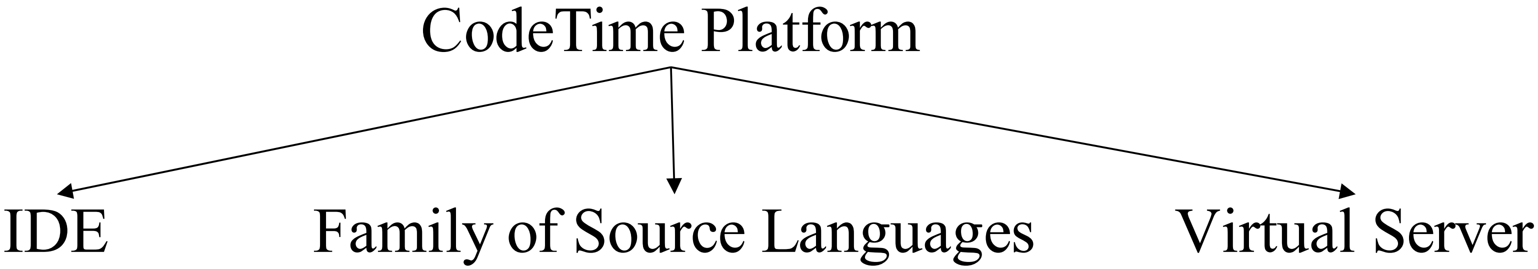
# *Scope of the Platform*

- The CodeTime platform covers all interactions with software:

| | |
|---|---|
| Creation | (Source Language, IDE) |
| Translation | (Source Compiler, Intermediate Format) |
| Testing | (Test harness in the IDE) |
| Maintenance | (IDE, language features, OS interf. Features) |
| Distribution | (Intermediate Format, IDE, OS interface) |
| Installation | (Back-end Compiler, Virtual Server) |
| Invocation | (Virtual Server, OS interface) |

# *Elements of the Platform*

- Three top-level components:



CodeTime Platform

IDE

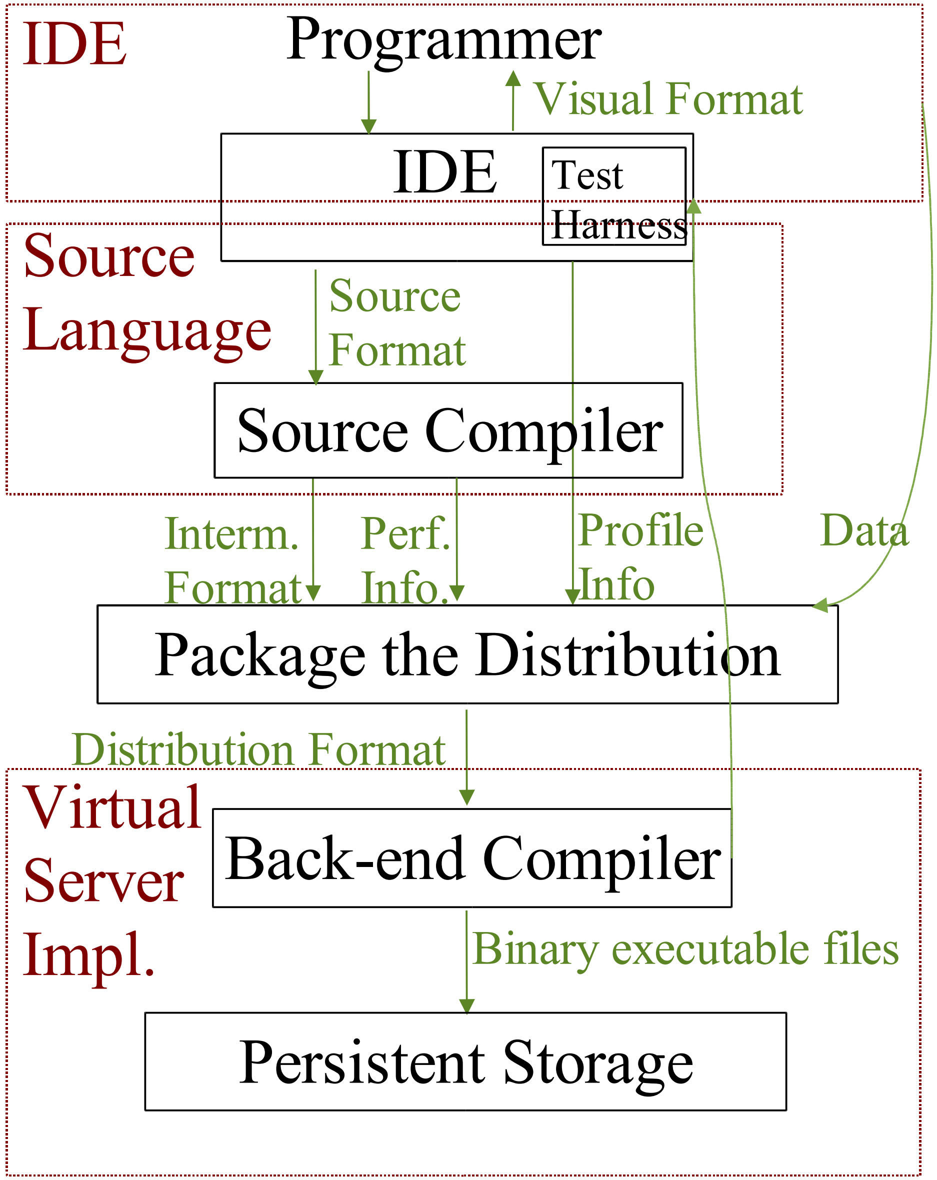
Family of Source Languages

Virtual Server

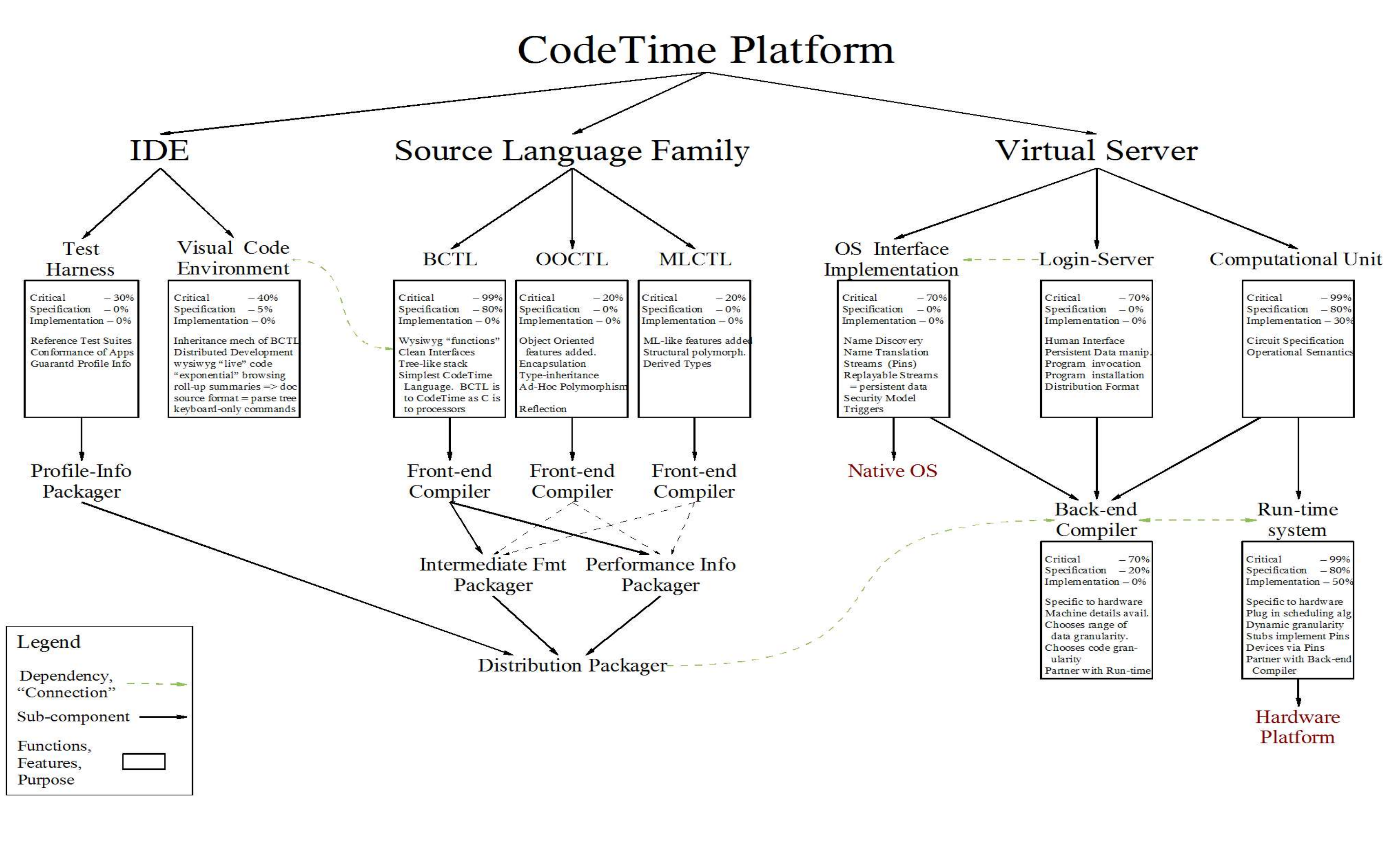
- Virtual Server is the core of the platform:
  - Embodies the computation model
  - OS interface
  - Holds persistent data
  - (each “machine” has a virtual server written specifically for it)

# Platform's Interactions with Software

- Creation
- Translation
- Testing
- Maintenance
- Distribution
- Installation
- Invocation



# Detailed Elements of the Platform



CodeTime Platform

IDE

Source Language Family

Virtual Server

Test Harness

Critical – 30%
Specification – 0%
Implementation – 0%

Reference Test Suites
Conformance of Apps
Guarantd Profile Info

Profile-Info Packager

Visual Code Environment

Critical – 40%
Specification – 5%
Implementation – 0%

Inheritance mech of BCTL
Distributed Development
wysiwyg "live" code
"exponential" browsing
roll-up summaries => doc
source format = parse tree
keyboard-only commands

BCTL

Critical – 99%
Specification – 80%
Implementation – 0%

Wysiwyg "functions"
Clean Interfaces
Tree-like stack
Simplest CodeTime Language. BCTL is to CodeTime as C is to processors

OOCTL

Critical – 20%
Specification – 0%
Implementation – 0%

Object Oriented features added.
Encapsulation
Type-inheritance
Ad-Hoc Polymorphism

Reflection

MLCTL

Critical – 20%
Specification – 0%
Implementation – 0%

ML-like features added
Structural polymorph.
Derived Types

Front-end Compiler

Front-end Compiler

Front-end Compiler

Intermediate Fmt Packager

Performance Info Packager

Distribution Packager

OS Interface Implementation

Critical – 70%
Specification – 0%
Implementation – 0%

Name Discovery
Name Translation
Streams (Pins)
Replayable Streams = persistent data
Security Model
Triggers

Native OS

Login-Server

Critical – 70%
Specification – 0%
Implementation – 0%

Human Interface
Persistent Data manip.
Program invocation
Program installation
Distribution Format

Computational Unit

Critical – 99%
Specification – 80%
Implementation – 30%

Circuit Specification
Operational Semantics

Back-end Compiler

Critical – 70%
Specification – 20%
Implementation – 0%

Specific to hardware
Machine details avail.
Chooses range of data granularity.
Chooses code granularity
Partner with Run-time

Run-time system

Critical – 99%
Specification – 80%
Implementation – 50%

Specific to hardware
Plug in scheduling alg
Dynamic granularity
Stubs implement Pins
Devices via Pins
Partner with Back-end Compiler

Hardware Platform

Legend

Dependency, "Connection"

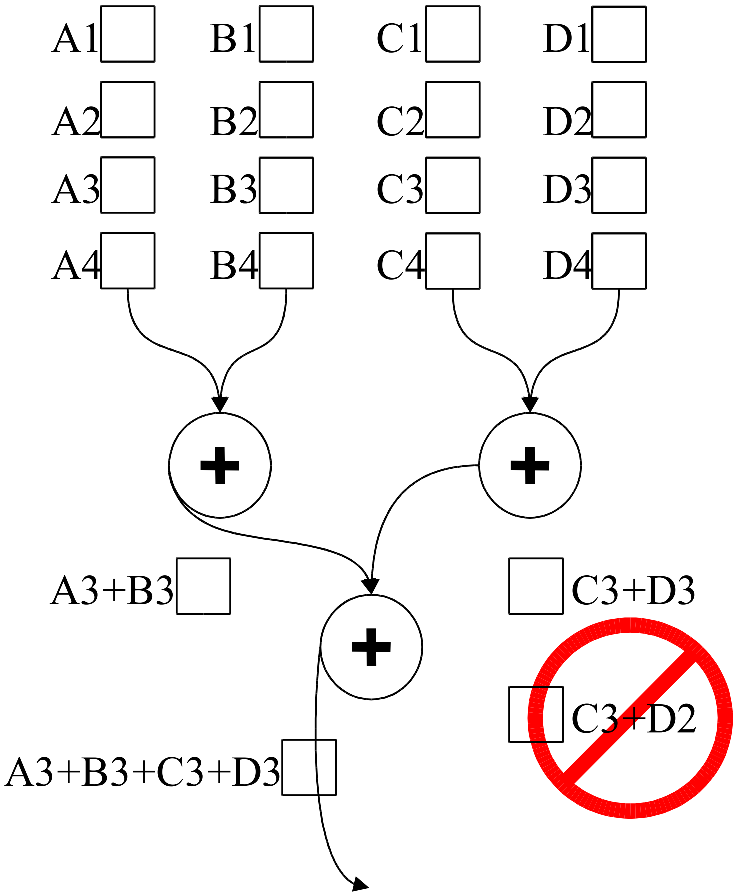
Sub-component

Functions, Features, Purpose

# Intro || Core Concepts

# *Effect of CodeTime Ext. to Dataflow*

## Classic Dataflow



A1 B1 C1 D1

A2 B2 C2 D2

A3 B3 C3 D3

A4 B4 C4 D4

+ +

A3+B3

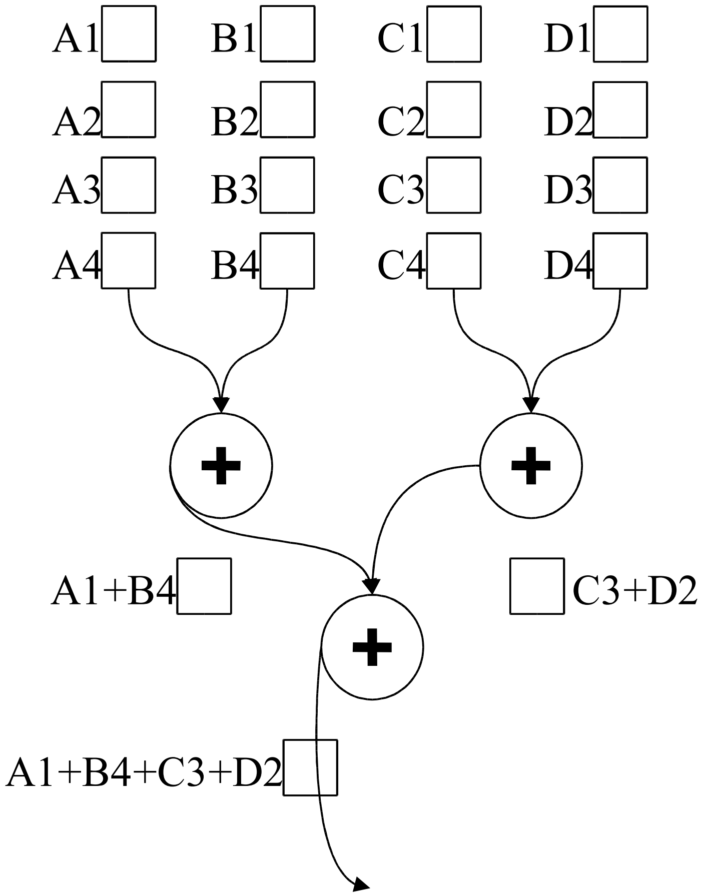
C3+D3

+

C3+D2

A3+B3+C3+D3

## With CodeTime Extension



A1 B1 C1 D1

A2 B2 C2 D2

A3 B3 C3 D3

A4 B4 C4 D4

+ +

A1+B4

C3+D2

+

A1+B4+C3+D2

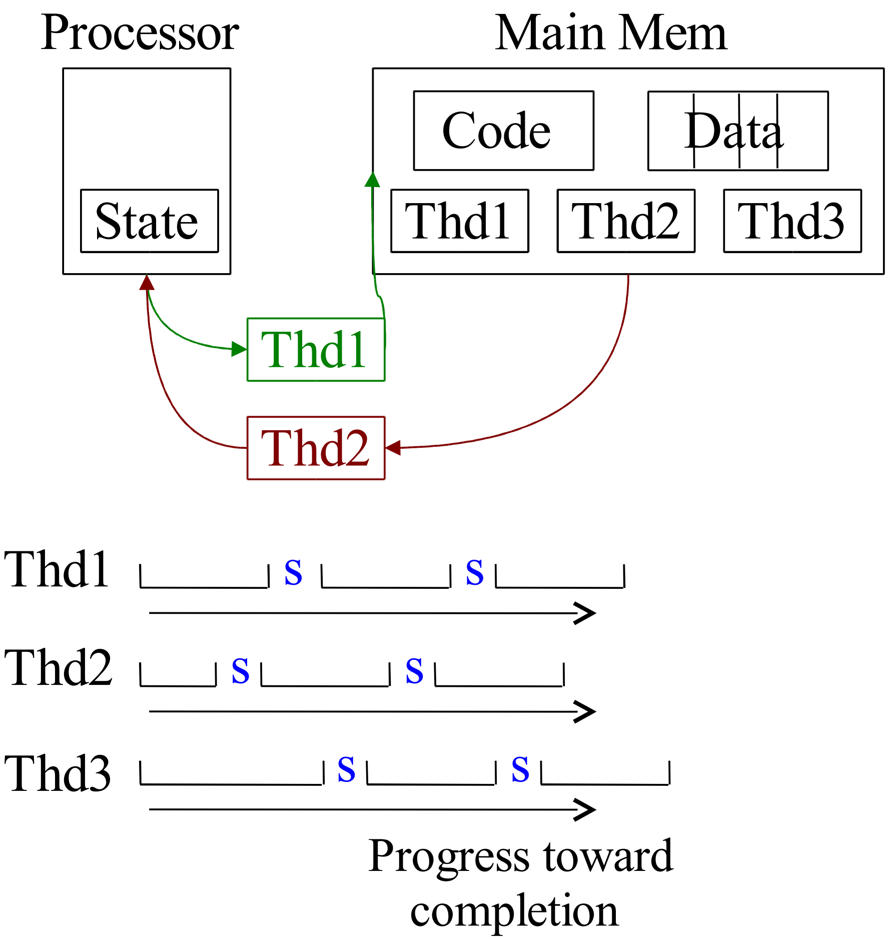
# *What's Extension Buy? Why Care?*

- Starting with straight Dataflow, get:
  - No Side Effects
  - Large amounts of parallelism (Instr level parallelism)
- Large Grain adds:
  - More familiar mental-model for programming
  - Translation to multi-processor machines more straight-forward
- CodeTime extension plus memory model add:
  - Fewer constraints on order of execution, but still correct result
  - Thread-level parallelism
  - Declarative control of thread-level parallelism (easier to program)
  - Straight-forward change of code-granularity (at install-time)
  - Straight-forward change of data-granularity (at run-time)

# *A Core Concept: Thread as Data*

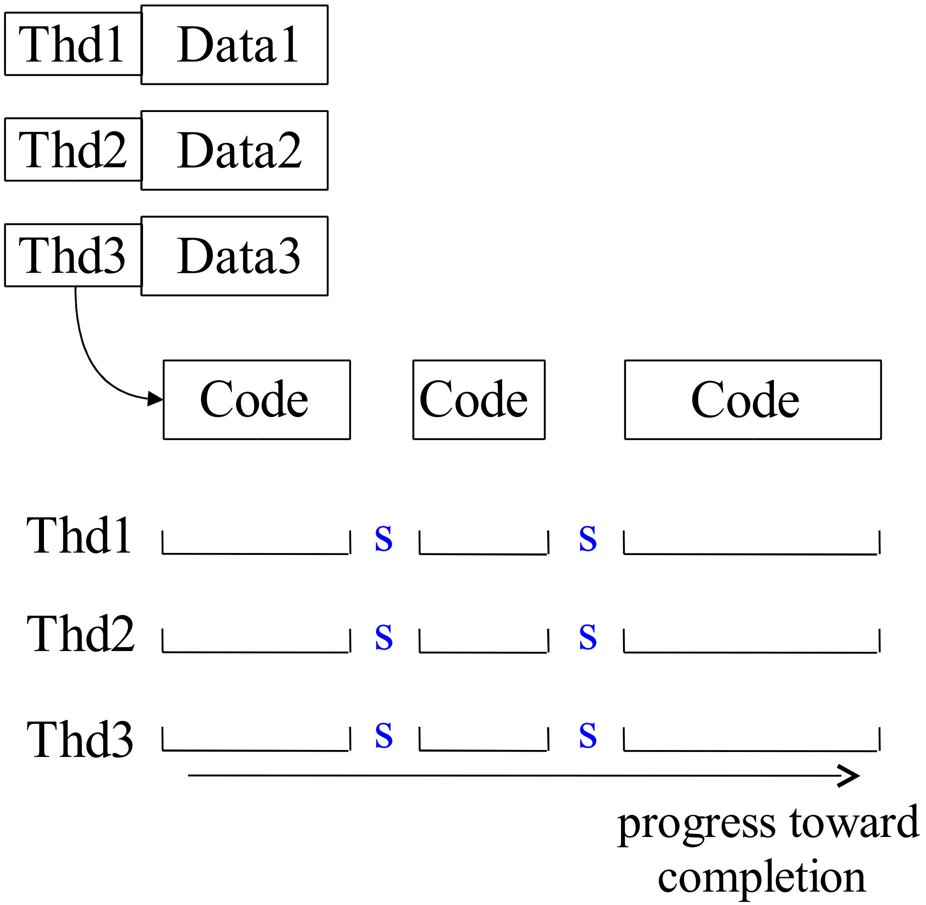
- Each function defines a custom “processor”
- In C.T. save this processor-state with the data, rather than with the code
- Index vars, relation of data to other data, etc, travel with the data
- Means *position in execution* = pos in circuit plus contents of data
- Processor-centric lang:: *position in execution* = time:: syncs, guards... control *position in execution* that threads get shared data.
- CodeTime:: *position in execution* is data, not *time*:: control sharing by conditions on data not *time order*
  - have data and **entire** state of the computation being performed on it, together
  - Pairing means self-contained “tasks”:: advance comp. a little bit at each location
  - Progress of task seen by location and data
  - “task” = thread:: notion of “thread” now *passive* data, rather than an *active* thing.

# *Threads in CodeTime*



Instrs accumulate as **time** advances

Scheduling decisions are made at regular time-intervals and at time when progress reaches a sync point



Instrs accumulate as **position** advances

Scheduling decisions are made each time data moves

Atomic sequences of Instructions, separated by Scheduling decisions

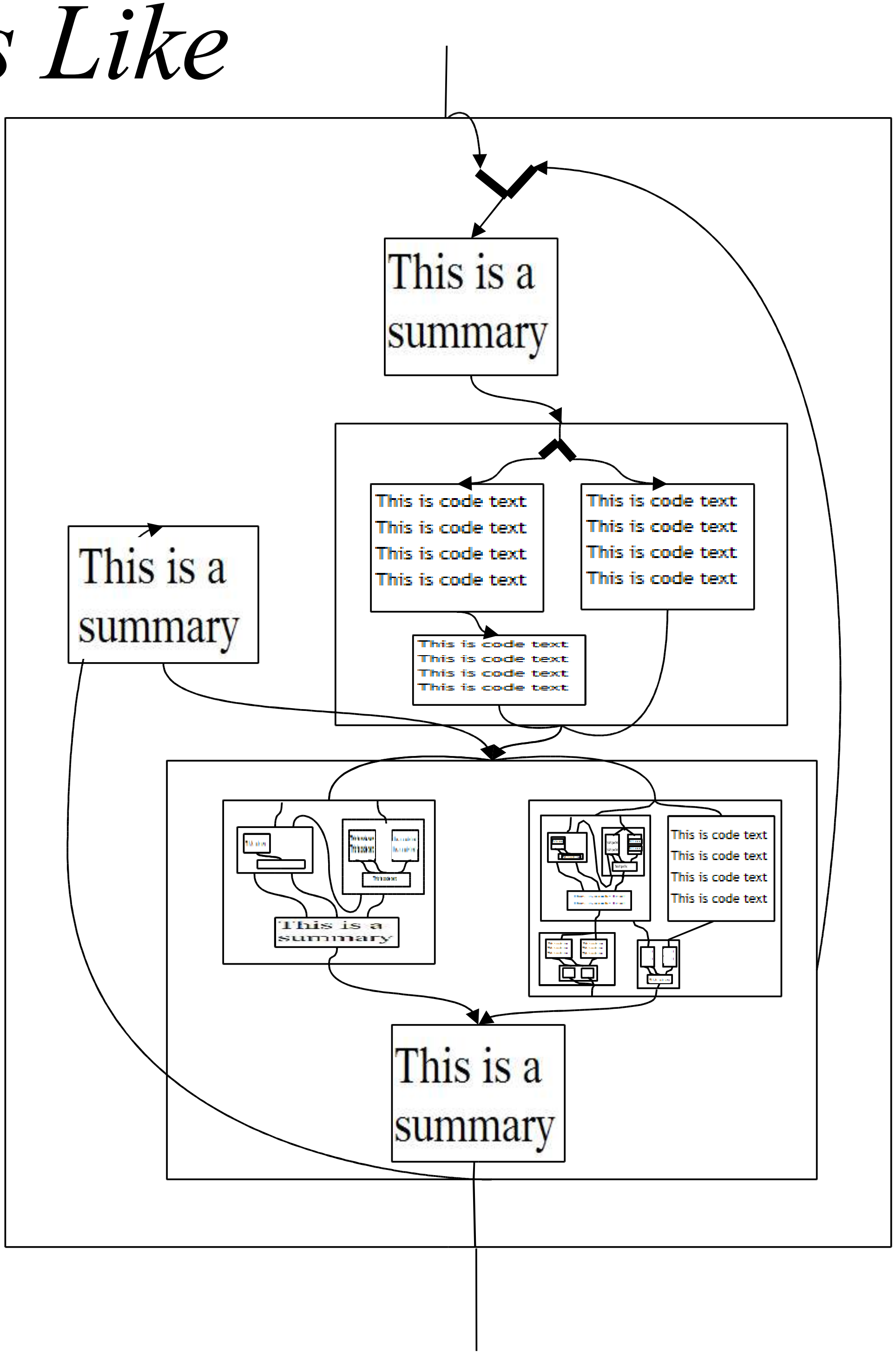
# Core Ideas || BCTL Lang.

# *Introduction to BCTL*

- **B**ase **C**ode**T**ime **L**anguage
- Low level: BCTL is to C.T. computation model as C is to processors
- Compiles to CodeTime's circuit-based intermediate format
- Visual language, intended for wysiwyg coding
- Memory model = collection of separate address spaces (each w/name)
- Organised into Function-Units (code-units) and Hierarchy-Units
- Tags, tag-code, and coord-code implement Dataflow extensions
  - Tags help hold current thread-state (for code-defined processor)
  - Coordination code declares when safe to join threads

# What Code Looks Like

- Structure of code apparent
- Body of "function" visible
- Link or copy: Link = Inheritance
- Keyboard navigation
- Quickly find code of interest
- Can roll-up to see summary
- Full-text search on summaries
- Coders motivated – searches useful to them
- Coders motivated – see and use summaries everyday



# *Example Program:  Vector Reduction*

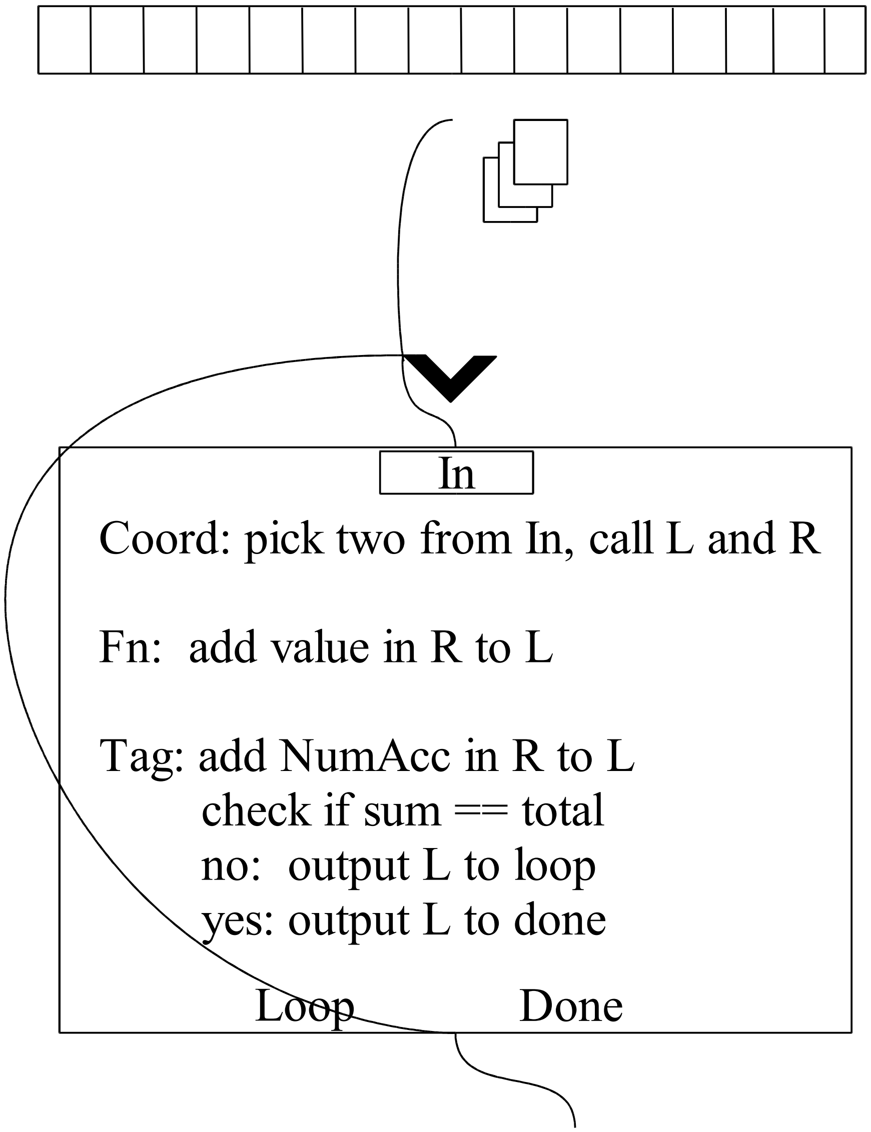
- Vector sliced into elems
- Elem = separate thread
- Pick two from input pool

- Put sum back into input pool
- Until all Elems summed

ElemContainer

Value:    float

NumAcc:  int
total:     const


In

Coord: pick two from In, call L and R

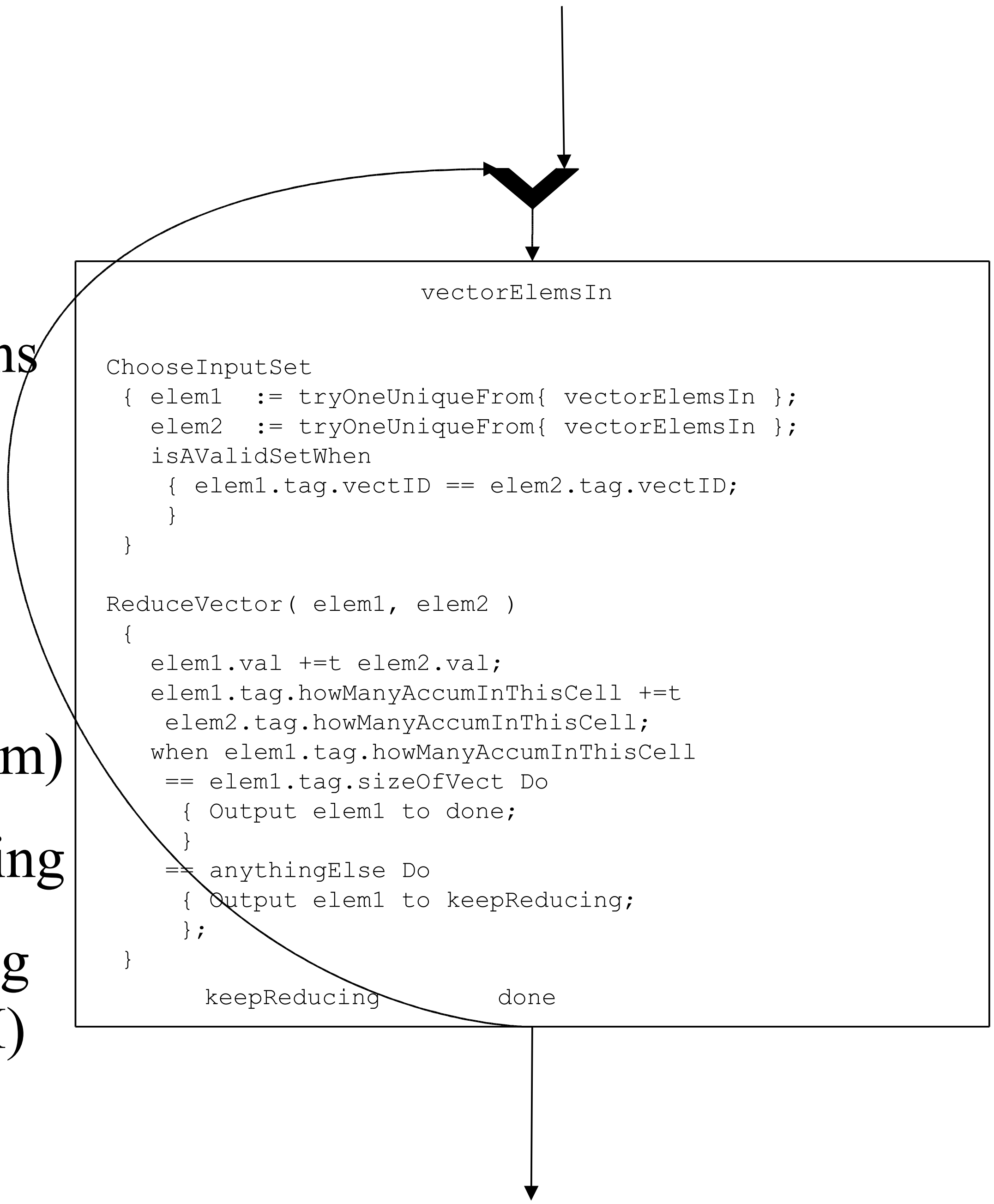
Fn:  add value in R to L

Tag: add NumAcc in R to L
        check if sum == total
        no:  output L to loop
        yes: output L to done

# Real Code

- Separate vectors via vectID
- Time is not defined
- Any number of f() invocations

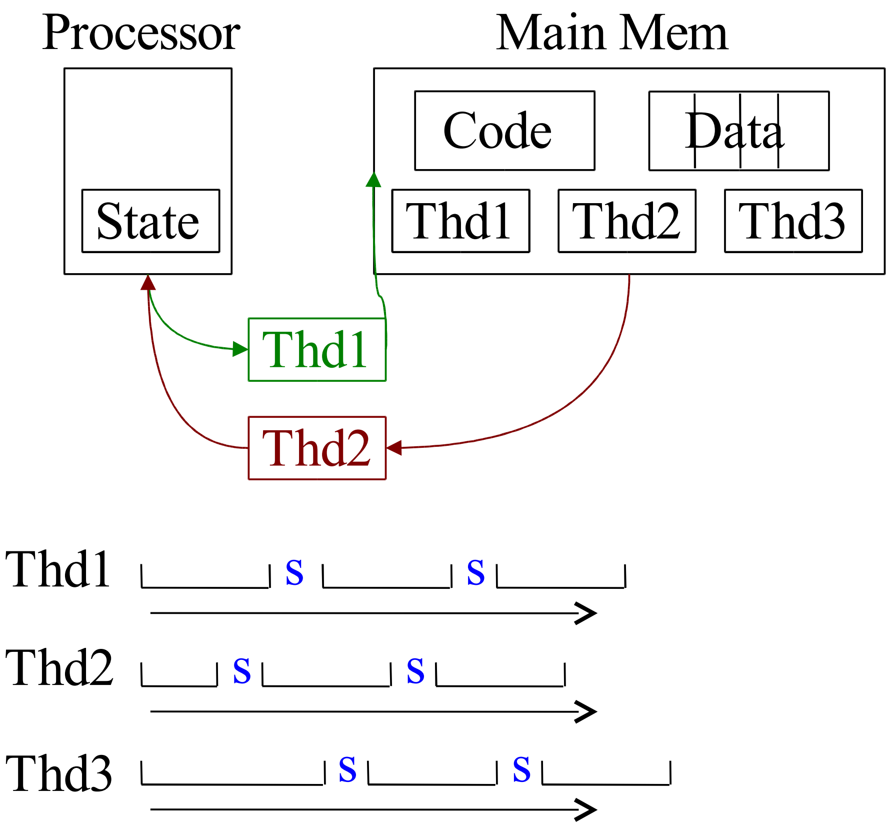
- Parallelism comes from the separate threads (one per elem)
- order undefined => any pairing
- Vs. Processor-centric: pairing precisely defined (try in MPI)



```
ChooseInputSet
 { elem1  := tryOneUniqueFrom{ vectorElemsIn };
   elem2  := tryOneUniqueFrom{ vectorElemsIn };
   isAValidSetWhen
    { elem1.tag.vectID == elem2.tag.vectID;
    }
 }

ReduceVector( elem1, elem2 )
 {
   elem1.val +=t elem2.val;
   elem1.tag.howManyAccumInThisCell +=t
    elem2.tag.howManyAccumInThisCell;
   when elem1.tag.howManyAccumInThisCell
    == elem1.tag.sizeOfVect Do
     { Output elem1 to done;
     }
    == anythingElse Do
     { Output elem1 to keepReducing;
     };
 }
```

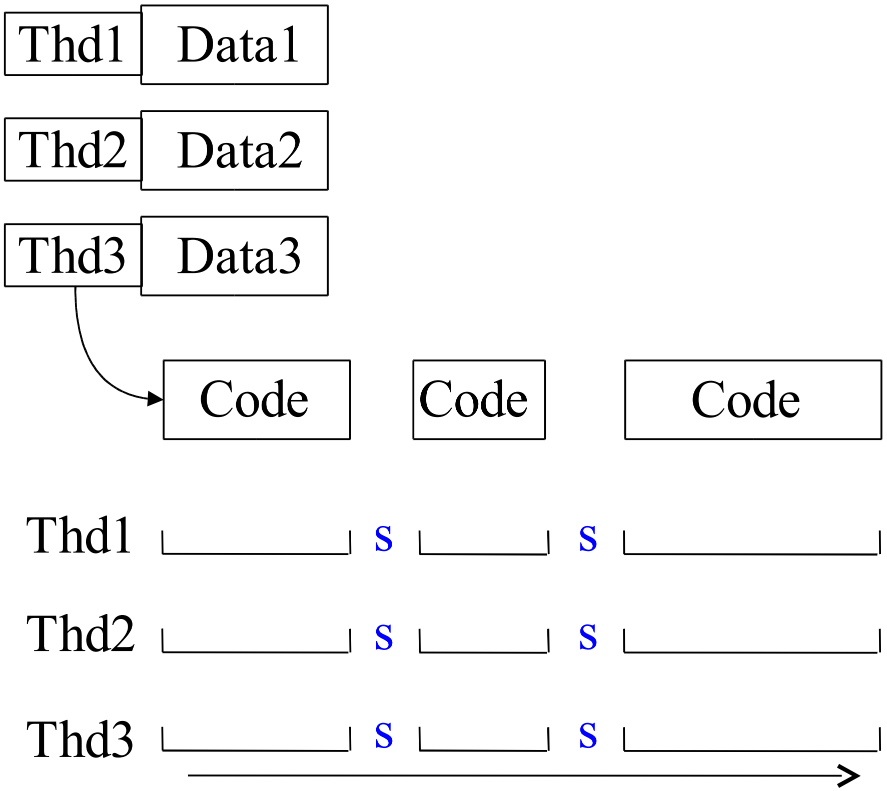
# Threads in CodeTime



Progress toward
completion

Instrs accumulate as **time** advances

Scheduling decisions are made at regular time-intervals and at time when progress reaches a sync point



progress toward
completion

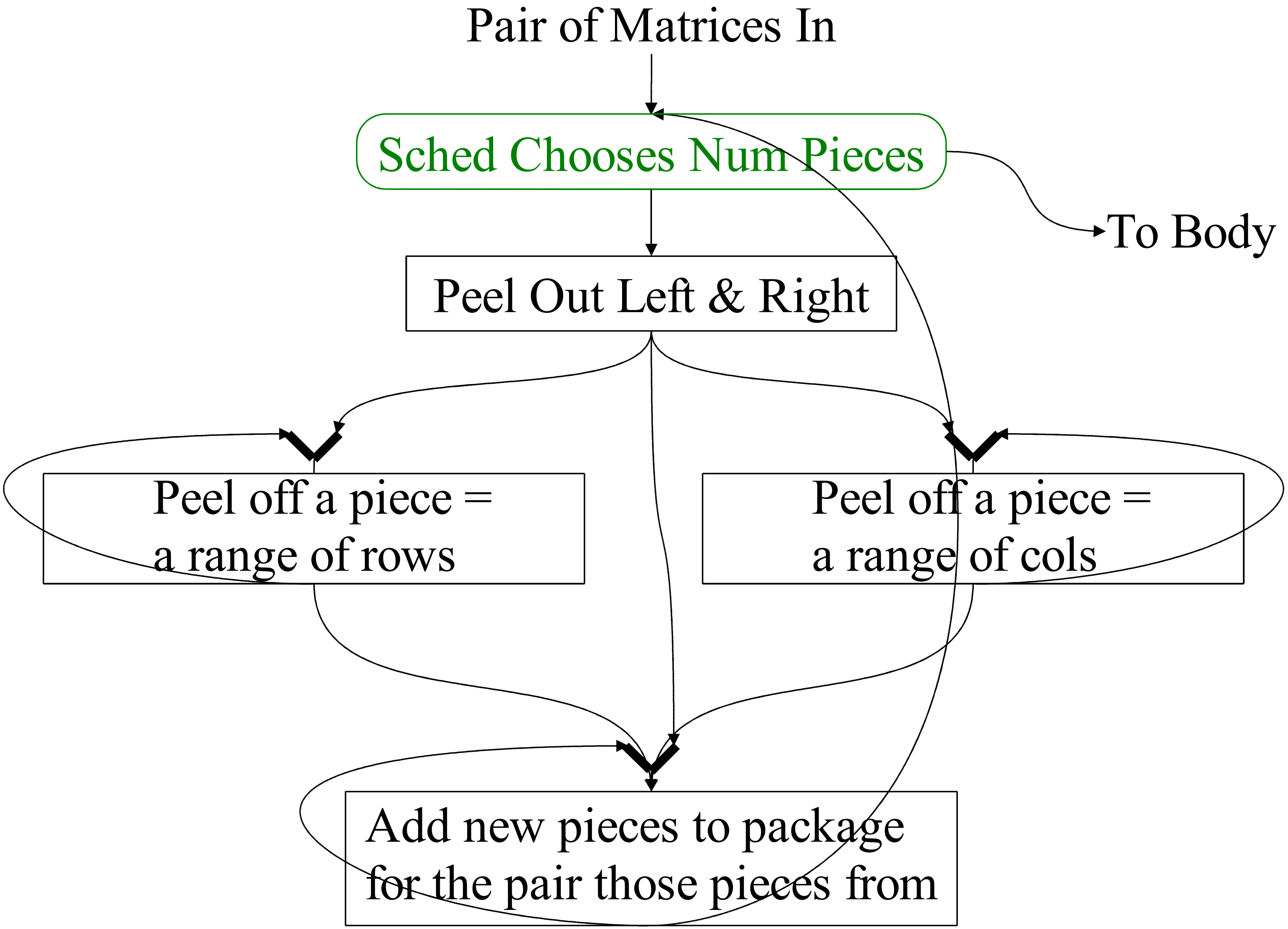
Instrs accumulate as **position** advances

Scheduling decisions are made each time data moves

Atomic sequences of Instructions, separated by Scheduling decisions

# BCTL Lang || Run-Time

# A Divider



Pair of Matrices In

Sched Chooses Num Pieces

To Body

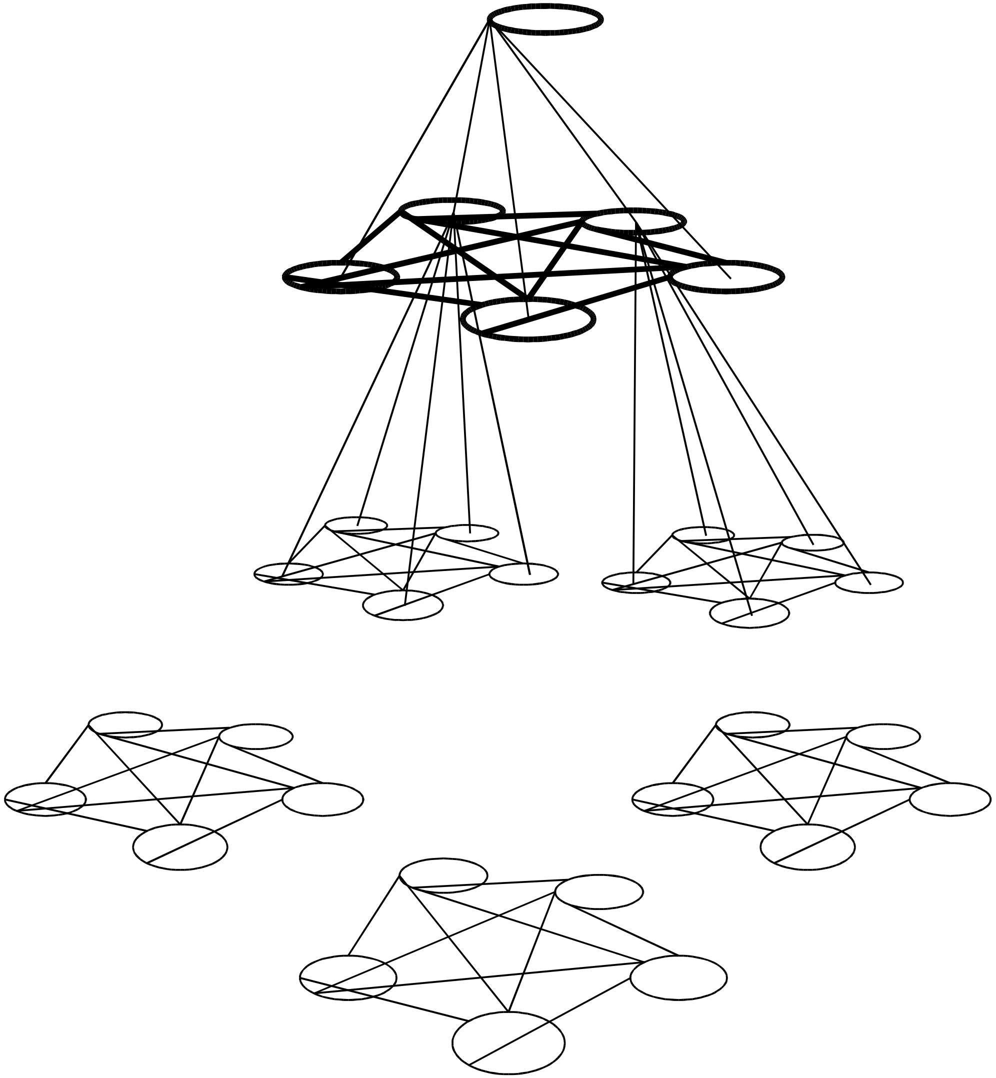
Peel Out Left & Right

Peel off a piece =
a range of rows

Peel off a piece =
a range of cols

Add new pieces to package
for the pair those pieces from

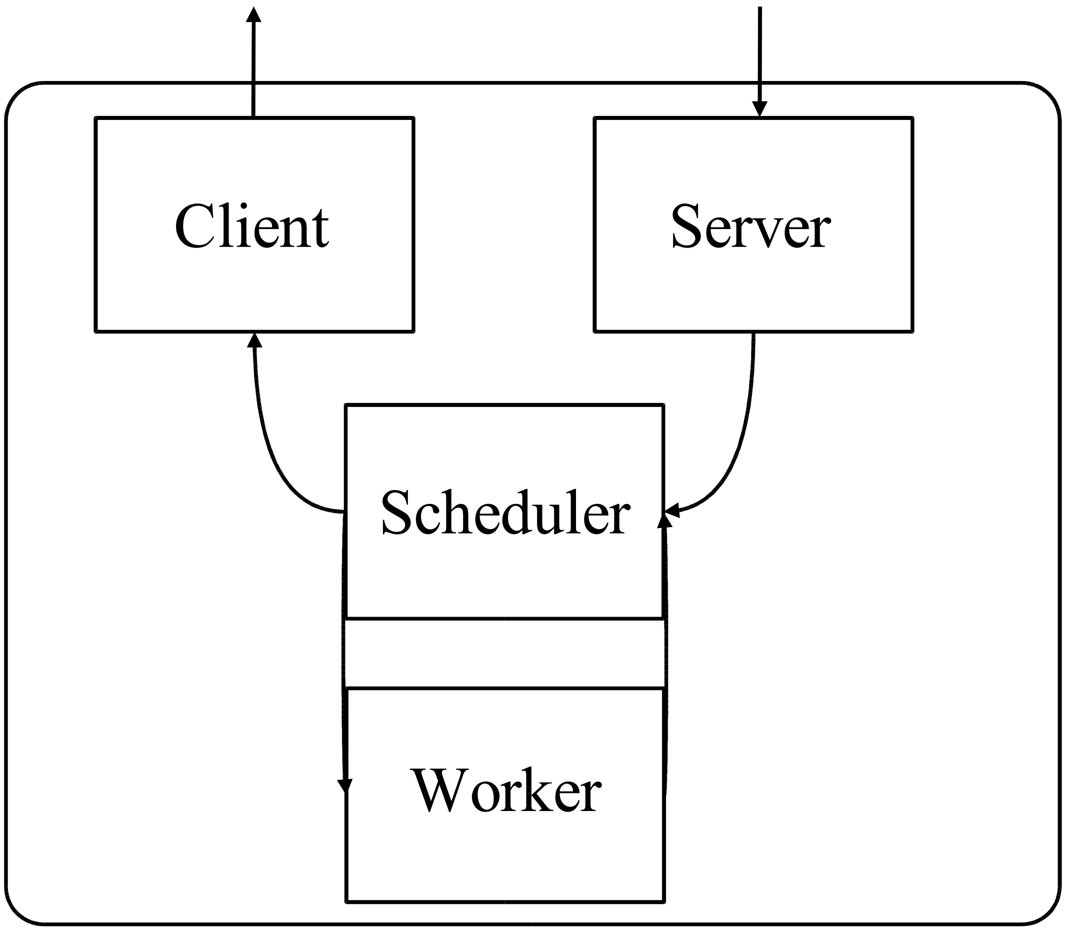
# The Tree-Graph Hierarchy of Peers



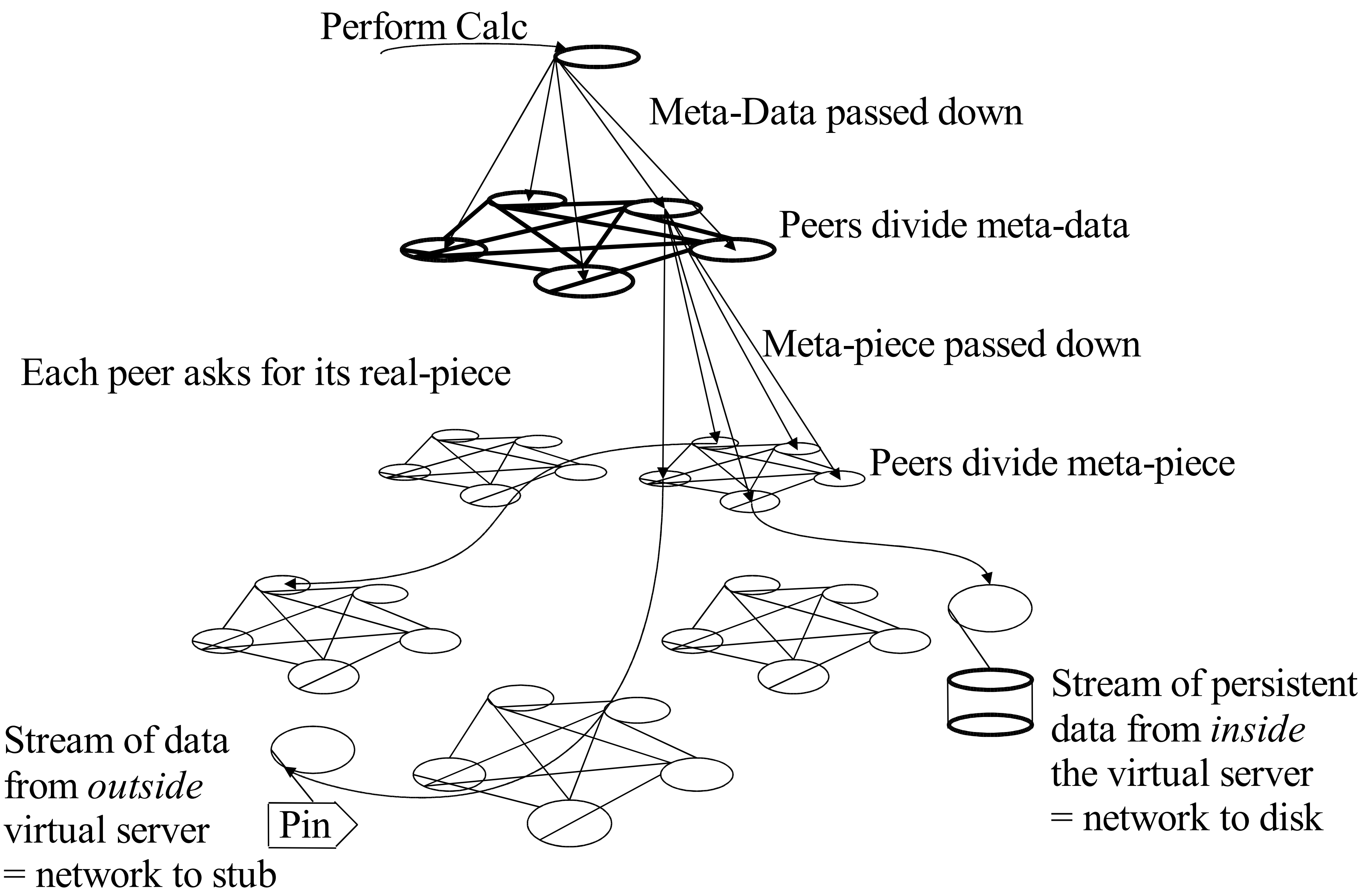
Upper-Level Peers

Leaf-Peers

# Peer Internals



# Dividing Data and Getting Pieces



Perform Calc

Meta-Data passed down

Peers divide meta-data

Meta-piece passed down

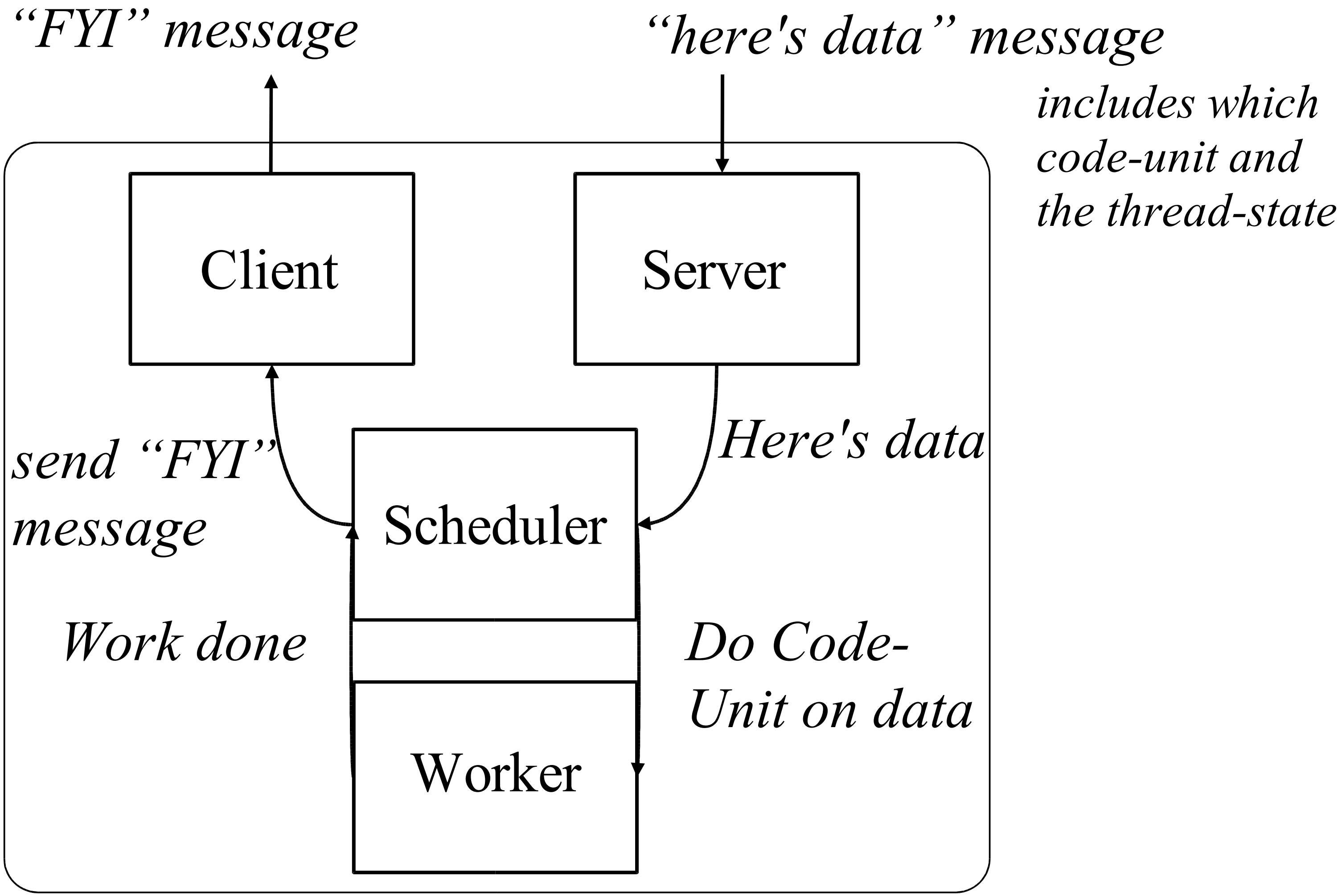
Each peer asks for its real-piece

Peers divide meta-piece

Stream of data
from *outside*
virtual server
= network to stub

Stream of persistent
data from *inside*
the virtual server
= network to disk

# *Performing Work*

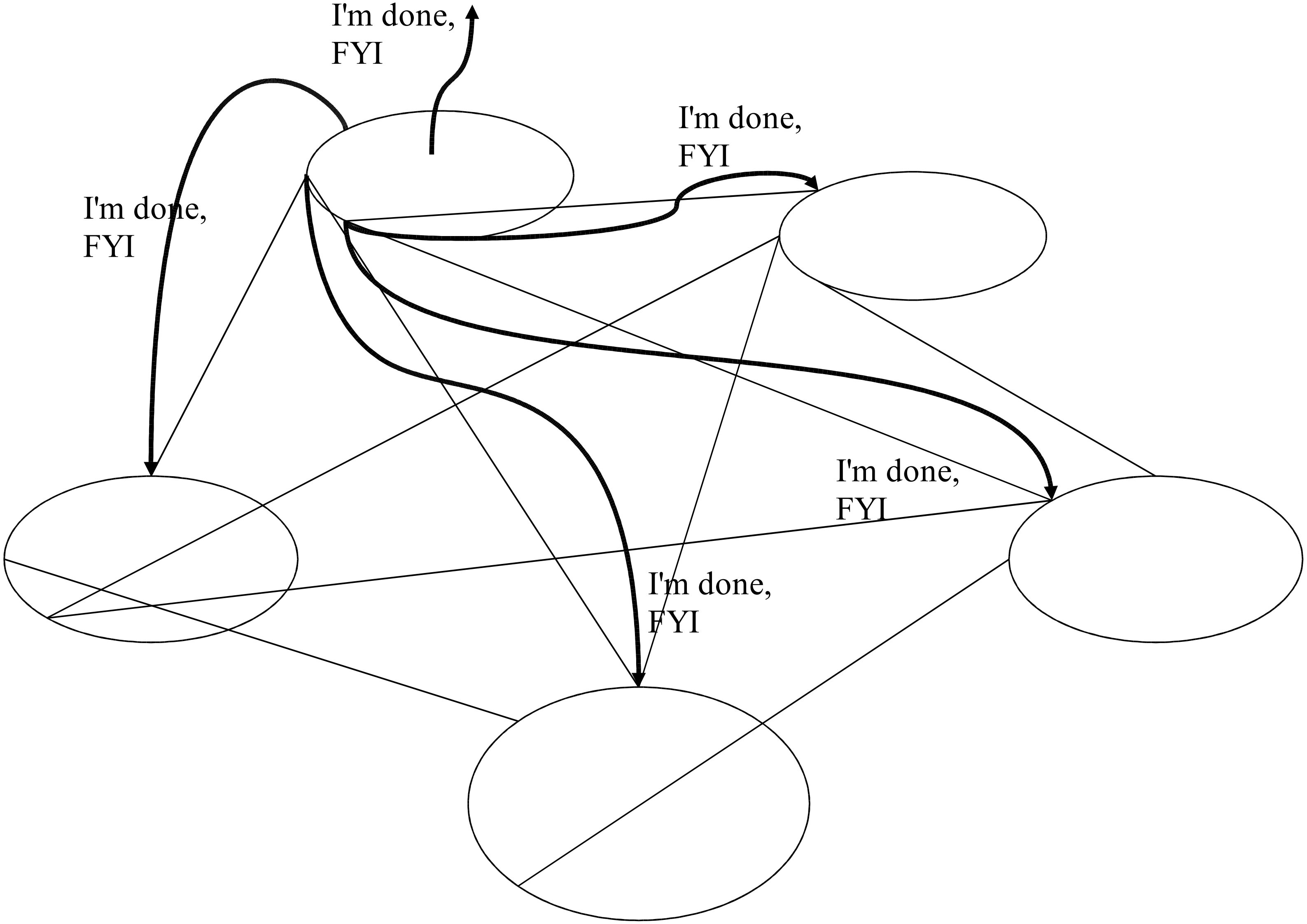


*“FYI” message*

*“here's data” message*

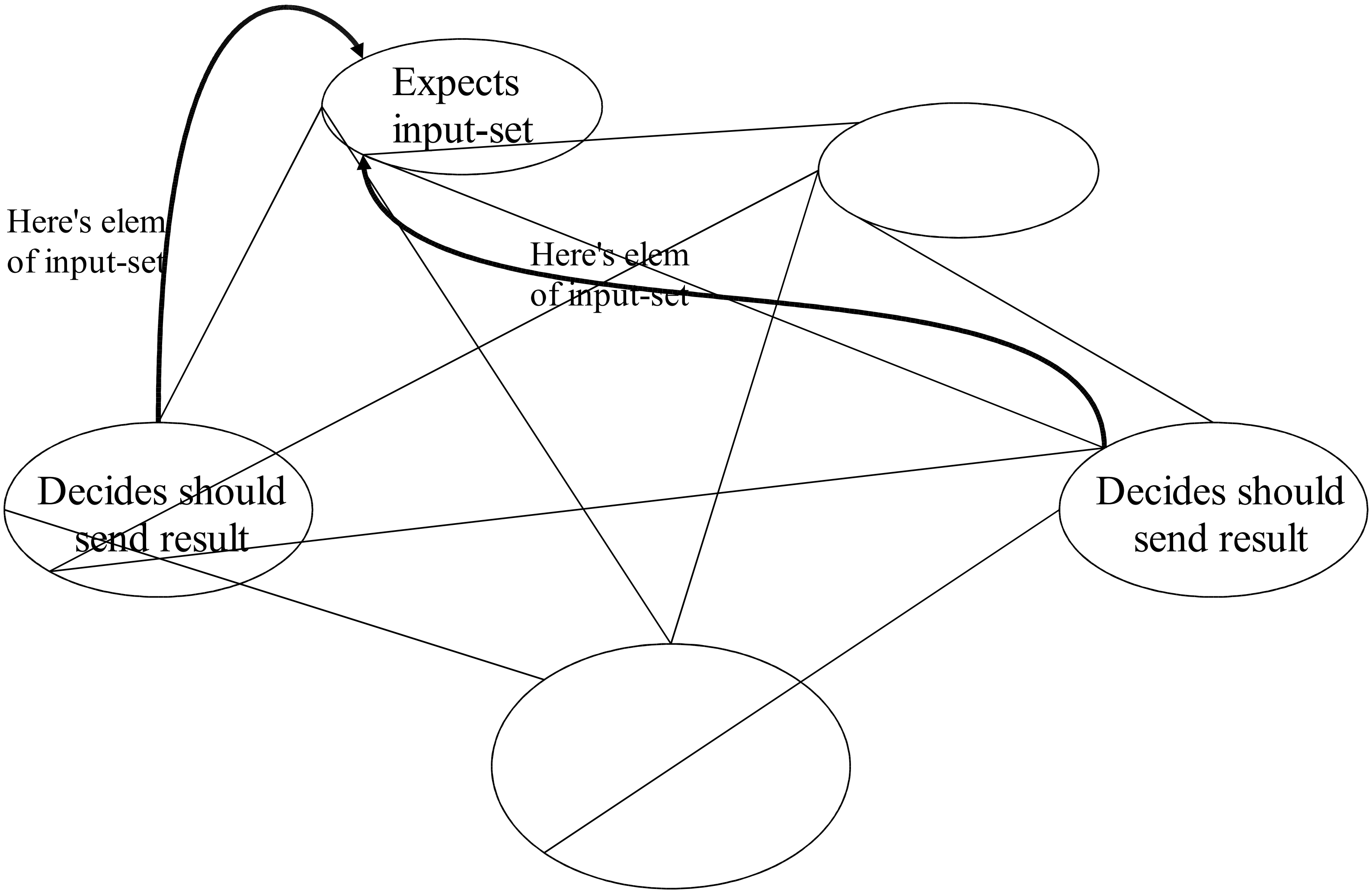
*includes which code-unit and the thread-state*

# Sending Out Completion Messages

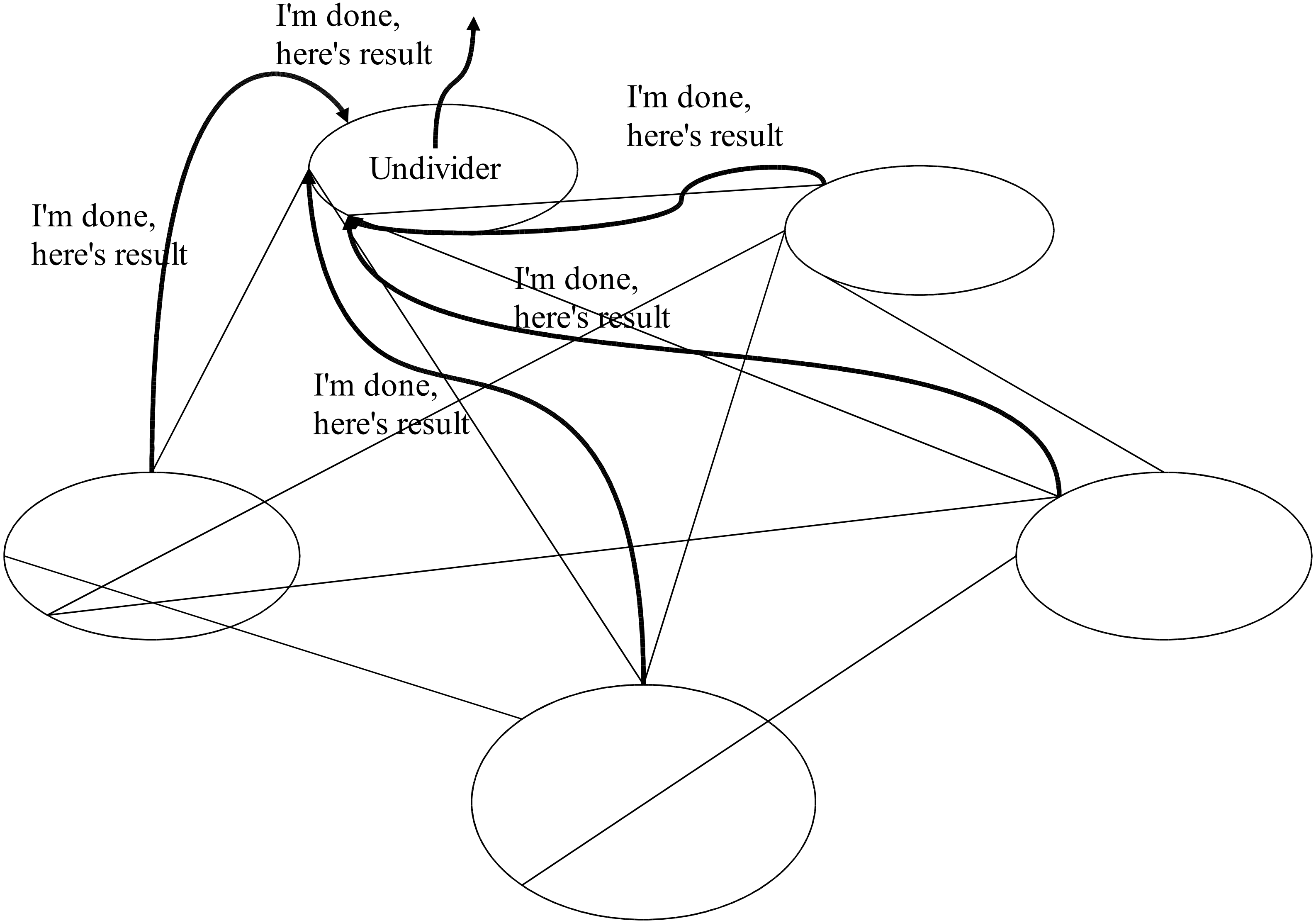


# Sending an Input-Set

All peers in group do same calc of who should get the input-set and who should send results



# Sending Results to the Undivider

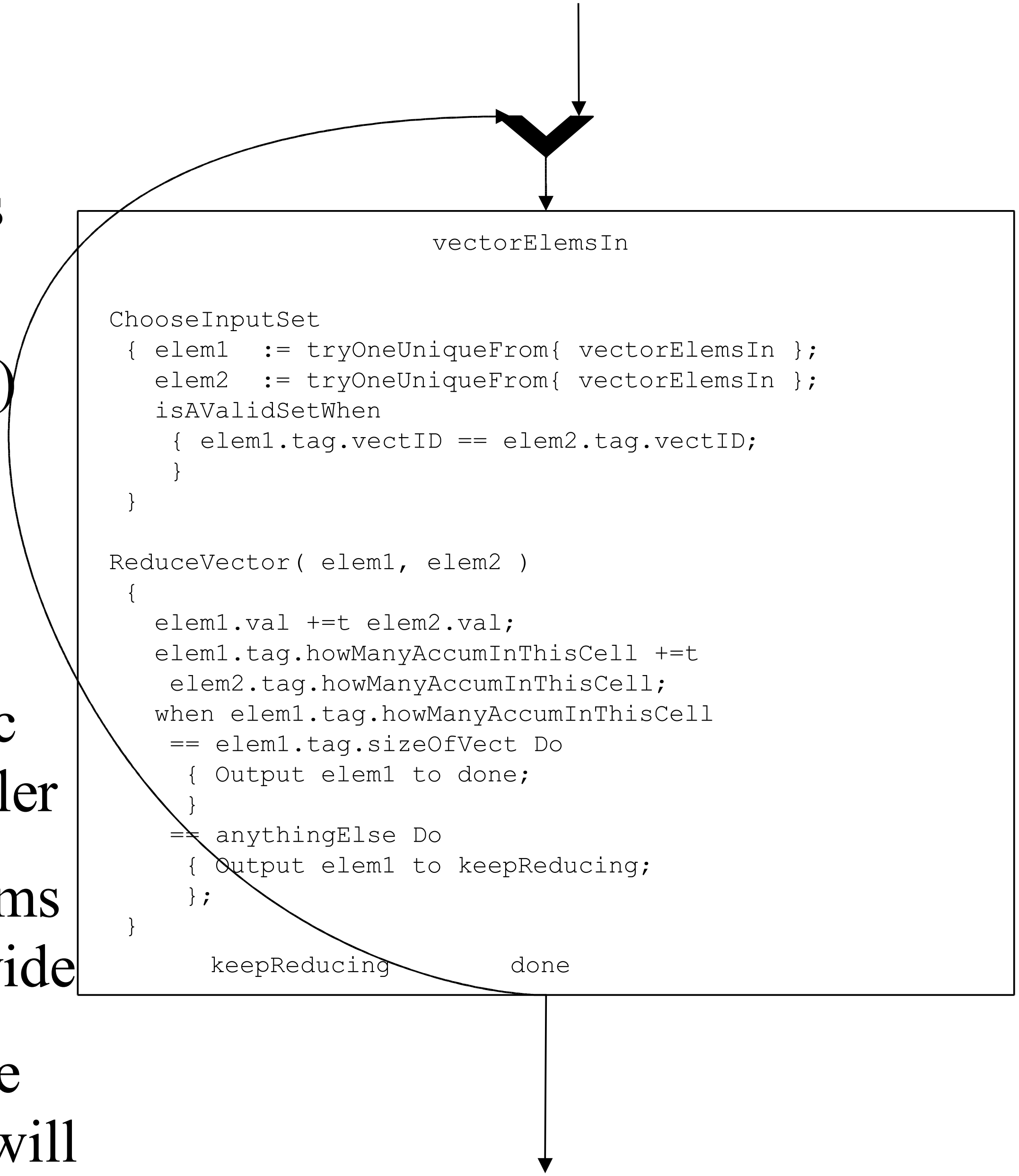


# Run-Time Behavior of Code

- Time is not defined
- Any number of f() invocations
- ChooseInputSet = contract with scheduler (no order said!)

*Therefore*

- Code Invariant to number of processors
- Combine Code-Units via static scheduling in back-end compiler
- Works with variable-sized elems = works in Divide-body-undivide
  - means run-time can change size of incoming elems at will



```
ChooseInputSet
 { elem1  := tryOneUniqueFrom{ vectorElemsIn };
   elem2  := tryOneUniqueFrom{ vectorElemsIn };
   isAValidSetWhen
    { elem1.tag.vectID == elem2.tag.vectID;
    }
 }

ReduceVector( elem1, elem2 )
 {
   elem1.val +=t elem2.val;
   elem1.tag.howManyAccumInThisCell +=t
    elem2.tag.howManyAccumInThisCell;
   when elem1.tag.howManyAccumInThisCell
    == elem1.tag.sizeOfVect Do
     { Output elem1 to done;
     }
    == anythingElse Do
     { Output elem1 to keepReducing;
     };
 }
```

# *Scheduling and Load Balancing*

- Code invariant to scheduling and load balancing algorithms
  - Coord-constraints state:
    - min scheduler must do
    - max scheduler must do
    - scheduler & load balancer free to choose order (as long as constraints satisfied)
- Easy to *automatically* change granularity via back-end comp. & sched.
  - Code invariant to number of processors
  - Data contains entire thread-state:: no shared code-state or data-state
  - Size of data (thread) chosen via divide-body-undivide pattern:: app-provided
  - Size of code-unit chosen via static scheduling in the back-end compiler
  - App-progr. provides variable number of variable size threads, and small code-units
  - Fit the machine by:: choose thread number = size; combine small loop-free code-units into larger loop-containing composite-code-units:: adjust comp/comm.

# *Performance*

- Performance:: fit order of tasks, fit size of tasks to machine details
- C.T.:: wide choice:: task order, task size (perhaps widest, maybe proof)
- Best choice:: based on:: code characteristics, machine details:: at:: run-time
- 1st, simple implementation:: install-time is almost run-time
  - Network latency, network bandwith:: available to B.E. Compiler and scheduler
  - Processor speed:: machine details:: available at install-time and run-time
  - Profile info:: characteristics of code:: loop behavior:: avail BE compiler, scheduler
  - Source-code info:: characteristics of code:: divide-body-undivide, CCA:: avail C & S
  - Processor load:: choose best for next task:: machine details:: avail to scheduler, at r-t
- Combine code-units to increase comp/comm ratio:: install-time
- Choose data-size to tune comp/comm ratio:: run-time
- Balance comp/comm:: comp overlaps comm | percent idle (freq sched)

# *Tying it All Together*

- Hardware Indep:: easy to *automatically* change granularity
  - Possible due to:
    - contract with sched,
    - self-contained threads (no side effects),
    - no complex checking of acceptable path (time order),
    - no transforms when change # threads (num of processors),
    - no search for how to divide (app provides),
    - combining small things easy:: breaking up large hard:: app gives small pieces
- Hardware Indep:: “Reference” server:: OS indep:: persistence
- Performance from
  - Back-end compiler and run-time:: full hardware knowl. & wide granularity choice
- Easy to program:: only give contract with scheduler about data
- Benefits of platform derive from more than just the extension to dataflow