



Concurrent Collections (CnC)

A programming model for
parallel programming

CnC



Kathleen Knobe
Intel



Ease of Use with Concurrent Collections (CuC)

Kathleen Knobe
Intel

Abstract

Parallel programming is hard. We present a new approach called Concurrent Collections (CnC). This paper briefly explains why writing a parallel program is hard in the current environment and introduces our new approach based on this perspective. In particular, a CnC program doesn't explicitly express the parallelism. It expresses the constraints on parallelism. These constraints remain valid regardless of the target architecture.

1. Why is parallel programming hard?

Many parallel languages embed parallel language constructs within the text of the serial code. Examples include MPI, OpenMP, PThreads, Ct etc. This embedding is the source of some unnecessary difficulties:

- Serial code requires a serial ordering. If there is no semantically required ordering among some blocks of code, an arbitrary ordering must be specified.¹
- Serial code modifies and refers to variables (locations), not values. Variables can be overwritten. This overwriting over-constrains the possible orderings.
- Serial code tightly couples the question of *if* we will execute code from *when* we will execute it. Arriving at some point in the control flow indicates both that, yes, we will execute this code and also that we will execute it now. This is true for loop iterations, recursive calls and invocations of other subroutines. These also constitute arbitrary ordering.

Converting serial code to parallel code involves uncovering alternate valid executions either by manually or automatically. In the presence of arbitrary ordering, this process requires a complex analysis (human or machine). Embedding parallel language constructs or pragmas in the midst of this problem again requires uncovering alternate valid executions. This is difficult to get right in the first place and to modify later. In addition, of course, the parallelism constructs might

- be for a constrained class or architectures (say, shared memory)
- focus on a limited type of parallelism (say, data parallelism)

So when the architecture changes, so must the code. For these reasons, embedding parallelism in serial code

¹ It not hard to find an ordering but it can be complicated for a program or a compiler to undo the ordering.

can limit both the language's effectiveness and its ease of use. In addition, these constraints might assume arbitrary constraints such as barriers after each loop or single-program-multiple-data (SPMD). Although this is not the focus of this paper, notice that these assumptions can also inhibit performance.

2. The essence of parallel execution

What does a runtime system need to know in order to execute a program in parallel? We are not yet asking how to specify the parallelism, how to optimize for any specific target, etc. We are just asking: What are the inputs to this decision?

We need to identify the semantically required scheduling constraints. These are:

- Data dependences (producer/consumer relations): One computation produces data consumed by another. Data is explicitly produced by a producer computation and explicitly consumed by (possibly multiple) consumer computations.
- Control dependences (controller/controlled relations): One computation determines if another will execute. To eliminate the tight coupling of the *if* and *when* control flow questions, control tags will be explicitly produced by a controller computation and will control the execution of a controlled computation. This puts the control and data dependences on the same level as in intermediate forms such as program dependence graphs [5].

The types of objects that need to be identified are:

- The computations, i.e., the high-level operations, in the application.
- The data structures that participate in data dependences among these high-level operations.
- The control tags that participate in control dependences among these high-level operations.

Overview

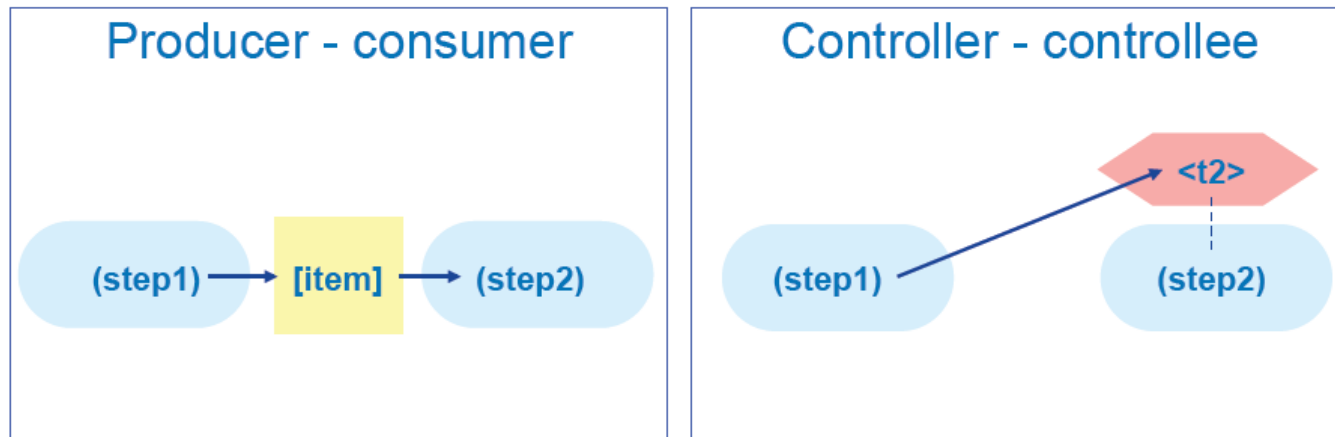
■ Ideas

- Separate *if* an operation is executed from *when* that operation is executed
 - Focus on ordering constraints dictated by semantics of application
 - Programming languages usually bind these together
 - Overburdens development effort
 - Limits implementation alternatives
- Dynamic single assignment
 - Use write-once values rather than variables (locations)
 - Avoids issues of synchronization, overwriting, etc.

Overview

■ Representations

- Diagram (“whiteboard”) version and text formats
- Relationships between high level operations (steps)
 - Data dependencies (producer-consumer relationship)
 - Control dependencies (controller-controllee relationship)
- High level operations (steps)
 - Purely functional
 - Implemented in conventional programming language



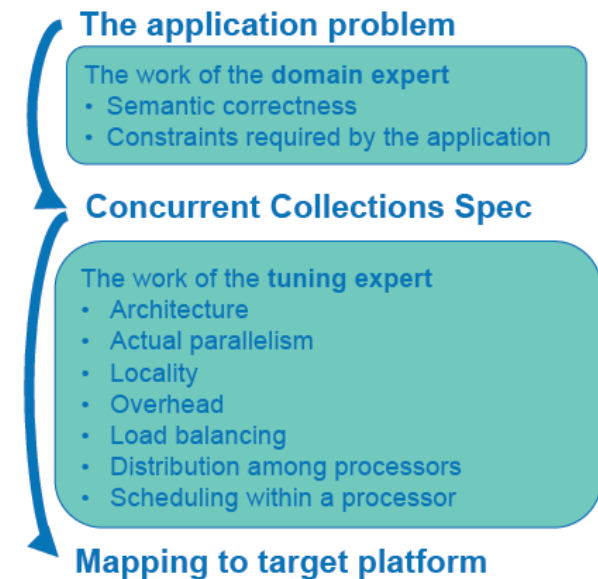
Note: figure from presentation by Kathleen Knobe (Intel) and Vivek Sarkar (Rice)

Overview

■ Advantages

- Allows roles and expertise of *domain expert* and *tuning expert* to be differentiated and combined by allowing each to focus on the aspects of the computation related to their expertise.

- Domain expert need not know about parallelism
- Tuning expert need not know about domain




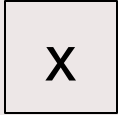
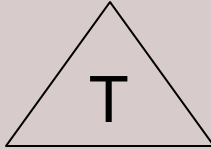

Note: figure from presentation by Kathleen Knobe (Intel) and Vivek Sarkar (Rice)

Overview

■ Advantages (cont.)

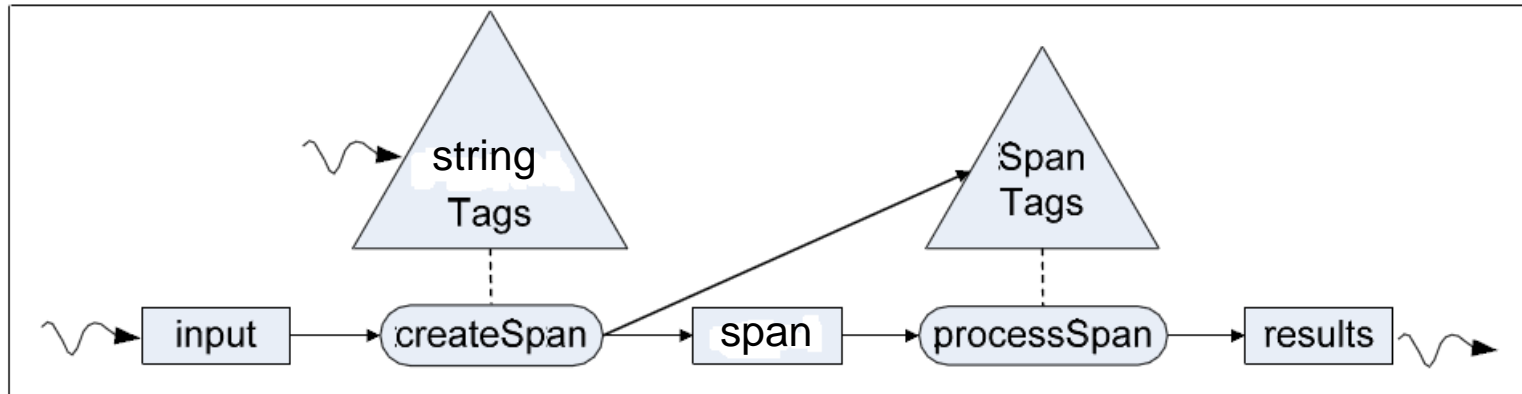
- Avoids specifying/reasoning/deducing which operations can execute in parallel
 - This is difficult to do
 - Depends on architecture
- Allows run-time support to be tailored for different architectures
- Creates portability across different architectures

Basic Structures

Element	CnC name	Graphical form	Textual form
computation	step		(foo)
data	item		[x]
control	tag		<T>
environment			env

Simple Example

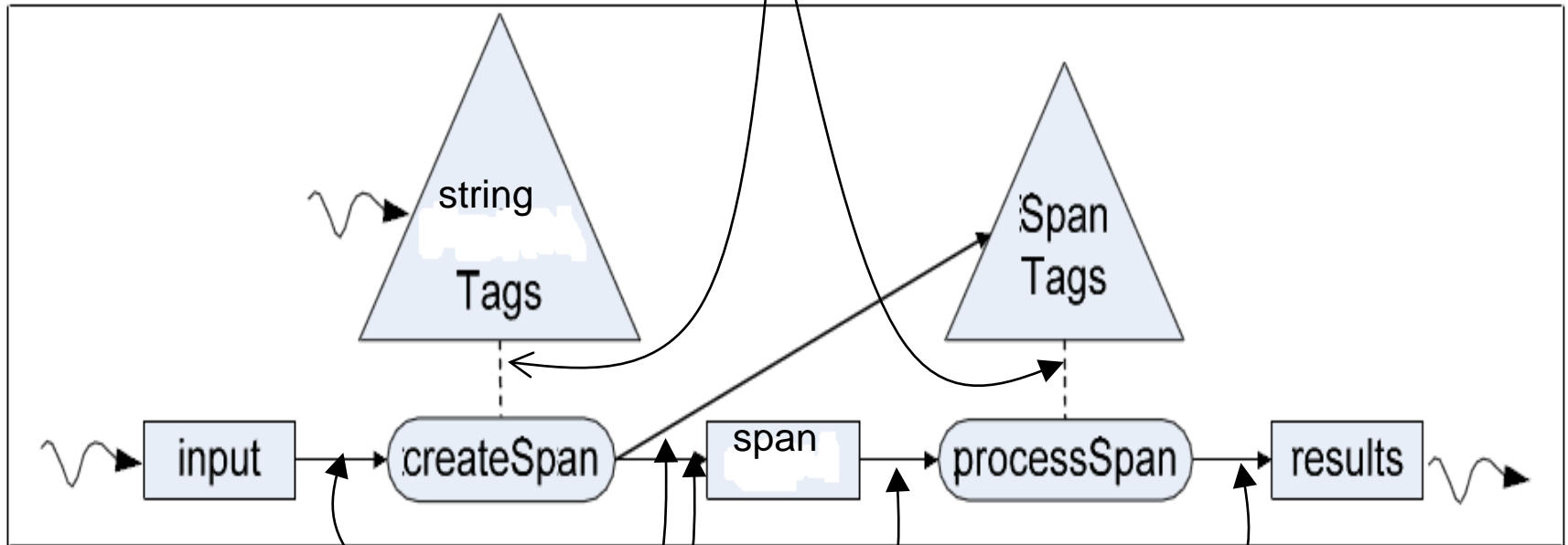
Produce odd length sequences of consecutive identical characters



	“aaa”	“aaa”
	“ff”	“qqq”
“aaaffqqqmmmmmmmm”	“qqq”	“mmmmmmmm”
	“mmmmmmmm”	

Relations

prescriptive

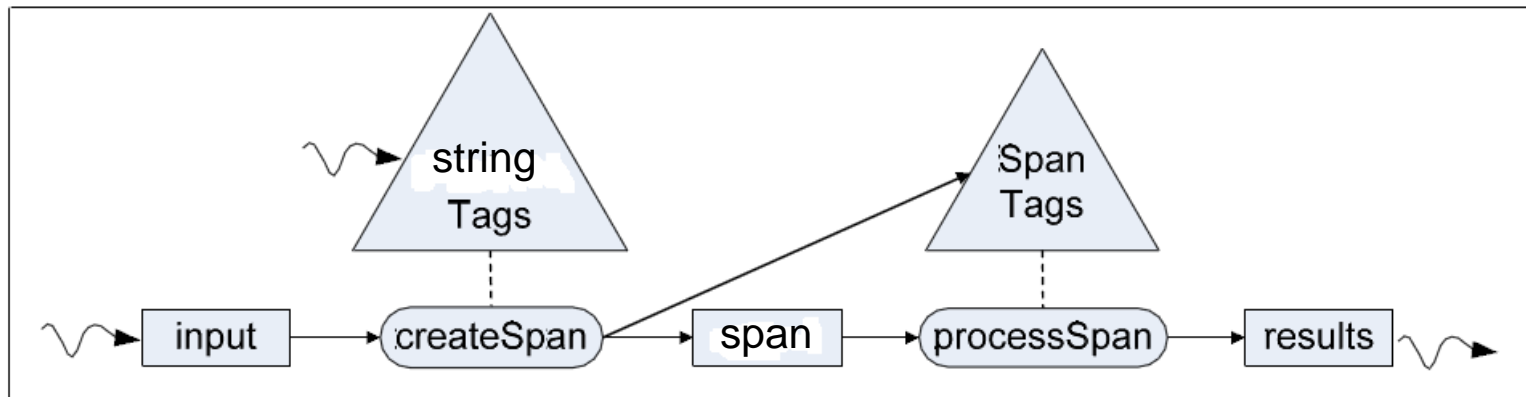


producer

consumer

Item Collections

- Multiple item instances correspond to different values of the item kind
- Each instance is distinguished by a user-defined instance tag



```
["aaaffqqqmmmmmmmm" : 1]
```

input

```
["aaa" : 1,1]
["ff" : 1,2]
["qqq" : 1,3]
["mmmmmmmm" : 1,4]
```

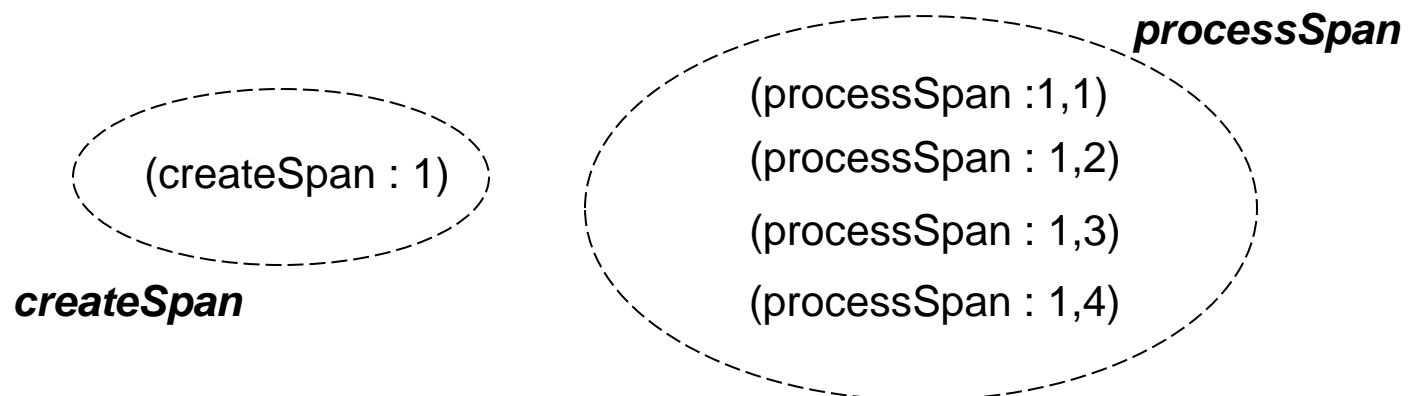
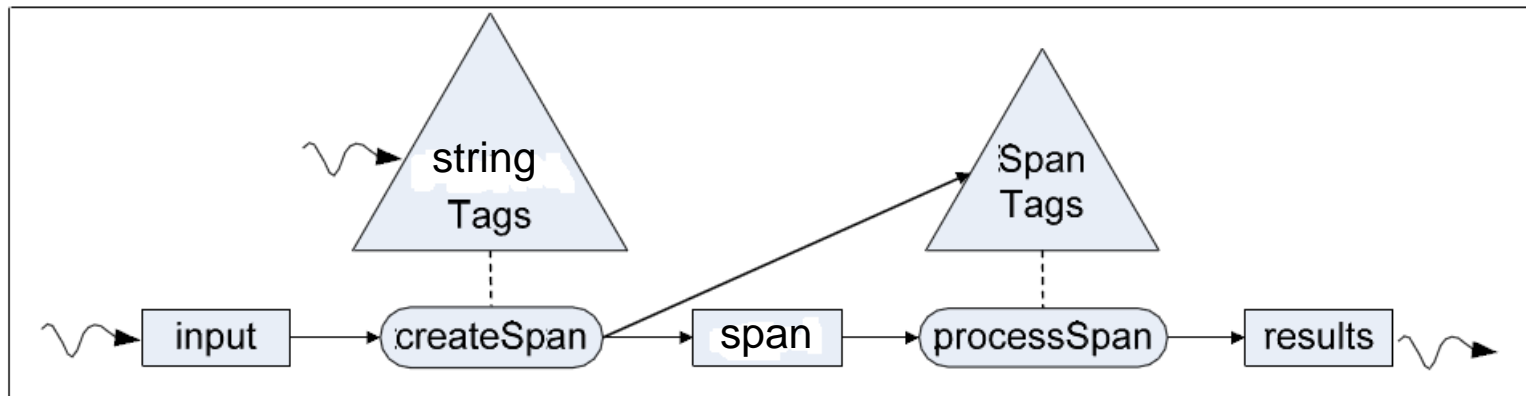
span

```
["aaa" : 1,1]
["qqq" : 1,3]
["mmmmmmmm" : 1,4]
```

results

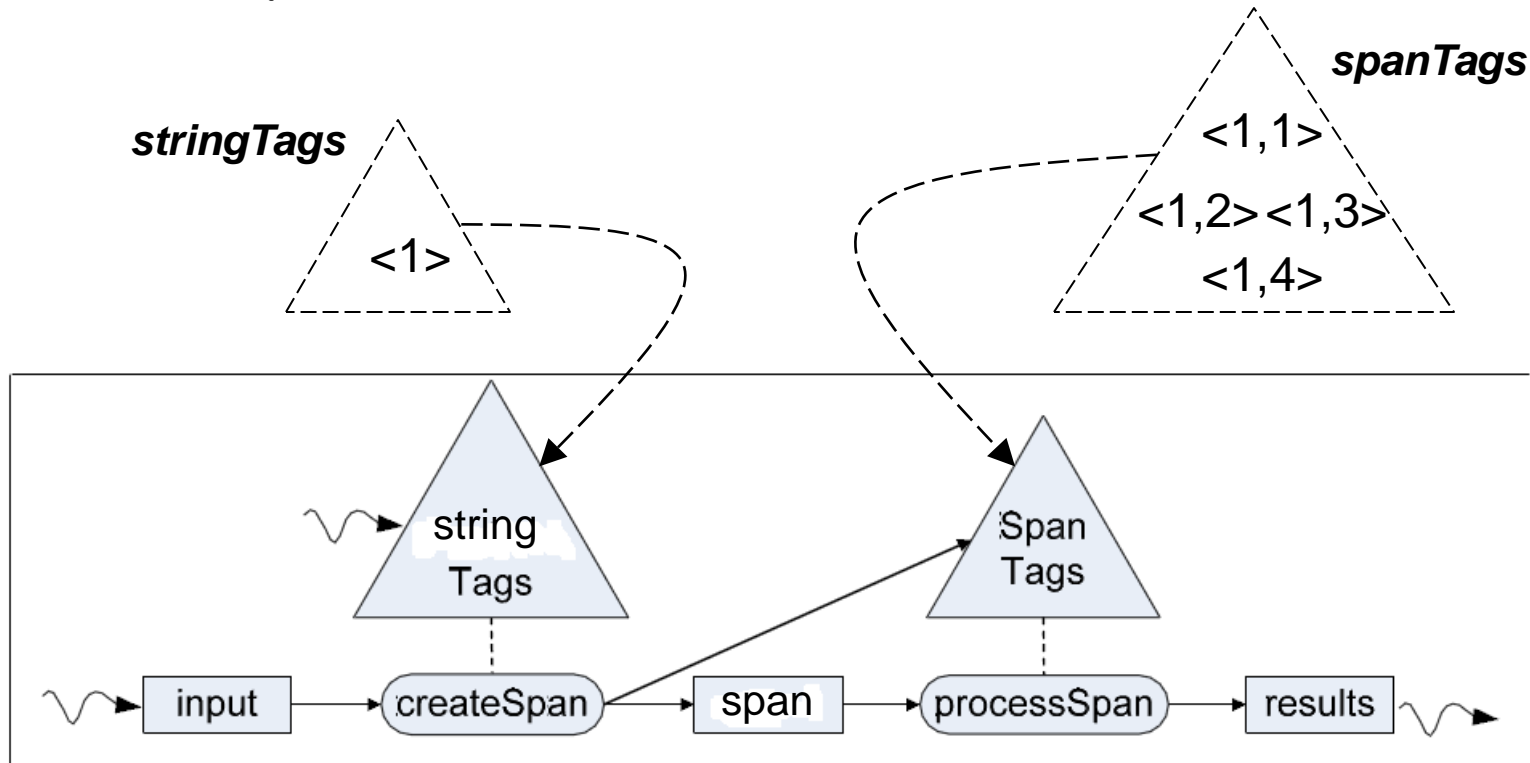
Step Collections

- Multiple steps instances correspond to different instantiations of the code implementing the step
- Each instance is distinguished by a user-defined instance tag



Tag Collections

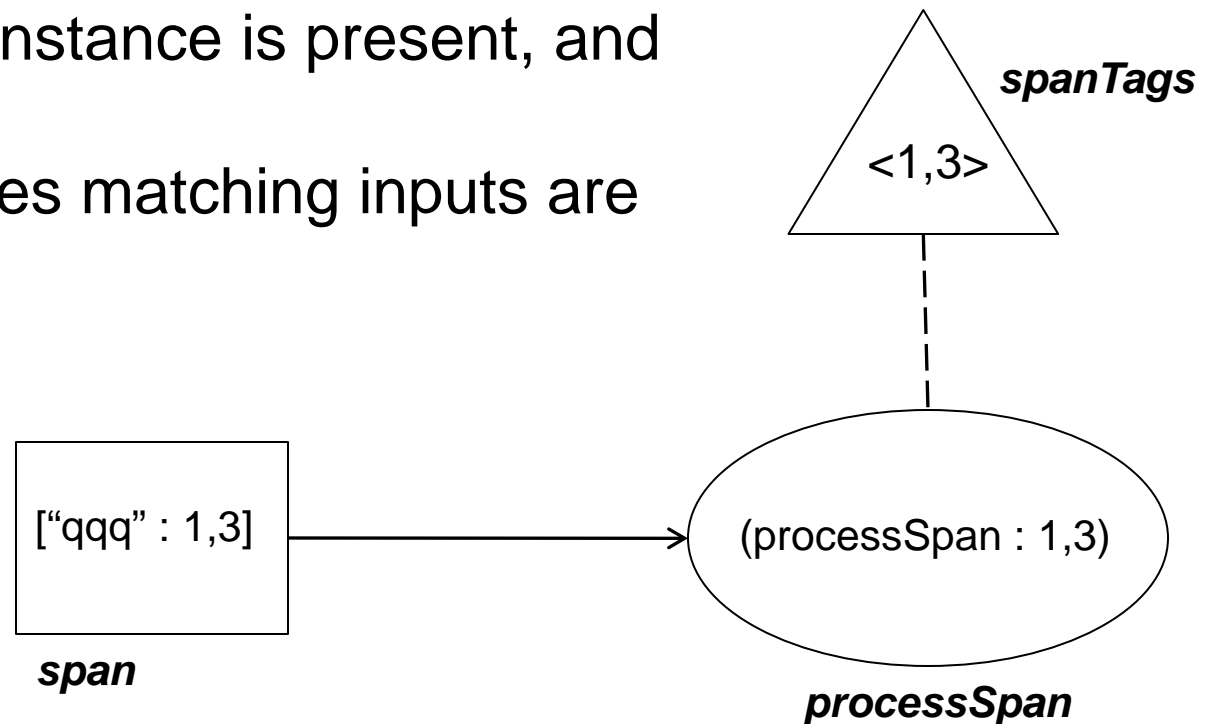
- Tag collections are sets of tags of the same type/structure as the step with which they are associated



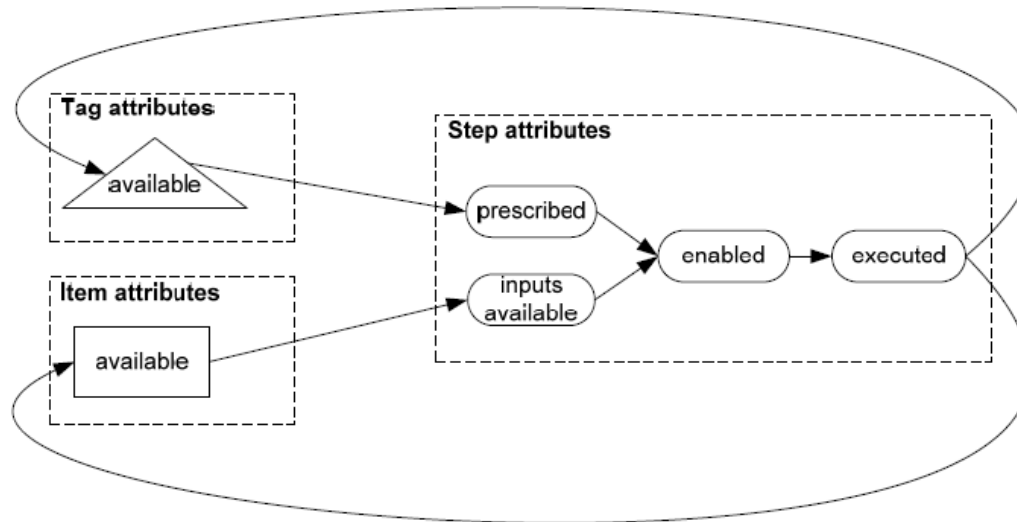
Execution Semantics

A step instance with a given tag will execute when

- a matching tag instance is present, and
- the step instances matching inputs are available



Semantics

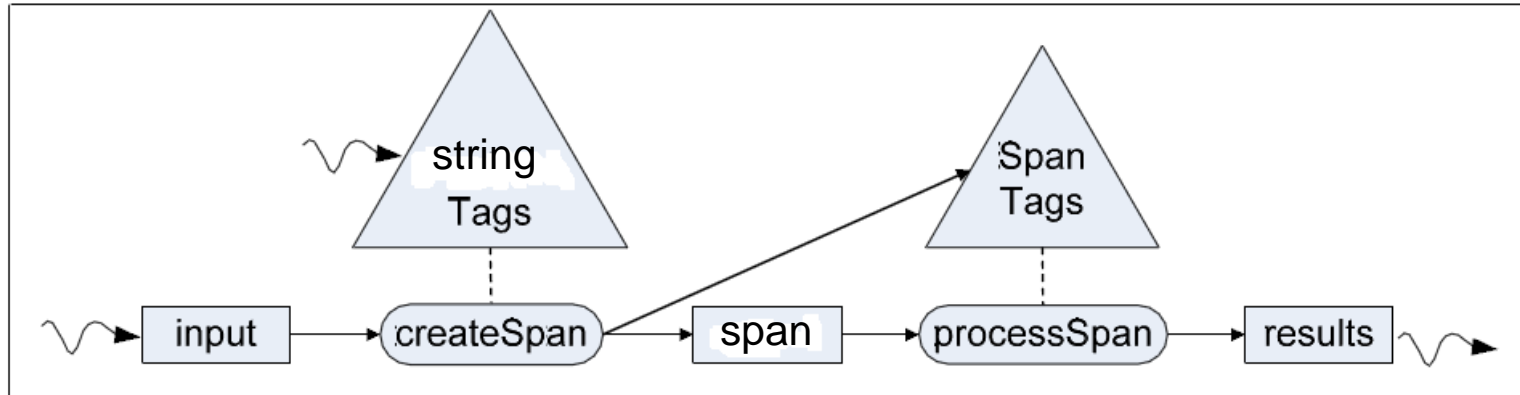


- When $(S : t_1)$ executes, if it produces $[I, t_2]$, then $[I, t_2]$ becomes *available*.
- When $(S : t_1)$ executes, if it produces $\langle T : t_2 \rangle$, then $\langle T, t_2 \rangle$ becomes *available*.
- If $\langle T \rangle$ prescribes (S) , when $\langle T : t \rangle$ is available then $(S : t)$ becomes *prescribed*.
- If forall $\{I, t_1\}$ such that $(S : t_2)$ gets $[I, t_1]$
 $[I, t_1]$ is *available* // if all inputs of $(S : t_2)$ are available
 then $(S : t_2)$ is *inputs-available*.
- If $(S : t)$ is both *inputs-available* and *prescribed* then is its *enabled*.
 Any *enabled* step is ready to execute.

Semantics

- Execution frontier: the set of instances that have any attributes and are not dead.
- Program termination: no step is currently executing and no unexecuted step is currently enabled.
- Valid program termination: a program terminates and all prescribed steps have executed.
- Instances that are dead may be garbage collected.
- Note: parallel execution is possible but not mandated; thus testing/debugging on a sequential machine is possible.

Sources of Parallelism



["aaaffqqqmmmmmmmm" : 1]
 ["bbbxxxxxffxxxxxy" : 2]

input

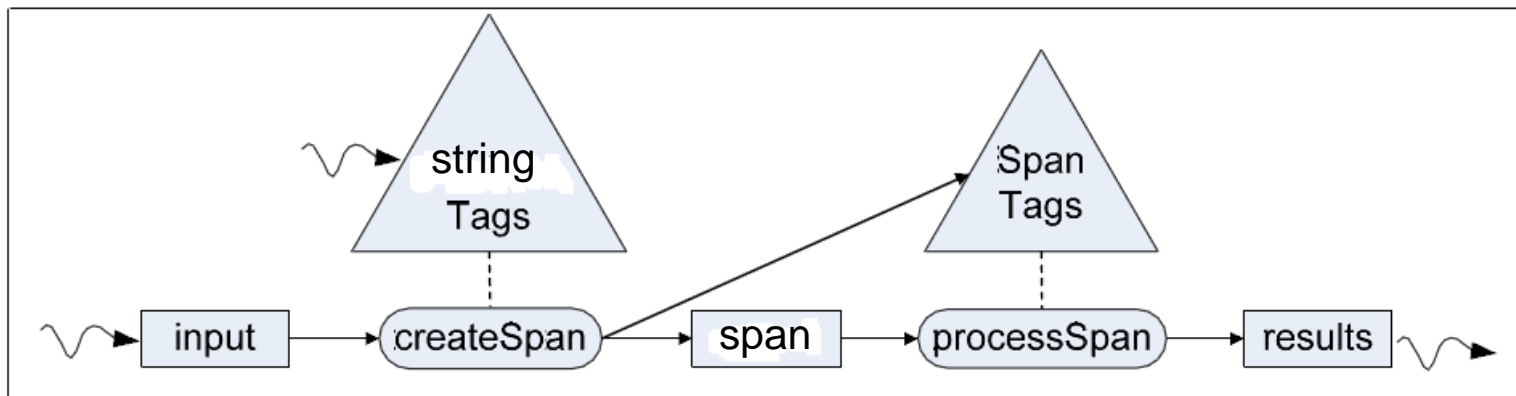
["aaa" : 1,1] ["ff" : 1,2]
 ["qqq" : 1,3]
 ["mmmmmm" : 1,4]
 ["bbb" : 2,1] ["xxxxx" : 2, 2]
 ["ff" : 2,3]

span

(processSpan : 1,1)
 (processSpan : 1,3)
 (processSpan : 2,2)
 (processSpan : 2,3)
 (createSpan : 2)

executing

Textual Representation



```

<stringTags: int stringID>;
<spanTags:   int stringID, int spanID>;
  
```

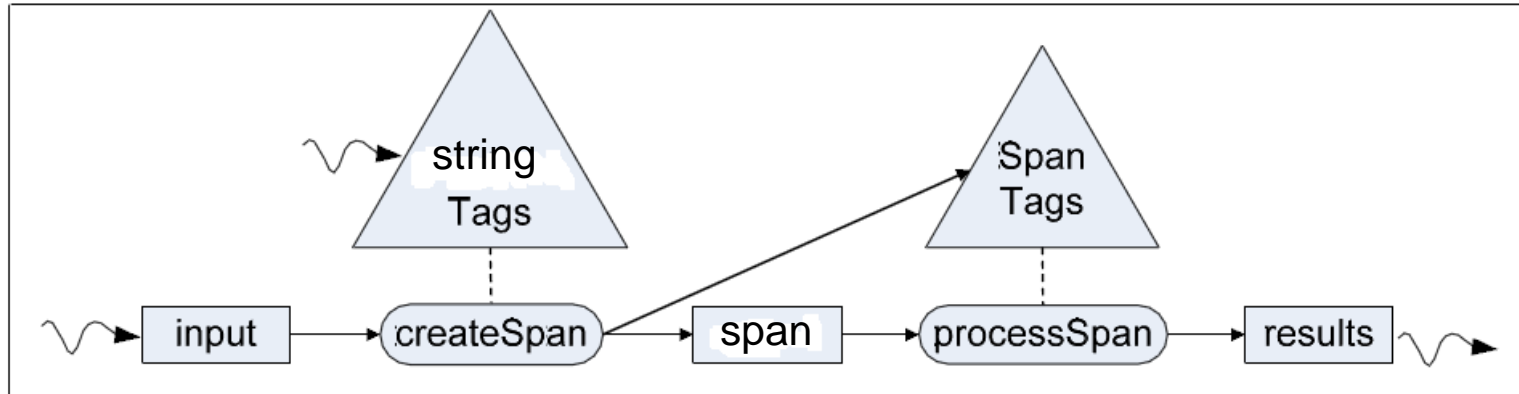
```

[input:      int stringID];
[span:       int stringID, int spanID];
[results:    int stringID, int spanID];
  
```

```

env -> [input], <stringTags>;
[results], <spanTags> -> env;
  
```

Textual Representation



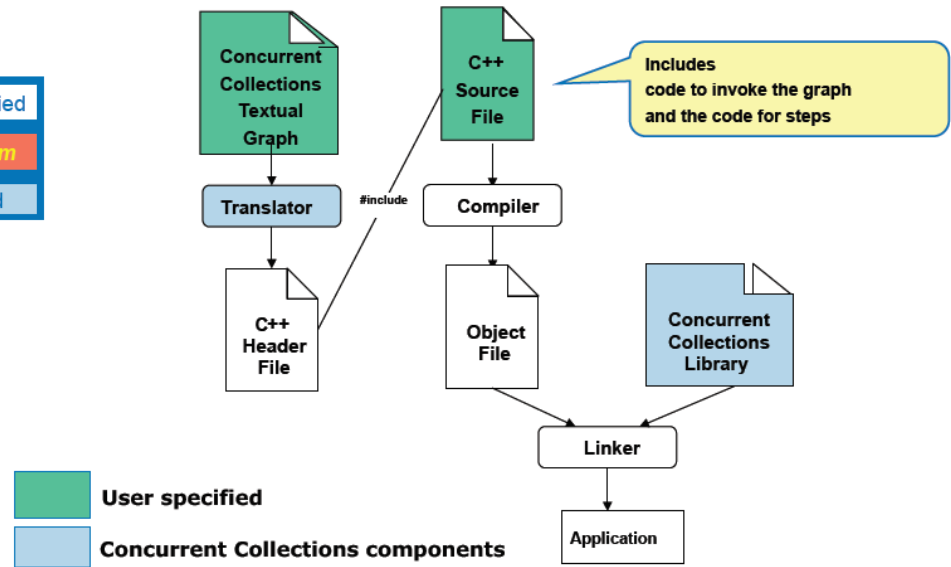
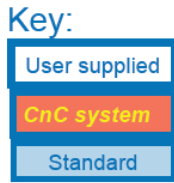
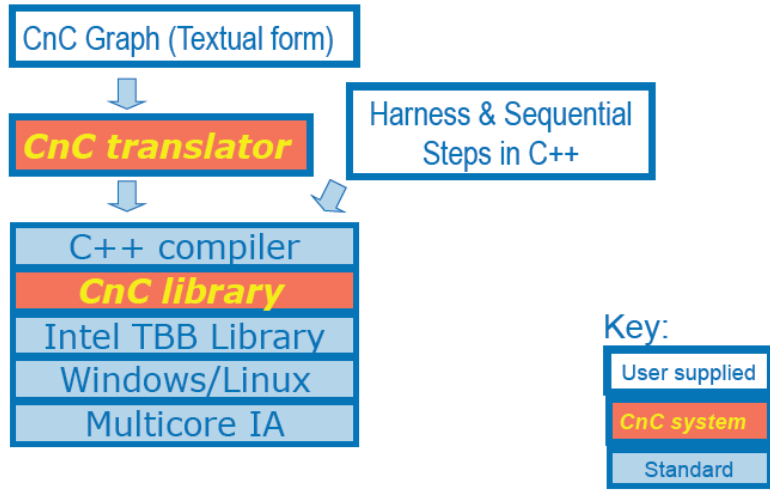
```

<stringTags> :: (createSpan);
<spanTags>   :: (processSpan);
  
```

```

[input: stringID]          -> (createSpan: stringID);
(createSpan: stringID)    -> <spanTags: stringID, spanID>;
(createSpan: stringID)    -> [span: stringID, spanID];
[span: stringID, spanID]  -> (processSpan: stringID, spanID);
(processSpan: stringID, spanID) -> [results: stringID, spanID];
  
```

Mechanics



Note: graphics from Kathleen Knobe (Intel), Vivek Sarkar (Rice), PLDI Tutorial, 2009

A coded step

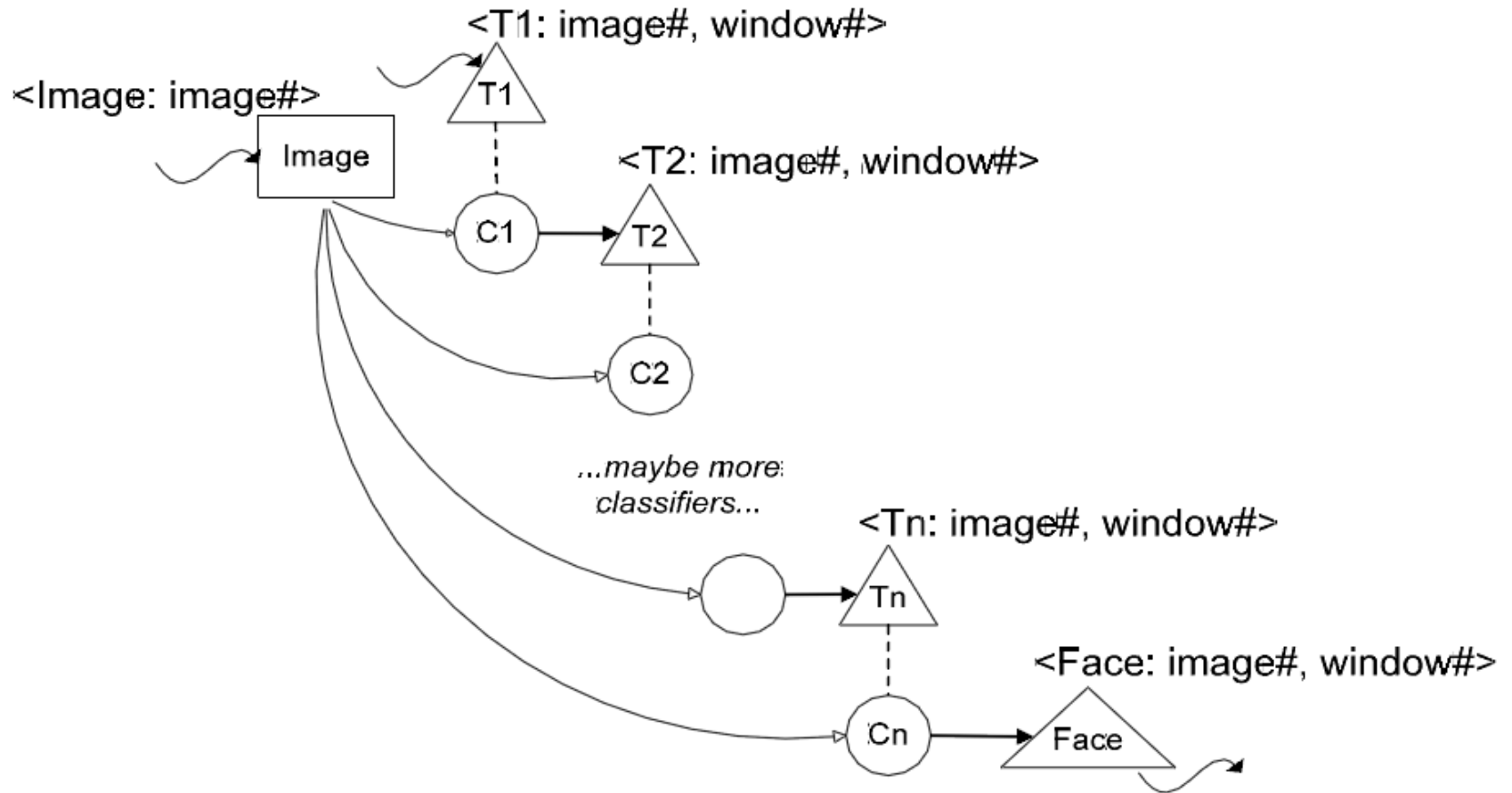
```

int createSpan::execute(const int & t, partStr_context & c) const
{ string in;
  c.input.get(t, in);

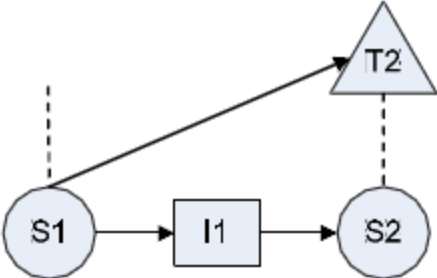
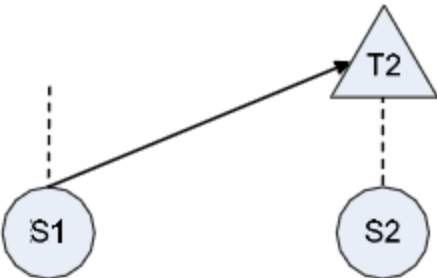
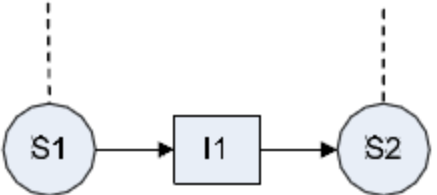

  if(! in.empty()) {
    char ch = in[0];
    int len = 0;
    unsigned int i=0;
    unsigned int j = 0;
    while (i < in.length()) {
      if (in[j] == ch) {
        i++; len++;
      } else {
        c.span.put(t, j, in.substr(j, len));
        c.spanTags.put (t,j);
        ch = in[i];
        len = 0; j = i;
      }
    }
    c.span.put(t, j, in.substr(j,len));
    c.spanTags.put(t, j);
  }
  return CnC::CNC_Success;
}

```

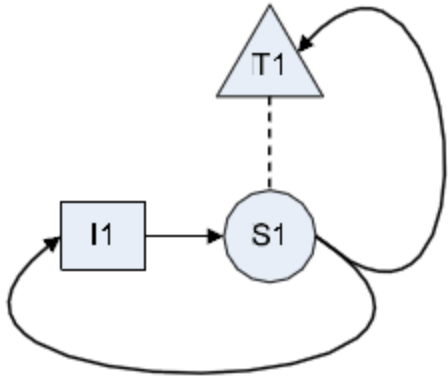
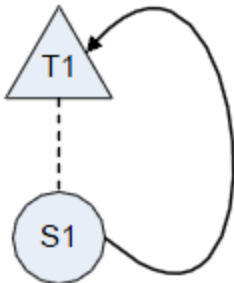
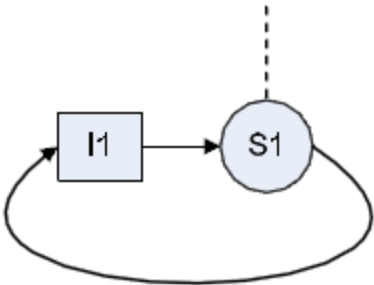

Another Example



Patterns – steps in different collections

<p>I: Distinct collections</p>	<p>A: Producer/Consumer</p>	<p>B: No Producer/Consumer</p>
<p>1: Controller/Controllee</p>		
<p>2: No controller/Controllee</p>		

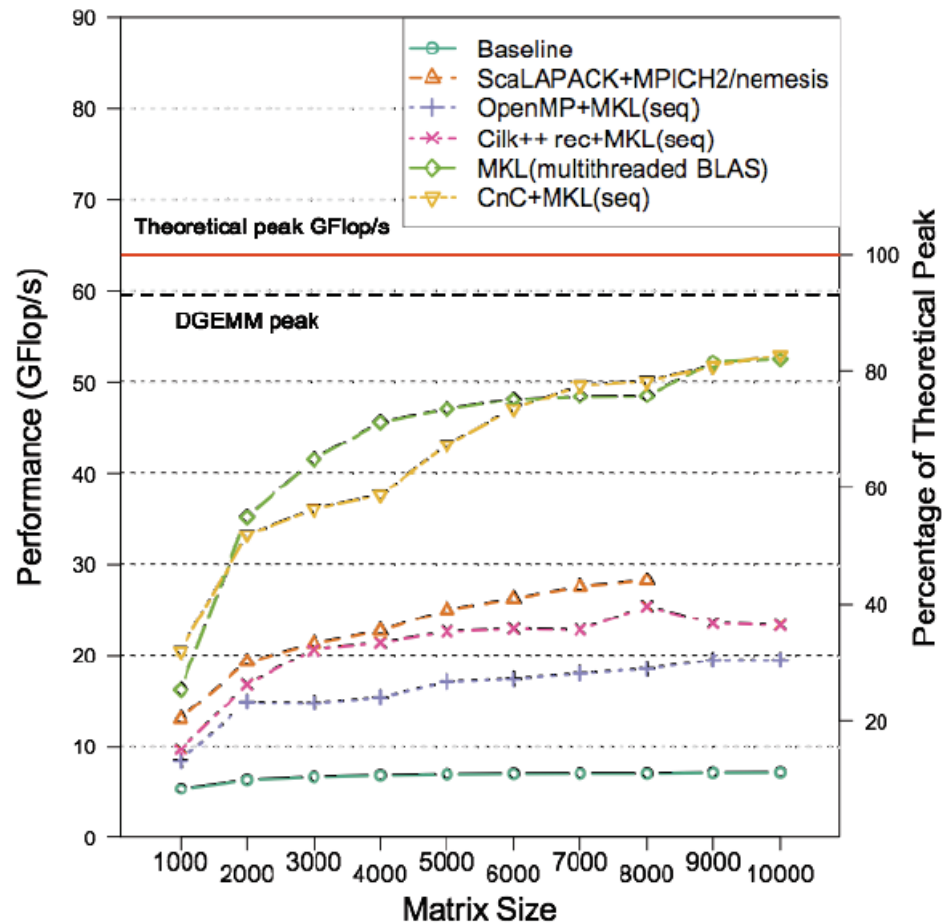
Patterns – steps in same collection

II: Same collection	A: Producer/Consumer	B: No Producer/Consumer
1: Controller/Controllee		
2: No Controller/Controllee		

Performance

Cholesky performance:

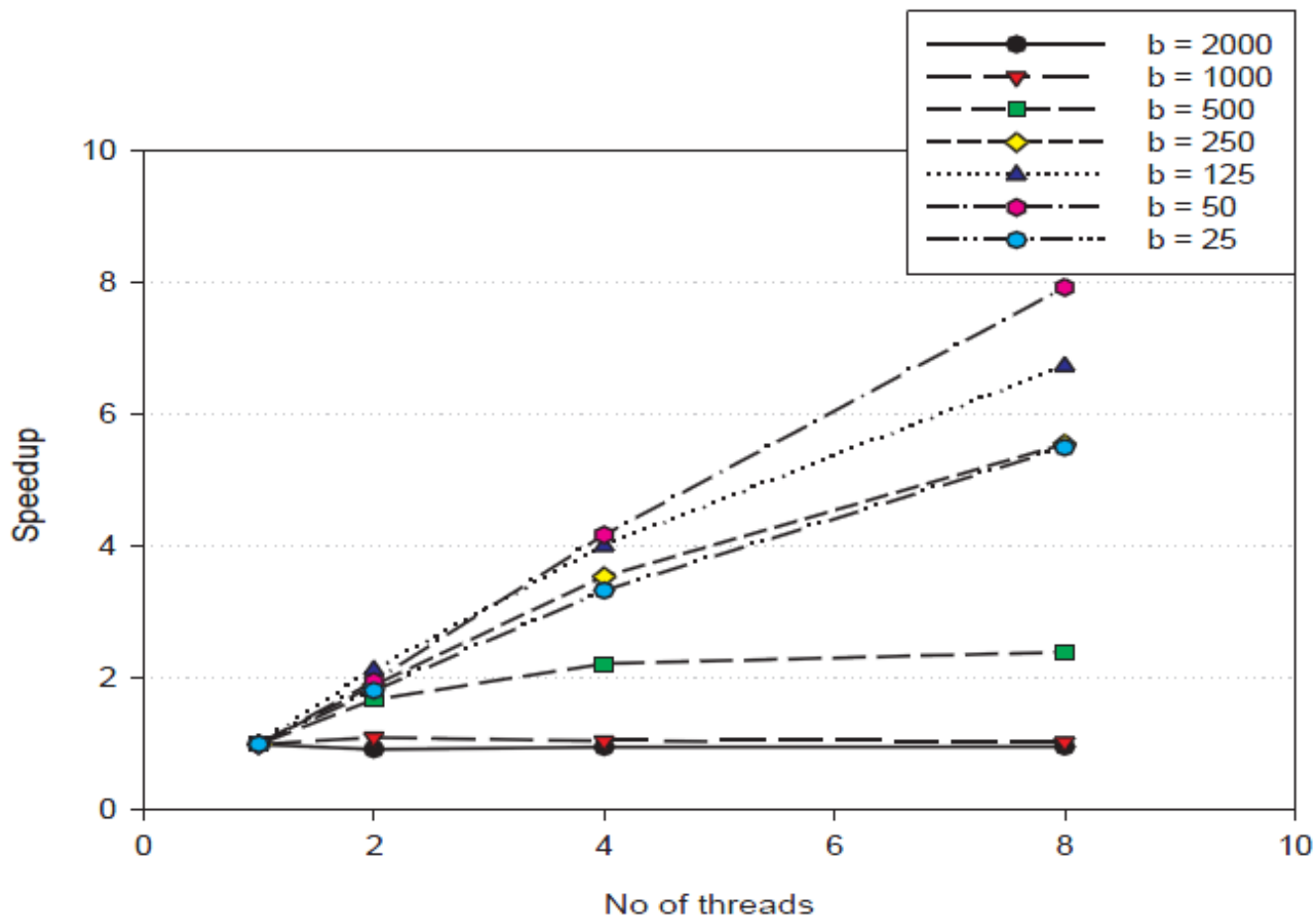
Intel 2-socket x 4-core Harpertown @ 2 GHz + Intel MKL 10.1



Acknowledgements: Aparna Chandramolishwaran, Rich Vuduc (Georgia Tech)

Performance

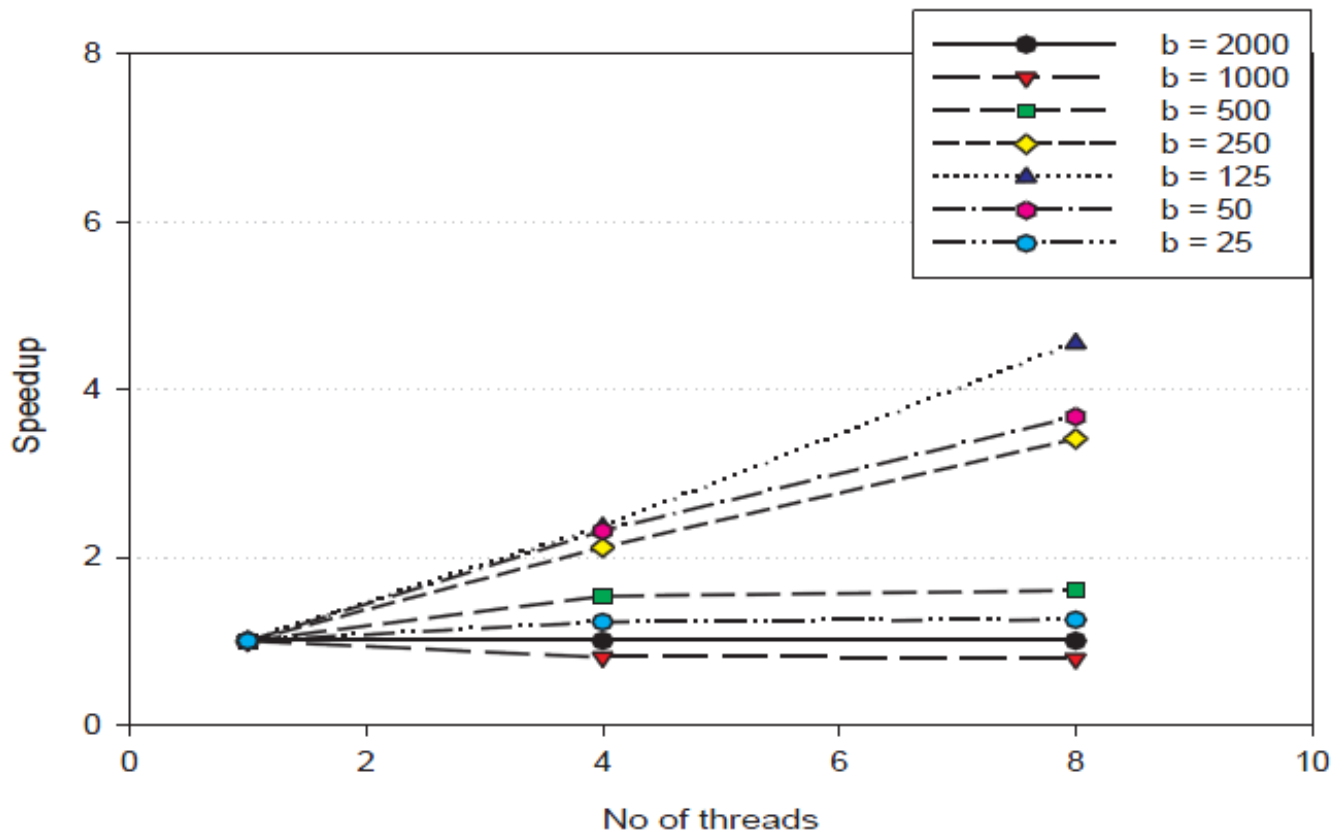
Cholesky Speedup (n = 2000)



TBB implementation, 8-way Intel dual Xeon Harpertown SMP system.

Performance

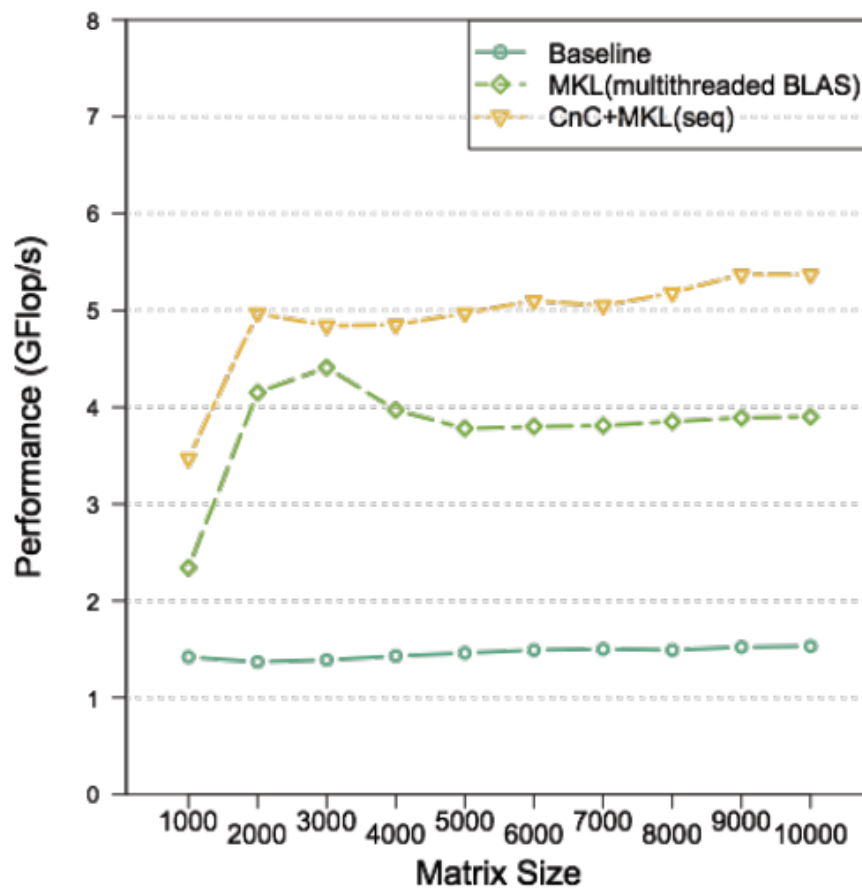
Cholesky Speedup (n = 2000)



Habenero-Java implementation, 8-way Intel dual Xeon Harpertown SMP system.

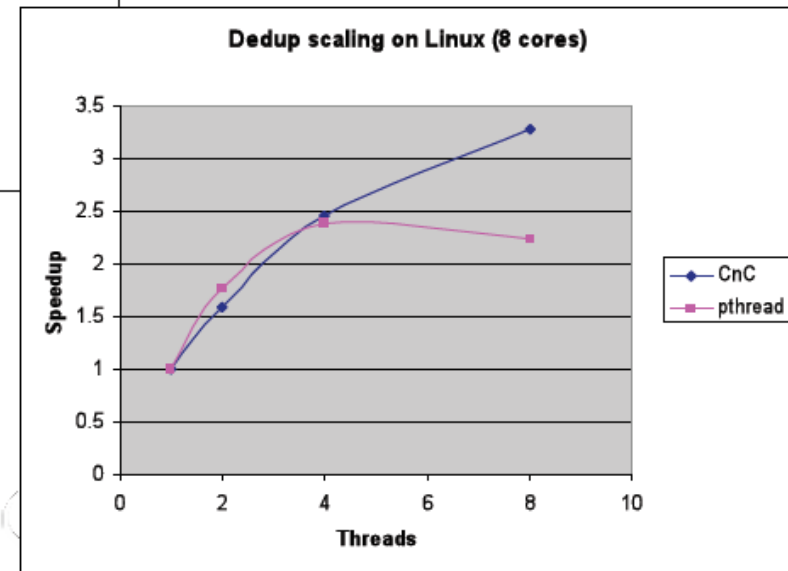
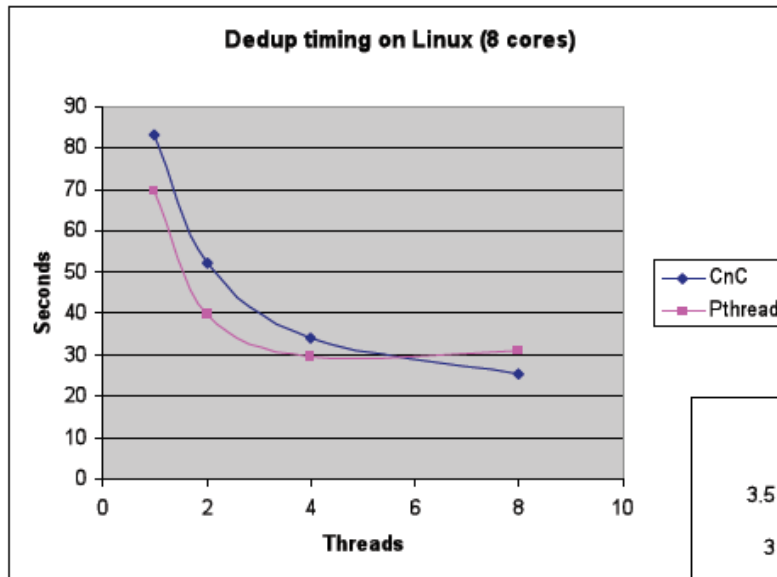
Performance

Eigensolver performance (dsygvx)
Intel Harpertown (2x4 = 8 core)



Acknowledgements: Aparna Chandramolishwaran, Rich Vuduc (Georgia Tech)

Performance



Input stream compression
using “deduplication”

Memory management

■ Problem

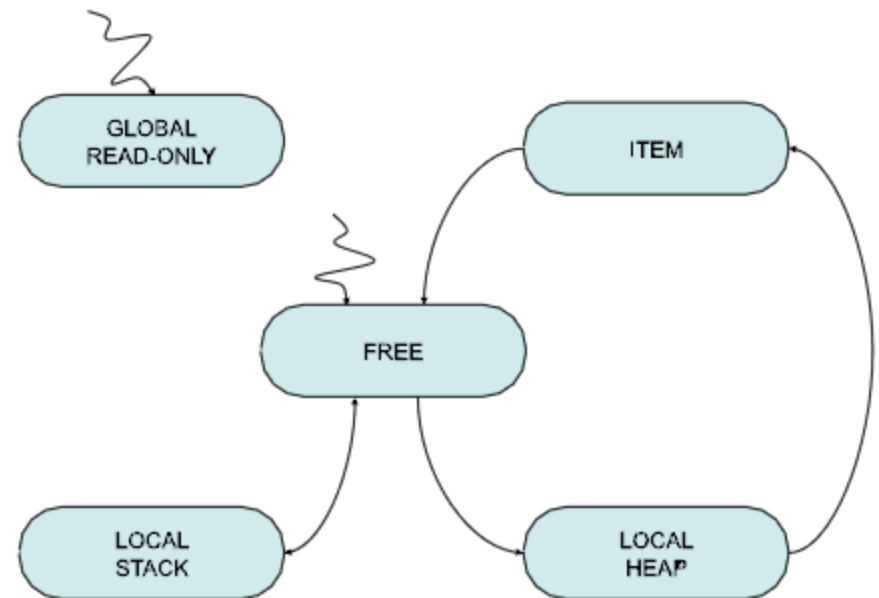
- the lifetime of a produced (data) item is not clear
- the (data) item may be used by multiple steps
- some step using a (data) item may not exist yet
- Serious problem for long-running computations

■ Solution

- Declarative annotations (*slicing annotations*) added to step implementations
- Indicates which (data) items will be read by the step
- Converted into reference counting procedures

Memory states

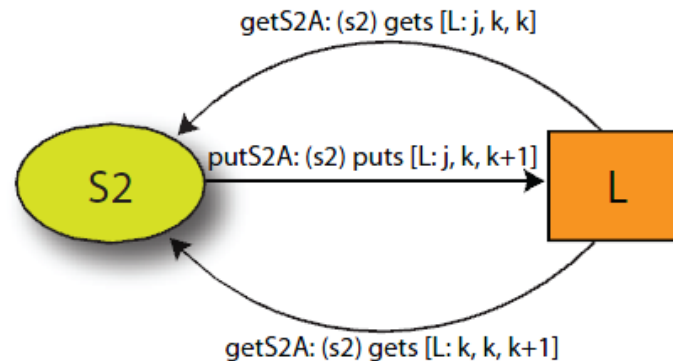
- 5 memory states
- Note: no transition from FREE to ITEM
- Assumes step implementation manages local stack and local heap correctly



Annotations

General form:

(S: I) is in *readers*([C: T], *constraints*(I,T)



$getS2A: (s2 : k, j) \subseteq readers([Lijk : t1, t2, t3])$, $t2 = t3 \wedge t3 = k \wedge t1 = j$

$getS2B: (s2 : k, j) \subseteq readers([Lijk : t1, t2, t3])$, $t1 = t2 \wedge t2 = k \wedge t3 = k + 1$

Conditions for removing an item

```

[I:i].dead =
  // indicates when an item
  // will not be used in the future
  for each (S) s.t. [I] -> (S)
    for each s in readers((S), [I])
      (S:s).complete

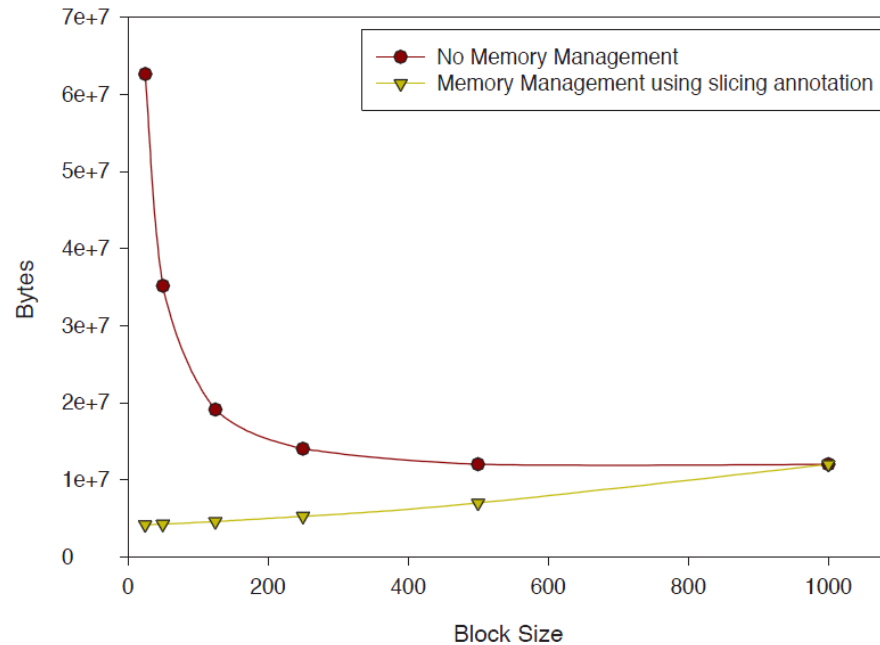
(S:s).complete =
  // indicates when it is known that the
  // step will not execute in the future
  (S:s).executed
  or ( <T>:: (S) and !<T:s>.available )

!<T:t>.available =
  // indicates when known that the tag will never
  // be available. This attribute can be put by
  // a step directly and is also propagated
  not(<T:t>.available) and
  for each (S) s.t. (S) -> <T>
    for each s in writers((S), <T>)
      (S:s).complete

```


Performance

Cholesky Factorization (N = 1000)



- Memory usage did not vary with number of cores
- Optimal (running time) tile size was 125 (for above case)
- Memory savings a factor of 7
- In other cases, memory savings a factor of 14