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 our new approach based on this perspective. In particular, a CnC program doesn't explicitly express the parallelism. It expresses the constraints on parallelism. These constraints remain valid regardless of the target architecture.

1. Why is parallel programming hard?

Many parallel languages embed parallel language constructs within the text of the serial code. Examples include MPI, OpenMP, PThreads, Ct 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 must be specified.[1]
- Serial code modifies and refers to variables (locations), not values. Variables can be overwritten. This overwriting over-constrains the possible orderings.
- Serial code tightly couples the question of *if* we will execute code from *when* we will execute it. Arriving at some point in the control flow indicates both that, yes, we will execute this code and also that we will execute it now. This is true for loop iterations, recursive calls and invocations of other subroutines. These also constitute arbitrary ordering.

Converting serial code to parallel code involves uncovering alternate valid executions 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 pragmas in the midst of this problem again requires uncovering alternate valid executions. This is difficult to get right in the first place and to modify later. In addition, of course, the parallelism constructs might

- be for a constrained class or architectures (say, shared memory)
- focus on a limited type of parallelism (say, data parallelism)

So when the architecture changes, so must the code. For these reasons, embedding parallelism in serial code can limit both the language's effectiveness and its ease of use. In addition, these constraints might assume arbitrary constraints such as barriers after each loop or single-program-multiple-data (SPMD). Although this is not the focus of this paper, notice that these assumptions can also inhibit performance.

[1] It not hard to find an ordering but it can be complicated for a program or a compiler to undo the ordering.

2. The essence of parallel execution

What does a runtime system need to know in order to execute a program in parallel? We are not yet asking how to specify the parallelism, how to optimize for any specific target, etc. We are just asking: What are the inputs to this decision?

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

- Data dependences (producer/consumer relations): One computation produces data consumed by another. Data is explicitly produced by a producer computation and explicitly consumed by (possibly multiple) consumer computations.
- Control dependences (controller/controllee relations): One computation determines if another will execute. To eliminate the tight coupling of the *if* and *when* control flow questions, control *tags* will be explicitly produced by a controller computation and will control the execution of a controllee computation. This puts the control and data dependences on the same level as in intermediate forms such as program dependence graphs [5].

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 dependences among these high-level operations.
- The control tags that participate in control dependences among these high-level operations.

The relationships among these objects that need to be identified are:

- producer/consumer relations
- controller/controllee relations

As we see below, these three types of objects and relations among them is exactly what Concurrent Collections provides.

3. What is Concurrent Collections?

CnC relies on a combination of ideas from tuple-space [6], streaming [7] and dataflow [8] languages. CnC programs are written in terms of high-level application-specific operations. These operations are partially ordered according to their semantically required scheduling constraints only. The data that flows among these operations is by value, not by location. There is no overwriting and no arbitrary serialization among the high-level operations. The high-level operations themselves are implemented in a serial language.

This approach supports an important separation of concerns. There are two roles involved in implementing a parallel program. One is the *domain expert*, the developer whose interest and expertise is in the application domain, e.g., finance, genomics, etc. The other is the *tuning expert*, whose interest and expertise is in performance, possibly performance on a particular platform. These may be distinct individuals or the same individual at different stages in application development. The tuning expert may in fact be software (static compiler analysis or dynamic runtime analysis). The Concurrent Collections programming model separates the expression of the semantics of the computation (by the domain expert) from the expression of the actual parallelism, scheduling and distribution for a specific architecture (by the tuning expert). This separation simplifies the work of the domain expert. Writing in this language does not require any reasoning about parallelism or any understanding of the target architecture. The domain expert is concerned only with her area of expertise (the semantics of the application). The tuning expert is given the maximum possible freedom to map the computation onto his target architecture. In this document we will focus on topics relevant to the domain expert.

3.1. Language concepts via an example

We will use face detection as an example application to describe the language. Detection is performed on a sequence of images. Each image is further subdivided into square sub-images (called *windows*) of any size and at any position within the image. Each window is processed by a sequence of classifiers. If any classifier in the sequence fails, the window does not contain a face and the remainder of the classifiers need not process that window. The goal of this approach is to rapidly reject any window not containing a face. This example is chosen because it relies heavily on control-dependences which enable Concurrent Collections to support more than pure streaming applications.

A program is specified by a graph with three types of nodes (computation *steps*, data *items* and control *tags*) and three types of edges (producer relations, consumer relations and prescription relations). We will introduce the language by showing the process one might go through to create a version of the face detector in this language. This discussion refers to Figure 3-1 which shows a simplified graphical representation of our face detection application.

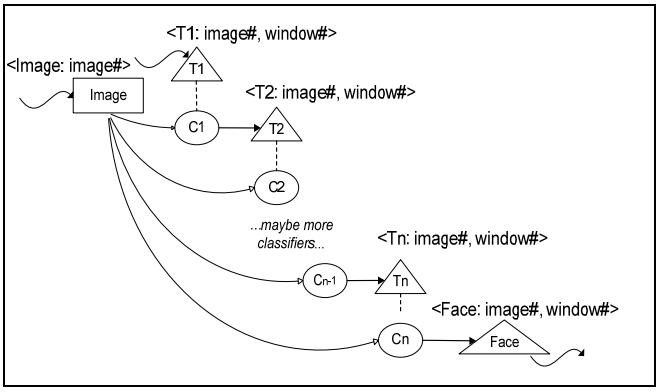

Figure 3-1 Face detection: graphical form

3.1.1. Creating a CnC graph

1. What are the high-level operations in the application?

The computation is partitioned into high-level operations called step collections. Step collections are represented as ovals. In this application, the step collections are the classifiers C1, …, Cn. We use the term step collection to indicate that it is a collection composed of distinct step instances which are the unit of scheduling, distribution and execution.

2. What data is produced/consumed by these operations?

Similarly, the user data is partitioned into data structures called *item collections.* Item collections are represented by rectangles. Again we use the term collection to indicate that it is a collection composed of distinct item instances. In this application there is only one item collection, image. Item instances are the units of storage, communication and synchronization. The producer

and consumer relationships between step collections and item collections are explicit. The consumer relationships are represented as directed edges into steps. Producer relations are represented as directed edges out from steps. The image items are consumed by the classifier steps. There are no items produced in this application.[2]

The environment (the code that invokes the graph) may produce and consume items and tags. These relationships are represented by directed squiggly edges. In our application, for example, the environment produces Image items.

At this point we have a description that is typical of how people communicate informally with one another at a whiteboard. The next two phases are required to make this informal description precise enough to execute.

3. What distinguishes instances of data and operations?

The computations represented by circles are not long-lived computations that continually consume input and produce output. This would constitute another arbitrary ordering. Instead, scheduling and distribution is on step instances. Synchronization and communication is on item instances.

We need to distinguish among the instances of a step collection and instances of an item collection. Each dynamic instance of a step or an item is uniquely identified by an application-specific tag. A *tag component* might indicate a node identifier in a graph, a row number in an array, an employee number, a year, etc. A complete *tag* might be composed of several components, for example, employee number and year or maybe xAxis, yAxis, and zAzis.

In our example, the instances of the image collection are distinguished by image#. The classifier step instances are distinguished by image# and window# pair. In this example, a classifier step inputs the whole image even though it operates only on one window within the image.

4. What are the actual instances of data and operations?

Knowing the tag components that allow us to distinguish among instances is not quite precise enough to execute. Knowing that we distinguish instances of classifier1 by values of image# and window# doesn't tell us if classifier1 is to be executed for image#2873, window#56. We have already introduced item collections for data and step collections for computation. Now we introduce *tag collections* for control to specify exactly which instances will execute.

Tag collections, sets of tag instances, provide the control mechanism. Tag collections are shown in triangles. The tag collections in this graph are T1, … , Tn . A *prescriptive relation* may exist between a tag collection T and a step collection S. The meaning of such a relationship is this: if a tag instance t, say image# 2873, window# 56, is in collection T, then the step instance s in S with tag value t, image# 2873, window# 56, will execute. A prescriptive relation is shown as a dotted edge between a tag collection and a step collection. A step collection S prescribed by a tag collection T must have tags of the same form as tags in T. Thus we know the form of the tags for the classifiers.

Usually control flow indicates not only *if* code executes but also *when*. In CnC, the control via tags only indicates *if* code executes. *When* it executes is up to a subsequent scheduler.

When we add a tag collection to our specification, we have to add the corresponding producer relation. For example, the environment produces T1 which indicates all the windows for all the images. The point of step collection C1 is to determine which of these windows definitely do not contain a face and which might contain a face. An instance of C1, say with tag image# i and window# w, will either produce tag T2 with tag image# i and window# w (indicating that it might be a face) or it will produce nothing (indicating that it is definitely not a face). So step collection C1 produces tag collection T2. The tag instances in T2 determine which instances of C2 will execute. Similarly other step collections and tag collections have producer relationships.

5. What are the relationships among instances?

To understand the constraints on parallelism we need more specifics about the relations among instances. *Tag functions* provide this information. In our example, the producer tag function that maps the tag of a classifier step, say (C1) to the tag of the tag collection <T2> is the identity function, e.g., (C1: i, w) can only produce <T2: i, w> not <T2: i+1, w> for example. Other applications, for example nearest neighbor computations, have more interesting tag functions. What is important is that tag functions require only domain knowledge, not understanding of parallelism.

At this point the importance of tag collections and tag instances should be clear. Tags make this language

[2] The directed edges from steps to the triangles are discussed below.

more flexible and more general than a streaming language. In addition, the tag mechanism separates the question of *if* a step will execute from *when* a step will execute. The domain-expert determines *if* a step will execute. The tuning-expert determines *when* it will execute. This separation not only allows for more effective tuning, it makes the job of the domain expert easier.

3.1.2. Textual representation

Concurrent Collections can be represented in a variety of distinct forms. A textual form of the graph represents each relationship in a separate statement using parens, square brackets and angle brackets instead of ovals, rectangles and triangles. For example,

```
(C1: image#, window#) • <T2: image#, window#>;
```

A translator converts this form to use a CnC class library. One can specify the graph directly in the CnC class library. We are currently investigating a graphical form that looks more like Figure 3-1.

3.1.3. Coding the high-level operators

In addition to specifying the graph, we need to code the steps in a serial language. The step has access to the values of its tag components. It uses `get` operations to consume items and `put` operations to produce items and tags. An example of step code showing the API for the current implementation is shown below.

```
void c1(facedetector_graph_t& graph,
     const Tag_t& step_tag) {
   // Retrieve the image
   image_t x = graph.image.Get(step_tag);
   // Check the image
   if (isFace(x))
     // Add the tag for next classifier
     graph.T2.Put(step_tag);
   return;
 }
```

3.1.4. Semantics and Parallelism

The vision expert going through the process above needs to know a lot about vision but nothing in the process involves any reasoning about parallelism. The resulting program makes the constraints on parallelism explicit, but not the parallelism itself. The constraints are either data dependences (steps produce items that are consumed by other steps) or control dependences (steps produce tags that prescribe other steps). This simple example only contains control dependences.

The semantics ensure the constraints on the schedule by maintaining attributes for each instance. As the program executes, item and tag instances become *available*. A step is *prescribed*, when its prescribing tag is available. A step may execute when it is *prescribed* and its input items are *available*. Attributes are only added so the process is monotonic. The *execution frontier* is the set of instances that have some attribute but are not yet *dead/executed*. The program is *valid* if, when it terminates, all its *prescribed* steps have *executed*. Note that this is a description of the semantics and not of any particular implementation. Some of the implementations of CnC are actually quite different from this description.

Consider the possible schedules for our example. There are no constraints among images and there are no constraints among windows in an image. The only constraint is that for a given window w of a given image i, the classifiers are executed in order. This ordering constraint arises because of the control-dependences, for example, we don't know if classifier (C2: i, w) will execute until (C1: i, w) completes.

3.1.5. Key aspects of the language

Concurrent Collections is a way of expressing a program:

- In terms of higher-level operators and data structures appropriate to the application.

This allows the programmer to continue to express the high-level operators of the program in any familiar serial programming language.

- In dynamic single assignment form.

The computation is expressed in terms of values, not locations. Each high-level operation is functional. The only side-effects are explicitly producing values. There is no overwriting so there are no race conditions and the results are deterministic regardless of schedule.

- In terms of ordering constraints based on the flow of data and control.

The operations of the computation are partially ordered, based only on the data flow among them. There is no need for analysis to undo an overly-constrained ordering.

- CnC isolates the question of *if* code will execute from *when* it will execute.

This provides ease and flexibility for scheduling.

- CnC acts as an interface.

Parallel constructs are not embedded with serial source. The details of the application are on one side of this interface. The mapping to a parallel platform is on the other. Because these are isolated, it is easier to modify one or the other independently.

This model delivers to the domain expert computation that

- is based on how people actually communicate with one another about their application,
- is deterministic and therefore gets identical results regardless of the schedule, distribution, configuration or architecture
- requires no reasoning about parallelism and
- is neutral with respect to target platform.

and delivers to the tuning expert (person or program)

- maximal flexibility for tuning. This flexibility comes about because only the constraints are explicit. There is no overwriting or arbitrary serialization to constrain scheduling and distribution decisions.

4. Mapping CnC to a parallel target

Given the objects specified in your CnC specification, there are a variety of ways to execute it in parallel. First it supports task parallelism, pipeline parallelism and data parallelism. The three things that need to be determined are: grain, distribution across processors, scheduling within processors. Unlike other languages that are designed with a fairly specific execution style in mind, CnC is designed to support many. The following distinct systems have been implemented:

	Memory	Grain	Distribution	Schedule
HP	distributed	static	static	static
HP	distributed	static	static	dynamic
Intel	shared	static	dynamic	dynamic
Rice	shared	static	dynamic	dynamic
GaTech	shared	dynamic	dynamic	dynamic

5. Applications

We have built a variety of runtime systems. The current system (see [1] for download and documentation) is built on Intel's TBB [2]. The set of applications in CnC is small but growing. It includes body tracking, Black-Scholes, game of life, Dedup, Cholesky factorization, Eigansolver, matrix inversion, conjugate gradient. Some are complete. Others are on the way. They show speedup comparable to TBB itself. More importantly they show good scalability. When we incorporate tools for the tuning expert, we anticipate even better performance.

Acknowledgements

Thanks are due to many who have contributed to this work.

Carl Offner and Alex Nelson for help with the initial model, and the design and implementation of several runtime systems.

Shin Lee, Steve Rose, Nikolay Kurtov and Leo Treggiari, the Intel® Concurrent Collections for C/C++ team.

Geoff Lowney, William Youngs, John Pieper, Frank Schlimbach, Steve Lang and Mark Hampton from Intel.

Vivek Sarkar, Zoran Budimlic from Rice.

Kishore Ramachandran, Hasnain Mandviwala, Aparna Chandramowlishwaran, and Rich Vuduc from Georgia Tech.

References

1. Intel ® Concurrent Collections for C/C++. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc

2. Intel Corporation. Threading building blocks. http://www.threadingbuildingblocks.org/

3. Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core implementations of the concurrent collections programming model. In *CPC '09: 14th International Workshop on Compilers for Parallel Computers.* Springer, January 2009.

4. Hasnain Mandviwala. Capsules: Expressing Composable Computations in a Parallel Programming Model. PhD thesis, Georgia Institute of Technology, 2008.

5. Ferrante, J., Ottenstein, K. J., and Warren, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (Jul. 1987), 319-349.

6. Carriero, N. and Gelernter, D. Linda in context, *Commun. ACM 32*, 4 (Apr. 1989)

7. StreamIt: A Language for Streaming Applications. William Thies, Michal Karczmarek, Saman Amarasinghe. *International Conference on Compiler Construction.*Grenoble, France. Apr, 2002.

8. Arvind, Gostelow, K. P. and Plouffe, W. The (preliminary) Id report. Technical Report 114, Department of Information and Computer Science, University of California, Irvine, CA, 1978.