Outline

1. Concurrent Collections Model and Languages
2. Execution Semantics and Properties
3. Implementation Approaches
4. Patterns and Examples
5. Research Topics
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
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**

(\textit{step1}) \rightarrow \textbf{[item]} \rightarrow (\textit{step2})

**Controller - controllee**

(\textit{step1}) \rightarrow (\textit{step2})
Three main influences

1. Streaming
2. Tuple spaces
3. Dataflow

We will discuss each of them as they become relevant.
Notation

<table>
<thead>
<tr>
<th></th>
<th>White Board</th>
<th>Textual</th>
<th>Slideware</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computation Step</strong></td>
<td>foo</td>
<td>(foo)</td>
<td>(foo)</td>
</tr>
<tr>
<td><strong>Data Item</strong></td>
<td>x</td>
<td>[x]</td>
<td>[x]</td>
</tr>
<tr>
<td><strong>Control Tag</strong></td>
<td>T</td>
<td>&lt;T&gt;</td>
<td>&lt;T&gt;</td>
</tr>
</tbody>
</table>

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.
Collections of dynamic instances

Static

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.

Dynamic
Influence 1: Streaming

• CnC is like Streaming
  – Internals of a step are hidden from the graph that describes their connectivity
  – Isolation
    (step1) doesn’t know that (step2) exists, it simply produces [item]
    (step2) doesn’t know that (step1) exists. It just consumes [item]

• CnC differs from Streaming
  – Not strictly FIFO
  – Specified not only data and computation but also control

• Reference for Streaming in CnC
Collections of dynamic instances

Static

(step1)

Dynamic

(step2)
Collection names and instance tags

(foo: 12)  
(foo: 14)  
(foo: 15)

[x: 12, 1]  
[x: 12, 2]  
[x: 14, 1]  
[x: 14, 2]  
[x: 15, 1]  
[x: 15, 2]

(bar: 12)  
(bar: 14)  
(bar: 15)

(s: 1)  
(s: 2)  
(s: 3)  
(s: 4)

(q: 1)  
(q: 2)  
(q: 3)  
(q: 4)

<t: 1>  
<t: 2>  
<t: 3>  
<t: 4>
Tag collections: new concept

• Tags are typically related to the semantics of application (indices in a loop nest, nodes in a tree or graph).

\[
\begin{align*}
\text{loop } k &= \ldots \\
\text{loop } j &= \ldots \\
\text{loop } i &= \ldots \\
Z(i, j, k) &= A(k, j) \\
\end{align*}
\]

• Name the control and isolate controller (step1) from controllee (step2)
  – Some computation determines what widgets exist without knowing what will happen to the widgets. (step1) produces \(<t2>\)
  – Then independently multiple computations operate on widgets. (step2) is a prescribed by \(<t2>\)
  – (step1), the widget-determining step, can be incorporated in distinct graphs that do different things with widgets.

• Tag collection is the analog of region node in PDGs. (one controller – multiple controllees).

• In C or Fortran (where “if” and “when” are not distinguished) you often see multiple instances of identical loop nests. A tag collection is a way of defining the set of instances once but using them multiple times.

• just as index used for loop iterations and for array subscripts, we use tags for tag collections and for tagging items and steps.
Influence 2: Tuple Spaces (Linda)

• CnC is like Linda
  – Both are coordination languages that specify computations and communications via a tuple/tag namespace
  – Both create new computations by adding new tuples/tags to the namespace

• CnC differs from Linda
  – In Linda, an in() operation atomically removes the tuple from the tuplespace, but a CnC get() operation does not remove the item from the collection
  – In Linda, there is no separation between tags and values in a tuple; the choice of tag is implicit in the use of wildcards
  – In CnC, there is a separation between tags and values, and control tags are first class constructs like data items
    • CnC used to be called TStreams where T stood for “tagged”

• References:
  – “Coordination languages and their significance”, David Gelernter and Nicholas Carriero, CACM, Feb 1992
Objects

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

**Step Collections**
- Every step collection is controlled by a tag collection
- Functional. Only side-effects are put objects
- Gets items. Puts tags and items. All gets before any put
- Are tagged. A step has access to its tag value

**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: Consumer

• Corresponds to gets in steps

• A step may consume multiple distinct item collections
  \([x], [y] \rightarrow (\text{foo})\)

• A step may consume multiple instances of items from a given collection
  \([x: \text{neighbors}(i)] \rightarrow (\text{foo}: i)\)

• Conservative
  \([x] \rightarrow (\text{foo})\) means that \((\text{foo})\) may consume from \([x]\)
  If there is no consumer relationship between \([y]\) and \((\text{foo})\) then
  \((\text{foo})\) may not consume from \([y]\)
Relationships: producer

• Corresponds to puts in steps

• A step may produce multiple distinct item/tags collections
  \((\text{foo}) \rightarrow [x], [y]\)

• A step may produce multiple instances of items/tags from a given collection
  \((\text{foo: i}) \rightarrow [x: \text{neighbors(i)}]\)

• Conservative
  \((\text{foo}) \rightarrow [x]\) means that \((\text{foo})\) may produce \([x]\)
  If there is no producer relationship between \([y]\) and \((\text{foo})\) then \((\text{foo})\) may not produce \([y]\)
Relationships: prescription

• A prescription relationship between tag collection T and step collection S means that if tag instance in T with tag component values \( t_1, \ldots t_n \) is in T then the step with component values \( t_1, \ldots t_n \) will execute.

• The relationship is always the identity function.

• A tag collection may prescribe multiple step collections.
An Application (simple in the extreme)

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

Input string
“aaaffqqqmmmmmmm”

Sequences of repeated characters
“aaa”
“ff”
“qqq”
“mmmmmmm”

Filtered sequences
“aaa”
“qqq”
“mmmmmmm”
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?

Input = “aaaffqqqmmmmmmm”

(createSpan) → [span] = “aaa”

(processSpan) → [results] = “aaa”

(processSpan) → [results] = “qqq”

(processSpan) → [results] = “mmmmmmm”
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.

```
[Input] = “aaaffqqqmmmmmmm”
```

```
[span: 1] = “aaa”
[span: 2] = “ff”
[span: 3] = “qqq”
[span: 4] = “mmmmmmm”
```

```
(results: 1] = “aaa”
(results: 3] = “qqq”
(results: 4] = “mmmmmmm”
```
Make it precise enough to execute

How do we distinguish among the instances?

- Common case: Control and data dependence are identical. May provide syntax for this in the future.
Make it precise enough to execute

How do we distinguish among the instances?

[createSpan]

[span: j]

[processSpan: j, s]

[results: s]

[<stringTag: j>]

[<spanTag: j>]

[<spanTag: s>]

[results: j, s]

How do we distinguish among the instances?

[<stringTag: 1>]

[<stringTag: 2>]

[<stringTag: j>]

[<spanTag: j>]

[<spanTag: s>]

[<spanTag: 1, 1>]

[<spanTag: 1, 2>]

[<spanTag: 1, 3>]

[<spanTag: 1, 4>]

[results: 1, 1] = “aaa”

[results: 1, 2] = “ff”

[results: 1, 3] = “qqq”

[results: 1, 4] = “mmmmmmm”

[<span: 1, 1>] = “aaa”

[<span: 1, 2>] = “ff”

[<span: 1, 3>] = “qqq”

[<span: 1, 4>] = “mmmmmmm”

[input: 1] = “aaa”

[input: 2] = “ff”

[input: 3] = “qqq”

[input: 4] = “mmmmmmm”

[input: 5] = “rrhhhhxxx”

…
Summarize questions: White board level

• What are the high level operations? (step collections)
  
  (createSpan) \rightarrow (processSpan)

• What are the chunks of data? (item collections)
  
  [span] \rightarrow [input] \rightarrow [results]

• What are the producer/consumer relationships?
  
  [span] \rightarrow (createSpan) \rightarrow [span] \rightarrow (processSpan)

• What are the inputs and outputs? considered to be produced and consumed by the environment
  
  [input] \rightarrow \rightarrow [results]
Summarize questions: distinguish among instances

becomes

[\text{input: } j] \xrightarrow{\text{createSpan: } j} [\text{span: } j, s] \xrightarrow{\text{processSpan: } j, s} [\text{results: } j, s]
Summarize questions: control

• Control: Every step is a controllee.

• What are the distinct control tag collections? (iterations spaces)

• What are the controllers? (produce the instances in the control tag collections)
One slide summary of questions

- **White board level**
  - What are the high level operations? (step collections)
  - What are the chunks of data? (item collections)
  - What are the producer/consumer relationships?
  - What are the inputs and outputs? (produced and consumed by env)

- **Distinguish among the instances**

- **Control**
  - Every step is a controllee.
  - What are the distinct control tag collections? (iterations spaces)
  - What are the controllers? (produce the instances in the control tag collections)
No thinking about parallelism
Only domain/application knowledge

Result is:
• Parallel
• Deterministic (wrt results)
• race-free
Experience with questions: cell tracker

- [Input Image: K] 
  - (Cell detector: K) 
  - [Histograms: K] 
  - (Cell tracker: K) 
    - [Labeled cells initial: K] 
    - (Arbitrator initial: K) 
      - [State measurements: K] 
        - [Motion corrections: K, cell] 
        - (Arbitrator final: K) 
          - [Final states: K] 
            - (Prediction filter: K, cell) 
              - [Predicted states: K, cell]

- <Cell tags Initial: K, cell> 
- <Cell tags Final: K, cell>
Available Parallelism: among substrings
Available Parallelism: among...
Scheduling decision

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
• Range of number of substrings per string
• Cost of communication
• Rate of arrival of input strings

More than enough parallelism
- Favor processing existing substrings or
- Favor starting new input strings

These issues do not alter the CnC specification
More on scheduling later
Separation of Concerns between Domain Expert and Tuning Expert

Goal:
serious separation of concerns.

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

The application problem

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

Mapping to target platform

The application problem

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

What are the distinct controls?

What steps are the controllers?

Each tag collection is a subset of the previous.

- `<C1Tag>` specifies the set of faces
- `<C3Tag>` is the output of the application.

Each step is a controllee.
Textual form of graph

// Inputs from environment
env \rightarrow [\text{input: } j];

// outputs to environment
[\text{results: } j, s] \rightarrow \text{env};

// controller/controllee relations
<\text{spanTags: } j, s> :: (\text{processSpan: } j, s);

// producer/consumer relations
[\text{span: } j, s] \rightarrow (\text{processSpan : } j, s) \rightarrow [\text{results : } j, s];
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

When this tag is available
When these items are available

This step will execute (sometime)

[\text{T}: 3, 28]
[\text{T}: 3, 29]

(FOO: 3, 28)

[\text{X}: 2, 28]
[\text{X}: 3, 28]
[\text{X}: 4, 28]

[\text{Y}: 3]

This tag becomes available

and this tag is available

When this step executes
this step becomes enabled
Influence 3: Dataflow

• CnC semantic model is like dataflow
  – No extra serialization --- execution only needs to wait for data becoming ready
  – Steps are functional

• CnC differs from dataflow
  – Control is elevated to first class
  – Control can be available or not just as data is available or not
  – Item collections allow more general indexing than dataflow arrays (I-structures)

• References:
Summary of attributes: Comparison with Dataflow

- `< >` \{available\}
- `()` \{prescribed\}
- `()` \{inputs-available\}
- `()` \{inputs-available, prescribed\} (AKA: enabled)
- `()` \{inputs-available, prescribed, executed\}
- `< >` \{available\}
- `[ ]` \{available\}

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.*
Comparison with Microsoft Dryad Execution Engine

• Overview of Dryad
  – General-purpose distributed execution engine for coarse-grain data-parallel applications.
  – Combines sequential “vertices” with directed communication “channels” to form an acyclic dataflow graph.
  – Support for distributed systems --- channels are implemented TCP, files, or shared memory pipes as appropriate.
  – Dryad graph is specified by an embedded language (in C++) using a combination of operator overloading and API calls.

• Differences with CnC
  – CnC supports cyclic graphs with first-class tagged controller-controller relationships and tagged item collections.
  – Current CnC implementations are focused on multicore rather than distributed systems
Outline

1. Concurrent Collections Model and Languages
2. Execution Semantics and Properties
3. Implementation Approaches
4. Patterns and Examples
5. Research Topics
CnC Abstract Interpreter

• Theoretical model used to define operational semantics for CnC
  – Actual CnC implementations will be discussed later

• Operational semantics is defined with respect to the state of all item, tag, and step collections in a CnC program execution
  – Assumes unbounded memory and unbounded number of processors

• Abstract interpreter continues execution as long as at least one step instance’s Enabled attribute is true

• Motivation for defining abstract interpreter and formal semantics --- establish key properties of the CnC execution model
  – Determinism
  – Parallelism
  – Redundancy
Definitions

• \((SC:T)\) denotes a step in *step collection* SC with *tag* T

• \([IC:T]\) denotes the item in *item collection* IC with *tag* T

• \(<TC:T>\) denotes the entry *in tag collection* TC with *tag* T

• \(\pi(SC)\) is the (unique) tag collection that *prescribes* step collection SC

• \(\text{consumedBy}(SC)\) is the *set of item collections* that will be consumed by a step in step collection SC via *get* operations

• \(\text{consumedBy}(SC:T)\) is the *set of item instances* that will be consumed by step instance \((SC:T)\)
  • In this formalization, we assume that a CnC program completely specifies all items that will be consumed by a step instance (*strict* input semantics)
  • A CnC program that does not satisfy this property can always be transformed to one that does

• \(\text{producedBy}(SC)\) is the *set of item and tag collections* that may be produced in step collection SC via *put* operations

• \(\text{prescribedBy}(TC)\) is the *set of step collections* that are *prescribed* by tag collection TC
CnC Program State

• The state $\sigma$ of an executing CnC program consists of the following primitive attributes of item, tag and step collections
  – $\sigma.\text{[IC:T].Avail} = \text{true}$ if a put operation has been performed on item collection $IC$ with tag $T$
  – $\sigma.\text{[IC:T].Value} = \text{value stored in item collection} \ IC \ \text{with tag} \ T$
    if $\sigma.\text{[IC:T].Avail} = \text{true}$, or $\perp$ (undefined) otherwise
  – $\sigma.\text{<TC:T>.Avail} = \text{true}$ if a put operation has been performed on tag collection $TC$ with tag $T$
  – $\sigma.\text{(SC:T).Done} = \text{true}$ if step SC(T) has executed and all its put operations have been performed

• The null state $\sigma_0$ for a CnC program is defined as follows
  – $\sigma_0.\text{[IC:T].Avail} = \text{false}$ for all item collections $IC$ and tags $T$
  – $\sigma_0.\text{[IC:T].Value} = \perp$ (undefined) for all item collections $IC$ and tags $T$
  – $\sigma_0.\text{<TC:T>.Avail} = \text{false}$ for all tag collections $TC$ and tags $T$
  – $\sigma_0.\text{(SC:T).Done} = \text{false}$ for all step collections $SC$ and tags $T$

• A value of $\sigma = \perp$ represents an error state in a CnC program in which all attributes are $\perp$ (undefined)
Derived attributes

For convenience, we define the following derived attributes that can be computed from the primitive state attributes Avail, Value and Done

- $(SC:T).Prescribed = <\Pi(SC):T>.Avail$, indicates if step $(SC:T)$ has been prescribed
  - Recall that $\Pi(SC)$ is the (unique) tag collection that prescribes step collection SC

- $(SC:T).InputsAvail = \land_{[ICj:Tj] \in consumedBy(SC:T)} [ICj:Tj].Avail$
  - indicates if each input item $[ICj:Tj]$ for step $(SC:T)$ is available

  - indicates if step $(SC:T)$ is ready to execute
  - enabled steps can execute in parallel or in any order

The prefix $\sigma.$ is omitted when it’s clear from the context which state is being referred to
Life Cycle of Primitive and Derived Attributes

- **Primitive Attribute**
  - Prescribed
  - Enabled
  - Available

- **Derived Attribute**
  - Available
  -InputsAvailable
  - Done
  - Available

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.*
Interaction of a CnC Program with its Environment: Initial State

• The initial state $\sigma_i$ of a CnC program is obtained by starting with $\sigma_0$, and then executing an *Environmental Input* computation, EI
  – The only CnC operations that can be performed by EI are *individual put* operations on item and tag collections

• Example

```
Environmental Input computation, EI:
put([input: 1], “aaaffqqqmmmmmmmm”);
put([input: 2], “rrhhhhxxx”);
put(<stringTag: 1>);
put(<stringTag: 2>);
```
Interaction of a CnC Program with its Environment: Final State

- The final state $\sigma_f$ of a CnC program is examined by executing an Environmental Output computation $EO$ that can perform the following special CnC operations:
  - *Nonblocking get operation on a specific item* $[IC:T]$, which returns the value if $[IC:T].Avail = \text{true}$ or an error code otherwise (instead of blocking)
  - *Iterated get operation on an item collection* $IC$, which returns all available items $[IC:T]$ in $IC$ with their values, $[IC:T].Value$
  - *Nonblocking get operation on a specific tag* $<TC:T>$ that returns the value of $<TC:T>.Avail$
  - *Iterated get operation on a tag collection* $TC$, which returns the tags for all available items $<TC:T>$ in $TC$

\[
\begin{align*}
\text{Environmental Output computation, } EO: \\
\text{Iterated get on results}
\end{align*}
\]
State Mapping for Enabled Step

• Execution of step (SC:T) in state $\sigma$ results in state $\sigma'$ following its execution where $\sigma' = \text{Apply}(\sigma, (SC:T))$ is derived as follows:
  
  – For each put($IC,T_a,V$) operation performed by (SC:T),
    $\sigma'.[IC:T_a].\text{Value}$ is updated from $\bot$ to $V$ and $\sigma'.[IC:T_a].\text{Avail}$ is set = true
  
  • Single Assignment Rule: If $\sigma.[IC:T].\text{Avail} = \text{true}$, then the entire state $\sigma'$ is set to $\bot$
  
  – For each put($TC,T_b$) operation performed by (SC:T),
    $\sigma'.<TC:T_b>.\text{Avail}$ is updated from false to true
  
  • Single Assignment Rule: If $\sigma.<TC:T_b>.\text{Avail} = \text{true}$, then the entire state $\sigma'$ is set to $\bot$
  
  – $\sigma'.(SC:T).\text{Done}$ is set to true when execution of step (SC:T) is completed
  
  • Note that all derived attributes (Prescribed, InputsAvail, InputsReady) impacted by step (SC:T)’s execution will resolve to updated values in $\sigma'$ after step (SC:T) completes execution
CnC Program Execution

- Red: prescribed
- Blue: inputs available
- Red & blue: enabled
- Yellow: executed
Merge of CnC Program States

We define $\sigma' = \text{Merge}(\sigma_1, \sigma_2)$, the merge of states $\sigma_1$ and $\sigma_2$ as follows:

- $\sigma'.[\text{IC:T}].\text{Avail} = \sigma_1.[\text{IC:T}].\text{Avail} \lor \sigma_2.[\text{IC:T}].\text{Avail}$

- $\sigma'.[\text{IC:T}].\text{Value} =$
  - $\sigma_1.[\text{IC:T}].\text{Value}$ if $\sigma_2.[\text{IC:T}].\text{Avail} = \text{false}$
  - $\sigma_2.[\text{IC:T}].\text{Value}$ if $\sigma_1.[\text{IC:T}].\text{Avail} = \text{false}$
  - $\sigma_1.[\text{IC:T}].\text{Value}$ if $\sigma_1.[\text{IC:T}].\text{Avail} \land \sigma_2.[\text{IC:T}].\text{Avail}$
    $\land \sigma_1.[\text{IC:T}].\text{Value} = \sigma_2.[\text{IC:T}].\text{Value}$

- $\sigma' = \bot$ if $\sigma_1.[\text{IC:T}].\text{Avail} \land \sigma_2.[\text{IC:T}].\text{Avail}$
  $\land \sigma_1.[\text{IC:T}].\text{Value} \neq \sigma_2.[\text{IC:T}].\text{Value}$

- $\sigma'.[\text{TC:T}].\text{Avail} = \sigma_1.[\text{TC:T}].\text{Avail} \lor \sigma_2.[\text{TC:T}].\text{Avail}$

- $\sigma'.(\text{SC:T}).\text{Done} = \sigma_1.(\text{SC:T}).\text{Done} \lor \sigma_2.(\text{SC:T}).\text{Done}$
Partial Order on Program States

We define a partial order relation \( \leq \) on two non-error program states \( \sigma_1 \leq \sigma_2 \) as follows:

- \( \sigma_1 \leq \sigma_2 \) if and only if all the following conditions are true for all item/tag/step collections IC, TC, SC and tags T
  - \( \sigma_1 \neq \bot \land \sigma_2 \neq \bot \)
  - \( \sigma_1.[IC:T].Avail \implies \sigma_2.[IC:T].Avail \)
  - \( \sigma_1.[IC:T].Value \neq \bot \implies \sigma_2.[IC:T].Value = \sigma_1.[IC:T].Value \)
  - \( \sigma_1.<TC:T>.Avail \implies \sigma_2.<IC:T>.Avail \)
  - \( \sigma_1.(SC:T).Done \implies \sigma_2.(SC:T).Done \)

The following results follow directly from the definition of \( \leq \)

- If \( \sigma' \neq \bot \) then \( \sigma_0 \leq \sigma' \)
- If \( \sigma' = \text{Apply}(\sigma, (SC:T)) \land \sigma' \neq \bot \land \sigma \neq \bot \) then \( \sigma \leq \sigma' \)
- If \( \sigma' = \text{Merge}(\sigma_1, \sigma_2) \land \sigma' \neq \bot \land \sigma_1 \neq \bot \land \sigma_2 \neq \bot \) then \( \sigma_1 \leq \sigma' \land \sigma_2 \leq \sigma' \)

Thus, step execution and state merges lead to states that are *monotonically increasing* w.r.t. \( \leq \) (when no error state is encountered)
State Transitions in a CnC Program

• During execution, the CnC program will start with initial state \( \sigma_i \), go through a number of state transitions, and \textit{terminate} with final state \( \sigma_f \) when no step instance \((SC:T)\) remains with \((SC:T).\text{Enabled} = \text{true} \) and \((SC:T).\text{Done} = \text{false} \).
  - Final state \( \sigma_f \) is said to be \textit{incomplete} if it contains a step instance with \( \text{Prescribed} = \text{true} \).

• Sequential Abstract Interpreter (SAI) executes a single enabled step \((SC:T)\) at a time in isolation from other steps.
  - Output state is computed for \((SC:T)\) using the state update function, \( \sigma' = \text{Apply}(\sigma, (SC:T)) \) where \( \sigma \) is the input state (output state from previous step).

• Parallel Abstract Interpreter (PAI) executes enabled steps asynchronously in parallel.
  - For each step \((SC:T)\), the PAI first computes the input state, \( \sigma = \text{Merge}(\sigma_P, \sigma_1, \sigma_2, \ldots) \) as the merge of all “predecessor” states of \((SC:T)\).
    - \( \sigma_P \) = output state of step that prescribed \((SC:T)\).
    - \( \sigma_j \) = output state of step that produced item \([ICj:Tj] \epsilon \text{consumedBy}(SC:T)\).
  - The output state for \((SC:T)\) is then computed as \( \sigma' = \text{Apply}(\sigma, (SC:T)) \).
Properties of CnC Execution Semantics

• Determinism
  – For a given CnC program and input store $\sigma_i$, the final output store $\sigma_f$ will be the same for all scheduled sequences of steps in the sequential abstract interpreter

• Parallelism
  – For a given CnC program and input store $\sigma_i$, the final output store $\sigma_f$ will be the same for the sequential abstract interpreter and the parallel abstract interpreter

• Redundancy
  – Consider an extension to the single assignment rule in which multiple puts to the same item $[\text{IC:T}_a]$ are permitted with the same value, and multiple puts to the same tag $<\text{TC:T}_b>$ are also permitted
  – For a given CnC program and input store $\sigma_i$, a modified abstract interpreter with this extension in which any put to a tag $<\text{TC:T}>$ is replicated a bounded number of times will lead to the same final output store $\sigma_f$ as the sequential abstract interpreter
Possible Extensions to Execution Semantics

• Allow interleaving of Environmental Input (EI) and Environmental Output (EO) computations

• Allow step to start executing even if InputsAvailable = false
  – Step will need to block on get operations for which the items are not available
Outline

1. Concurrent Collections Model and Languages
2. Execution Semantics and Properties
3. Implementation Approaches
4. Patterns and Examples
5. Research Topics
Current implementations

• CnC is a programming model that can be embodied in multiple programming languages

1. Intel® Concurrent Collections for C++
2. Rice Concurrent Collections for Java
3. Rice Concurrent Collections for .NET (preliminary)
1. Intel® Concurrent Collections for C++

CnC Graph (Textual form) → CnC translator → C++ compiler → Intel TBB Library → Windows/Linux → Multicore IA → Harness & Sequential Steps in C++


*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners
Textual form of CnC 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];
CnC C++ Build Model

Concurrent Collections Textual Graph

Translator

C++ Header File

C++ Source File

Compiler

Object File

Linker

Concurrent Collections Library

Application

Includes code to invoke the graph and the code for steps

User specified

Concurrent Collections components
Migrating legacy code to CnC

CnC specification

\[
\text{<spanTag: } j,s> \quad::\quad (\text{processSpan: } j,s); \\
\text{[span:} j,s] \quad\rightarrow\quad (\text{processSpan: } j,s); \\
(\text{processSpan: } j,s) \rightarrow [\text{result: } j,s];
\]

Original scalar routine

```cpp
string processSpanOrig(string inStr) { ... //unchanged}
```

Wrapper code for CnC

```cpp
Step processSpan( Graph_t &g, Tag_t t) {
    string outStr = processSpanOrig(g.span.get(t));
    if (strlen(outStr) > 0) g.result.put(t);
}
```
Translator generates coding Hints

```c
StepReturnValue_t processSpan(
    partStr_graph_t& graph,
    const Tag_t& step_tag)
{
    // For each input item for this step retrieve the item using the proper tag
    // User code to create item tag here
    string ... = graph.span.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.results.Put(Tag_t(...), ...);

    if ((span.length())%2 != 0) { // odd length
        graph.results.Put(step_tag, span);
    }

    return CNC_Success;
}
```
Cholesky performance:

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

Theoretical peak GFlop/s

Performance (GFlop/s)

Matrix Size

DGEMM peak

Acknowledgments:
Aparna Chandramolishwaran,
Rich Vuduc
Georgia Tech
Eigensolver performance (dsygvx)
Intel Harpertown (2x4 = 8 core)

Acknowledgments:
Aparna Chandramolishwaran,
Rich Vuduc
Georgia Tech
2. Rice Concurrent Collections for Habanero-Java (HJ)

CnC Graph (Textual form)

CnC translator

HJ compiler
CnC library
Habanero Runtime
Java VM
Win/Lin/OSX/AIX...
Multicore Hardware

Harness & Sequential Steps in HJ

Key:
User supplied
CnC system
Standard
Habanero Multicore Software Research project (habanero.rice.edu)

Parallel Applications
(Seismic analysis, Medical imaging, Finite Element Methods, …)

Challenge: Develop new programming technologies and pedagogical foundations for portable parallelism on future multicore hardware

Scalable runtime system:
1) Work-stealing extensions for async-finish task parallelism
2) Locality control with places
3) Mutual exclusion with location-based isolation
4) Collective and point-to-point synchronization with phasers

Multicore Platforms

Two-level programming model
Implicitly Parallel Coordination Language for Joe,
CnC (Intel Concurrent Collections) +
Explicitly Parallel Programming Languages for Stephanie,
Habanero-Java (from X10 v1.5) and Habanero-C

Habanero Static Compiler & Parallel Intermediate Representation
Habanero Runtime & Dynamic Compiler

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners*
Overview of Habanero-Java Language for Steps

• Derived from X10 v1.5 (http://x10-lang.org/)

• Extension of sequential subset of Java 1.4 language with access to Java 5 and Java 6 libraries
  – New data types
    • complex --- primitive data type
    • point --- standard class for int tuples with special syntax e.g., [1,2]
  – New constructs for explicit parallelism
    • async, delayed async, future
    • finish
    • isolated
    • phasers
    • places
  – Domain expert only needs to use points for tags and finish in harness
  • Remaining parallel constructs are generated by CnC compiler
  • Tuning expert has the option of using parallel constructs within CnC steps if they choose
Async and Finish

**Stmt ::= async Stmt**

- Creates a new child activity that executes statement $S$
- Returns immediately
- $S$ may reference final variables in enclosing blocks
- Activities cannot be named
- Activity cannot be aborted or cancelled

**Stmt ::= finish Stmt**

- Execute $S$, but wait until all (transitively) spawned asyncs have terminated.
- Rooted exception model
  - Trap all exceptions thrown by spawned activities.
  - Throw an (aggregate) exception if any spawned async terminates abruptly.
- Implicit finish between start and end of main program
Delayed Async

**Stmt** ::= async when *(Cond)* **Stmt**

- Creates a new child activity for statement Stmt
- Returns immediately
- New activity can only start execution of Stmt when Cond becomes true

⇒ Activity does not get inflated into an active task before Cond becomes true

Example
- Consider the case when Stmt contains a blocking operation such as a CnC get operation e.g.,
  S1;
  C.get(...); // May block
  S2;

- “async Stmt” can be translated to a delayed async as follows:

```plaintext
async {
  S1;
  async when(C.containsTag(...)) S2;
}
```
## Mapping Concurrent Collections to Habanero-Java

<table>
<thead>
<tr>
<th>CnC Construct</th>
<th>Translation to Habanero-Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Point object</td>
</tr>
<tr>
<td>Prescription</td>
<td>Async or Delayed async</td>
</tr>
<tr>
<td>Item Collection</td>
<td>java.util.concurrent.ConcurrentHashMap</td>
</tr>
<tr>
<td>put() on Item Collection</td>
<td>Nonblocking put() on ConcurrentHashMap</td>
</tr>
<tr>
<td>get() on Item Collection</td>
<td>Blocking or nonblocking get on ConcurrentHashMap</td>
</tr>
</tbody>
</table>

### Error Checking in CnC-HJ

- A step can only access an input/output collection if there’s an appropriate declaration in the CnC graph
- An exception is thrown if two put operations are performed with the same tag on the same item/tag collection
CnC Habanero-Java Build Model

Concurrent Collections Textual Graph

Concurrent Collections

Habanero Java Source Files

Code to invoke the graph
Code to put initial values in graph
Code to implement abstract steps

Concurrent Collections Library

Habanero Java Runtime Library

JAR builder

Java application

User specified

Concurrent Collections components
Habanero-Java Source Code for CnC Steps

• Programmer provides one step class per CnC step collection
  – extends abstract class generated from CnC specification

• Step class must include a compute() method
  – specifies the code for the step

• Optionally, it can contain a ready() method
  – specifies a condition to delay execution of step until ready() returns true
  – used as condition for delayed async
    • ready() method can be used by tuning expert and CnC implementation to obtain more efficient implementation than regular async’s
    • ready() just returns true by default
3. Rice Concurrent Collections for .Net (In progress)

• Parser
  – Written in FSYacc (F# version of YACC)
  – Slightly modified grammar relative to Intel CnC for C++
    • New syntax for array types (e.g. `[int[]] A` for item collection containing int array)
    • Only one direction inputs (e.g. `[A] → (B), not (B) ← [A]`)

• Code Generation
  – Uses CodeDom for source code generation for multiple source targets
    • Targets F#, C#, VB.NET, managed C++ (bold languages have been tested)

• Runtime
  – Uses Task Parallel Library (TPL) for task creation/scheduling
  – ItemCollection is hand coded concurrent hash table with locking
  – Tags are F# tuples (.NET 4.0 will have proper tuples in CIL)
  – Tag collection Put immediately queues a new task with TPL

• Performance
  – Currently gets 3.3x speedup on Cholesky benchmark on 8-core machine

• Acknowledgments
  – David Peixotto (dmp@cs.rice.edu)
CnC Implementation Challenges

• **Scalable runtime implementation for multicore parallelism**

• **Garbage collection of dead items**
  – “Declarative Aspects of Memory Management in the Concurrent Collections Parallel Programming Model”, Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, Leo Treggiari
  – More details in following slides

• **Copy avoidance and update-in-place optimizations**

• **Scheduling optimizations for parallelism and locality**

• **Additional challenges discussed in Research Topics**
Garbage Collection of Dead Items

• Problem: Items in collections are accessed by tags --- how to tell when an item is dead?

• Current C/C++ version of CnC supports user supplied *getCount* function. User knows how many steps will get the item but not when they will execute.

• Proposed approach in DAMP 2009 paper
  – Add a declarative slicing annotation to CnC that identifies set of items that a step can read with constraints based on the step’s tag
  – Runtime computes the inverse of this map to determine when no future steps could possibly access the item
  – Research effort --- this extension has not been incorporated in CnC
Slicing Annotation

• Consider an item instance in collection \([C: T]\), and an instance step \((S: I)\).

• The annotation

\[(S: I) \subseteq \text{readers}([C: T]), constraints(I, T)\]

indicates that step instance \((S: I)\) may perform a get operation on item \([C: T]\) if \(\text{constraints}(I, T) = \text{true}\).
Example of Slicing Annotations for Cholesky

- \( Lijk \) is an item collection indexed by three dimensional tags, \(<t1, t2, t3>\)

- The constraint relates step tags and item tags for items that are read by a given step
Use of Slicing Annotations for Memory Management

• Transform slicing annotations into expressions for instance counts

• Transform instance counts into reference counts

• Free item when all known (possible) reader steps of item have completed
Memory Requirements for 2000x2000 Cholesky Factorization w/ and w/o Garbage Collection of Dead Items

Cholesky Factorization (N = 2000)

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.
CnC supports not only different schedules but a wide range of runtime *styles*

<table>
<thead>
<tr>
<th></th>
<th>grain</th>
<th>distribution</th>
<th>schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP TStreams</td>
<td>static</td>
<td>static</td>
<td>static</td>
</tr>
<tr>
<td>HP TStreams</td>
<td>static</td>
<td>dynamic</td>
<td>dynamic</td>
</tr>
<tr>
<td>Intel CnC</td>
<td>static</td>
<td>dynamic</td>
<td>dynamic</td>
</tr>
<tr>
<td>Rice CnC</td>
<td>dynamic</td>
<td>dynamic</td>
<td>dynamic</td>
</tr>
</tbody>
</table>
Outline

1. Concurrent Collections Model and Languages
2. Execution Semantics and Properties
3. Implementation Approaches
4. Patterns and Examples
5. Research Topics
Cholesky factorization

- Input matrix B of size nxn where n = p*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($L_{jk}$, $L_{ik}$, $B_{ij}$);
Cholesky: graphical form

< kTag: k>

(convChol: k)

< kjTag: k, j>

(Trisolv: k, j)

<kjiTag: k, j, i>

(update: k, j, i)
Line 2: Conventional Cholesky

$K = 1$
Line 4: Triangular Solve

<table>
<thead>
<tr>
<th>$K = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>$K = 1$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$K = 1$</td>
</tr>
</tbody>
</table>
Line 6: Symmetric Rank k Update

<table>
<thead>
<tr>
<th>K = 1</th>
<th>K = 1</th>
<th>K = 1</th>
<th>K = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>K = 1</td>
<td>K = 1</td>
<td>K = 1</td>
<td>K = 1</td>
</tr>
</tbody>
</table>
### Line 2: Conventional Cholesky

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K = 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Line 4: Triangular Solve

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>K = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>K = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>K = 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Line 6: Symmetric Rank k Update

<table>
<thead>
<tr>
<th></th>
<th>K = 2</th>
<th></th>
<th>K = 2</th>
<th></th>
<th>K = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>K = 2</td>
<td></td>
<td></td>
<td>K = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners*
CnC asynchronous execution

<table>
<thead>
<tr>
<th>K = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
CnC asynchronous execution

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>K = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## CnC asynchronous execution

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K = 1</td>
<td></td>
</tr>
<tr>
<td>K = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.*
CnC asynchronous execution: k=1 not done. k=2 starts.
Cholesky: Intel 2-socket x 4-core Harpertown @ 2 GHz + Intel MKL 10.1
CnC-based Cholesky timeline (n=1000): Intel 2-socket x 4-core Harpertown @ 2 GHz + Intel MKL 10.1 for sequential components
Eigensolver performance (dsygvx)
Intel Harpertown (2x4 = 8 core)
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.
Dedup performance

Dedup timing on Linux (8 cores)

Dedup scaling on Linux (8 cores)
Applications

- Body tracking
- Heat diffusion
- Face detection
- PIRO feature extraction
- Black-Scholes
- Dedup compression
- Cholesky
- Eigenvalue solver
- Matrix inversion
- Conjugate gradient
- Game Of Life
Eight basic relationships between two steps

- See tutorial on the web site organized around these relationships
- Shows a complete but very simple example for each one.
<table>
<thead>
<tr>
<th>Control</th>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td><img src="image1.png" alt="Diagram 1" /></td>
<td><img src="image2.png" alt="Diagram 2" /></td>
</tr>
<tr>
<td>no</td>
<td><img src="image3.png" alt="Diagram 3" /></td>
<td><img src="image4.png" alt="Diagram 4" /></td>
</tr>
<tr>
<td>Control</td>
<td>Data</td>
<td>yes</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>yes</td>
<td></td>
<td><img src="image1.png" alt="Diagram" /></td>
</tr>
<tr>
<td>no</td>
<td><img src="image3.png" alt="Diagram" /></td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Different step collections
No constraints

• Potential task parallelism
• No ordering constraints
Different step collections

Control dependence

• If

(S1: j) -> <t2: j>

(S1: 8) may or may not put <t2: 8>

• Nested loop

(S1: j) -> <t2: j, i>

(S1: 8) puts <t2: 8, 1>, <t2: 8, 2>, <t2: 8, 3>, …

• Arbitrary

(S1: 8) may put <t2: 45, 526, 87>
Different step collections
Data dependence

• Standard streaming paradigm (pipeline parallelism)

Assume the same prescribing tag collection
Different step collections
Both control and data dependence

• Substring span
Same step collection
No constraints

- Loop parallel / data parallel
- Instances of this step collection can execute in parallel
- No cycle that we have to analyze to determine that it can be computed in parallel
Same step collection
Control dependence

- While loop
- Data items used in the control cycle are produced outside of the control cycle
Same step collection
Data dependence

- Iterative loop
- Control tags used in the data cycle are produced outside of the data cycle
Same step collection
Control and data dependences

See next few slides
While loop

Initial item

Optionally output tags

Output items

<whileTag>

(whileBody)

[whileItem]
Divide and conquer

Root tag

<divideTag>

(divide)

[divideItem]  Leaf tags

Leaf items

<conquerTag>

(conquer)

[conquerItem]  Leaf tags

Leaf items

Root tag

Root item

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.
Outline

1. Concurrent Collections Model and Languages
2. Execution Semantics and Properties
3. Implementation Approaches
4. Patterns and Examples
5. Research Topics

Pop up a level and look at what we’ve done
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
  – Too specific wrt type of parallelism in the application and wrt the type target architecture

The solution

• Isolate roles
  – Easier for the domain expert
  – Maximum flexibility for tuning expert
Raise the level of the programming model just enough to avoid over-constraints

Concurrent Collections
(only semantically required constraints)

explicitly serial languages
(over-constrained)

explicitly parallel languages
(over-constrained)
Intel® Concurrent Collections

The application problem
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

Mapping to target platform

Goal:
**serious separation of concerns:**
The **domain expert** does not need to know about **parallelism**
The **tuning expert** does not need to know about the **domain**.

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.*
Outline

1. Concurrent Collections Model and Languages
2. Execution Semantics and Properties
3. Implementation Approaches
4. Patterns and Examples
5. Research Topics
A model not a language

• Don’t care about syntax
  – GUI, Textual representation, Class library

• Variety of different choices based on trade-offs among:
  – Efficiency
  – Ease-of-use
  – Generality
  – Guarantees
  – …

• The tag of a get can depend on the tag of the step. Can it also depend on contents of a previously gotten item?
• Do we check for single assignment or allow multiple puts as long as the contents are the same?
• Do we allow for continuous I/O?
• Do we require tags of puts to be static function of step tag?

Any suggestions for the names?
Possible projects and research efforts

- Above CnC: Applications & applications support
- Within CnC: Functionality
- Within CnC: Theory
- Below CnC: Runtime
- On the side of CnC: Tools
Above CnC:

- Applications
- Domain-specific variants (e.g., finance, graphics, database operations, medical imaging)
Possible efforts: Within CnC: Functionality

• Hierarchy

• Support for reuse
  – Bottom-up (libraries)
  – Top-down (frameworks)
  – Sideways (connect inputs and outputs of distinct graphs)

• Reductions

• Continuous applications

• Checking for compliance with the rules (e.g., single assignment, no side-effects)

• Incorporating controlled non-determinism

• Improved garbage collection (especially with continuous applications)

• Support for memory optimizations (e.g., in-place algorithms)
Future: Hierarchical step collections

[Input] → (createSpan) → [span: s] → (processSpan: s) → [results: s]
Future: Hierarchical step collections

- (createSpan) is prescribed by prescriber of parent
- j is not visible inside (only s is visible)
- Runtime manages j to distinguish among instances
- is a renaming

(input: j) \[\rightarrow\] (createSpan) \[\rightarrow\] (span: s) \[\rightarrow\] (processSpan: s) \[\rightarrow\] [results: s] \[\rightarrow\] [results: j, s]
Future: reductions

Semantics:
(reduceWholeCollection) step instance
gets whole item collection [input]

Execution:
As each instance of [input] arrives, it is
incorporated into the result. [result] is only
put when all instances of [input] are available

Based on hierarchy. A step at one level is doing a get at another.
Future: reductions

Semantics:
Separate reduction for each j

Execution:
As before, each instance is incorporated as it arrives

Issue: we need to know when all instances in the collection [input] are available
Possible efforts: Within CnC: theory

- Variety of languages
  - theoretical aspects of continuous programs
  - theoretical aspects of CnC with controlled non-determinism
- Universal CnC program
- A CnC interpreter in CnC
Name distinction vs tag distinction

- Merge collections
- Name distinction becomes tag distinction
- Same grain
- Same potential parallelism
- Same information

\[ \text{[x: } j] \quad \text{[y: } j] \quad \text{[z: } j] \]

\[ \text{<Tony: } j> \quad \text{(fooB: } j) \quad \text{(fooA: } j) \]

\[ \text{[x: } j] \quad \text{[y: } j] \quad \text{[z: } j] \]

\[ \text{<Tony: } j> \quad \text{(foo: AB, } j) \]

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.*
Name distinction vs tag distinction

- Merge collections
- Name distinction becomes tag distinction
- Same grain
- Same potential parallelism
- Lose information that (fooB) does not get [y]

![Diagram showing name distinction vs tag distinction]

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.*
Universal CnC specification

- End result of maximal merging of any program
- Can we input and execute any CnC program with this universal spec?
- What would we need to add?
Possible efforts: Below CnC: Runtime

• Additional targets

• Checkpoint / restart (existed at HP)

• Checkpoint / continue (new)

• Disk-based execution

• Scheduling / distribution
  – Speculative scheduler
  – Demand-driven scheduler
Possible efforts: Below CnC: Runtime

- Additional targets
- Checkpoint / restart (existed at HP)
- Checkpoint / continue (new)
- Disk-based execution
- Scheduling / distribution
- Speculative scheduler
- Demand-driven scheduler
Dynamic GPU/vector operations

Red step collection
Blue step collection

- All enabled steps
- Enabled Blue steps
- Enabled Red steps

Single bucket
Bucket per collection
ILP

Red step collection
Blue step collection

Compiled independently for ILP

Step fusion yields additional parallelism
Red-blue step collection

For ILP, compiled together assuming independence between red and blue
Pick

- an enabled blue step
- an enabled red step or
- one of each
Forward Execution

Execution frontier:
set of all object instances
with any attribute

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

Execution frontier: set of all object instances with any attribute
But not yet dead/done

- Red: prescribed
- Blue: inputs available
- Red&blue: enabled
- Yellow: executing
- Gray: done&dead

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners*
Checkpoint: save execution frontier

- As objects become part of the execution frontier, we can save them on disk.
- We maintain an independent execution frontier on disk.
Restart: from disk

- Assume the executing program dies.
- We can restart from the execution frontier on disk.
Disk-based execution

• Long-lived but low priority job

• Just disk part of the execution (no memory-based component)

• Maintains state

• When processor is available, speculate that there is time to complete a step. If not, drop the step.
Convert repetitive similar computation to long-lived job

Example:

• Daily check of international database for new SARS virus

• Execute $O(O(n^2))$ computation on all known strains to produce possible phylogenetic tree

• Repeats $O((n-1)^2)$ operations that were done the previous day

• Write a single, $O(n^2)$ version that is long-lived and never finishes but outputs today’s results.

• Tomorrow it reuses the old results and only computes the pairwise interactions with the new strain.
Future: Departmental CnC server

• Departmental => trusted resource sharing

• Optimize use of departmental resources among jobs.

• Hidden tags (introduced earlier for hierarchy) can be used to distinguish jobs.

• Alternately, department-wide tagging scheme allows use of results that others have computed without knowing that they are available.
Possible efforts: on the side of CnC: tools

- CnC debugger
- CnC performance tuner
- CnC visualizer
- CnC diff
The academic community

• Rice University
  • Vivek Sarkar
  • Zoran Budlimic
  • David Peixotto
  • Sagnak Tasirlar (also Intel intern)

• Georgia Tech
  • Rich Vuduc
  • Aparna Chandramowlishwaran (also Intel intern)
  • Kishore Ramachandran
  • Hasnain Mandviwala (intern + PhD thesis)

• Colorado State University
  • Michelle Strout
  • Jon Roelofs

• Novosibirsk State University
  • Nikolay Kurtov (intern)

• Washington University – St Louis
  • Kunal Agrawal (MIT)
The Intel community

• DPD
  Shin Lee
  Steve Rose
  Leo Treggiari
  Ilya Cherny
  Frank Schlimbach
  Ganesh Rao
  Nikolay Kurtov

• SPI
  Kath Knobe
  Geoff Lowney
  Mark Hampton

• Others
  Steve Lang
  John Pieper

The HP community

• Cambridge Research Lab
  Kath Knobe
  Carl Offner
  Alex Nelson
Papers

• **Capsules: Expressing Composable Computations in a Parallel Programming Model.**
  Hasnain A. Mandviwala, Umakishore Ramachandran, and Kathleen Knobe. LCPC’07

• **TStreams: A Model of Parallel Computation (Preliminary Report).**
  Knobe, Kathleen; Offner, Carl D. HPL-2004-78R1

• **Weak Dynamic Single Assignment Form.**
  Carl Offner, Kathleen Knobe, HPL-2003-169(R.1)August 1, 2005*

• **Compiling to TStreams, a New Model of Parallel Computation.**
  Knobe, Kathleen; Offner, Carl HPL-2005-138

• **TStreams: How to Write a Parallel Program**
  Knobe, Kathleen; Offner, Carl D. HPL-2004-193

• **Multi-core Implementations of the Concurrent Collections Programming Model.**
  Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. CPC’09.

• **Declarative aspects of memory management in the concurrent collections parallel programming model.**
  Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari.
  The 4th workshop on Declarative aspects of multicore programming. Savannah, GA, USA. 2009

• **Ease of use with Concurrent Collections (CnC).**
  Kathleen Knobe. First USENIX Workshop on Hot Topics in Parallelism (HotPar ‘09)
Sites

- **Intel Windows version:**
  
  whatif.intel.com
  

- **Intel Linux version:**
  
  contact kath.knobe@intel.com

- **Rice web site**
  
  [http://habanero.rice.edu/cnc](http://habanero.rice.edu/cnc)
CnC Workshop

July 23-24

Hudson, MA

Day 1 tutorial and hands-on workshop for newcomers
Day 2 For those working in CnC at the application level and those doing CnC related research

Contact: kath.knobe@intel.com
Tutorial Acknowledgments

• Zoran Budlimic
• Aparna Chandramowlishwaran
• Shin Lee
• Geoff Lowney
• David Peixotto
• Leo Treggiari
• Rich Vuduc