The Concurrent Collections (CnC) Parallel Programming Model

Kathleen Knobe
Intel

kath.knobe@intel.com
Cholesky Performance

Aparna Chandramowlishwaran
Rich Vuduc
(Georgia Tech)

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

~2x speedup over MKL

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

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.
Why is Concurrent Collections interesting for Autotuners?

• Goal of autotuning: find best options within the constraints of the application

• Goal of Concurrent Collections: Separation of concerns
  – domain-expert: semantic requirements only
  – tuning-expert: maximal flexibility (minimizes constraints) for tuning

• What is a tuning-expert?
  – The same person as the domain-expert at a different time
  – A different person
  – A naïve runtime
  – A sophisticated static analyzer
  – An autotuner

Concurrent Collections minimizes the constraints for the autotuner
Intel® Concurrent Collections

The application problem
The work of the **domain expert**
- Semantic correctness
- Constraints required by the application

The work of the **tuning expert**
- Architecture
- Actual parallelism
- Locality
- Overhead
- Load balancing
- Distribution among processors
- Scheduling within a processor

Concurrent Collections Spec

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.
The problem

- Serial languages over-constrain orderings
  - Require arbitrary serialization
  - Allow for overwriting of data
  - The decision of *if* and *when* to execute are bound together

- Parallel programming languages embedded within serial languages
  - Inherit problems of serial languages
  - Too specific wrt type of parallelism in the application type target architecture
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)
Outline

• Language concepts

• Runtime
  – Abstract execution model
  – Actual systems

• 2 examples of flexibility
  – Checkpointing
  – Interesting target architectures
Outline

• Language concepts

• Runtime

• Examples of flexibility
The Big Idea

- Don’t specify what operations run in parallel difficult and depends on target

- Specify only the semantic ordering requirements 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
Notation

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

White Board | Textual | Slideware

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners.*
Exactly two sources of ordering requirements

- **Producer / Consumer (Data Dependence)**
  Producer must execute before consumer

- **Controller / Controllee (Control Dependence)**
  Controller must execute before controllee
Tag collections: new concept

Tags are:
- Generalization of iteration space
- Typically related to the semantics of application.

Loop nests

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

components of tag \(<t2>\) are k and j

- Loop iteration space is a sequence \(<1,1,1>, <1,1,2>, \ldots\)
- 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 \(
\{<1,1>, <1,2>, \ldots\}\)
- The step code has access to its tags
- The time the associated body is executed is yet to be determined.
Tag collections: new concept

Tag collections are like iteration spaces but more general

• Not just loop indices
  – Trees (e.g., recursion)
  – Graphs (e.g., irregular mesh)
  – Sets (employees, molecules, frames of images)

• Not sequential
  – But rather unordered

• Named
  – To facilitate reuse
Collections of dynamic instances

Static

Dynamic

\(\text{(s: 1)}\) \(\text{(s: 2)}\) \(\text{(s: 3)}\) \(\text{(s: 4)}\)

\(\text{(q: 1)}\) \(\text{(q: 2)}\) \(\text{(q: 3)}\) \(\text{(q: 4)}\)
Collections of dynamic instances

Static

(s1) → [i] → (s2)

Dynamic

(s1: 5) → (s1: 12) → (s1: 3) → [i: 6] → [i: 13] → [i: 7] → [i: 4] → [i: 5] → (s2: 7) → (s2: 4) → (s2: 13)
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
"aaaffqqqmmmmmmmm"

Sequences of repeated characters
"aaa"
"ff"
"qqq"
"mmmmmmmm"

Filtered sequences
"aaa"
"qqq"
"mmmmmmmm"
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] = “aaaffqqqmmmmmmmm”

[span] = “aaa”

[span] = “ff”

[span] = “qqq”

[span] = “mmmmmmmm”

[results] = “aaa”

[results] = “qqq”

[results] = “mmmmmmmm”

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

```
<stringTag: j,s> :: (processSpan: j,s);
[span:j,s] -> (processSpan: j,s);
(processSpan: j,s) -> [result: j,s];
```

**Original scalar routine**

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

**Glue code**

```cpp
Step processSpan(Graph_t &g, Tag_t t) {
    string outStr = processSpanOrig(g.span.get(t));
    if (strlen(outStr) > 0) g.result.put(t);
}
```
Make it precise enough to execute

How do we distinguish among the instances?

[Input 1] = “aaaffqqqmmmmmmm”
[Input 2] = “rrhhhhxxx” …

<StringTag: 1>
<StringTag: 2>

...
No thinking about parallelism
Only domain/application knowledge

Result is:
• Parallel
• Deterministic (wrt results)
• Race-free
Experience with graph design questions
Outline

- Language concepts
- Runtime
- Examples of flexibility
Semantic model: instances acquire attributes

When this tag is *available*

When these items are *available*

This step will execute (sometime)

When this step executes

this step becomes *enabled*

[<T: 3, 28>]

[<T: 3, 29>]

and this tag is *available*

This tag becomes *available*
Execution model

- Red: prescribed
- Blue: inputs available
- Red&blue: enabled
- Yellow: executing
- Gray: done&dead
CnC supports not only different runtimes but a wide range of runtime *styles*

<table>
<thead>
<tr>
<th></th>
<th>memory</th>
<th>grain</th>
<th>distribution</th>
<th>schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP TStreams</td>
<td>distributed</td>
<td>static</td>
<td>static</td>
<td>static</td>
</tr>
<tr>
<td>HP TStreams</td>
<td>distributed</td>
<td>static</td>
<td>static</td>
<td>dynamic</td>
</tr>
<tr>
<td>Intel CnC</td>
<td>shared</td>
<td>static</td>
<td>dynamic</td>
<td>dynamic *</td>
</tr>
<tr>
<td>Rice CnC</td>
<td>shared</td>
<td>static</td>
<td>dynamic</td>
<td>dynamic</td>
</tr>
<tr>
<td>GaTech CnC</td>
<td>shared</td>
<td>dynamic</td>
<td>dynamic</td>
<td>dynamic</td>
</tr>
</tbody>
</table>

* soon: the ability to plug in an application-specific or a domain-specific *scheduler*
Outline

• Language concepts
• Runtime
• Examples of flexibility
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.
Checkpoint/continue

- Checkpoint/restart implies full stop on failure
- Checkpoint/continue would continue around the failed component
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

Step fusion yields additional parallelism
Red-blue step collection

Compiled independently for ILP

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

Pick
• an enabled blue step
• an enabled red step or
• one of each

Enabled Blue steps

Enabled Red steps
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
  - Ryan Newton

- **Others**
  - Steve Lang
  - John Pieper

The HP community

- Carl Offner
- Alex Nelson

The academic community

- **Rice University**
  - Vivek Sarkar
  - Zoran Budimlic
  - Sagnak Tasirlar
  - David Peixotto

- **Georgia Tech**
  - Rich Vuduc
  - Aparna Chandramowlishwaran

- **Colorado State**
  - Michelle Strout

- **Novosibirsk**
  - Nikolay Kurtov

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