The Concurrent Collections Programming Model

Michael G. Burke
Rice University
Houston, Texas

Kathleen Knobe
Intel Corporation
Hudson, Massachusetts

Ryan Newton
Intel Corporation
Hudson, Massachusetts

Vivek Sarkar
Rice University
Houston, Texas

CnC, TStreams

Concurrent Collections (CnC) is a parallel programming model, with an execution semantics that is influenced by dynamic dataflow, stream-processing, and tuple spaces. 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 invocations of that computation that consume and produce data items. A data collection corresponds to a set of data items, indexed by item tags, that can be accessed via put and get operations; once put, data items cannot be overwritten and
DISCUSSION

Introduction  Parallel computing has become firmly established since the 1980’s as the primary means of achieving high performance from supercomputers. Concurrent Collections (CnC) was developed to address the need for making parallel programming accessible to non-professional programmers. One approach that has historically addressed this problem is the creation of domain specific languages (DSLs) that hide the details of parallelism when programming for a specific application domain.

In contrast, CnC is a model for adding parallelism to any host language (which is typically serial and may be a DSL). In this approach, the parallel implementation details of the application are hidden from the domain expert, but are instead addressed separately by users (and tools) that serve the role of tuning experts. The basic concepts of CnC are widely applicable. Its premise is that domain experts can identify the intrinsic data dependences and control dependences in their application (irrespective of lower-level implementation choices). This identification of dependences is performed by specifying a CnC graph for their 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. This deterministic semantics and the separation of concerns between the domain and tuning experts are the primary characteristics that differentiate CnC from other parallel programming models.

CnC specification graph  The three main constructs in a CnC specification 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.

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 whole 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.

Collections indexed by tags  Within each collection, control, data, and step instances are each identified by a unique tag. In most CnC implementations, tags may be of any data type that supports an equality test and hash function. Typically, 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). 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 tag indexing that step instance. The tag argument contains the information necessary to compute the tags of all the step's input and output data. For example, in a stencil computation a tag "i, j" would be used to access data at positions "i+1, j+1", "i-1, j-1" and so on. In a CnC specification file a step collection is written (foo) and the components of its tag indices can optionally be documented, as in (foo: row, col).

- Putting a 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 as $<C>$ or as $<C:x,y,z>$, where $x$, $y$, $z$ comprise the tag. Instances of a control collection contain no information except their tag, so the word “tag” is often used synonymously with “control instance”.

- A data collection is an associative container indexed by tags. The contents indexed by a 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 $\rightarrow$ 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 step
<myCtrl> :: (myStep);
// myStep gets items from myData, and puts 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 the relationship between tag indices:

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

**Model execution** During execution, the state of a CnC program is defined by attributes of step, data, and control instances. (These attributes are not directly visible to the CnC programmer.) Data instances and control instances each have an attribute $Avail$, which has the value $true$ if and only if a put operation has been performed on it. A data instance also has a $Value$ attribute representing the value assigned to it where $Avail$ is true. When the set of all data instances to be consumed by a step instance and the control instance that prescribes a step instance have $Avail$ attribute value $true$, then the value of the step instance attribute $Enabled$ is set to $true$. A step instance has an attribute $Done$, which has the value $true$ if and only if all of its put operations have been performed.

Instances acquire attribute values monotonically during execution. For example, once an attribute assumes the value $true$, it remains $true$ unless an
execution error occurs, in which case all attribute values become undefined. Once the Value attribute of a data instance has been set to a value through a put operation, assigning it a subsequent value through another put operation produces an execution error, by the single assignment rule. The monotonic assumption of attribute values simplifies program understanding, formulating and understanding the program semantics, and is necessary for deterministic execution.

Given a complete CnC specification, the tuning expert maps the specification to a specific target architecture, creating an efficient schedule. This is quite different from the more common approach of embedding parallel constructs within serial code. Tag functions provide a tuning expert with additional information needed to map the application to a parallel architecture, and for static analysis they provide information needed to optimize distribution and scheduling of the application.

**Example** The following simple example illustrates the task and data parallel capabilities of CnC. This application takes a set (or stream) 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.

```plaintext
<stringTags> :: (splitString);  // step 1
<wordTags>   :: (uppercaseWord); // step 2
// The environment produces initial inputs and retrieves results:
env → <stringTags>, [inputs];
env ← [results];
// Here are the producer/consumer relations for both steps:
[inputs] → (splitString) → <wordTags>, [words];
[words]   → (uppercaseWord) → [results];
```

The above text corresponds directly to the graph in Figure 1. Note that separate strings in [input] can be processed independently (data parallelism), and, further, the (splitString) and (uppercase) steps may operate simultaneously (task parallelism).

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 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 tag for all output words. Thus, in the specification we could annotate the collections with tag components indicating the pair structure of word tags: e.g. $(\text{uppercaseWord: stringID, wordNum})$.

Figure 2 contains C++ code implementing the steps \texttt{splitString} and \texttt{uppercase}. 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.

**Mapping to target platforms** There is wide latitude in mapping CnC to different platforms. For each 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 target platforms, including static, dynamic, or a hybrid of static/dynamic systems with respect to the above choices.

The way an application is mapped will determine its execution time, memory, power, latency, and bandwidth utilization on a target platform. The mappings are specified as part of the static translation and compilation as well as dynamic scheduling of a CnC program. In dynamic run-time systems,
// The execute method 'fires' when a tag is available.
// The context c represents the CnC graph containing item and tag collections.
int splitString::execute(const int & t, partStr_context & c) const
{
    // Get input string
    string in;
    c.input.get(t, in);

    // Use C++ standard template library to extract words:
    istringstream iss(in);
    vector<string> words;
    copy(istream_iterator<string>(iss),
         istream_iterator<string>(),
         back_inserter<vector<string> >(words));

    // Finally, put words into an output item collection:
    for(int i=0; i < words.size(); i++) {
        pair<int, int> wtag(t, i);
        c.wordTags.put(wtag);
        c.words.put(wtag, words[i]);
    }
    return CnC::CNC_Success;
}

// Convert word to upper case form:
int uppercaseWord::execute(const pair<int, int> & t, partStr_context & c) const
{
    string word;
    c.words.get(t, word);
    strUpper(word);
    c.results.put(t, word);
    return CnC::CNC_Success;
}

int main()
{
    partStr_context c;
    for(...) {
        c.input.put(t, string); // Provide initial inputs
        c.wait(); // Wait for all steps to finish
        ... // Print results
    }
}

Figure 2: C++ Code implementing steps and environment. Together with a CnC specification file the above forms a complete application.
the mappings are influenced through scheduling strategies, such as LIFO, FIFO, work-stealing, or priority-based.

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. Parallel tasks can be spawned eagerly upon prescription, or delayed until the data needed by the task is ready. The get operations on a data collection could be blocking (in cases when the task executing the step is to be spawned before all the inputs for the step are available) or non-blocking (the run-time system guarantees that the data is available when get is executed). Both the C++ and Java implementations have a roll-back and replay policy, which aborts the step performing a get on an unavailable data item and puts the step in a separate list associated with the failed get. When a corresponding put is executed, all the steps in a list waiting on that item are restarted. The Java implementation also has a “delayed async” policy [1], which requires the user or the translator to provide a boolean ready() method that evaluates to true once all the inputs required by the step are available. Only when ready() for a given step evaluates to true does the Java implementation spawn a task to execute the step.

Initialization and Shutdown: All implementations require some code for initialization of the CnC graph: creating step objects and a graph object, as well as performing the initial puts into the data and control collections.

In the C++ implementation, ensuring that all the steps in the graph have finished execution is done by calling the run() method on the graph object, which blocks until all runnable steps in the program have completed. In the Java implementation, ensuring that all the steps in the graph have completed is done by enclosing all the control collection puts from the environment in a Habanero-Java finish construct [14], which ensures that all transitive spawned tasks have completed.

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.

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. Managed run-time systems such as Java or .NET will not solve this problem, since an item collection retains pointers to all instances. Recovering memory used by data instances is a separate problem from traditional garbage collection. There are two approaches identified thus far to determine when data instances are dead and can safely be released (without breaking determinism). First, [8] introduces a declarative slicing annotation for CnC that can be transformed into a reference counting procedure for memory management. Second, the C++ implementation provides a mechanism for specifying use counts for data instances, which are discarded after their last use. Irrespective of which of these mechanisms is used, data collections can be released after a graph has finished running. Frequently, an application uses CnC for finite computations inside a serial outer loop, thereby reclaiming all memory between iterations.

Related Work Table 1 is used to guide the discussion in this section. This table classifies programming models according to their attributes in three dimensions: Declarative, Deterministic and Efficient. A few representative examples are included for each distinct set of attributes. The reader can extrapolate this discussion to other programming models with similar attributes in these three dimensions.

A number of lower-level programming models in use today — e.g., Intel TBB [18], .Net Task Parallel Library [19], Cilk, OpenMP [3], Nvidia CUDA, Java Concurrency [17] — are non-declarative, non-deterministic, and efficient. Here a programming model is considered to be efficient if there are known implementations that deliver competitive performance for a reasonably broad set of programs. Deterministic Parallel Java [7] is an interesting variant of Java; it includes a subset that is provably deterministic, as well as constructs that explicitly indicate when determinism cannot be guaranteed for certain code regions, which is why it contains a “hybrid” entry in the Deterministic column.

The next three languages in the table — High Performance Fortran (HPF)
Table 1: Comparison of several parallel programming models.

<table>
<thead>
<tr>
<th>Parallel prog. model</th>
<th>Declarative</th>
<th>Deterministic</th>
<th>Efficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel TBB [18]</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>.Net Task Par. Lib.</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Cilk</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>OpenMP [3]</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>CUDA</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Java Concurrency [17]</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Det. Parallel Java [7]</td>
<td>No</td>
<td>Hybrid</td>
<td>Yes</td>
</tr>
<tr>
<td>High Perf. Fortran [15]</td>
<td>Hybrid</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>X10 [6]</td>
<td>Hybrid</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Linda [10]</td>
<td>Hybrid</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Asynch. Seq. Processes [2]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>StreamIt[12]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LabVIEW [20]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CnC</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
deterministic. Asynchronous Sequential Processes [2] is a recent model with a clean semantics, but without any efficient implementations. In contrast, the remaining three entries are efficient as well. StreamIt [12, 13] is representative of a modern streaming language, and LabVIEW [20] is representative of a modern dataflow language. Both streaming and dataflow languages have had major influence on the CnC design. The CnC semantic model is based on dataflow in that steps are functional and execution can proceed whenever data is ready.

However, CnC differs from dataflow in some key ways. The use of control tags elevates control to a first-class construct in CnC. In addition, item collections allow more general indexing (as in a tuple space) compared to dataflow arrays (I-structures). CnC is like streaming in that the internals of a step are not visible from the graph that describes their connectivity, thereby establishing an isolation among steps. A producer step in a streaming model need not know its consumers; it just needs to know which buffers (collections) to perform read and write operations on. However, CnC differs from streaming in that put and get operations need not be performed in FIFO order, and (as mentioned above) control is a first-class construct in CnC. Further, CnC’s dynamic put/get operations on data and control collections serves as a general model that can be used to express many kinds of applications that would not be considered to be dataflow or streaming applications.

**Future Directions** Future work on the CnC model will focus on incorporating more power in the specification language (module abstraction, libraries of patterns) and integration with persistent data models.

Combinations of static/dynamic treatment of scheduling and step/data distribution will continue to be explored. Run-time strategies, such as for reducing overhead for finer-grained parallelism and for memory management, will be developed.

Static graph analysis will play a role in performance optimization in the future.

**RELATED ENTRIES**

- dependences
- dataflow languages
• Cilk
• Linda
• OpenMP
• High-Performance FORTRAN
• X10
• Determinsitic Parallel Java

BIBLIOGRAPHIC NOTES AND FURTHER READING

The basic principles behind the Concurrent Collections programming model are outlined in [4]. The model is built on past work done at Hewlett Packard Labs on TStreams, described in [16].

A technique is presented in [8] for memory management of data items with lifetimes that are longer than a single computation step.

[5] is a pioneering paper in dataflow languages. For a description of coordination languages and their use, see [11].

Many members of the Concurrent Collections community gathered at workshops held in 2009 (http://habanero.rice.edu/cnc09) and 2010 (http://cnc10.rice.edu/cnc10).

BIBLIOGRAPHY

References


