Concurrent Collections (CnC)

A programming model for parallel programming
Ease of Use with Concurrent Collections (CnC)

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 a new approach based on this perspective. In particular, a CnC program does not explicitly express the parallelism. It expresses the concurrency as partial substitution. These expressions result in valid programs no matter what the target architecture can handle, both the language's effectiveness and its ease of use. In addition, these expressions make sense arbitrary constraints such as barriers after each loop or simple program-parallel-time (SPPM). Although this is not the focus of this paper, note that these assumptions can also inhibit performance.

1. Why is parallel programming hard?

Many parallel languages enable parallelism within the text of the actual code. Examples include MPJ, OpenMP, Pthreads, C 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 may be specified.
- Serial code modifies and reads values in variables (variables are not values). Variables can be overwritten. Thus, sometimes, w_{i} requires the possible assignments.
- Serial code typically compares the question of what the value of variable would be when we assign it. Assigning at some point in the current flow indicates that, yes, we will assign the code and set the values to assign that name. This is true for multiple assumptions with new variables or other interactions. These also constitute arbitrary ordering.

Converting serial code to parallel code involves uncoercing statements valid exceptions 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 programs into the set of this problem may require uncoercing statements valid exceptions. This is difficult to get right in the first place and to modify later. In addition, of course, the parallelism constructs might be f1a, constrained classes or architectures (say, shared memory) or f2a, on a limited type of parallelism (say, data parallelism). For the former, uncoercing statements valid exceptions is not hard to find an ordering but can be complicated for a program on a complex to write the ordering.

The essence of parallel execution

What does a system need in order to execute a program in parallel? We are not yet asking how to specify the promise, how to optimize for any specific targets etc. We are just asking: What are the trade-offs in this decision?

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

- Data dependencies (producer-consumer relations). A producer computes produce data consumed by another. Data is explicitly produced by a producer computation and explicitly consumed by (possibly multiple) consumer computations.
- Control dependencies (unwritable control relations). One computation depends on another if it will ensure. To determine the right calling of the f and when control flows, control flow will be explicitly produced by a controller computation and will control the execution of a controller computation. This can be the control and data dependencies on the same level as in co-routine forms such as program dependence graphs (t).

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 dependencies among these high-level operations.
- The control flows that participate in control dependencies among these high-level operations.
Concurrent Collections

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)
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

<table>
<thead>
<tr>
<th>Element</th>
<th>CnC name</th>
<th>Graphical form</th>
<th>Textual form</th>
</tr>
</thead>
<tbody>
<tr>
<td>computation</td>
<td>step</td>
<td><img src="image" alt="foo" /></td>
<td>(foo)</td>
</tr>
<tr>
<td>data</td>
<td>item</td>
<td><img src="image" alt="x" /></td>
<td>[x]</td>
</tr>
<tr>
<td>control</td>
<td>tag</td>
<td><img src="image" alt="T" /></td>
<td>&lt;T&gt;</td>
</tr>
<tr>
<td>environment</td>
<td></td>
<td></td>
<td>env</td>
</tr>
</tbody>
</table>
Simple Example

Produce odd length sequences of consecutive identical characters

```
“aaaffqqqmmmmmmm”
“aaa”
“ff”
“qqq”
“mmmmmmmm”
```

```
“aaa”
“qqq”
“mmmmmmmm”
```
Relations

prescriptive

input -> createSpan -> span -> processSpan -> results

producer -> createSpan

consumer -> processSpan

string Tags

Span Tags
Item Collections

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

![Diagram showing the process of creating and processing spans with instance tags.](image-url)
Tag Collections

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

```
stringTags

<1>

stringTags

<1,1>
<1,2><1,3>
<1,4>

input createSpan span processSpan results
```
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

\[ \text{span} \rightarrow \text{[“qqq” : 1,3]} \rightarrow \text{(processSpan : 1,3)} \]
When \((S : t1)\) executes, if it produces \([I, t2]\), then \([I, t2]\) becomes *available*.

When \((S : t1)\) executes, if it produces \(<T : t2>\), then \(<T, t2>\) becomes *available*.

If \(<T>\) prescribes \((S)\), when \(<T : t>\) is available then \((S : t)\) becomes *prescribed*.

If \(\forall \{I, t1\} \text{ such that } (S : t2) \text{ gets } [I, t1] \)
\([I,t1]\) is *available*  // if all inputs of \((S : t2)\) are available
then \((S : t2)\) 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

Concurrent Collections

Dennis Kafura – CS5204 – Operating Systems

input

[“aaaffqqqmmmmmmmm” : 1]
[“bbbxxxxxxffxxxxxyy” : 2]

span

[“aaa” :1,1]
[“ff” : 1,2]
[“qqq” : 1,3]
[“mmmmmmmm” : 1,4]
[“bbb” : 2,1]
[“xxxxxx” : 2, 2]
[“ff” : 2,3]

executing

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

```
<stringTags: int stringID>;  
<spanTags:   int stringID, int spanID>;  
[input:      int stringID];  
[span:       int stringID, int spanID];  
[results:    int stringID, int spanID];  

eqv -> [input], <stringTags>;  
[results], <spanTags> -> eqv;
```
<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];
Concurrent Collections

Mechanics

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

Key:
- User supplied
- CnC system
- Standard

- Concurrent Collections Textual Graph
- C++ Source File
- Compiler
- Object File
- Linker
- Application

- Includes code to invoke the graph and the code for steps

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

```c++
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

<Image: image#>

<T1: image#, window#>

C1

<T2: image#, window#>

C2

...maybe more: classifiers...

<Tn: image#, window#>

Cn

<Face: image#, window#>
### Patterns – steps in different collections

<table>
<thead>
<tr>
<th>I: Distinct collections</th>
<th>A: Producer/Consumer</th>
<th>B: No Producer/Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Controller/Controllee</td>
<td><img src="#" alt="Diagram A1" /></td>
<td><img src="#" alt="Diagram B1" /></td>
</tr>
<tr>
<td>2: No controller/Controllee</td>
<td><img src="#" alt="Diagram A2" /></td>
<td><img src="#" alt="Diagram B2" /></td>
</tr>
</tbody>
</table>
## Patterns – steps in same collection

<table>
<thead>
<tr>
<th>II:</th>
<th>A:</th>
<th>B: No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Same collection</strong></td>
<td>Producer/Consumer</td>
<td>Producer/Consumer</td>
</tr>
<tr>
<td>1:</td>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td>Controller/Controller</td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
<td><img src="image5" alt="Diagram" /></td>
</tr>
<tr>
<td>2: No Controller/Controller</td>
<td><img src="image6" alt="Diagram" /></td>
<td><img src="image7" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image8" alt="Diagram" /></td>
<td><img src="image9" alt="Diagram" /></td>
<td><img src="image10" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Performance

Cholesky performance:

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

Acknowledgements: Aparna Chandramolishwaran, Rich Vuduc (Georgia Tech)
Performance

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

Eigen solver performance (dsygvx)
Intel Harpertown (2x4 = 8 core)

Acknowledgements: Aparna Chandramolishwaran, Rich Vuduc (Georgia Tech)
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)

\[
\text{getS2A: } (s2 : k, j) \subseteq \text{readers}([L_{ij}k : t1, t2, t3]) \quad \text{,} \quad t2 = t3 \land t3 = k \land t1 = j \\
\text{getS2B: } (s2 : k, j) \subseteq \text{readers}([L_{ij}k : t1, t2, t3]) \quad \text{,} \quad t1 = t2 \land t2 = k \land t3 = k + 1
\]
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
Concurrent Collections

Performance

- 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