# Concurrent Collections (CnC)

*The material presented in this document is an adaptation from chapter 2 of a master's thesis: Implementing Asynchronous Checkpoint/Restart for the Concurrent Collections Model [1].*

CnC is a system for describing the structure of parallel computation, or coordinating the data- and control-flow between the individual steps of a computation [2, 3]. A CnC application specifies a set of discrete step functions, and the data collections used as input to and output from those step functions.[1] The CnC coordination language describes the relationship between a specific invocation of a step function, its input and output data, as well as parent/child relationships between to other step function invocations.

## 1 Key Properties of CnC

In this section we outline several distinctive characteristics of the CnC programming model. These are the same characteristics that make the CnC programming model well suited for expressing large-scale parallel computations. The characteristics include graph representation, single-assignment data, monotonically growing state, discrete computation steps, and side-effect-free computation steps.

### 1.1 Graph Representation of the Application

A fundamental characteristic of a CnC application is that the entire computation flow is represented as a graph. An application is partitioned into collections of computation steps and data items, each of which describe a

[1] This model varies slightly from the traditional CnC model in that it lacks *control collections*; however, this elision in our model reflects the absence of control collections in the Habanero variants of CnC developed at Rice University, on which this work is based. For a brief overview of control collections, and a discussion of the equivalence of this simplified CnC model with the traditional model, please see Appendix A in [1].

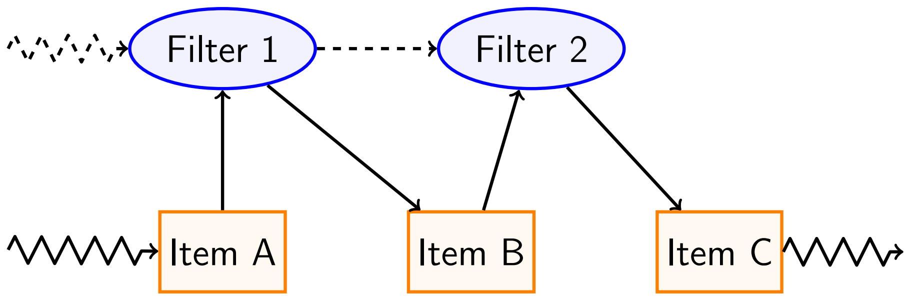


Figure 1: The abstract graph representation of a simple data-filtering CnC application. The two ellipses represent step collections for two separate levels of filtering. The three rectangles represent item collections for holding all of the input data (*Item A*), the results of the first filter pass (*Item B*), and the final output from the second filter pass (*Item C*). Solid edges represent puts to and gets from item collections. Dashed edges represent prescriptions (creation) of new step instances. Jagged edges represent the interactions with the application environment that encloses the CnC graph.

class of step (computation) or item (data) instances. These collections serve as the nodes of the graph. The *prescribe* (step creation), *put* (item creation) and *get* (item read) relationships among the collections are represented as edges in the graph. Figure 1 shows a graph representation for a simple CnC application. By default, we mean a static program graph, when referring to a CnC graph. When necessary, we will differentiate between a static CnC graph, which defines a CnC program, and a dynamic CnC graph, which defines a CnC program execution.

By providing a high-level graphical representation of the application, the user provides the CnC runtime with all the necessary information to automatically track the incremental progress of the application. In the case of a failure, the runtime can restart the computation by simply restarting all of the computation steps that were running at the time of the failure, and providing the input data for those steps to run to completion.

## 1.2 Single-Assignment Data

In a traditional imperative computation model, an application calculates incremental solutions to a problem by updating (or mutating) its in-memory data, eventually resulting in the final output. In contrast, CnC takes a functional rather than imperative approach to modeling state. All data available at the level of the CnC computation graph is single-assignment, meaning that once a data item is created it is never updated. An individual computation step is free to mutate data local to that step, but all such mutations must be fully encapsulated within the step.

### 1.3 Monotonically Growing State

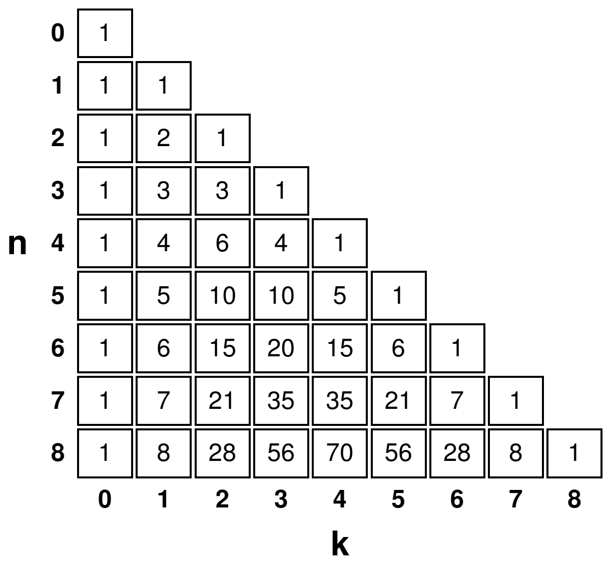
A property that follows from the single-assignment property is the monotonicity property. Since data cannot be updated after appearing in the graph, the overall state of a CnC application appears to only add new data, never removing or mutating previous data. A CnC implementation can optionally free data that is no longer required, though this process can, in general, be more complicated than garbage collection in functional languages [4].

### 1.4 Discrete and Side-Effect-Free Computation Steps

A traditional application may be implicitly divided into several logical computations, but the CnC programming model makes these divisions explicit. The CnC runtime takes advantage of discrete computation steps to run computation steps in parallel on multicore hardware. The fact that CnC applications have discrete computation steps with explicit inputs allows us to restart any given computation at the CnC step granularity. In addition, computation steps in CnC are side-effect-free because the only observable outputs of CnC steps are their items put and steps prescribed. This means that if a computation failed mid-step, there is no possibility that some incremental updates made by the step will corrupt the global CnC graphs state. These properties allow us to safely restart an application that failed at any point in the computation.

## 2 A Sample CnC Application

To better describe the CnC programming model, we now introduce a simple CnC application as an example. This sample application computes binomial coefficients—i.e., the values of ${}_nC_k$ ($n$ choose $k$)—via Pascal's Triangle.



| n \ k | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | | | | | | | | |
| 1 | 1 | 1 | | | | | | | |
| 2 | 1 | 2 | 1 | | | | | | |
| 3 | 1 | 3 | 3 | 1 | | | | | |
| 4 | 1 | 4 | 6 | 4 | 1 | | | | |
| 5 | 1 | 5 | 10 | 10 | 5 | 1 | | | |
| 6 | 1 | 6 | 15 | 20 | 15 | 6 | 1 | | |
| 7 | 1 | 7 | 21 | 35 | 35 | 21 | 7 | 1 | |
| 8 | 1 | 8 | 28 | 56 | 70 | 56 | 28 | 8 | 1 |

Figure 2: The first nine rows of Pascal's Triangle. The entries of Pascal's Triangle correspond to the binomial coefficients, such that the entry at row $n$, column $k$ is equal to ${}_nC_k$.

## 2.1 Review of Pascal's Triangle

Figure 2 shows the first nine rows of Pascal's Triangle. If $P(n,k)$ is the value at row $n$, column $k$ of Pascal's Triangle (where both the row and column numbers are zero-based), then for all $n \geq k \geq 0$:

$$P(n,0) = P(n,n) = 1 \tag{1a}$$

$$P(n,k) = P(n-1,k-1) + P(n-1,k) \tag{1b}$$

Equations (1a) and (1b) exactly match the recursive definition for the binomial coefficients [5]; hence, the entry of Pascal's Triangle at row $n$, column $k$ corresponds to the binomial coefficient ${}_nC_k$ [6].

## 2.2 Structure of the CnC Graph

As explained earlier in this section, every CnC application must specify a set of step collections, corresponding to the functions used in computation, and a set of item collections, corresponding to the data on which the steps operate. To compute the value of ${}_nC_k$, we must compute $n$ rows and $k$ columns of Pascal's Triangle. Since the only type of data we use in this computation (both for building the triangle and in the result) is the set of values in the triangle, we only need a single item collection to hold that data, which we can call *pascal-entries*. Since we have two different equations for computing the entries of the triangle, we have one step collection for computing values based on equation (1a), and another based on equation (1b). We call the first step collection *edge-step* since it computes the values along the left and right edges of the triangle, and the second *inner-step* since it computes the remaining values inside the triangle. A high-level sketch of the CnC graph for this application is illustrated in figure 3.

In CnC, instances of step and item collections are differentiated by a unique *tag*, often represented by an integer tuple; however, to differentiate step and item collections, we typically refer to the tag of an item instance as a *key*. We identify instances of both the item and step collections by the row and column of the corresponding entry in Pascal's Triangle; therefore, the tags and keys for instances in all three collections are integer pairs of the form $\langle row, col \rangle$. For simplicity, we use the notation $(\!|\, \mathsf{S}\colon \mathsf{T} \,|\!)$ to denote an instance of step collection $S$ with the tag $T$, where the round brackets correspond to the round nodes used for steps in the graphical representation (as shown in figure 3). Similarly, we use the notation $[\![\, \mathsf{I}\colon \mathsf{K} \,]\!]$ to denote an instance of item collection $I$ with the key $K$, or $[\![\, \mathsf{I}\colon \mathsf{K} {\rightarrow} \mathsf{V} \,]\!]$ to denote that the item instance has the value $V$, where the square brackets correspond to the rectangular nodes used for items in the graphical representation.

To give our application a more dynamic feel, each step instance with tag $\langle row, col \rangle$ prescribes the step instance with tag $\langle row + 1, col \rangle$. Since each row of Pascal's Triangle has one more column than the previous row, steps with tags where $row = col$ also need to prescribe the step with tag $\langle row{+}1, col{+}1 \rangle$. Each step instance also puts a single data item to the *pascal-entries* collection, with a key matching the step's tag, and the value ${}_{row}C_{col}$. Figure 4 illustrates these relationships among the step and item collections, with the mapping between step tags and item keys shown explicitly.

It is often useful in a CnC application to parameterize some aspects of the graph structure. For example, one might want to parameterize the dimensions of the input matrices to a matrix kernel in order to make the code more generic. In our application, we want to parameterize the values $n$ and $k$, which allows us to stop computation at row $n$ of Pascal's triangle. We do this by setting values for $n$ and $k$ in the CnC graph's context, which is available to all CnC functions. These parameters are considered constant throughout the graph execution. The step functions in our application use

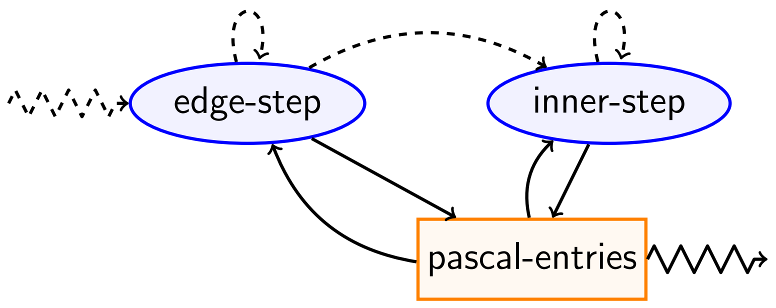
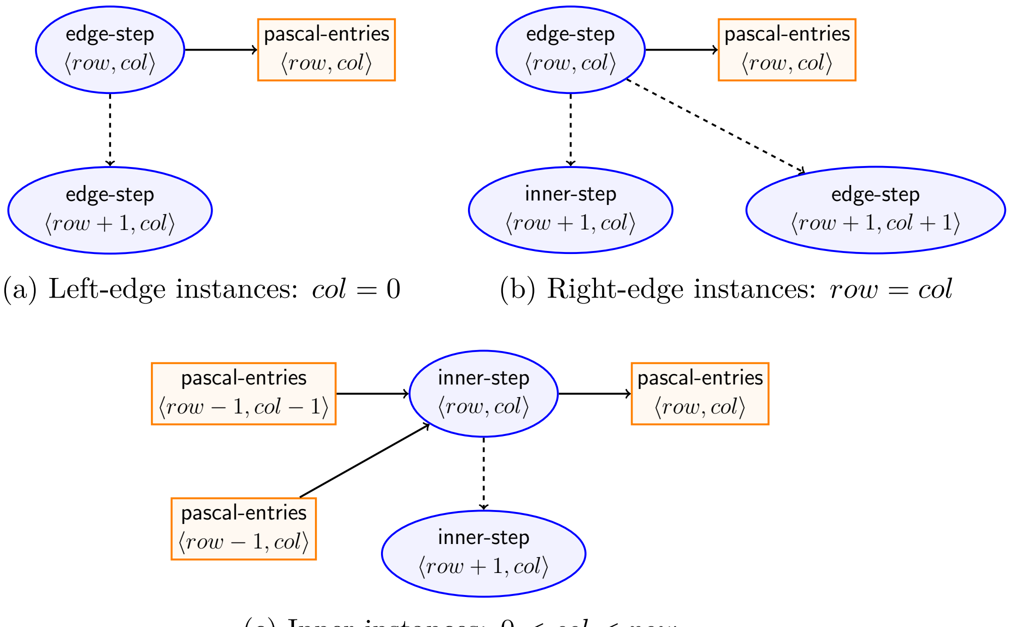


Figure 3: The abstract graph representation of the Pascal's Triangle CnC application. Ellipses represent step collections (computation), and rectangles represent item collections (data). Dashed edges represent step prescriptions (creation), and solid edges represent puts to or gets from item collections. Jagged edges represent interactions with the application environment that encloses this CnC graph.


edge-step ⟨row, col⟩
pascal-entries ⟨row, col⟩
edge-step ⟨row + 1, col⟩
edge-step ⟨row, col⟩
pascal-entries ⟨row, col⟩
inner-step ⟨row + 1, col⟩
edge-step ⟨row + 1, col + 1⟩
pascal-entries ⟨row − 1, col − 1⟩
inner-step ⟨row, col⟩
pascal-entries ⟨row, col⟩
pascal-entries ⟨row − 1, col⟩
inner-step ⟨row + 1, col⟩

(a) Left-edge instances: $col = 0$

(b) Right-edge instances: $row = col$

(c) Inner instances: $0 < col < row$

Figure 4: The *prescribe*, *put* and *get* relationships among the step and item collections in the Pascal's Triangle CnC application. Note that the topmost entry of the triangle, where $row = col = 0$, is actually a special case that does the *edge-step* prescription from the left edge and the *edge-step* prescription from the right edge without an *inner-step* prescription.

these parameter values to compute whether or not to prescribe a new step instance corresponding to the next row of the triangle.

## 2.3 Executing CnC Steps

Before a CnC step instance is executed, that step must be *prescribed* (created) and all of its input data items must be available. The CnC runtime tracks the status of step and item instances via *attributes* attached to the instances. When some step instance (or the environment) *prescribed* a step in collection $S$ with tag $T$, step $\llparenthesis$ S: T $\rrparenthesis$ is created with the *control ready* attribute. If a step has a single input dependence on $\llbracket$ I: K $\rrbracket$, the step gains the *data ready* attribute when an item with key $K$ has been put to item collection $I$. If a step has two or more such dependencies, the step is data ready only when all of the input items have been put. If a step has zero input dependencies then it is always considered data ready. Once a step is both control ready and data ready, it gains the *ready* attribute, and is only then eligible to execute.

In our Pascal's Triangle application, an instance of *edge-step* is ready as soon as it is prescribed because it has no input dependencies on the item collection. Since instances of *inner-step* depend on two item instances from *pascal-entries*, an *inner-step* instance is only ready to execute after it has been prescribed and both of the corresponding item instances have been put.

## 2.4 Interaction with the Environment

A CnC graph is typically embedded within a driver application, and we refer to the portions of the application that interact with the CnC graph as the *environment*. When CnC program execution is completed, the environment must put all data, prescribe all steps, and set any parameters necessary to properly initialize the CnC graph. The environment may also get values from item collections, which acts as an output mechanism for the graph.

In our Pascal's Triangle application, the environment initializes the graph's $n$ and $k$ parameters, then prescribes an $\llparenthesis$ edge-step: 0,0 $\rrparenthesis$, which corresponds to the topmost entry of the triangle. From that point, the CnC runtime has all the information it needs to compute the value for ${}_nC_k$. When the CnC graph has completed its execution, the environment gets $\llbracket$ pascal-entries: n,k $\rrbracket$, which holds the computed value of ${}_nC_k$.

## 2.5 Example Execution

We will now outline an example of an execution trace for our Pascal's Triangle application. For simplicity in tracing the execution, we assume that the runtime has only a single worker thread, meaning that only one step can run at a time. This assumption eliminates any possible concurrency among steps in the computation and simplifies reasoning about program execution and execution trace creation.

We pick ${}_2C_1$ as the target value for this execution, therefore the environment initializes an instance of our CnC graph with the parameters $n = 2$ and $k = 1$. The environment also prescribes ⦅edge-step: 0,0⦆ to start the graph's execution. Since ⦅edge-step: 0,0⦆ has been prescribed and has no input dependencies, it is ready to execute. The graph execution is described textually below, and graphically in figure 5.

**⦅edge-step: 0,0⦆**

  puts ⟦pascal-entries: 0,0→1⟧;
  prescribes ⦅edge-step: 1,0⦆ and ⦅edge-step: 1,1⦆.

All steps in row 0 have now run to completion.

**⦅edge-step: 1,0⦆**

  puts ⟦pascal-entries: 1,0→1⟧;
  prescribes ⦅edge-step: 2,0⦆.

**⦅edge-step: 1,1⦆**

  puts ⟦pascal-entries: 1,1→1⟧;
  prescribes ⦅inner-step: 2,1⦆ and ⦅edge-step: 2,2⦆.

All steps in row 1 have now run to completion.

**⦅edge-step: 2,0⦆**

  puts ⟦pascal-entries: 2,0→1⟧;
  prescribes no steps since $row = n = 2$.

⦅inner-step: 2,1⦆ depends on ⟦pascal-entries: 1,0⟧ and ⟦pascal-entries: 1,1⟧, but since both items were already put, it is ready to execute.

**⦅inner-step: 2,1⦆**

  gets ⟦pascal-entries: 1,0→1⟧ and ⟦pascal-entries: 1,1→1⟧;

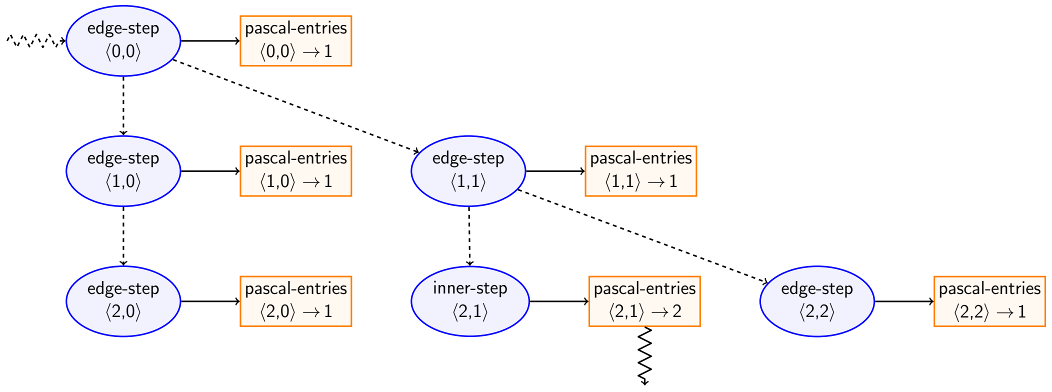


Figure 5: Dynamic CnC graph for the computation of ${}_2C_1$.

> puts ⟦ pascal-entries: 2,1⟶2 ⟧;
> prescribes no steps since $row = n = 2$.

⦇ **edge-step: 2,2** ⦈

> puts ⟦ pascal-entries: 2,2⟶1 ⟧;
> prescribes no steps since $row = n = 2$.

All steps in row 2 have now run to completion. Since all prescribed steps have run to completion, the CnC graph execution is finished. The environment gets item instance ⟦ pascal-entries: 2,1⟶2 ⟧ and correctly yields the answer ${}_2C_1 = 2$.

## 3 The CnC Continuum

CnC describes a programming paradigm rather than a specific runtime implementation. As a result, there is quite a bit of flexibility in how a particular CnC runtime may behave, and what requirements it might impose. One example of this is the static or dynamic nature of the CnC graph. CnC has no restrictions about how much of a application's graph structure must be computable statically versus computed dynamically at runtime. This results in a variety of requirements in the existing CnC implementations pertaining to the specification of inputs and outputs of CnC step functions. Some implementations require that some or all of step tags and item keys to be computed statically, whereas others allow all the inputs and outputs of a step instance to be computed dynamically.

## References

[1] N. Vrvilo, "Implementing Asynchronous Checkpoint/Restart for the Concurrent Collections Model," Master's thesis, Rice University, Houston, TX, 2014. https://habanero.rice.edu/vrvilo-ms.

[2] M. Burke, K. Knobe, R. Newton, and V. Sarkar, "Concurrent collections programming model," in *Encyclopedia of Parallel Computing* (D. Padua, ed.), pp. 364–371, Springer US, 2011. http://goo.gl/UF4L0I.

[3] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3, pp. 203–217, 2010.

[4] D. Sbîrlea, K. Knobe, and V. Sarkar, "Folding of tagged single assignment values for memory-efficient parallelism," in *Euro-Par 2012 Parallel Processing*, pp. 601–613, Springer, 2012.

[5] E. W. Weisstein, "Binomial Coefficient. From *MathWorld*–A Wolfram Web Resource." http://mathworld.wolfram.com/BinomialCoefficient.html. Accessed: 2014-03-15.

[6] E. W. Weisstein, "Pascal's Triangle. From *MathWorld*–A Wolfram Web Resource." http://mathworld.wolfram.com/PascalsTriangle.html. Accessed: 2014-03-15.