
COMP 322: Principles of Parallel Programming

Lecture 24: The Concurrent Collections (CnC) Programming Model

Fall 2009

<http://www.cs.rice.edu/~vsarkar/comp322>

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu



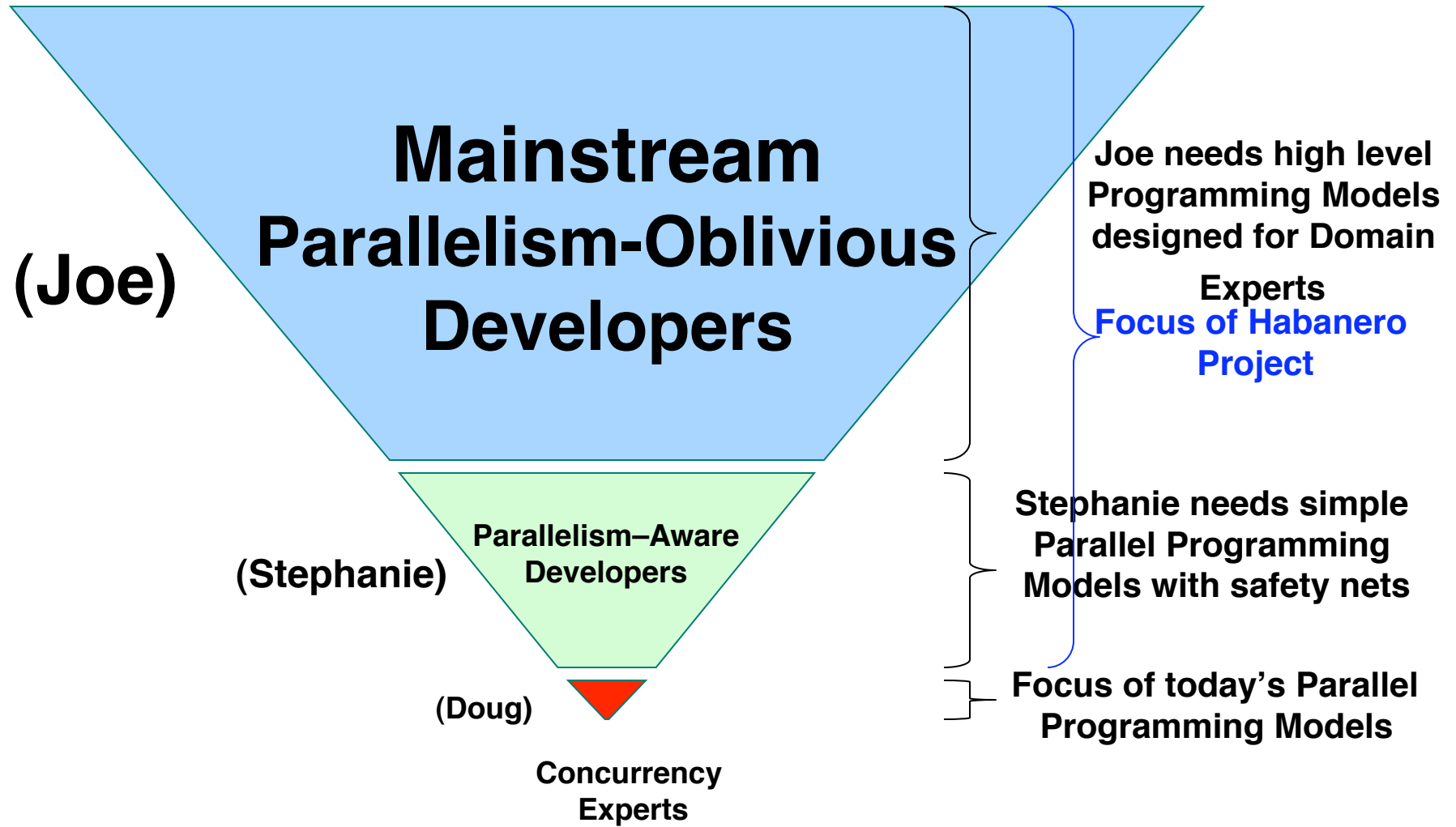
Announcements

- **Last lecture: Thursday, Dec 3rd (Course review)**
- **Programming Assignment #2 checkpoint**
 - Account on target system?
 - Copy of sequential code?
 - Successful execution of sequential code on target system as baseline?
- **Programming Assignment # 2 deadline**
 - Official deadline: 5pm, Dec 4, 2009
 - No-penalty extension: 5pm, Dec 11, 2009
- **Take-home final exam**
 - To be given on Dec 11, 2009
 - Due by 5pm, Dec 16, 2009
- **Optional project presentations**
 - Doodle poll sent for 10am - 4pm on Dec 14-16
 - Preferred time on Dec 14th?

Acknowledgments

- Slides from PLDI 2009 tutorial, “The Concurrent Collections Parallel Programming Model - Foundations and Implementation Challenges”, Kathleen Knobe and Vivek Sarkar, June 2009
 - <http://www.cs.virginia.edu/kim/publicity/pldi09tutorials/#M1>
- Intel® Concurrent Collections for C++ (& TBB)
 - <http://whatif.intel.com>
- Rice Habanero implementation of CnC model
 - <http://habanero.rice.edu/cnc>

Parallel Software Challenge & Inverted Pyramid of Parallel Programming Skills



The Big Idea in CnC

- 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

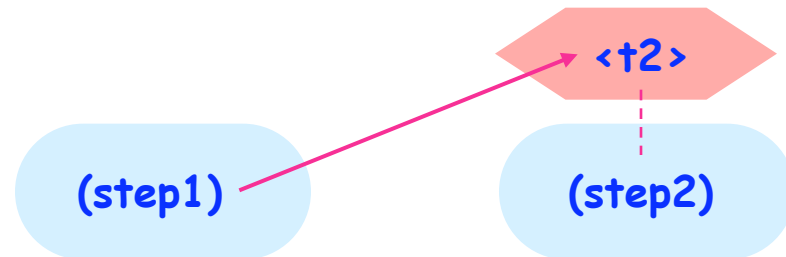
Exactly two sources of ordering requirements

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

Producer - consumer



Controller - controllee

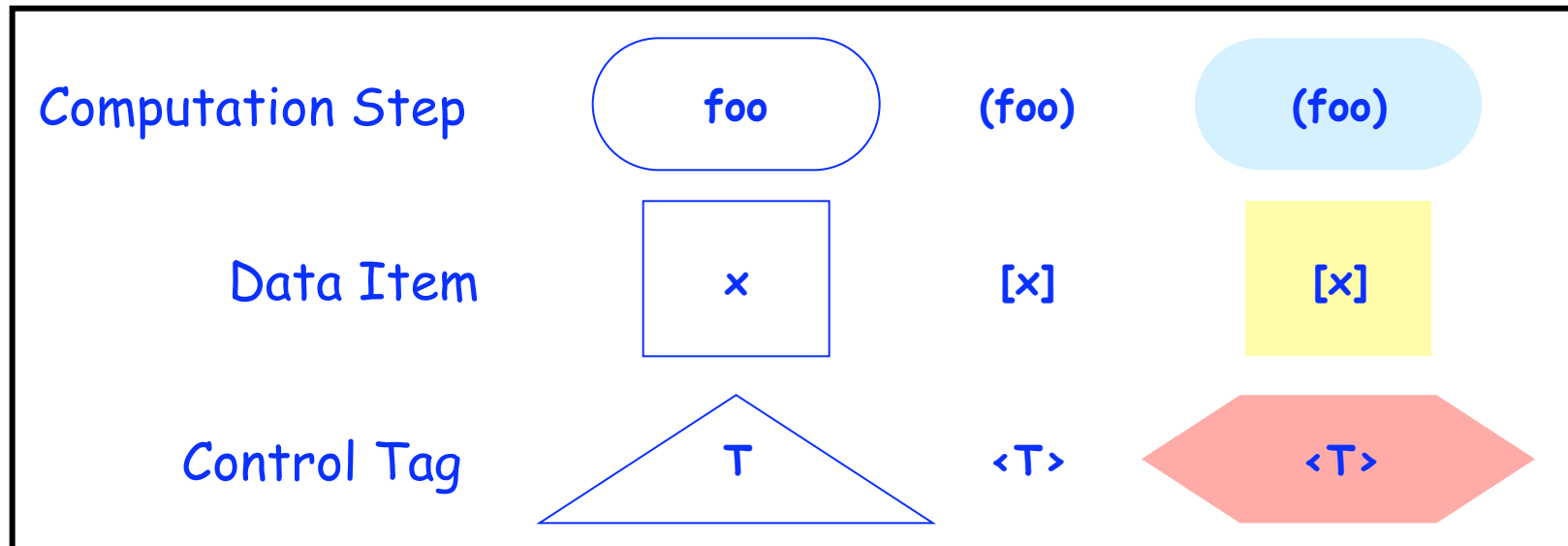


Notation

White Board

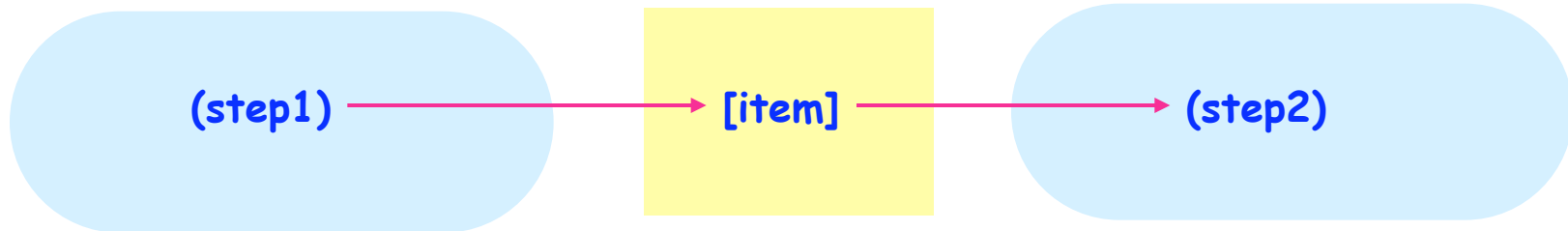
Textual

Slideware

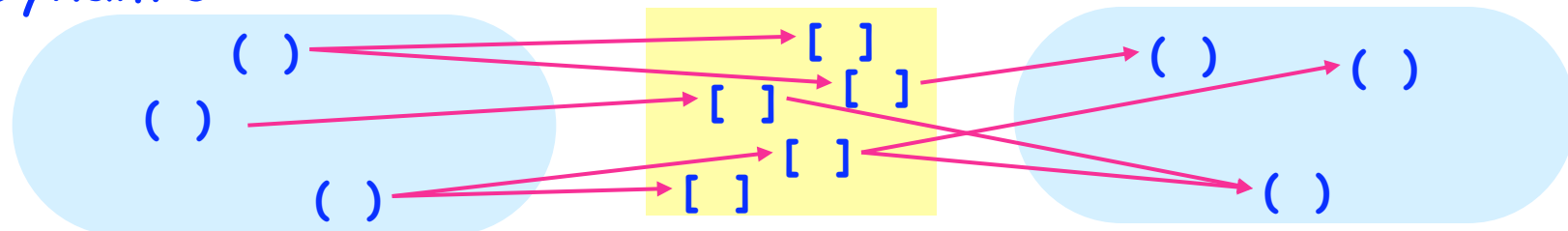


Collections of dynamic instances of Producers and Consumers

Static

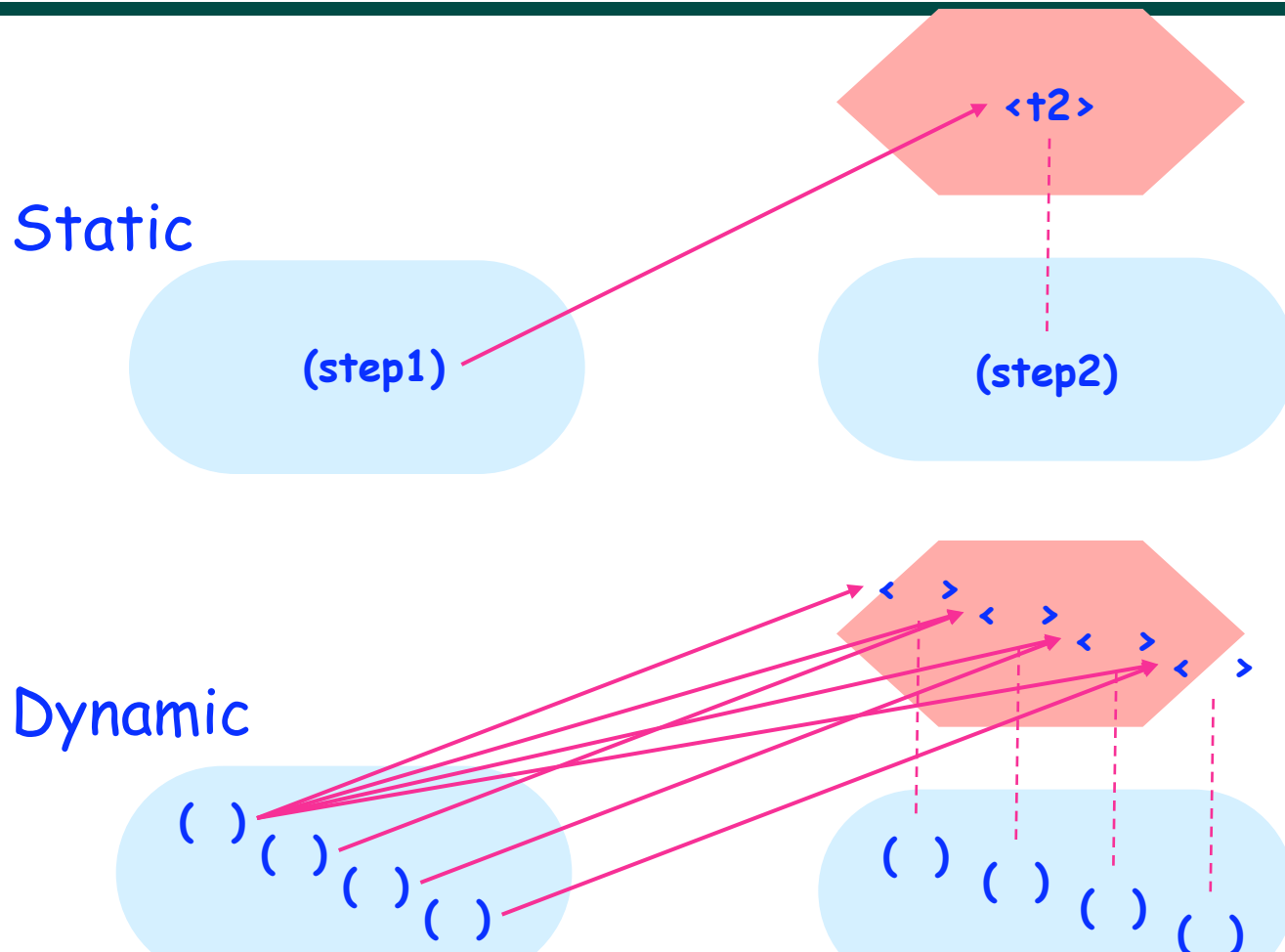


Dynamic



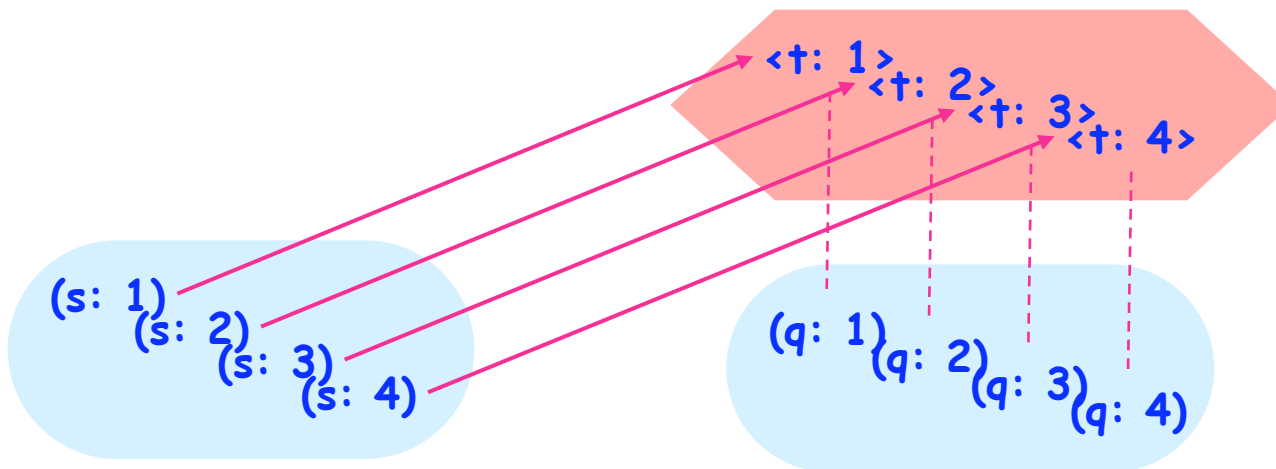
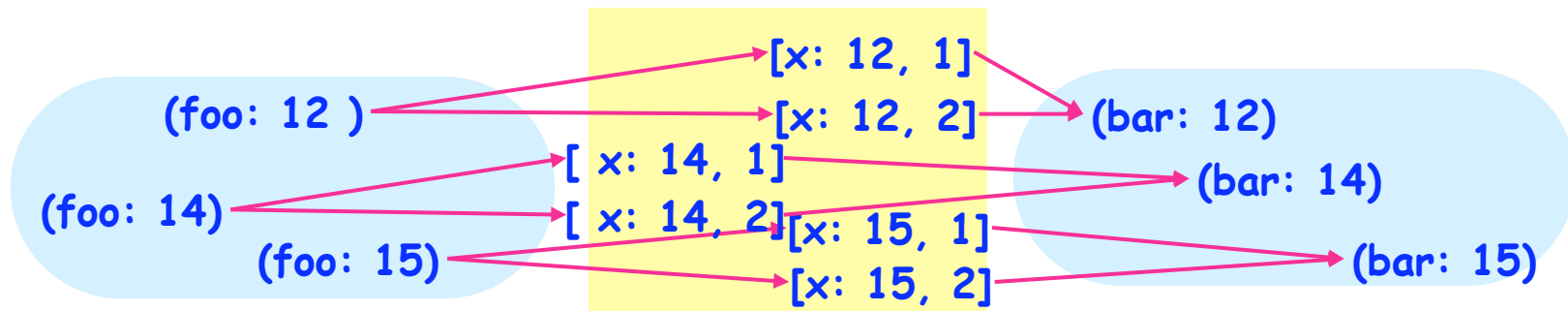
A step instance may produce multiple item instances
A step instance may consume multiple item instances
Dynamic single assignment: each item instance is produced once.

Collections of dynamic instances of Controllers and Controllees



A step instance may prescribe one or more step instances
A step instance is prescribed by exactly one step instance
Dynamic single assignment: each step instance is prescribed once

Dynamic Instances are identified by Name:Tag pairs



NOTE: Tags are passed as parameters to prescribed steps

Summary of CnC Collections

Step Collections

(foo)

- Are tagged. A step has access to its tag value
- Performs gets and puts
- Functional. Only "side-effects" are puts
- CnC programs are deterministic

Item Collections

[x]

- Means of communication among step instances
(data dependence)
- Supports puts and gets
- Dynamic single assignment

Tag Collections

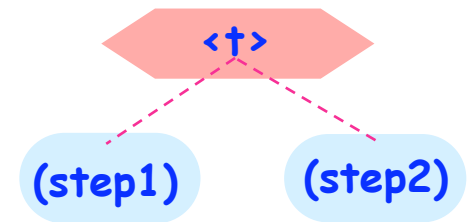
<T>

- Means of communication among step instances
(control dependence)
- Supports puts
- Determines what step instances will execute

Summary of CnC Relationships

Prescription

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



Consumer

[item] → (step2)

- 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

(step1) → <t2>

- 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)]

More on Prescription Tags

Tags are typically related to the semantics of application.

(step1)

(step2)

<t2>

Loop nests

```
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

(step1) prescribes an
instance of (step2)
with control tag
<t2> = <i,j,k>

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

- Tag collections are a generalization of iteration spaces
- Not just loops but also trees, graphs, sets ...

- Loop iteration space is a *sequence* <1,1,1>, <1,1,2>, ...
- The body of the loop has access to its indices
- As each tuple in the sequence is created, the associated body is executed.
- The related tag collection is a *set* {<1,1,1>, <1,1,2>, ...}
- The step code has access to its tag components
- The time the associated body is executed is yet to be determined.

An Application (partition_string)

Break up an input string
- sequences of repeated single characters
Filter allowing only
- sequences of odd length

Input
string

"aaaffqqqmmmmmm"

Sequences of
repeated
characters

"aaa"

"ff"

"qqq"

"mmmmmm"

Filtered
sequences

"aaa"

"qqq"

"mmmmmm"

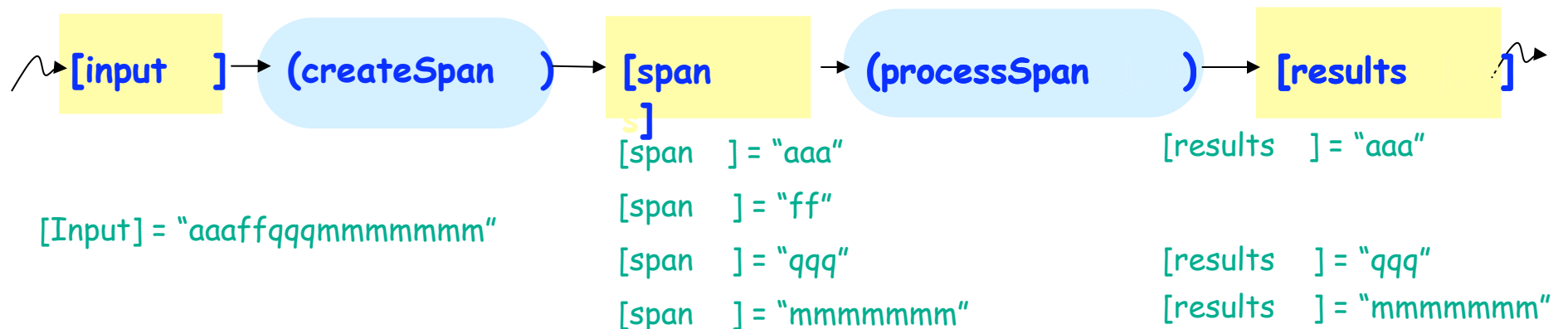
How people think about their application: The white board drawing

What are the high level operations?

What are the chunks of data?

What are the producer/consumer relationships?

What are the inputs and outputs?

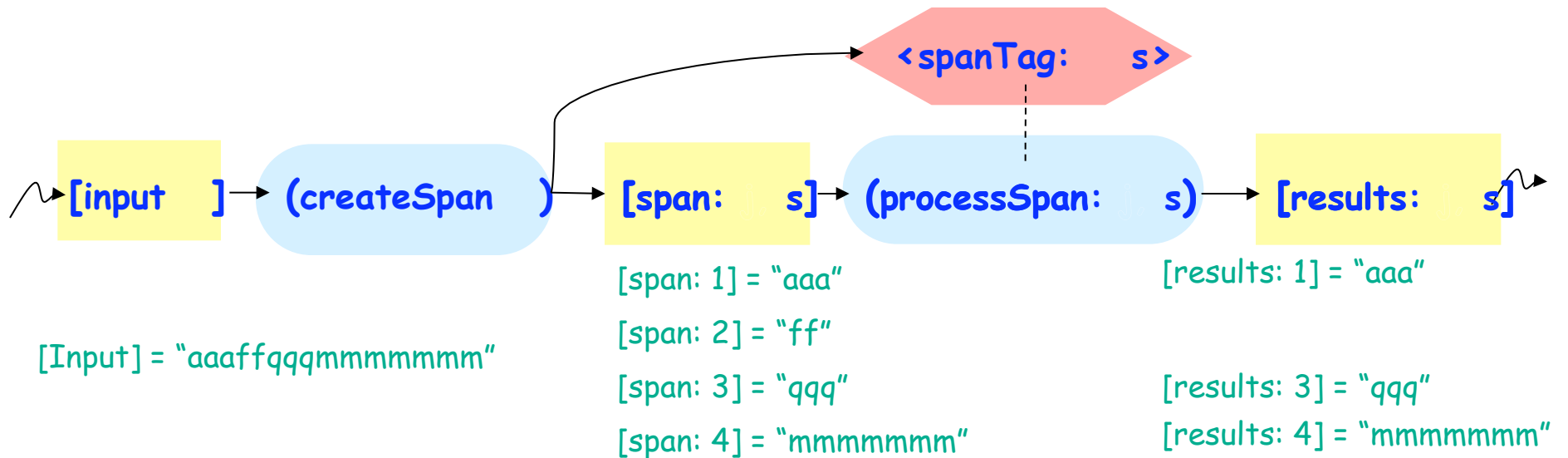


Make it precise enough to execute

How do we distinguish among the instances?

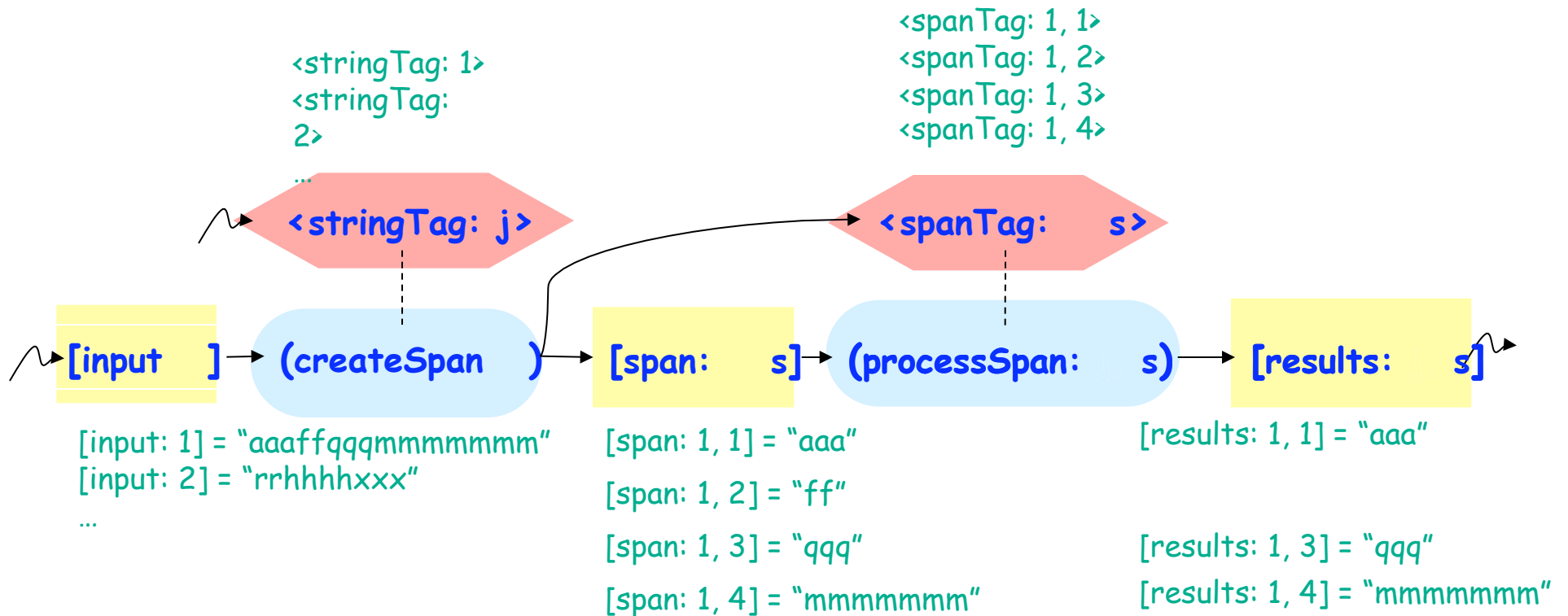
- Here the tag values are arbitrary.
- Often the tags have semantic meaning in the application.

<spanTag: 1>
<spanTag: 2>
<spanTag: 3>
<spanTag: 4>



Make it precise enough to execute

How do we distinguish among the instances?



Textual form of graph

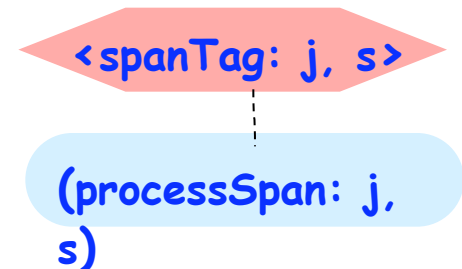
```
// Inputs from environment  
env -> [input: j];
```



```
// outputs to environment  
[results: j, s] -> env;
```



```
// controller/controllee relations  
<spanTags: j, s> :: (processSpan: j, s);
```



```
// producer/consumer relations
```

```
[span: j, s] -> (processSpan : j, s) -> [results : j, s];
```



Textual graph of partition_string

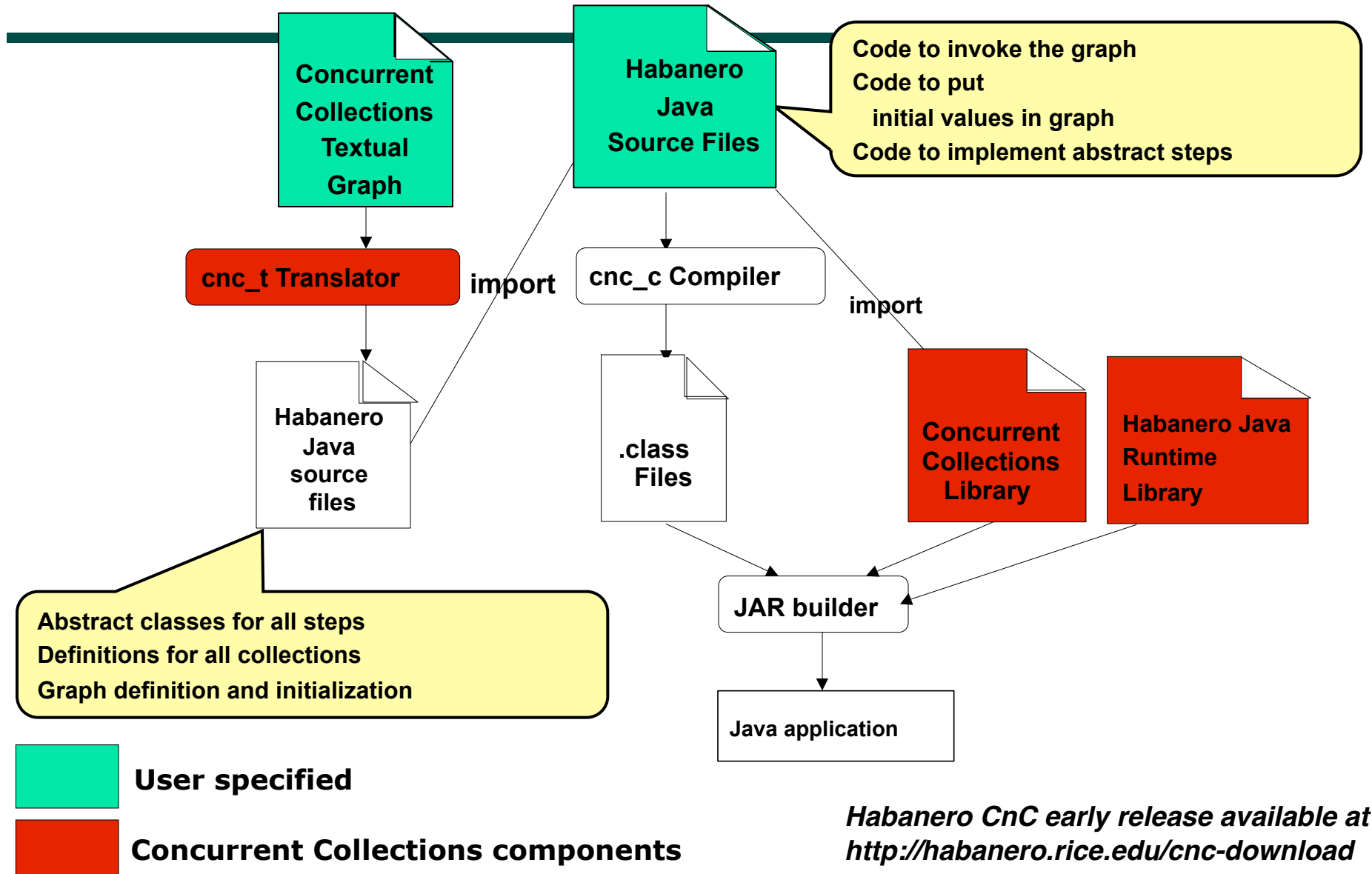
```
// declarations
[input: singleton];           // single input string
[span: int spanID];          // sub-strings with the same character
[result: int spanID];        // sub-strings of odd length
<stringTag: singleton>;      // tag which identifies the input string
<spanTag: int spanID>;       // tags which identify sub-strings

// steps and controlling tags
<stringTag > :: (createSpan);
<spanTag> :: (processSpan);

// input and output for steps
[input] -> (createSpan) -> <spanTag>, [span];
[span] -> (processSpan) -> [result];

// What comes from the environment and what goes to the environment
env -> [input], <stringTag>;
[result] -> env;
```

CnC Habanero-Java Build Model



Another Application (also very simple)

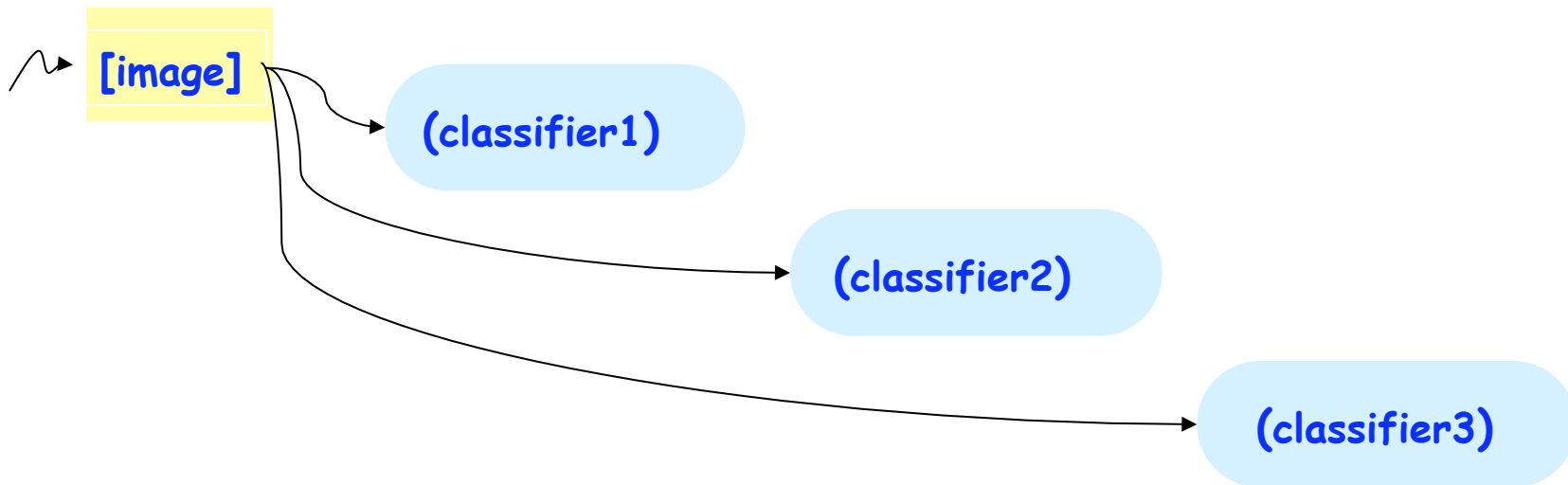
- Consider all possible square windows within an image (all sizes and possibly overlapping)
- A sequence (cascade) of classifiers where each can determine that the window does not contain a face
- A window successfully reaching the end of the cascade of classifiers, it is considered to be a face.

How people think about their application: The white board drawing

What are the high level operations?

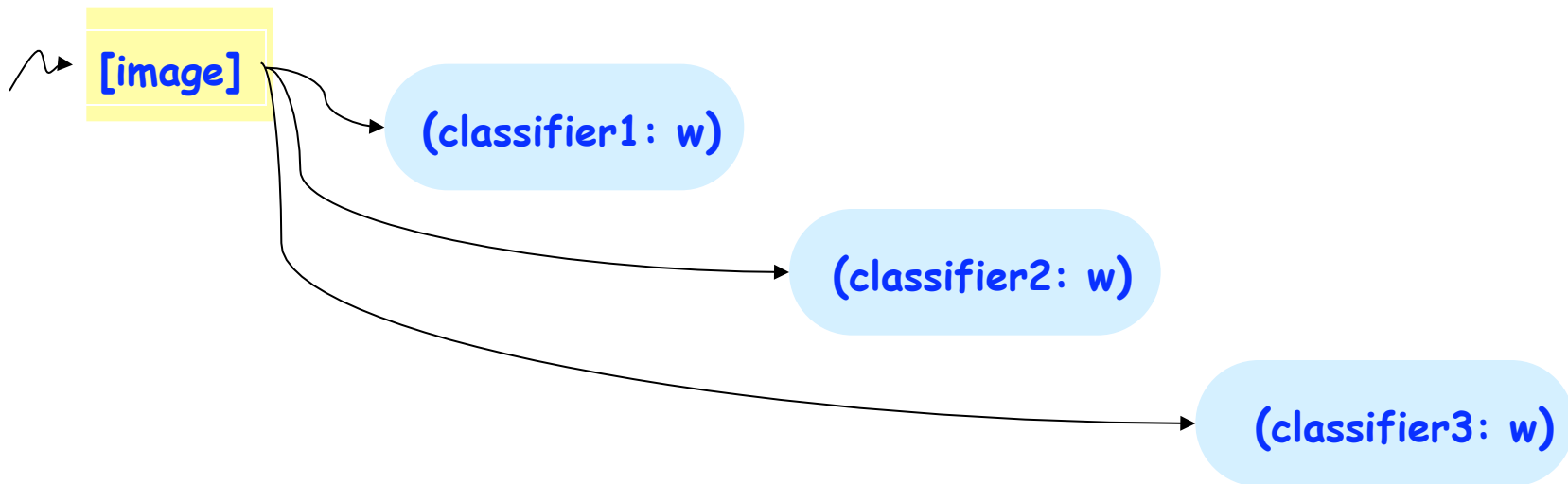
What are the chunks of data?
What are the producer/consumer
relationships?

What are the inputs and outputs?



Make it precise enough to execute

How do we distinguish among the instances?

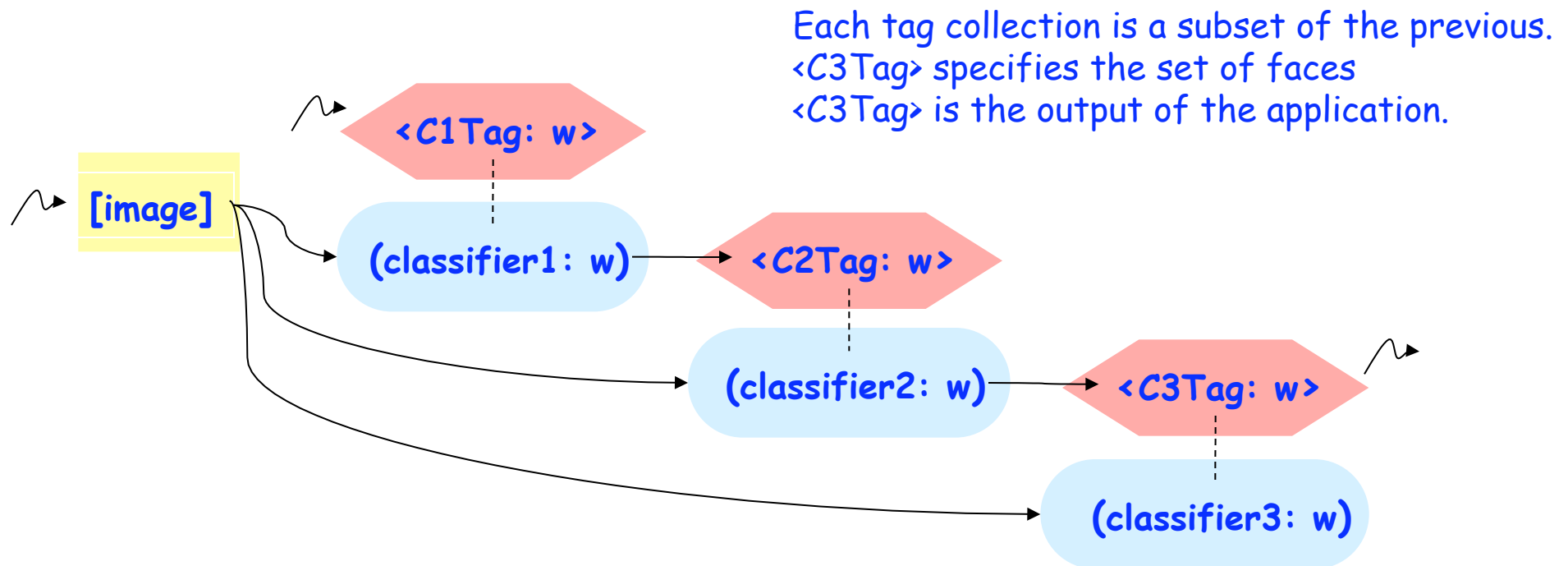


Make it precise enough to execute

Control: Every step is a controllee

What are the distinct controls?

What steps are the controllers?



CnC Properties

No thinking about parallelism
Only domain/application knowledge

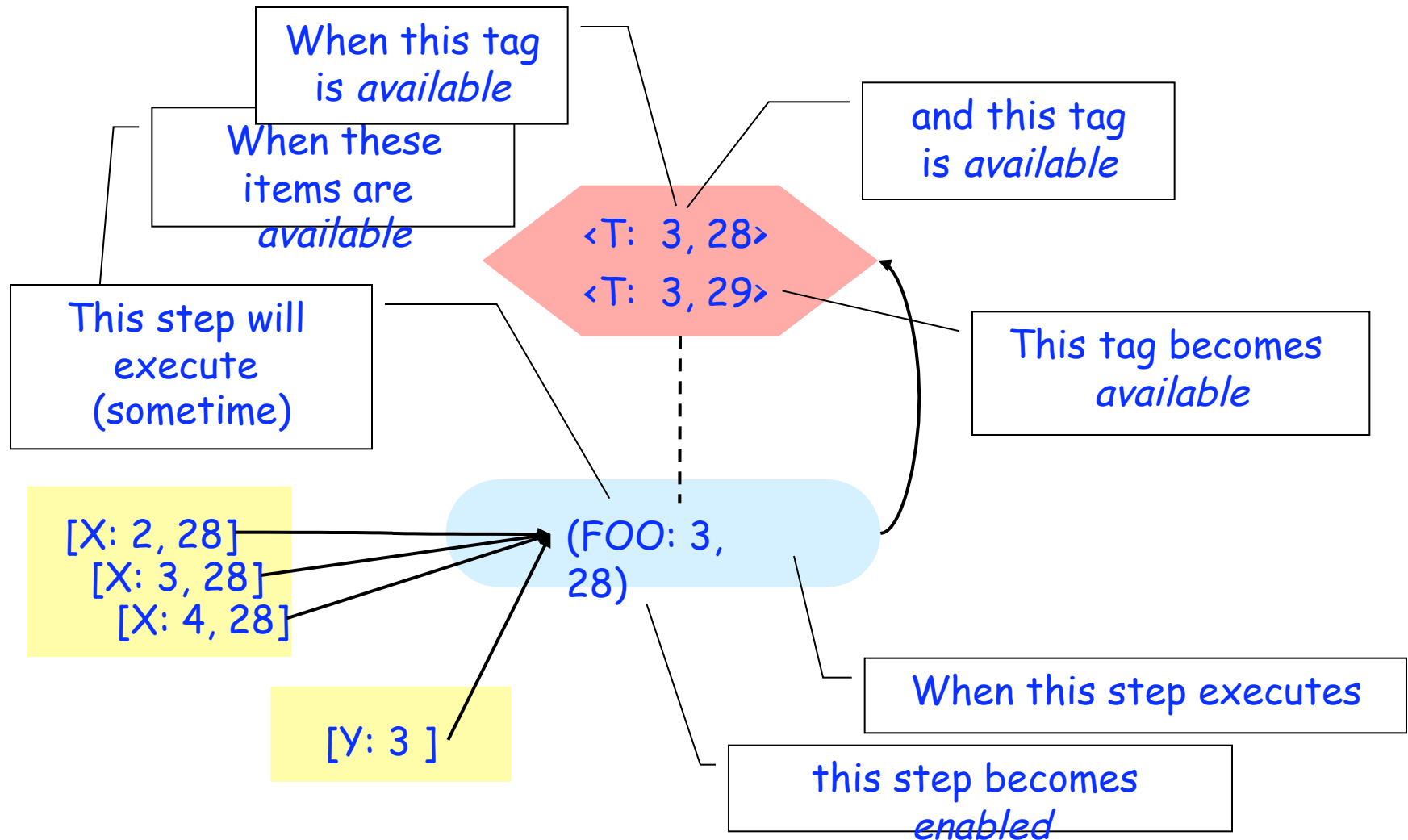
Result is:

- Parallel
- Deterministic
- Race-free

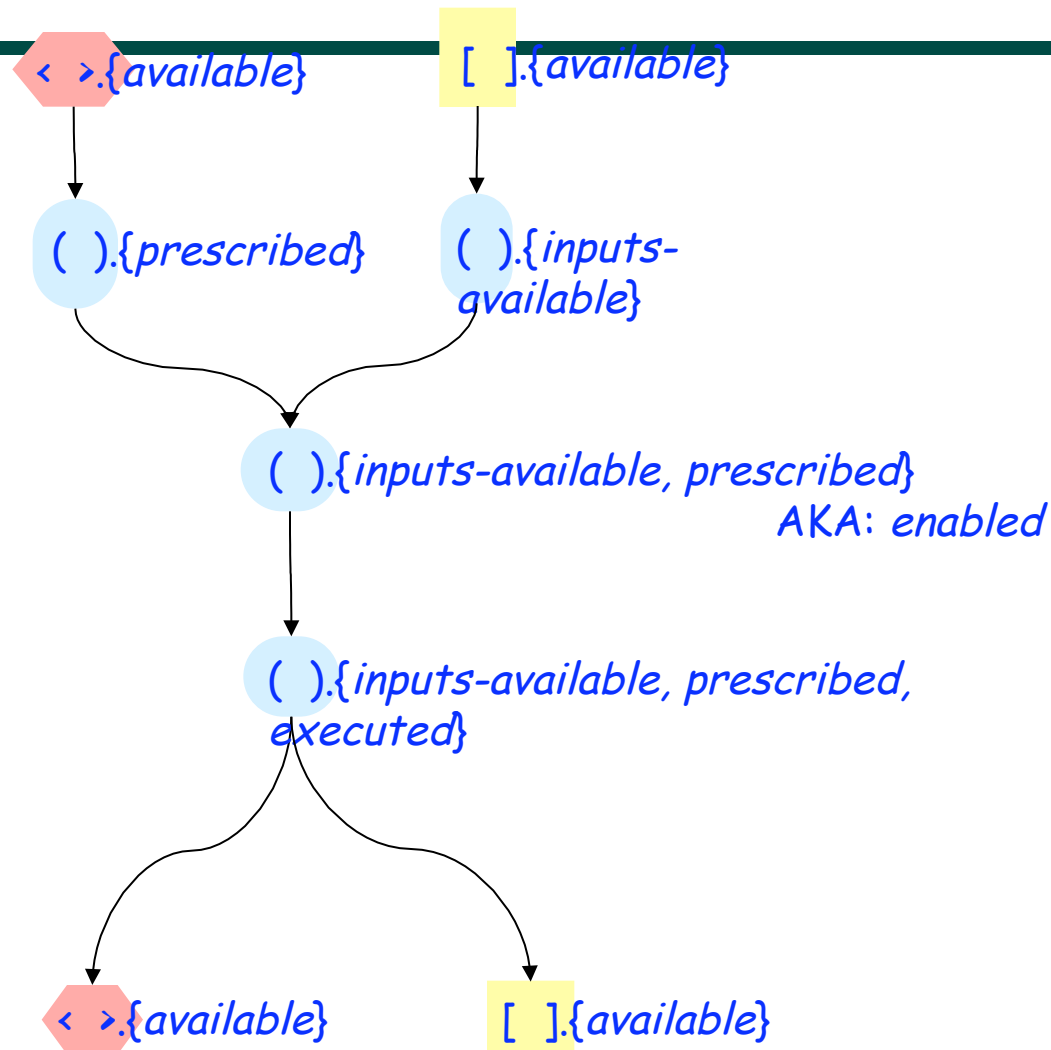
The Environment

- Environment (env) is just a normal main program
- In addition it starts and finishes the CnC program graph
- Using the same API used by the step code, it
 - Puts items and tags (env acts as a producer)
 - Gets items and tags (env acts as a consumer)
- It has access to the CnC API for puts and gets, it doesn't obey any of the constraints of CnC.
 - It isn't tagged
 - It isn't single assignment
 - It isn't functional
- Env may not know the tags of instances to get
 - Iterated get on a collection

Semantic model: instances acquire attributes monotonically



Summary of attributes as propagated

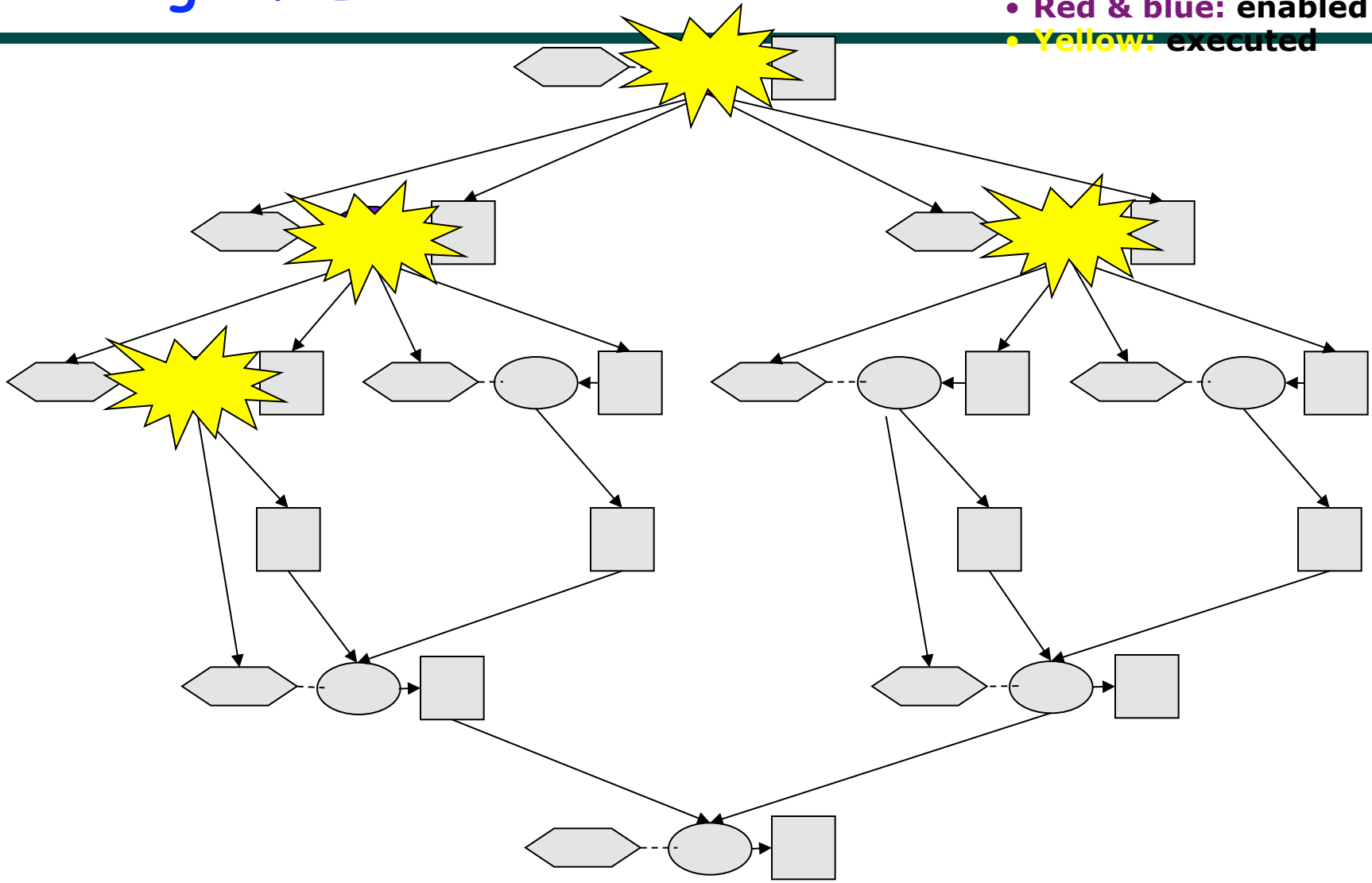


Attributes

- Attributes monotonically increase as execution proceeds
 - More instances
 - More attributes of instances
- Attributes can be used to support
 - Reclamation of storage from dead items/tags
 - Speculative execution
 - Demand-driven execution
 - Checkpoint/restart

CnC Program Execution

- **Red:** prescribed
- **Blue:** inputs available
- **Red & blue:** enabled
- **Yellow:** executed



Scheduling decision

More than enough parallelism

Possible goals

- Minimize latency
- Maximize utilization
- Improve predictability
- Minimize memory footprint
- Minimize overhead of scheduler

Other inputs to decision

- Number of processors
- Hierarchical aspects of architecture
- Typical number of substrings per string
- Cost of communication
- Rate of arrival of input strings

- These issues are isolated from the CnC spec
- CnC spec provides maximum flexibility for mapping

CnC supports not only different schedules but a wide range of runtime *styles*

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

* Soon you'll be able to plug your own scheduler into our C++ version

CnC plays well with . . .

Other languages

1. Intel® Concurrent Collections for C++ (& TBB)
 - <http://whatif.intel.com>
2. Rice Concurrent Collections for Java (& Habanero-Java)
 - <http://habanero.rice.edu/cnc>
3. Rice Concurrent Collections for .NET (preliminary)
4. Intel® Concurrent Collections for Haskell (preliminary)

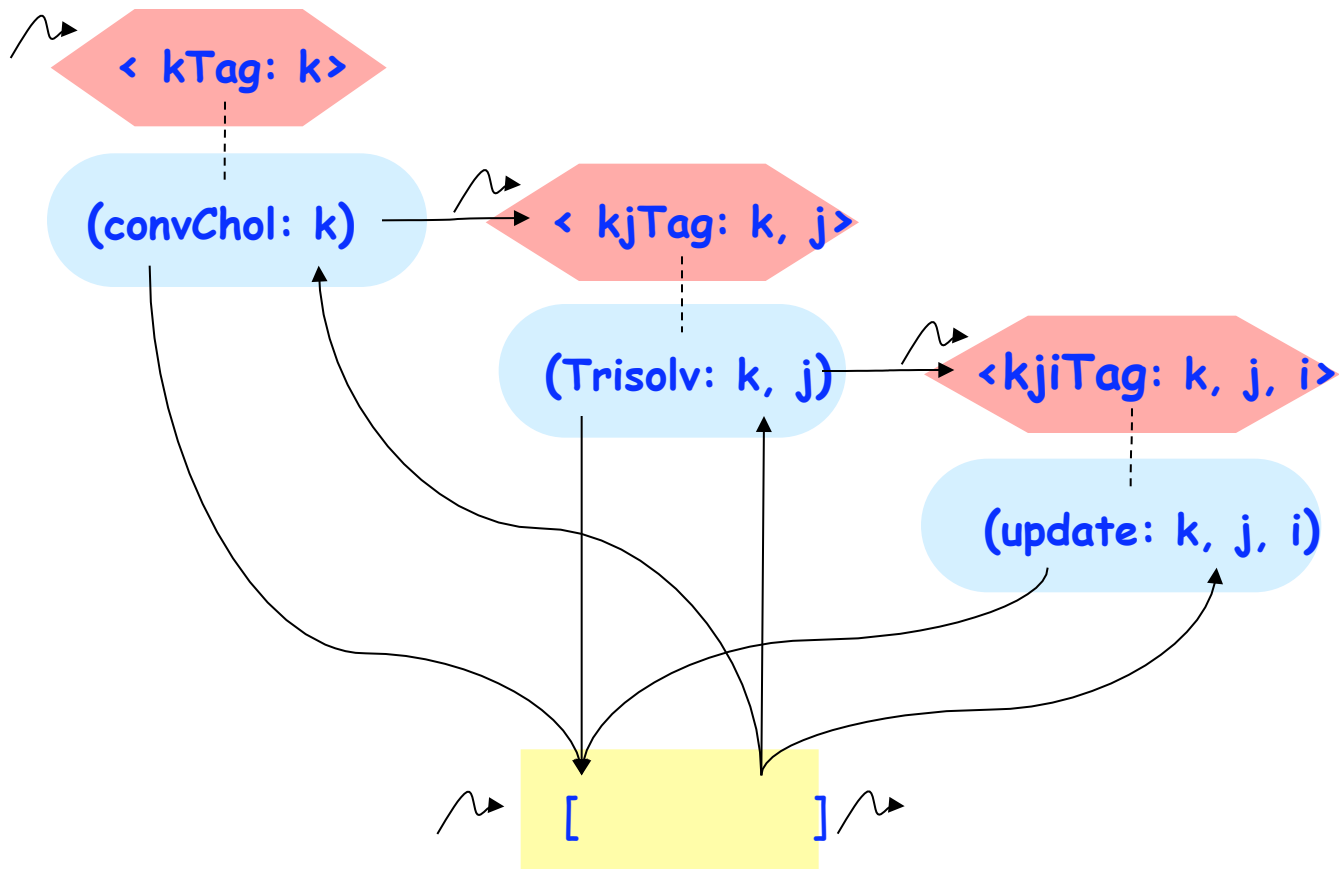
A wide variety of targets . . .

Cholesky factorization

- Input matrix B of size $n \times n$ where $n = p \cdot b$ for some b which denotes tile size
 - Output lower triangular matrix L
1. for $k = 1$ to p do
 2. conventionalCholesky(B_{kk} , L_{kk}):
 3. for $j = k + 1$ to p do
 4. triangularSolve(L_{kk} , B_{jk} , L_{jk}):
 5. for $i = k+1$ to j do
 6. symmetricRank-kUpdate(L_{jk} , L_{ik} , B_{ij}):

Aparna Chandramolishwaran
Rich Vuduc
Georgia Tech

Cholesky: graphical form



Line 2: Conventional Cholesky

K = 1			

Line 4: Triangular Solve

K = 1			
K = 1			
K = 1			
K = 1			

Line 6: Symmetric Rank k Update

K = 1			
K = 1	K = 1		
K = 1	K = 1	K = 1	
K = 1	K = 1	K = 1	K = 1

Line 2: Conventional Cholesky

	K = 2		

Line 4: Triangular Solve

	K = 2		
	K = 2		
	K = 2		

Line 6: Symmetric Rank k Update

	K = 2		
	K = 2	K = 2	
	K = 2	K = 2	K = 2

CnC asynchronous execution

K = 1			

CnC asynchronous execution

K = 1			

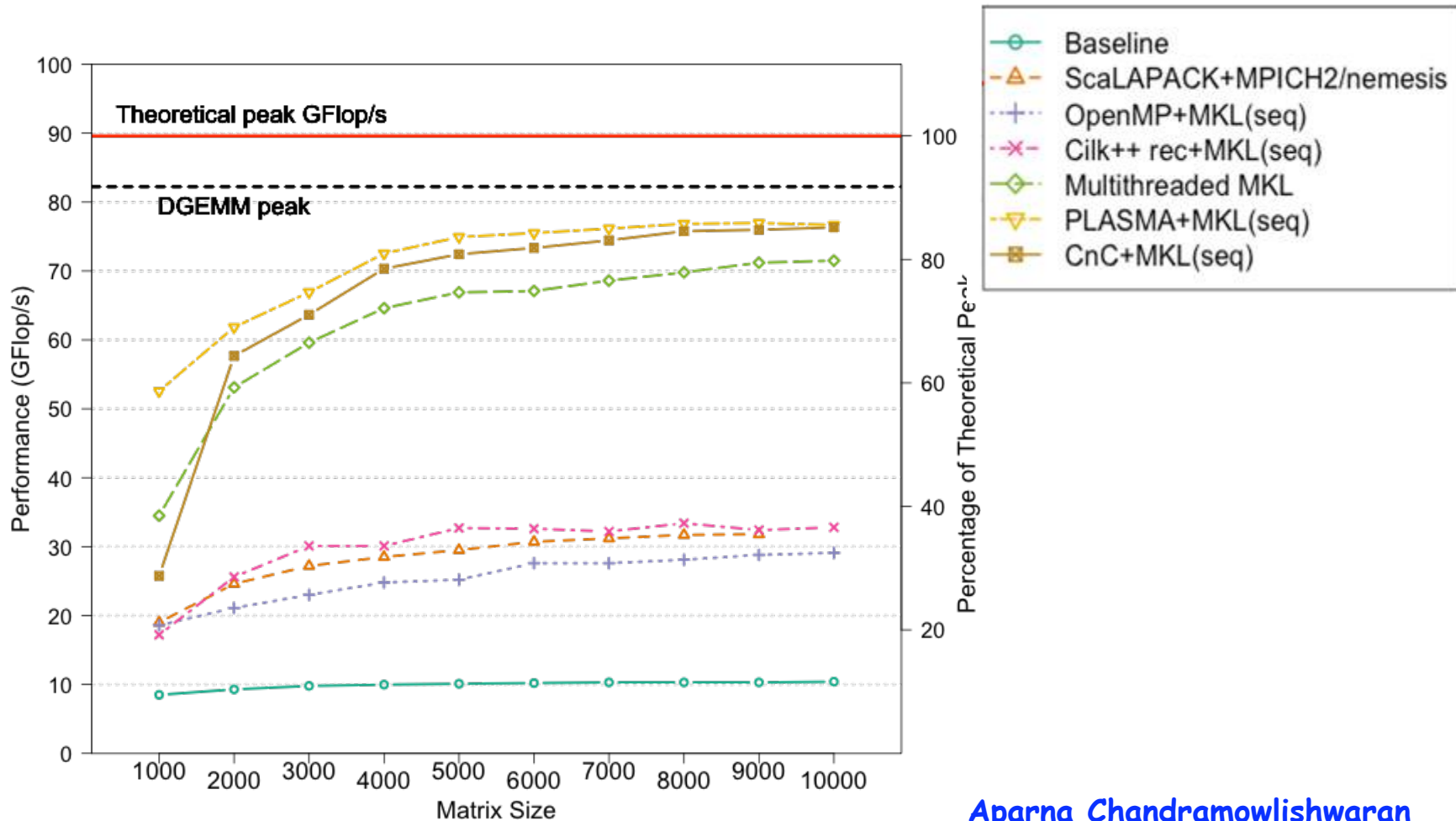
CnC asynchronous execution

	K = 1		
K = 1			

CnC asynchronous execution: $k=1$ not done. $k=2$ starts.

	K = 2		
	K = 1		
K = 1			

Cholesky Performance



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

Aparna Chandramowliswaran
Rich Vuduc
(Georgia Tech)

Dedup



- Use cheap resources (processing power) to make more efficient use of scarce resources (storage & bandwidth).
- Already in use in commercial products.
- Detects and eliminates redundancy in a data stream with a next-generation technique called 'deduplication'
- Input is an uncompressed archive containing various files
- Pipeline parallelism with multiple thread pools
- Huge working sets, significant communication

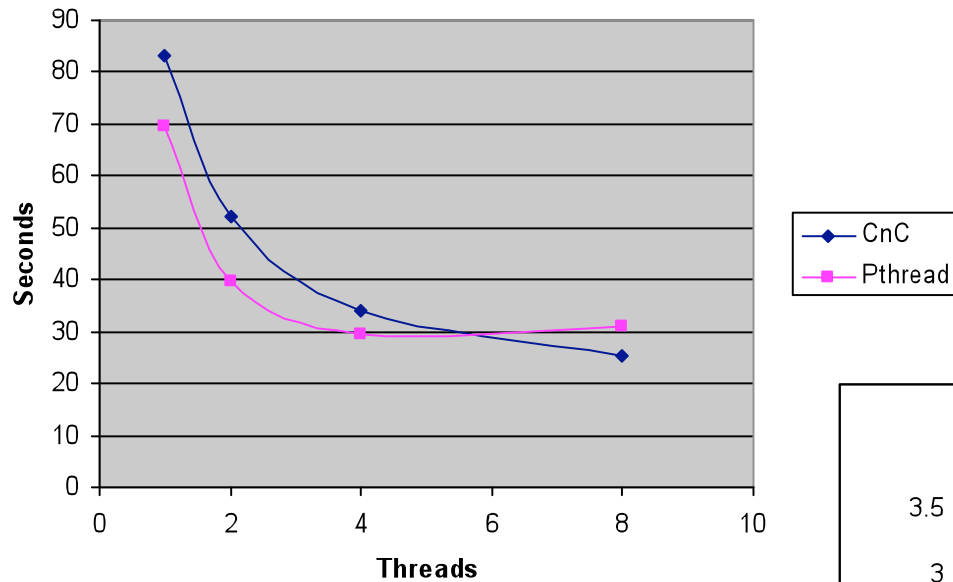
data domain



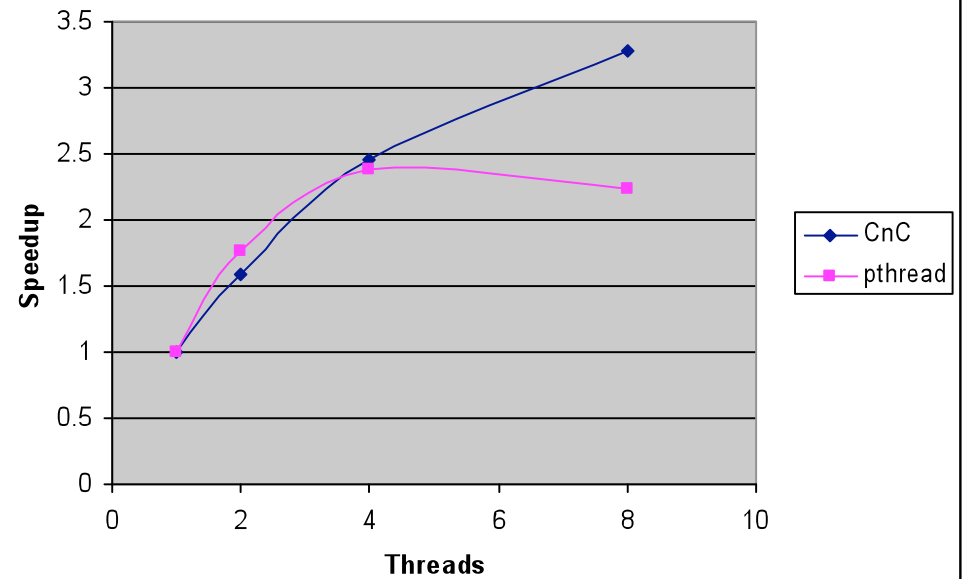
Next-generation storage and networking products already use data deduplication.

Dedup performance

Dedup timing on Linux (8 cores)



Dedup scaling on Linux (8 cores)



CnC Applications Experience

Body tracking
Heat diffusion
Face detection
PIRO feature extraction
Black-Scholes
Dedup compression
Cholesky
Eigenvalue solver
Matrix inversion
Conjugate gradient
Game Of Life
Medical imaging