

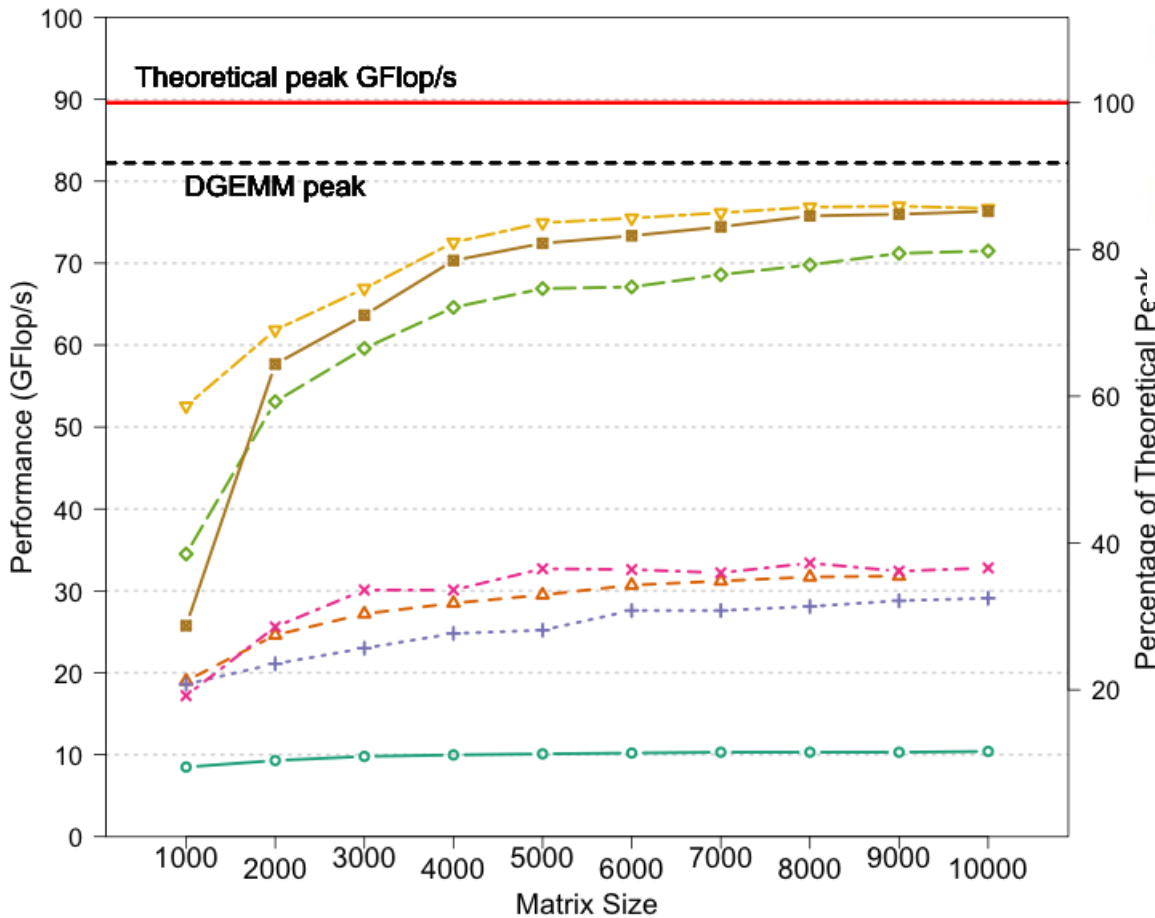
Concurrent Collections (CnC)

Kath Knobe
Frank Schlimbach
Mario Deilmann

Technology Pathfinding and Innovation (TPI)
Software and Services Group (SSG)
Intel



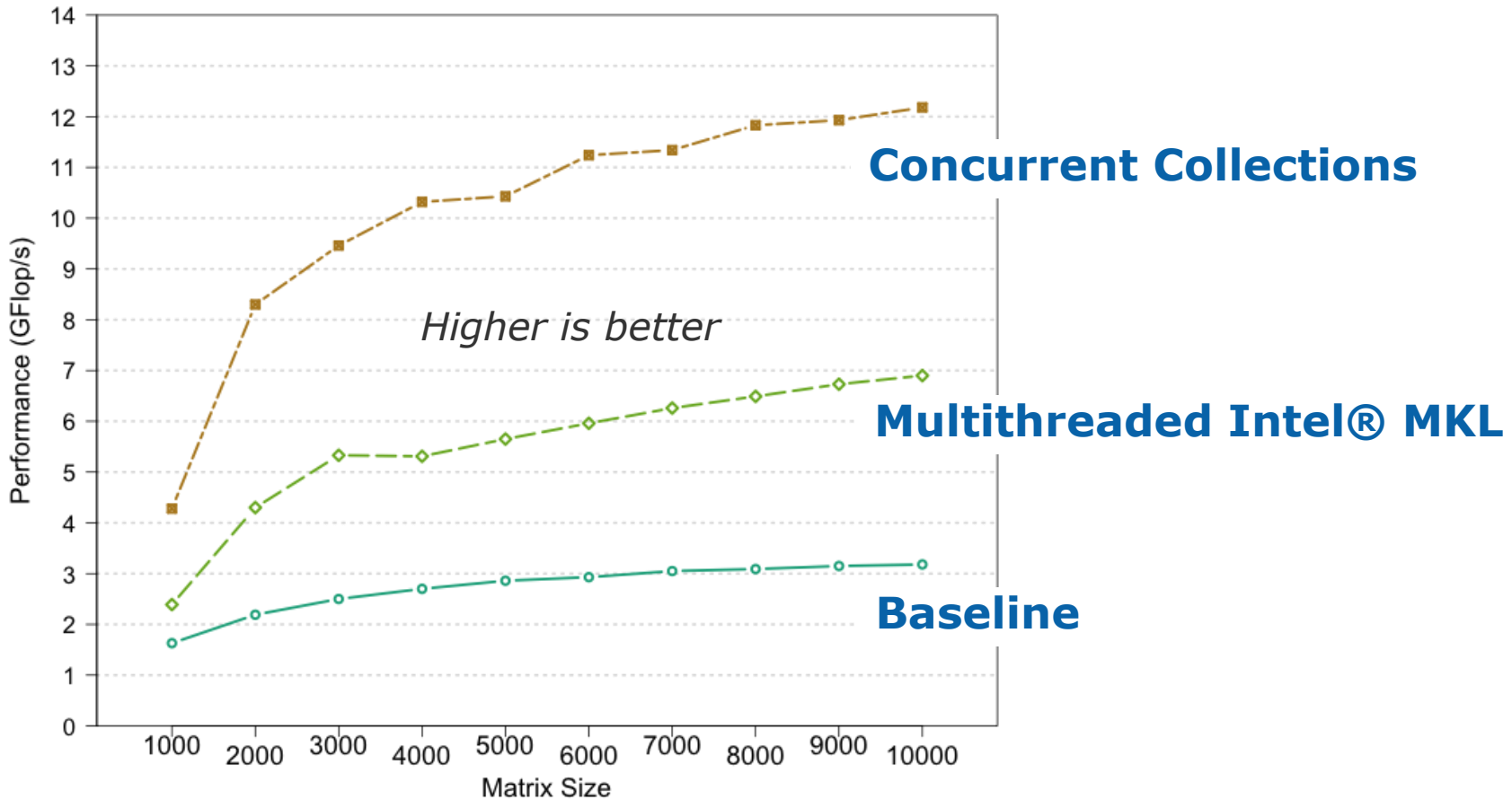
Cholesky Performance



**Intel 2-socket x 4-core Nehalem
@ 2.8 GHz + Intel MKL 10.2**

**IPDPS'10 (best paper)
Aparna Chandramowliswaran**

Eigen solver Performance



Intel 2-socket x 4-core Nehalem @ 2.8 GHz +
Intel® Math Kernel Libraries 10.2

Software and Services Group

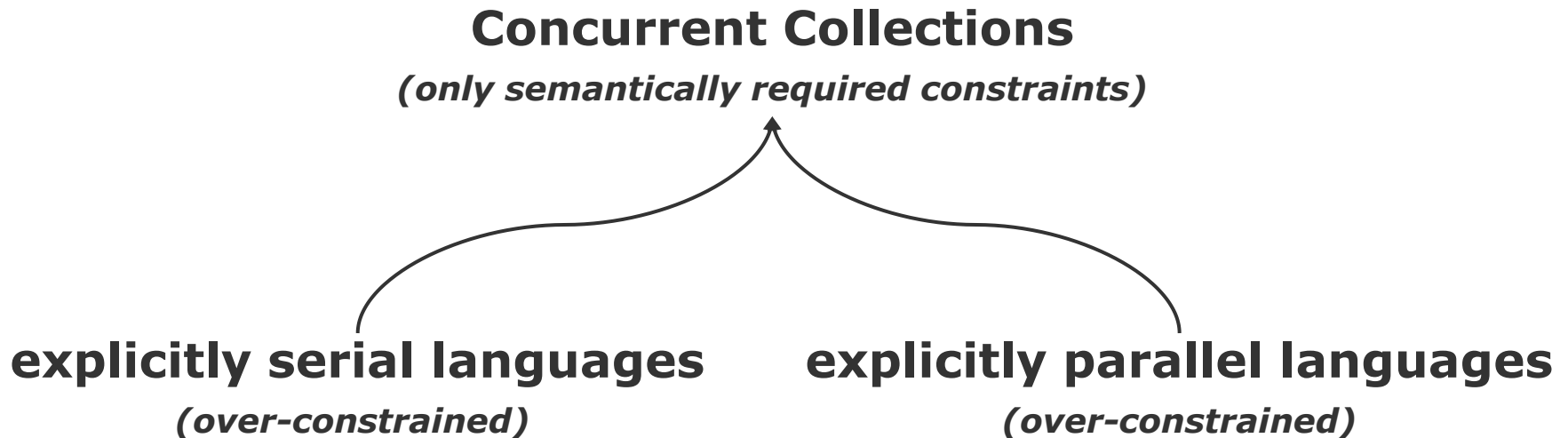


The Problem

- Most serial languages over-constrain orderings
 - Require arbitrary serialization
 - Allow for overwriting of data
 - The decision of *if* and *when* to execute are bound together
- Most parallel programming languages are embedded within serial languages
 - Inherit problems of serial languages
 - They are too specific wrt type of parallelism in the application and wrt the type target architecture
 - For the tuning expert, they don't provide quite enough control

The Solution

**Raise the level of the programming model
just enough to avoid over-constraints**



Agenda

- **Introduction**
- **The Big Idea**
- **Simple C++ Example**
- **Execution**

So What's the Big Idea?

- Don't specify what operations run in parallel
 - *Difficult and depends on target*
- Specify the semantic ordering constraints only
 - *Easier and depends only on application*

Semantic ordering constraints:

The meaning of the program require some computations to execute before others.

Exactly Two Sources of Semantic Ordering Requirements

- **Producer / Consumer (Data Dependence)**
Producer must execute before consumer
- **Controller / Controllee (Control Dependence)**
Controller must execute before controllee

Separation of Concerns Between Domain Expert and Tuning Expert

Goal: **separation of concerns**

The work of the **domain expert**

- Semantic correctness
- Constraints required by the application

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

The work of the **tuning expert**

- Architecture
- Actual parallelism
- Locality
- Overhead
- Load balancing
- Distribution among processors
- Scheduling within a processor

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

- CnC maximizes flexibility for good performance

Separation of Concerns Between Domain Expert and Tuning Expert

Goal: **separation of concerns**

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

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

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

- CnC maximizes flexibility for good performance

Separation of Concerns Between Domain Expert and Tuning Expert

Goal: **separation of concerns**

The application problem

The work of the **domain expert**

- Semantic correctness
- Constraints required by the application

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

Concurrent Collections Spec

The work of the **tuning expert**

- Architecture
- Actual parallelism
- Locality
- Overhead
- Load balancing
- Distribution among processors
- Scheduling within a processor

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

- CnC maximizes flexibility for good performance

Separation of Concerns Between Domain Expert and Tuning Expert

Goal: **separation of concerns**

The application problem

The work of the **domain expert**

- Semantic correctness
- Constraints required by the application

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

Concurrent Collections Spec

The work of the **tuning expert**

- Architecture
- Actual parallelism
- Locality
- Overhead
- Load balancing
- Distribution among processors
- Scheduling within a processor

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




- CnC maximizes flexibility for good performance

Mapping to target platform

Notation

Graphical

Textual

Computation Step		(foo)
Data Item		[x]
Control Tag		<T>

Exactly Two Sources of Ordering Requirements

- **Producer / Consumer (Data Dependence)**
Producer must execute before consumer
- **Controller / Controllee (Control Dependence)**
Controller must execute before controllee

Producer - consumer



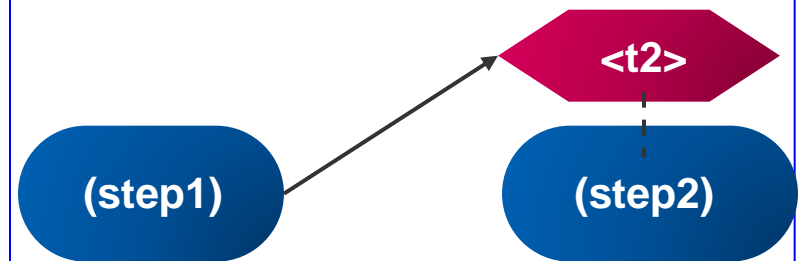
Exactly Two Sources of Ordering Requirements

- **Producer / Consumer (Data Dependence)**
Producer must execute before consumer
- **Controller / Controllee (Control Dependence)**
Controller must execute before controllee

Producer - consumer



Controller - controllee



Tag collections: new concept

(step1)

(step2)



```
loop k = ...
  loop j = ...
    loop i = ...
      Z(i, j, k) =
        = A(k, j)
    end
  end
end
end
```

- Loop iteration space is a **sequence** $\langle 1,1,1 \rangle, \langle 1,1,2 \rangle, \dots$
- Restricted to integer indices. Body accesses values.
- IF = WHEN

Tag collections: new concept

```
loop k = ...  
  loop j = ...  
    loop i = ...  
      Z(i, j, k) = ...  
      ... = A(k, j)  
    end  
  end  
end  
end
```

components of tag
<t2> are k, j and i

(Step2)

Z(i, j, k) = ...
... = A(k, j)

(step1)

(step2)

<t2>

A tag collection is a generalization of an iteration space

- **An unordered set**, not an ordered sequence
- **Not just integer indices**. Also graph nodes, tree nodes, set elements, ... Body accesses values.
- **IF ≠ WHEN**

Tag collections: new concept

```
loop k = ...
```

```
  loop j = ...
```

```
    loop i = ...
```

```
      Z(i, j, k) = Z(i-1, j, k) + ...  
      ... = A(k, j)
```

```
    end
```

```
  end
```

```
end
```

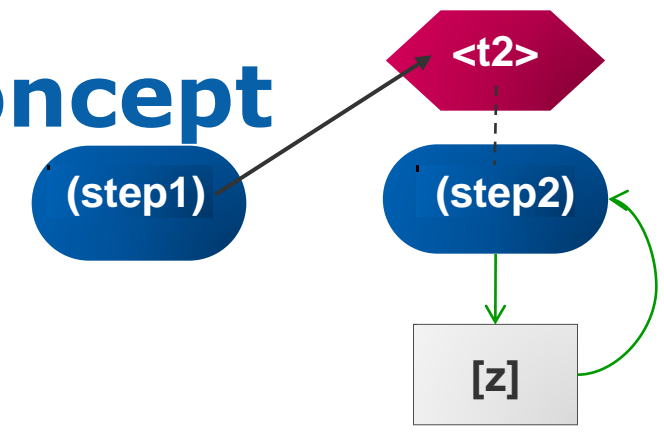
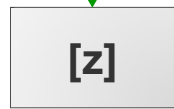
components of tag
<t2> are k, j and i

(Step2)

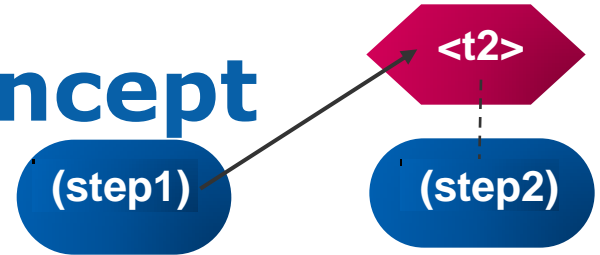
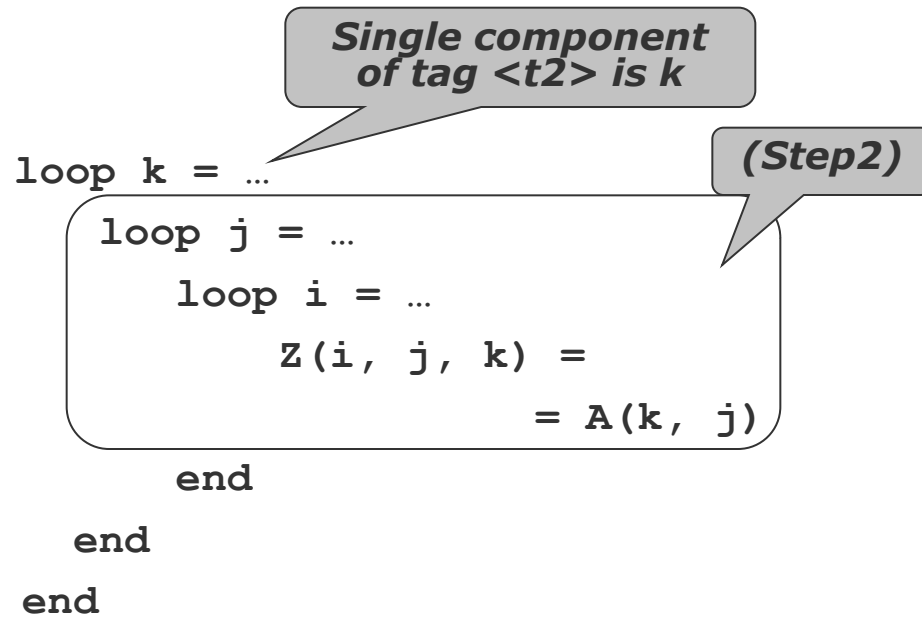
(step1)

(step2)

[z]



Tag collections: new concept



Agenda

- **Introduction**
- **The Big Idea**
- **Simple C++ Example**
- **Execution**

Cholesky factorization

Cholesky factorization

Cholesky			

Cholesky factorization

Cholesky			
Trisolve			

Cholesky factorization

Cholesky			
Trisolve	Update		

Cholesky factorization

	Cholesky		

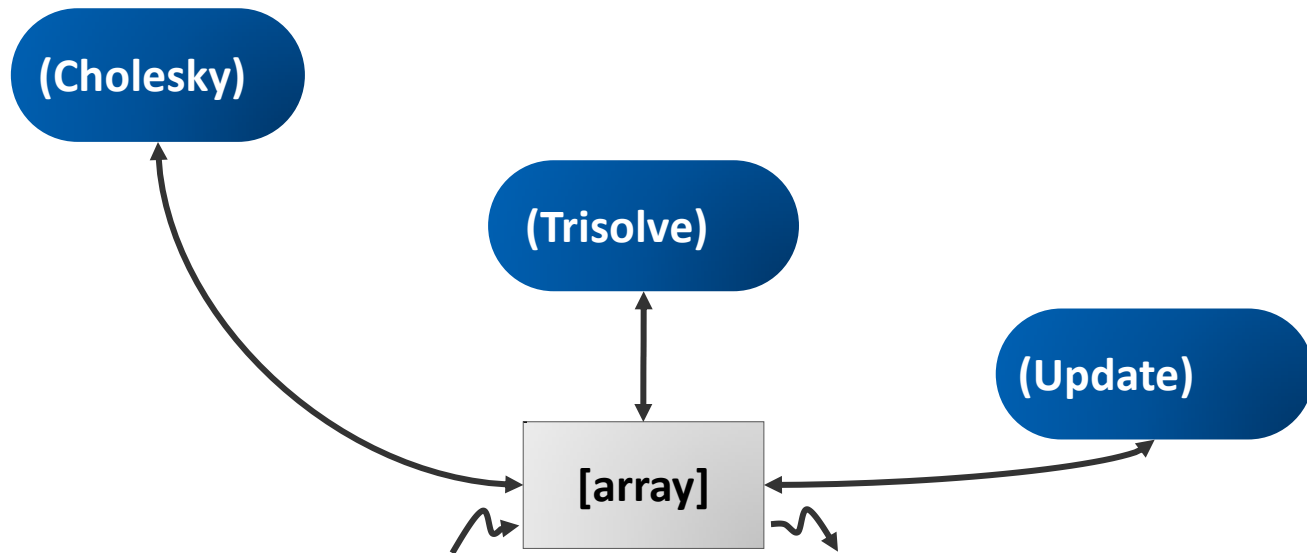
Example: The White Board Drawing

What are the high level operations?

What are the chunks of data?

What are the producer/consumer relationships?

What are the inputs and outputs?

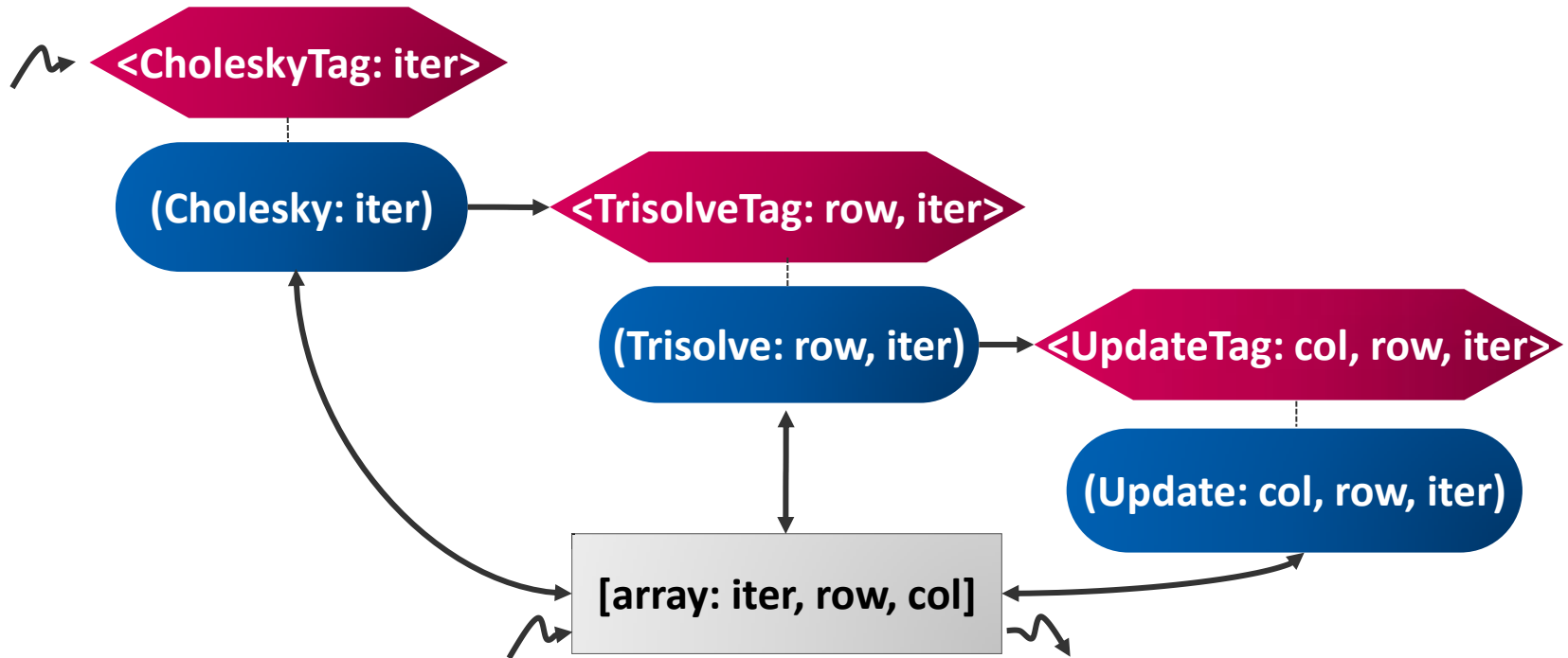


Make it precise enough to execute

Distinguish among the instances?

What are the distinct control tag collections?

What steps produce them?

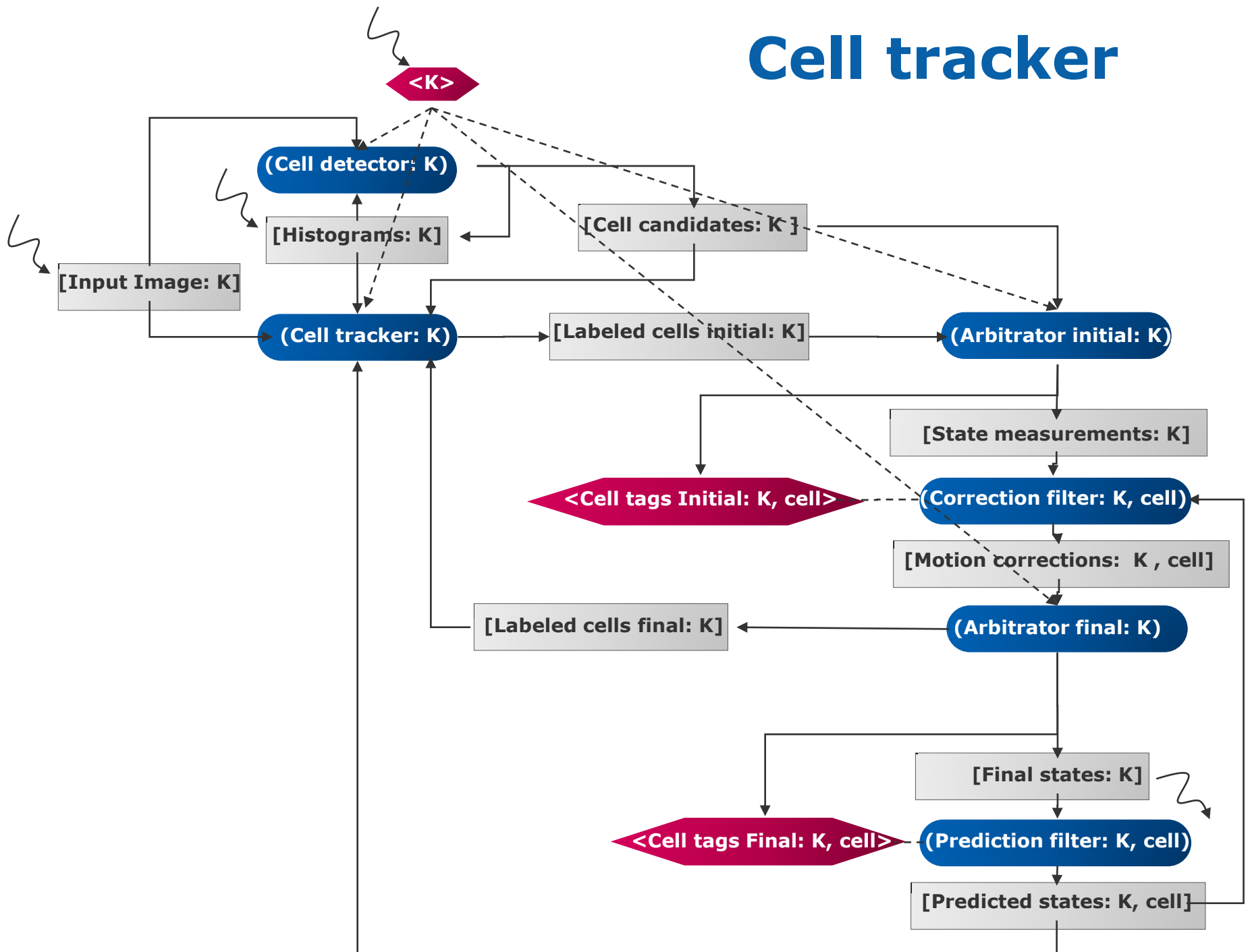


**Essentially making it ready for parallel execution
But without explicit thinking about parallelism
Focus on domain/application knowledge**

Result is:

- **Parallel**
- **Deterministic (wrt results)**
- **Race-free**

Cell tracker



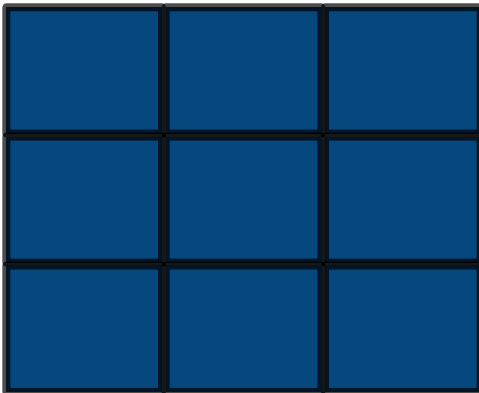
Another Application: Face Tracker

- Look for face in all possible windows of an image
Example: in a 3 x 3 image there are 14 windows
- Sequence of classifiers. Any can determine: not face

Nine

1 x 1

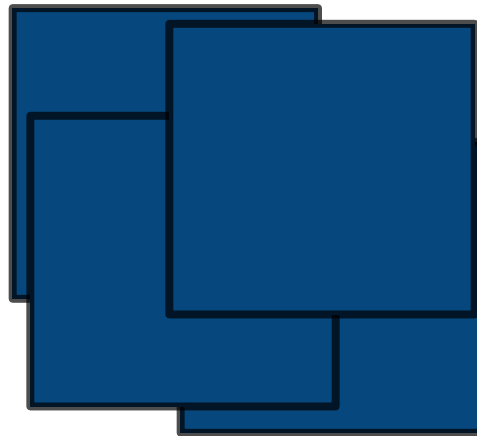
windows



Four

2 X 2

windows



One

3 x 3

window



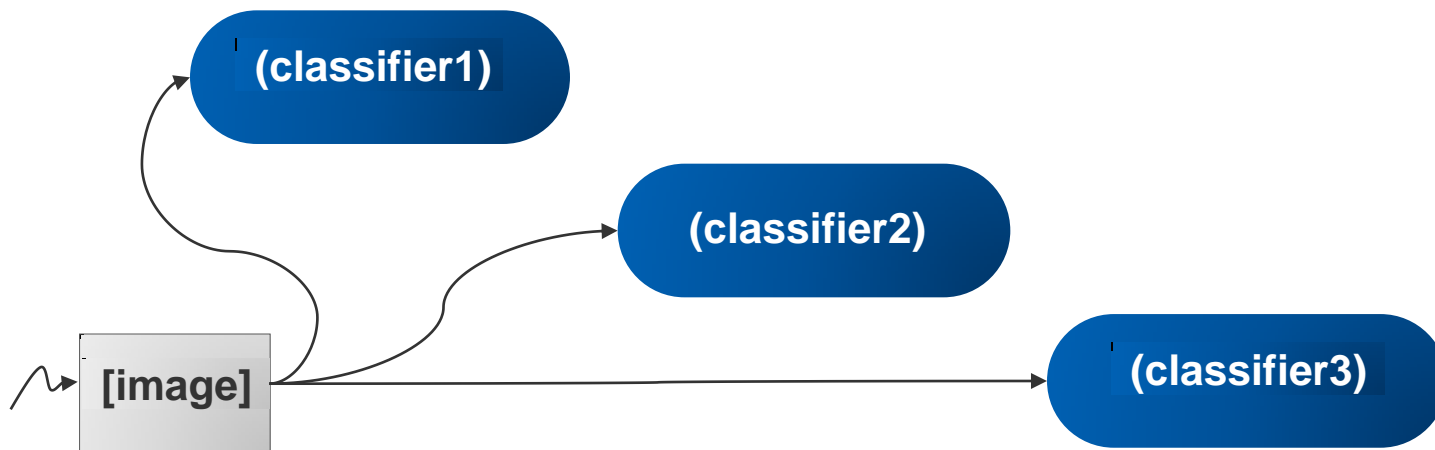
Example: The White Board Drawing

What are the high level operations?

What are the chunks of data?

What are the producer/consumer relationships?

What are the inputs and outputs?

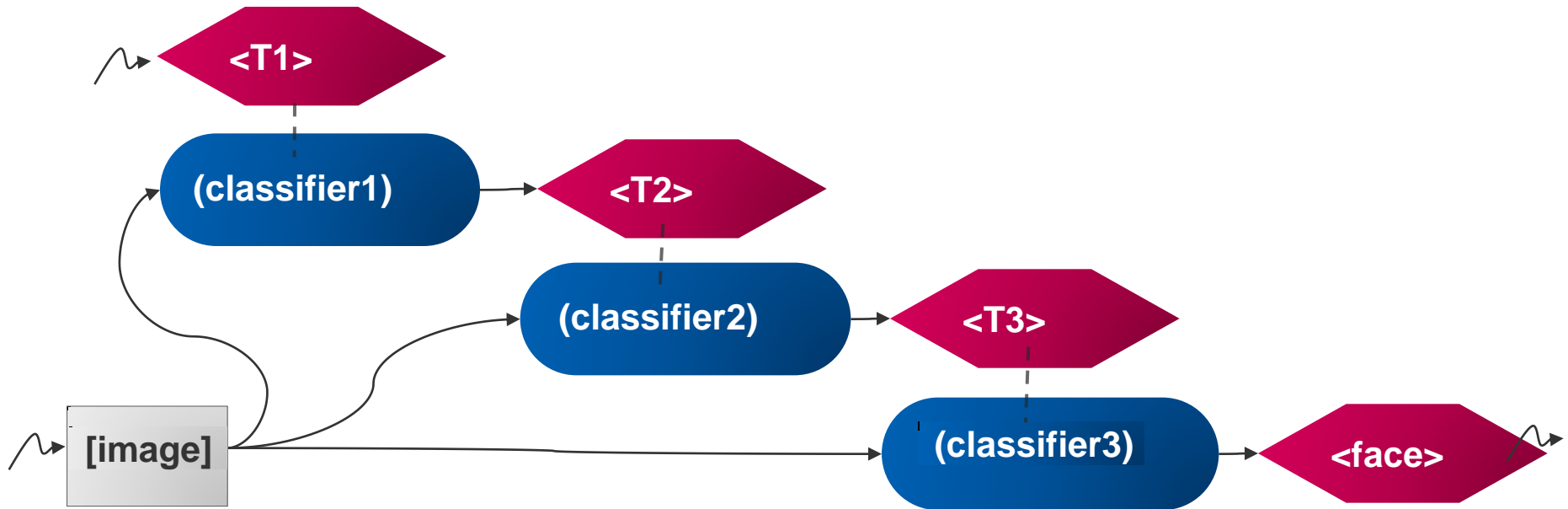


Make it precise enough to execute

Distinguish among the instances?

What are the distinct control tag collections?

What steps produce them?



Objects – summary

(foo)

Step Collections

- Are tagged. A step has access to its tag value
- Performs gets and puts
- Functional. Only side-effects are put objects

[x]

<T>

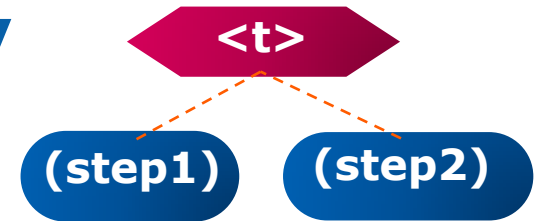
Item Collections

- Means of communication among step instances (data dependence)
- Dynamic single assignment
Each instance is associated with exactly one contents.
- Are tagged

Tag Collections

- Means of communication among step instances (control dependence)
- A tag collection may control multiple step collections
- Determines what step instances will execute

Relationships – summary



Prescription

- > Every step collection is prescribed.
- > The relationship is always the identity function.
- > A tag collection may prescribe multiple step collections.



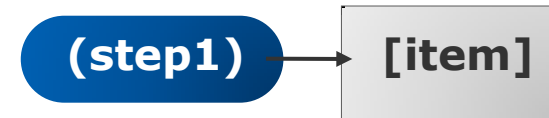
Consumer

- Corresponds to gets in steps
- A step may consume multiple distinct item collections

`[x], [y] -> (foo)`

- A step may consume multiple instances of items from a given collection

`[x: neighbors(i)] -> (foo: i)`



Producer

- Corresponds to puts in steps
- A step may produce to multiple distinct collections

`(foo) -> [x], [y]`

- A step may produce multiple instances to a given collection

`(foo: i) -> [x: neighbors(i)]`

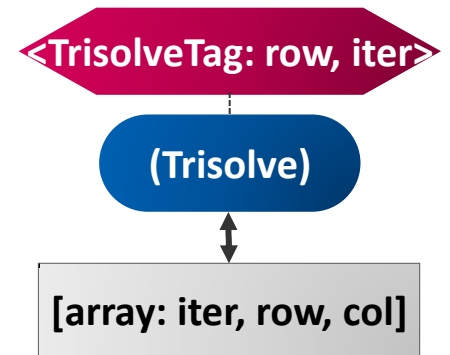
.cnc file

```
// delcarations
[BlockedMatrix<double>* array: int, int, int];

// Control relations
<TrisolveTag: row, iter> :: (Trisolve: row, iter);

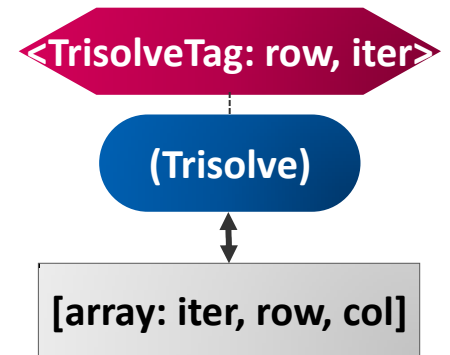
// Producer and consumer relations
[array: row, iter, iter],
[array: iter, iter, iter+1] -> (Trisolve: row, iter);

(Trisolve: row, iter) -> [array: row, iter, iter+1];
```



Coding Hints: Trisolve

```
StepReturnValue_t Trisolve(  
    cholesky_graph_t& graph,  
    const Tag_t& TS_tag) {  
    // For each input item for this step retrieve the item using the proper tag  
    // User code to create item tag here  
  
    BlockedMatrix<double>* ... = graph.array.Get(Tag_t(...));  
  
    // Step implementation logic goes here  
    ...  
    // For each output item for this step, put the new item using the proper tag  
    // User code to create item tag here  
  
    graph.array.Put(Tag_t(...), ...);  
  
    return CNC_Success;  
}
```



A model not a language: 1

Variety of ways to access the model

- > Textual representation
- > Class library
- > Graphical user interface

A model not a language: 2

There are variants within the model

- Distinct trade-offs
 - > Efficiency
 - > Ease-of-use
 - > Generality
 - > Guarantees
 - > ...
- Possible different functionality
 - > Continuous (or not)
 - > Tag functions are analyzable (or not)
 - > Real-time (or not)
 - > ...

Program execution: Attributes are monotonically acquired

Item instance



Program execution: Attributes are monotonically acquired

Item instance

available



Program execution: Attributes are monotonically acquired

Item instance

available



Tag instance



Program execution: Attributes are monotonically acquired

Item instance

available



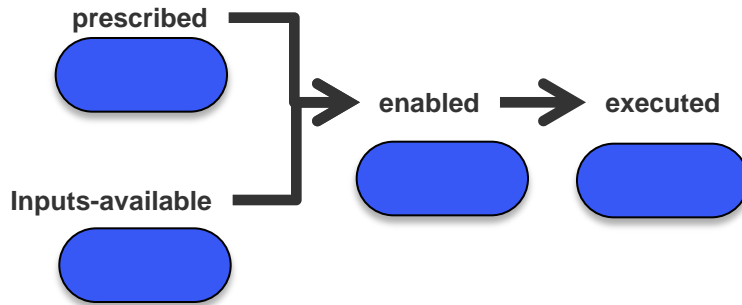
Tag instance

available



Program execution: Attributes are monotonically acquired

Step instance



Item instance

available



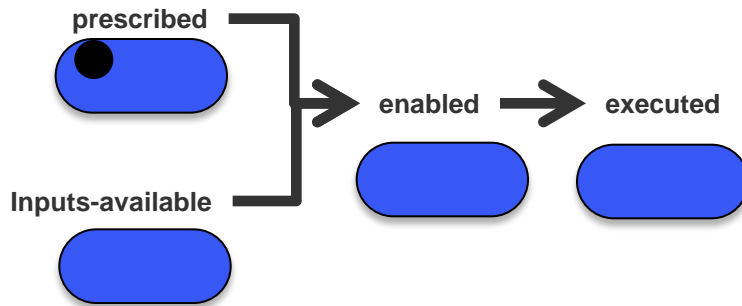
Tag instance

available



Program execution: Attributes are monotonically acquired

Step instance



Item instance

available



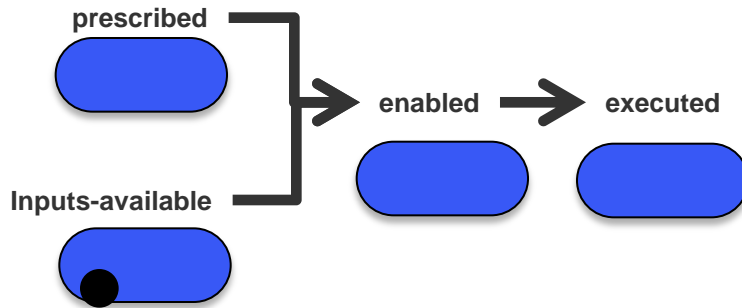
Tag instance

available



Program execution: Attributes are monotonically acquired

Step instance



Item instance

available



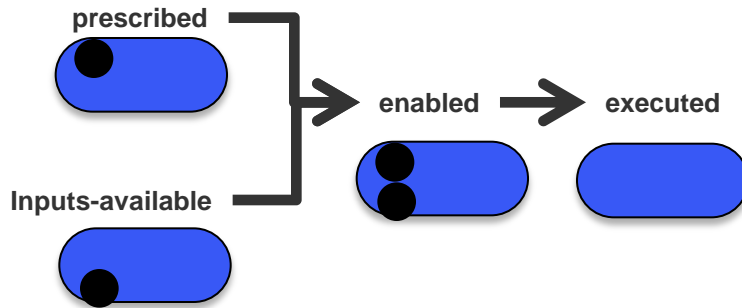
Tag instance

available



Program execution: Attributes are monotonically acquired

Step instance



Item instance

available



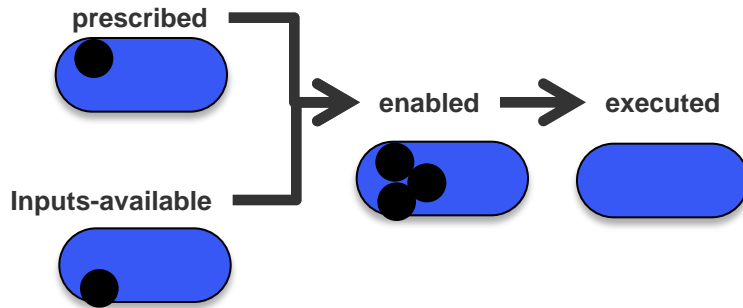
Tag instance

available



Program execution: Attributes are monotonically acquired

Step instance



Item instance

available



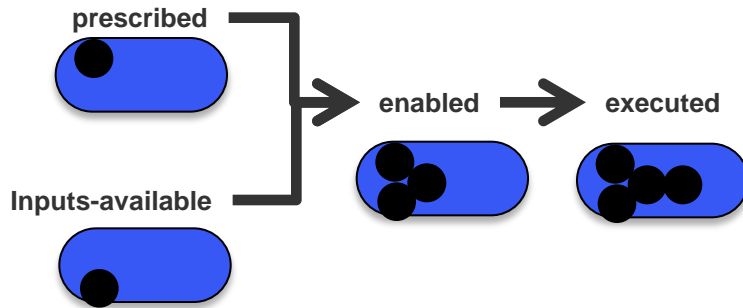
Tag instance

available



Program execution: Attributes are monotonically acquired

Step instance



Item instance

available



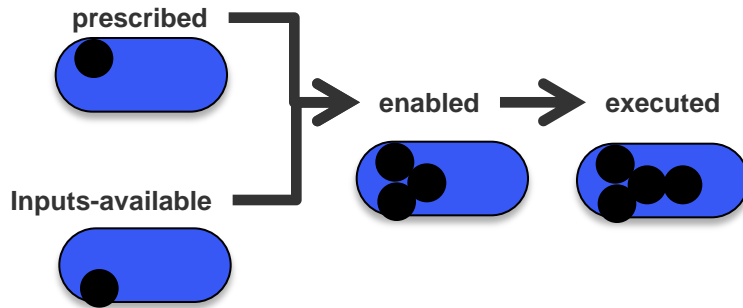
Tag instance

available



Program execution: Attributes are monotonically acquired

Step instance



Item instance

dead



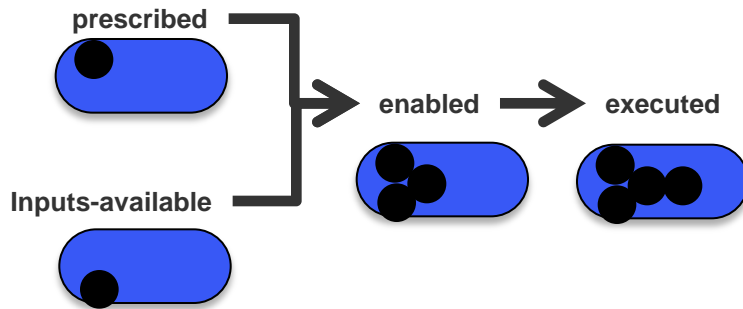
Tag instance

available



Program execution: Attributes are monotonically acquired

Step instance



Item instance

dead



Tag instance

dead



Agenda

- **Introduction and history**
- **The Big Idea**
- **Simple C++ Examples**
- **Execution**

Plays well with a variety of runtime approaches

	grain	distribution	schedule
HP TStreams	static	static	static
HP TStreams	static	static	dynamic
Intel CnC	static	dynamic	dynamic ¹
Rice CnC	static	dynamic	dynamic
Georgia Tech CnC	dynamic ²	dynamic	dynamic

² [Mandviwala et. al. LCPC '07]

¹We want to allow you to plug your own scheduler

Plays well with a variety of tuning experts

- The domain expert / later time
- Different person / different expertise
- Static analysis
- Dynamic Runtime

(This is the option currently available on our website.)

- ...

Lack of unnecessary constraints maximizes flexibility for tuning

Possible goals

- Maximize parallelism
- Minimize latency
- Maximize utilization
- Improve predictability
- Minimize memory footprint
- Minimize power consumption

Possible parallelism

- Data / loop parallelism
- Task parallelism
- Pipeline parallelism
- Tree based parallelism

Rescheduling serial executions

- Memory hierarchy opt
- Power

CnC Plays Nicely With Serial Languages

- Intel: C++ / TBB
- Rice: Java / Habanaro
- Intel: Haskell (preliminary)
- Rice: .NET (preliminary)
- ...

CnC Plays Nicely With target architectures

- Shared memory / distributed memory
- Homogeneous / heterogeneous
- Flat / hierarchical
- ...

The Intel community

- TPI

Kath Knobe

Geoff Lowney

Mark Hampton

Ryan Newton

Frank Schlimbach

- ICL

Chih-Ping Chen

Melanie Blower

Shin Lee

Steve Rose

Leo Treggiari

Ganesh Rao

Nikolay Kurtov

Jeff Arnold (soon)

Mario Deilmann

The HP community

Carl Offner

Alex Nelson

The academic community

- Rice University

Vivek Sarkar

Zoran Budimlic

Sagnak Tasirlar

David Peixotto

Mike Burke (+ IBM)

- Georgia Tech

Rich Vuduc

Aparna Chandramowlishwaran

- Novosibirsk

Nikolay Kurtov

- UCLA

Jens Palsberg

CnC'10 Workshop

**Co-located with
Languages and Compilers for
parallel Computers (LCPC)**

at Rice University in October 2010

kath.knobe@intel.com

Intel (C++):

<http://whatif.intel.com>

Rice (Java):

<http://habanero.rice.edu/cnc.html>

