Lecture 26: Dataflow Programming with Intel Concurrent Collections

Instructor: Vivek Sarkar
Graduate TAs: Sanjay Chatterjee, Raghavan Raman

Acknowledgment: This lecture handout has been largely extracted from [2].

1 Introduction

The Concurrent Collections (CnC) model[1] was developed to address the need for making parallel programming accessible to *domain experts* or *non-professional programmers*. One approach that has historically addressed this problem is the creation of *domain specific languages* (DSLs), such as Matlab, R, SQL, and Google's MapReduce framework that hide the details of parallelism when programming for a specific application domain. In contrast, CnC is a model for adding parallelism to a sequential language using primitives that should be accessible to people who are not Computer Science majors.

The basic premise of CnC is that domain experts can identify the intrinsic *data dependences* and *control dependences* in an application, without worrying about what parallel programming constructs should be used to satisfy those dependences. The dependences are specified in a CnC *graph* for an application. Parallelism is implicit in a CnC graph. A CnC graph has a deterministic semantics, in that all executions are guaranteed to produce the same output state for the same input.

2 Description

The three main constructs in the CnC programming model are *step collections*, *data collections*, and *control collections*. A step collection corresponds to a computation, and its instances correspond to dynamic invocations of that computation that consume and produce data items (analogous to instances of *async* tasks). A data collection corresponds to a set of *data items*, indexed by *item tags* (analogous to *keys* in key-value pairs), that can be accessed via *put* and *get* operations (analogous to *data driven futures*). Once put, data items cannot be overwritten, they are required to be *immutable*. A control collection corresponds to a *factory* [3] for step instances. A put operation on a control collection with a *control tag* results in the *prescription* (creation) of step instances from one or more step collections with the control tag passed as an input argument. In most CnC implementations, item tags and control tags may be of any data type that supports an equality test.

These collections and their relationships are defined statically as a CnC *graph* in which a node corresponds to a step, data or item collection, and a directed edge corresponds to a put, get, or prescribe operation.

CnC graph The three main constructs in a CnC graph are *step collections*, *data collections*, and *control collections*. These collections and their relationships are defined statically. But for each static collection, a set of dynamic *instances* is created as the program executes.

A step collection corresponds to a specific computation, and its instances correspond to invocations of that computation with different input arguments. A control collection is said to *control* a step collection—adding an instance to the control collection *prescribes* one or more step instances i.e., causes the step instances to eventually execute when their inputs become available. The invoked step may continue execution by adding instances to other control collections, and so on.

[1] CnC is a dataflow model, and is not in any way related to libraries such as Java's Concurrent Collections.

Steps also dynamically read (`get`) and write (`put`) data instances. The execution order of step instances is constrained only by their producer and consumer relationships, including control relations. A complete CnC specification is a graph where the nodes can be either step, data, or control collections, and the edges represent *producer*, *consumer* and *control* relationships.

A CnC program includes the specification, the step code and the environment. Step code implements the computations within individual graph nodes, whereas the *environment* is the external user code that invokes and interacts with the CnC graph while it executes. The environment can produce data and control instances. It can consume data instances and use control instances to prescribe conditional execution.

Typically, control tags have a specific meaning within the application. For example, they may be tuples of integers modeling an iteration space (*i.e.,* the iterations of a nested loop structure). Control tags can also be points in non-grid spaces—nodes in a tree, in an irregular mesh, elements of a set, etc. Collections use tags as follows:

- A step begins execution with one input argument—the control tag indexing that step instance. The control tag argument contains the information necessary to compute the item tags for all the step's input and output data. For example, in a one-dimensional stencil computation the control tag "`i`" could be used to access data with item tags, "`i+1`" and "`i-1`" . In a CnC specification, () parentheses are used to denote a step collection *e.g.,* `(foo)`.
- Putting a control tag into a control collection will cause the corresponding steps (in all controlled step collections) to eventually execute when their inputs become available. A control collection `C` is denoted with `<>` angle brackets *e.g.,* `<C>`.
- A data collection is an associative container indexed by item tags. The contents indexed by an item tag `i`, once written, cannot be overwritten (dynamic single assignment). The immutability of entries within a data collection, along with other features, provides determinism. In a specification file a data collection is referred to with square-bracket syntax: `[x:i,j]`.

Using the above syntax, together with `::` and → for denoting prescription and production/consumption relations, we can write CnC specifications that describe CnC graphs. For example, below is an example snippet of a CnC specification showing all of the syntax.

```
// control relationship: myCtrl prescribes instances of myStep
<myCtrl> :: (myStep);
// myStep gets items from myData, and puts control tags in myCtrl and items in myData
[myData] → (myStep) → <myCtrl>, [myData];
```

Further, in addition to describing the graph structure, we might choose to use the CnC specification to document relationships between control and item tags:

```
[myData: i] → (myStep: i) → <myCtrl: i+1>, [myData: i+1];
```

3 Example

The following simple example in Figure 1 illustrates the task and data parallel capabilities of CnC. This application takes a set of strings as input. Each string is split into words (separated by spaces). Each word then passes through a second phase of processing that, in this case, puts it in uppercase form.

The only keyword in the CnC specification language is `env`, which refers to the *environment*—the world outside CnC, for example, other threads or processes written in a serial language. The strings passed into CnC from the environment are placed into `[inputs]` using any unique identifier as a tag. The elements of `[inputs]` may be provided in any order or in parallel. Each string, when split, produces an arbitrary

```
// program inputs and outputs
env -> [inputs];
env -> <stringTags>;
[results] -> env;
// control relations (prescriptions)
<stringTags> :: (splitString);
<wordTags> :: (uppercase);
// producer/consumer relations
[inputs] -> (splitString) -> <wordTags>, [words];
[words] -> (uppercase) -> [results];
```

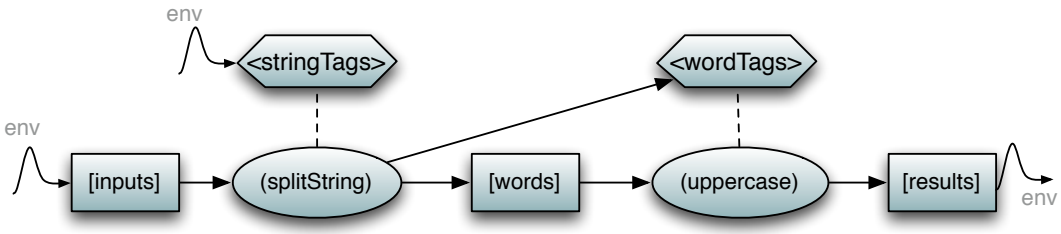

Figure 1: A CnC graph as described by a CnC specification. Dotted edges represent prescription (control/step relations), and arrows represent production and consumption of data. Squiggly edges represent communication with the environment (the program outside of CnC)

.

number of words. These per-string outputs can be numbered 1 through N—a pair containing this number and the original string ID serves as a globally unique item tag for all output words.

```
1  CnCReturnValue splitString(String tag, InputCollection inputs,
2    TagCollection wordTags, OutputCollection words) {
3          // Get input string
4          final String in = (String) inputs.Get(tag);
5          if (in.length() != 0) {
6              // construct words
7              char ch = in.charAt(0);
8              int len = 0; int i = 0; int j = 0;
9              while (i < in.length()) {
10                 if (in.charAt(i) == ch) {
11                     i++; len++;
12                 } else {
13                     words.Put(j+"", in.substring(j, j+len));
14                     wordTags.Put(j + "");
15                     ch = in.charAt(i); len = 0; j = i;
16                 }
17             }
18             // Put the last entry in words
19             words.Put(j+"", in.substring(j, j+len));
20             wordTags.Put(j + "");
21         }
22         return CnCReturnValue.Success;
23     }
24 }
```

Listing 1: HJ code to generate Computation Graph $G3$ from Homework 2 using DDFs

Listing 1 contains Java code to implement step `splitString`. (Intel's Concurrent Collection release only supports C++ code [4], but a variant has been developed at Rice University that supports Java code in CnC steps [1].) The step implementations, specification file, and code for the environment together make up a complete CnC application. Current implementations of CnC vary as to whether the specification file is required, can be constructed graphically, or can be conveyed in the host language code itself through an API.

4 Mapping to Target Platforms

There is wide latitude in mapping CnC to different platforms. For each parallel computing platform, there are several issues to be addressed: grain size, mapping data instances to memory locations, steps to processing elements, and scheduling steps within a processing element. A number of distinct implementations are possible for both distributed and shared memory parallel systems, including static, dynamic, or a hybrid of static/dynamic systems with respect to the above choices.

Implementations of CnC typically provide a translator and a run-time system. The translator uses a CnC specification to generate code for a run-time system API in the target language. As of the writing of this article, there are known CnC implementations for C++ (based on Intel's Threading Building Blocks), Java (based on Java Concurrency Utilities), .NET (based on .NET Task Parallel Library), and Haskell.

Step Execution and Data Puts and Gets: There is much leeway in CnC implementation, but in all implementations, step prescription involves creation of an internal data structure representing the step to be executed. Since parallel steps can be spawned before their inputs are available, Data Driven Futures (DDFs) [5] are a perfect vehicle for implementing CnC steps. All inputs of a CnC step are then enumerated in the `await` clause of the DDF used to implement a CnC step.

Initialization and Shutdown: All implementations require some code for initializing the CnC graph in a `finish` scope. The initialization creates the necessary runtime data structures for a CnC graph, and performs the initial `puts` into the data and control collections that are to be performed by the environment.

Safety properties: In addition to the differences between step implementation languages, different CnC implementations enforce the CnC graph properties differently. All implementations perform run-time system checks of the single assignment rule, while the Java and .NET implementations also enforce tag immutability. Finally, CnC guarantees determinism as long as steps are themselves deterministic—a contract strictly enforceable only in Haskell (among the languages for which CnC implementations are available).

Memory reuse: Another aspect of CnC run-time systems is garbage collection. Unless the run-time system at some point deletes the items that were put, the memory usage will continue to increase. This is because individual items can be accessed by application-specific tags such as strings and integers, and, without additional information, a garbage collector will not know for a sure that a specific item tag will never be used in the future. (DDFs do not have this problem since they are only accessed by opaque object references, rather than tags.)

One approach to assist with garbage collection of CnC items is *use counts*. In this approach, the user specifies the number of `get()` operations expected on an item as part of a `put()` operation. This information enables the CnC runtime system to free the item after the last `get()` operation is performed. However, there are some potential hazards with this approach if the user provides an incorrect count. If the count is too large, the item will never be freed. If it is too small, a `get()` operation may result in an exception if the item is freed prematurely.

References

[1] Habanero Concurrent Collections project. URL `http://habanero.rice.edu/cnc`.

[2] Michael G. Burke, Kathleen Knobe, Ryan Newton, and Vivek Sarkar. The Concurrent Collections Programming Model. Technical Report 10-12, Department of Computer Science, Rice University, December 2010.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] Intel. Intel (R) Concurrent Collections for C/C++. http://softwarecommunity.intel.com/articles/eng/3862.htm.

[5] Sagnak Tasirlar. Scheduling macro-dataflow programs on task-parallel runtime systems. M.S. Thesis, Department of Computer Science, Rice University, May 2011 (expected).