CnC-Hadoop: a Graphical Coordination Language for Distributed Multiscale Parallelism

Riyaz Haque
UCLA
rh12@rice.edu

David M Peixotto   Vivek Sarkar
Rice University 
{dmp,vsinkar}@rice.edu

Abstract
The information-technology platform is being radically transformed with the widespread adoption of the cloud computing model supported by data centers containing large numbers of multicore servers. While cloud computing platforms can potentially enable a rich variety of distributed applications, the need to exploit multiscale parallelism at the inter-node and intra-node level poses significantly new challenges for software. Recent advances in the Google MapReduce and Hadoop frameworks have led to simplified programming models for a restricted class of distributed batch-processing applications. However, these frameworks do not support richer distributed application structures beyond map-reduce, and do not offer any solutions for exploiting shared-memory multicore parallelism at the intra-node level.

In this paper, we extend past work on Intel’s Concurrent Collections (CnC) programming model to address the multiscale programming challenge using a model called CnC-Hadoop. CnC is a declarative and implicitly parallel coordination language that supports flexible combinations of task and data parallelism while retaining determinism. CnC computations are built using steps that are related by data and control dependence edges, which are represented by a CnC graph. While there have been a number of past efforts that proposed the use of graphical coordination languages for distributed computing, none of them targeted frameworks like Hadoop that enable scalable and fault-tolerant processing of large volume of data. This is likely because the basic bulk-synchronous structure of map-reduce frameworks make them less conducive to supporting execution of graph-based programs with point-to-point interactions. We address this problem by extending CnC with reduction steps and accumulator items, and by proposing the introduction of a partitioning phase in the tool chain that maps CnC steps to bulk-synchronous stages in a Hadoop framework while allowing for multicore execution of steps within each stage. Preliminary proof-of-concept results show that CnC-Hadoop can be a promising approach to address the challenge of distributed multiscale parallelism.

1. Introduction
The information-technology platform is being radically transformed with the widespread adoption of the cloud computing model supported by data centers containing large numbers of multicore servers. While cloud computing platforms can potentially enable a rich variety of distributed applications, the need to exploit multiscale parallelism at the inter-node and intra-node level poses significantly new challenges for software. Recent advances in the Google MapReduce and Hadoop frameworks have led to simplified programming models for a restricted class of distributed batch-processing applications. However, these frameworks do not support richer distributed application structures beyond map-reduce, and do not offer any solutions for exploiting shared-memory multicore parallelism at the intra-node level.

Intel’s Concurrent Collections (CnC) [1, 10] is a declarative and implicitly parallel coordination language that supports flexible combinations of task and data parallelism while retaining determinism. CnC computations are built using steps that are related by data and control dependence edges, which in turn are represented by a CnC graph. CnC is provably deterministic. While this restricts CnC’s scope, it is more general than other deterministic programming models including dataflow and stream-processing, and can incorporate static and dynamic forms of task, data, loop, pipeline, and tree parallelism. However, none of the current implementations of CnC targets a cloud computing framework like Google MapReduce or Hadoop.

In this paper, we extend past work on Intel’s Concurrent Collections (CnC) programming model to address the multiscale programming challenge using a model called CnC-Hadoop. CnC is a declarative and implicitly parallel coordination language that supports flexible combinations of task and data parallelism while retaining determinism. CnC computations are built using steps that are related by data and control dependence edges, which are represented by a CnC graph. While there have been a number of past efforts that proposed the use of graphical coordination languages for distributed computing, none of them targeted frameworks like Hadoop that enable scalable and fault-tolerant processing of large volume of data. This is likely because the basic bulk-synchronous structure of map-reduce frameworks make them less conducive to supporting execution of graph-based programs with point-to-point interactions. We address this problem by extending CnC with reduction steps and accumulator items, and by proposing the introduction of a partitioning phase in the tool chain that maps CnC steps to bulk-synchronous stages in a Hadoop framework while allowing for multicore execution of steps within each stage. Preliminary proof-of-concept results show that CnC-Hadoop can be a promising approach to address the challenge of distributed multiscale parallelism.

The rest of the paper is organized as follows. Section 2 summarizes background on the CnC and Hadoop MapReduce programming models. Section 3 introduces our proposed CnC-Hadoop programming interface. Section 4 summarizes our implementation of

1 An earlier version of CnC was called TStreams [12]
the CnC-Hadoop programming interface. Section 5 presents preliminary experimental results for CnC-Hadoop, and our conclusions are contained in Section 7.

2. Background

2.1 Concurrent Collections Programming (CnC) Model

In this section, we give a brief summary of the CnC model, as described in [2]. As in dataflow and stream-processing languages, a CnC program is a graph of communicating kernels. The three main constructs in CnC are step collections, data collections, and control collections. These collections and their relationships are defined statically. For each static collection, a set of dynamic instances is generated at runtime.

A step collection corresponds to a specific procedure, and its instances correspond to invocations of that procedure with different inputs. A control collection is said to prescribe a step collection—adding an instance to the control collection will cause a corresponding step instance to eventually execute with that control instance as input. The invoked step may continue execution by adding instances to other control collections, and so on.

Steps also dynamically read and write data instances in data collections. If a step might access data within a collection, then a (static) dependence exists between the step and that data collection. The execution order of step instances is constrained only by their data and control dependences. 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 prescription dependences. The following is an example snippet of CnC specification graph, where bracket types distinguish the three types of collections (< > for control collections, () for step collections, and [] for data collections.

\[
\text{myCtrl} ::= \text{(step)}; \quad \text{// prescribe step} \\
\quad \text{// producer/consumer dependence} \\
\quad \text{[data1] \to (step) \to [ctrl2], [data2]};
\]

The domain expert writes the step implementation code (procedure body) and puts the steps together in a CnC graph as above. It is in this sense that CnC is a coordination language. The domain expert says nothing about how operations are scheduled, which depends on the target architecture. The tuning expert\(^2\) then maps the CnC specification to a specific target architecture, creating an efficient schedule. Thus, the specification serves as an interface between the domain and tuning experts. This is quite different from the more common approach of embedding parallelism constructs within serial code.

Within each collection, control, data, and step instances are distinguished by unique tags. These tags generally have meaning within the application. For example, they may be database keys or tuples of integers modeling an iteration space. Each type of collection uses tags as follows:

- **Putting** a tag into a control collection will cause the corresponding steps (in prescribed step collections) to eventually execute. A control collection C with tag i is denoted \(< C : i >\).
- Each step instance is a computation that takes a single tag (originating from the prescribing control collection) as an argument. The step instance of collection \(foo\) at tag i is denoted \(foo : i\).
- A data collection is an associative container indexed by tags. The entry for a tag i, once written, cannot be overwritten thereby satisfying the dynamic single assignment rule. The immutability of entries within a data collection is necessary for determinism. An instance in data collection \(x\) with tag i is denoted \([x : i]\).

Because control collection tags are effectively synonymous with control instances we will use the terms interchangeably in the remainder of this paper. We will also refer to data instances simply as items, and operations on collections as puts and gets.

A CnC specification can optionally include tag functions [5] and use them to specify the mapping between a step instance and the data instances that it consumes or produces. A tag function can be the identity function, or can define nearest neighbor computations, a parent/child in a tree, neighbors in a graph, or any other relationship useful in the application.

2.2 Hadoop MapReduce Programming Model

2.2.1 MapReduce

MapReduce [3] is a programming model for large-scale distributed batch processing. It is suited for applications that process large amounts of independent data in a parallel manner. This is achieved by distributing a large data input file into fixed-size chunks among the nodes in the cluster. The distribution is performed transparently by the runtime with the help of the underlying file system. Each chunk in turn is conceptually organized into a list of records. Logically a record is a \(<\text{key}, \text{value}\>\) pair and can have any suitable format based on application logic. Only the following two interfaces are available to the programmer:

- **map**\(\text{key k, value v, output Set<k', v'>}\): map is the process that is run independently for each record. Each map takes as input a record \(<k, v>\) and computes a set of outputs records \(<k', v'>\).
- **reduce**\(\text{key k', Set<value v'>, output Set<k', v'>}\): reduce performs an aggregation over a list of values \(v'\) and generates a final output record \(<k', v'>\). Similar to map, since all values for a key \(k'\) are moved to the same node by the runtime after the map task, no data transfer between nodes is required during the reduce task. The final output of the entire computation is the set of records \(<k', v'>\).

Note that each instance of map and reduce is independent and sequential. Parallelism is created implicitly by the framework through the scheduling of tasks concurrently on different nodes. The programmer only needs to implement the map and reduce interfaces, in any language of one’s choosing. Without having to deal with explicit parallelism constructs, writing distributed programs is simplified. Since individual tasks do not need to communicate with each other, this approach is scalable to a very large number of nodes. In addition to high scalability, MapReduce also provides implicit support for fault tolerance and load-balancing among the cluster nodes. Figure 1 shows the steps involved in the execution of a Map-Reduce job on a 2-node cluster.

2.2.2 Hadoop

Hadoop [6] is a Java-based implementation of MapReduce. The Hadoop framework comprises of the following components:

- **Hadoop Distributed File System (HDFS):** HDFS [6] is a distributed file system based on the Google File System [8]. An HDFS file is divided into fixed-sized chunks across the nodes in the cluster. The default size of the chunks is 64MB. Additionally, to provide fault tolerance, several copies (default 3) of
In this section we describe the extensions to CnC needed to implement the Map-Reduce model and the interface used by the programmer to describe Map-Reduce computations.

3.1 Extensions to CnC

Accumulator Collections

Accumulator collections are item collections with some unique properties.

1. Multiple put operations are allowed for the same tag, even though doing so may appear to violate the dynamic single assignment rule.

2. The get operation returns a set of values instead of a single value.

3. A get will block until all steps that write to the collection have completed.

In standard CnC, multiple puts for the same tag would be an error, but accumulator collections allow multiple puts for the same tag. All of the values that were put for a given tag will be accumulated and returned as the result of a get for that tag. The get operation for accumulator collections is changed to always return a set of values (if only a single put is done for a tag then the result will be a singleton set). The get on an accumulator collection will block until all steps writing to the collection have completed. Thus a get will always return the full set of values and any two gets for the same tag will always return the same set of values.

Reduction Steps

We conclude this section with a brief summary and evaluation of how multicore parallelism is exploited in MapReduce jobs. Hadoop offers two configuration parameters, 

\[
\text{mapred.tasktracker.map.tasks.maximum} \quad \text{and} \quad \text{mapred.tasktracker.reduce.tasks.maximum}
\]

make that can be set in the hadoop-site.xml file. If \(m\) and \(r\) are set to the number of processor cores in a node, \(p\), then the Hadoop runtime system will partition the intra-node workload in the map and reduce phases to contain each \(p\) tasks to exploit \(p\)-way multi-core parallelism. To verify the effectiveness of this level of parallelism, we ran the canonical WordCount example described in [3] on an input file of size 3 GB that was created by concatenating text from several files taken from the Gutenberg project [9]. The evaluation was performed on a small-scale distributed Hadoop cluster consisting of 4 nodes with 4 cores per node, using HDFS as its underlying file system.

Table 1 displays the execution times for the WordCount example measured for \(m = r = 1\), \(m = r = 2\), and \(m = r = 4\). \((m = r = 2\) is the default setting for Hadoop on our test system.) As can be seen, a respectable speedup of 3.7× is obtained by changing the configuration parameters. However, these parameters cannot help with situations where less-structured dynamic parallelism needs to be exploited within a map task or reduce task.

Table 1. Execution times in seconds for WordCount on an input of size 3 GB as a function of the \(m\) and \(r\) parameters

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m = r = 1)</td>
<td>910.730 s</td>
</tr>
<tr>
<td>(m = r = 2)</td>
<td>475.183 s</td>
</tr>
<tr>
<td>(m = r = 4)</td>
<td>246.617</td>
</tr>
</tbody>
</table>

In addition to these Hadoop also provides built-in support for certain specific record types, an interface for performance monitoring and advanced features like partitioners to expose some load balancing capability to the programmer.

Figure 1. Map-Reduce job on a 2-node cluster. (1) Read input record from local HDFS chunk (2) Perform map on each record (3) Shuffle output data across nodes (4) Perform reduce (5) Write output to local HDFS store.

Each chunk are created on different nodes. In the HDFS setup one node called NameNode is designated as the master and the rest of the nodes are configured as workers called DataNodes. The NameNode maintains the meta-data about the file system and the DataNodes store the actual data chunks. This partitioning of files by HDFS determines where the map tasks are scheduled by the Hadoop runtime.

- **JobTracker**: JobTracker is a process created by the Hadoop runtime on a node designated as a master node. This process is responsible for creating individual processes on the other nodes, tracking their progress and providing fault tolerance.

- **TaskTracker**: TaskTracker is a process created at each worker node that performs the actual map and reduce tasks. A TaskTracker reports heartbeats to the JobTracker at regular intervals. If it fails to do so, the JobTracker can decide whether the TaskTracker has crashed and take appropriate action for fault tolerance.

- **Mapper**: The Mapper interface defines the map() method which must be overridden by the programmer in order to implement a map task.

- **Reducer**: The Reducer interface defines the reduce() method which must be overridden by the programmer in order to implement a reduce task.

- **JobConf**: Every map-reduce job needs to define a driver class which configures and initiates a map-reduce job. This configuration information is submitted to the Hadoop runtime through a JobConf object.
Reduction steps enable efficient reductions over the items in an accumulator collection. The reduction step differs from a standard step collection in two ways.

1. Reduction steps are not prescribed by a control collection.
2. A reduction step may only read from accumulator collections in the CnC graph.

A reduction step does not get prescribed explicitly by the programmer. Instead, when a value is put into the accumulator collection that the reduction step reads from, the reduction step will be prescribed for that tag. Multiple puts for the same tag will only prescribe a single reduction step for that tag.

Reduction steps have a restricted form that emphasizes their role as a reduction operation. A reduction step may only read from an accumulator collection in the CnC graph. It is not permitted to read from arbitrary item collections. The reason for this restriction is that the body of a reduction step does not have access to any item collections. The step body is passed two items from the accumulator collection when it executes and it returns the reduction of those two items. The reduction step is used by the runtime to compute a reduction over all of the items in an accumulator collection. The final reduction value is written to the output collection connected to the reduction step in the CnC graph. For convenience, we require that any path in the static CnC graph that includes a reduction step cannot be a cycle. However, a cycle in the graph that includes only normal (non-reduction) steps is permitted.

Reduction steps are defined this way to allow the runtime the flexibility to choose how to implement the reduction. A reduction can be performed eagerly as the items are put into the accumulator collection, or it can be done lazily after all the items have been written to the collection. It would also be possible to provide efficient versions of common reduction operations, (e.g. sum, max) that could be implemented without the overhead of invoking a reduction step for each pair of items.

Distributions

Each item collection can specify a distribution function which will be used to partition data keys if the collection is used in a reduce operation. This corresponds to the Map reduce abstraction of a Partitioner class, which helps divide the key space for effective reduce operation. Distributions could be implemented without the overhead of invoking a reduction step for any path in the static CnC graph.

3.2 Syntax

We propose adding a special syntax to the CnC graph language to support accumulator collections and reduction steps. The new syntax simplifies the task of the CnC translator in enforcing the rules for using accumulator collections and reduction steps. A reduction can be expressed in CnC as

```
[[Accumulator]] -> (Reduction) -> [Output]
```

which says that the collection Accumulator is reduced using the reduction Reduction and the result of the reduction is placed in the item collection Output.

If the CnC graph contains a non-reduction step that reads from an accumulator collection then that step can perform gets on the collection as described in Section 3.1. Reduction steps are distinguished from normal steps by the double parentheses syntax in the graph.

3.3 Operational Semantics

We take an operational view of a CnC program and describe the execution by sketching a state transition system. The state of a CnC program is a 4-tuple

```
State = (Steps, Items, Accums, Graph)
```

The Steps element contains all of the currently executing steps, the Items and Accums elements contains the item and accumulator collections, and the Graph element is the static graph as described by the programmer. Each executing step is a tuple (LocalStore, Code) that contains the local variables for that step and the code to execute. The code for a step body contains instructions of the form:

- **Put c tag var** - Put the variable var from local store to collection c with tag.
- **Get c tag var** - Get the item in the collection c with tag tag and store it in the local store with the name var.
- **Compute f** - Perform the computation f on the local store. This allows for arbitrary computations with local variables and returns and updated local store.

A full CnComputation can be described as follows:

```
state = initialState
while (!FinalState(state)) {
    state = Schedule(Next(state))
}
```

The `initialState` is setup according to the Put instructions performed by the environment. The `FinalState` predicate is true when there are no more steps to execute in the Steps portion of the program `State`. The Schedule and Next functions are responsible for updating the computation state. The Next function takes the current state and returns the next state based on the next instruction in the first step of the Steps component of the state. Defining the Next function based on just the first step simplifies its definition. We allow arbitrary scheduling of steps through the Schedule function which takes the Steps component and permutes the order of the steps as desired. The Schedule function could be a simple round robin, a random schedule, or a “smart” schedule based on the dependencies in the graph. The Next function is at the core of the semantics and we describe it in detail below.

In the description below we use pattern matching to deconstruct the state. A : is used for list matching. For example Compute f:insns would be a list of instructions whose first element is Compute f and whose remaining elements is captured by the variable insns. An empty list is denoted by []. A pattern match can be augmented with guard conditions, which are preceded by a !. The guard condition must be true for the match to be successful. Finally, in the function definition we use a form of list comprehension syntax. A list comprehension contains an expression on the left side and a generator with predicates on the right side, separated by a |. The values are generated and any value that passes all the predicates is passed to the expression on the left hand side. We can now give the full definition of the Next function.

**Item Collection Put** A put to an item collection must verify that there is no value already defined for that item collection at the given tag to satisfy the single assignment condition. The state is updated to reflect the new value added to the item collection.
where ics’ = ics with ic updated so ic[t] = val

Item Collection Get There are two cases to consider for a get from an item collection. If the value is available in the collection at the given tag, then we simply add a binding in the local store. If the variable is not available then we return the state unchanged with the idea that the Schedule function can schedule the task that writes that value so that we will be able to eventually proceed.

Next((ls, Get c t v:insns):ts, ics, acs, g)
| Member(c, ics)
| c[t] != undefined
= ((ls’, insns):ts, ics, acs, g)
where ls’ = ls with ls[v] = c[t]

Next((ls, Get c t v:insns):ts, ics, acs, g)
| Member(c, ics)
| c[t] = undefined
= ((ls’, insns):ts, ics, acs, g)

Step Prescription When a put is done on a tag collection, we look up the code for all the steps prescribed by the tag collection. These steps are given an initial local store that maps the variable ‘tag’ to the actual tag value.

Next((ls’, Get c t v:insns):ts, ics, acs, g)
| Member(c, ics)
| Member(c, acs)
where ics’ = ics with collections updated for all item collections in new
new = [ic[t] = reduce(r, ac[t])
| ic <- ic’s
| ic’ <- ics
| ic’ <- ic’s
| r <- Reductions(g)]

Accumulator Collection Put An accumulator collection allows multiple puts for the same value, so we do not need to verify that there is no value currently in the collection for the given tag. The value from the local store is added to the collection of values (if any) already present in the collection under the given tag.

Next((ls, Compute f:insns):ts, ics, acs, g)
= ((ls’, insns):ts ++ new, ics, acs, g)
where new = [("tag" -> t, code) | code <- steps]
steps = Prescriptions(g,c)

Local Computation A local computation acts on the local store of a single step. We simply replace the local store of the step with the result of applying the computation to the current local store.

Next((ls’, Compute f:insns):ts, ics, acs, g)
| ic <- ic’s
| Member(ac, acs)
where ls’ = f(ls)

Accumulator Collection Get A get on an accumulator collection is similar to a get on an item collection. The key difference is when we can write the value from the collection into the local store. In the case of an item collection we know that only one value will be written for a given tag so we can just check to see if a value exists for the tag. However, an accumulator can collect multiple puts for the same tag so we must use some other mechanism for deciding when all values have been put to the collection. We use the Complete predicate to decide when an accumulation has finished. Complete is a key predicate and we will say more about it below.

Next((ls, Get c t v:insns):ts, ics, acs, g)
| Member(ac, acs)
| Complete(c)
= ((ls’, insns):ts, ics, acs, g)
where ics’ = ics with ac updated so ac[t] += val

Next((ls, Get c t v:insns):ts, ics, acs, g)
| Member(c, ics)
Next((ls, Get c t v:insns):ts, ics, acs, g)
| Member(c, acs)

Computing Complete for Hadoop-CnC

The Complete predicate is used in two places for accumulator collections. First, it is used when performing a get from the collection to make sure that all values are present so that consumers will see the same result of doing a get. Second, it is used at step completion to see what reductions could possibly be run. The reduction steps are not spawned as normal steps with control tags, but rather are run when the accumulator collection is complete. Obviously we can only write out the result of the reduction once all of the input values have been processed.

Conceptually, the Complete predicate should be true when there will be no more data put to the collection. In general computing Complete is a hard problem, but we can use the structure of the graph and the properties of Hadoop to efficiently compute the Complete predicate. Hadoop guarantees that all the map tasks will complete before any reduce task starts. Since the map tasks will be generating the data we know that once all the map tasks are finished no more data will be generated. We use the distinct map/reduce phases of Hadoop to substitute for computing the Complete predicate.

Using the Hadoop phases to compute the Complete predicate requires us to partition the CnC graph into distinct phases. The full CnC graph is partitioned into phases so that there are no dependencies between steps in the same phase. For example, if we have this CnC graph

(p) -> [[A]] -> ([r]) -> [0] -> (c)

then we would have to partition p and c into separate phases since there is a producer/consumer relationship between the steps.

3.4 Patterns for Collective Communication in CnC

In this section we describe how accumulator collections can be used to implement barrier, gather, and (along with a reduction step) reduction operations.

Barrier

The semantics of accumulator collections allow them to be used as barriers between different phases of a computation. The
key property of accumulator collections that allows them to act as barriers is that that get operation will block until all steps that write into the collection have completed execution. The accumulator collection can then be used as a barrier by having all steps in phase one write a dummy value to the collection and have all steps in phase two read (and throw away) the dummy value.

The code below shows an example of a barrier operation. The steps in phase one write to the Barrier collection using the arbitrary tag "BARRIER" and the steps in phase two read from the collection using the same tag.

```java
// CnC Graph
(phase1) -> [[[Barrier]]] -> (phase2)

// Step Code
phase1(tag, Barrier) {
    Barrier.put("BARRIER", null)
}
phase2(tag, Barrier) {
    Barrier.get("BARRIER")
}
```

**Gather**

The gather operation takes data from many locations and gathers it into a single place. This operation is particularly important for distributed computing where a step can only access data located in the same place that it executes. Accumulator collections support the gather operation because they allow multiple puts for the same tag. When a step reads from the the accumulator collection with a given tag it will have access to all the data put with that tag.

The example below shows how accumulator collections can be used to implement a data redistribution using gather and scatter operations. All of the gather steps put their data to the Gather collection under the same (arbitrary) tag "GATHER". The scatter step reads all of the values in the Gather collection and redistributes them by writing them to the Scatter collection with unique tags. Future steps can then read the individual data items from the Scatter collection which will be local to the step.

```java
// CnC Graph
(gather) -> [[[Gather]]] -> (scatter) -> [Scatter]

// Step Code
gather(tag, Gather) {
    Gather.put("GATHER", value)
}
scatter(tag, Gather, Scatter) {
    vals = Gather.get("GATHER")
    i = 0
    foreach val in vals
        Scatter.put(i++, val)
}
```

**Reduction**

Reductions can of course be implemented using accumulator collections and reduction steps. The collection serves to gather all of the values for a given tag and the reduction step is used to perform the actual reduction operation. Note that the reduction could also be implemented as a gather operation described above without the need for the special reduction step. Using the reduction step allows the runtime more freedom on how the actual reduction is implemented and could result in significant space savings if the reduction is performed eagerly as the values arrive.

The example below shows how a reduction operation can be implemented. The producer steps write all of the values to be reduced to the Accum collection using one of three tags. Since the reduction is done on a per-tag basis, all inputs will be reduced to three values in this example (i.e. since the puts are done modulo three). The reduce step is invoked as needed by the runtime to perform the actual reduction. Once all values have been written to Accum, the runtime can write the value of the reduction to the Reduction output collection.

```java
// CnC Graph
(producer) -> [[[Accum]]]
[[Accum]] -> ((reduce)) -> [Reduction]
[Reduction] -> (consumer)

// Step Code
producer(tag, Accum) {
    Accum.put(tag % 3, value)
}
reduce(v1, v2){ v1 + v2 }
consumer(tag, Reduction) {
    r = Reduction.get(tag % 3)
}
```

### 3.5 Patterns for Map-Reduce in CnC

In this section we describe how accumulator collections and reduction steps can be used to implement the Map-Reduce paradigm. We begin by reviewing the basic form of Map-Reduce[13].

```java
mapReduce mapper reducer =
    reducePerKey -- 3. Apply reducer to each group
    groupByKey -- 2. Group intermediate data per key
    mapPerKey -- 1. Apply mapper to each key/value pair
```

The Map-Reduce computation is built from a mapper and a reducer functions. The mapper maps key/value pairs to list of key/value outputs. The reducer reduces a list of values to a single value. The entire Map-Reduce computation is a pipeline that first applies the mapper to each key/value pair, then groups all of the output values by their output keys, and finally applies the reducer to the values for each key.

The translation of the basic Map-Reduce structure to CnC is shown below. The environment writes the initial input values and starts one mapper step for each key. The mapper step writes its output into the groupByKey accumulator collection which gathers the output for each key. The reducer reduction step will be executed once for each key in the groupByKey collection and will write its output to the output collection. The environment can then read the output directly from this collection.

```java
// Input
env -> [input];
env -> <mapPerKey> :: (mapper);

// Map-Reduce
[v1 input: k1] -> (mapper: k1)
    -> [[v2 groupByKey: k2]]
    -> (reducer: k2)
    -> [v2 output: k2];

// Output
[output] -> env;
```

A few more details about each phase are given below.

**Mapper**

The mapper is used to spawn a single Map job. Only Driver activities can define and invoke a Mapper. For every mapper defined
by a driver activity, one map-reduce job is spawned. The activity registered to the mapper serves as the starting point for the said job.

Accumulator

Accumulator corresponds to the gather step of the Map-Reduce job. Each accumulator is registered to a Mapper. Any activity inside a map job gets a reference to the accumulator associated with that map job. Doing a put on this accumulator pushes values into the accumulator collection. This corresponds to generating intermediate key-value pairs in Map-Reduce. Once all the activities initiated by the mapper have completed, the reducer executes which carries out the specified reduction. The driver can then use the output as input to another map-reduce job thus chaining different jobs together.

Reducer

The reducer is executed by the runtime once all values have been written to the accumulator collection.

Larger Example

The example below shows the classic Map-Reduce word count program as implemented in CnC. The `countWords` step is the mapper that will split a document into words and write the (word, count) pairs to the counts accumulator collection. In this example we perform two reductions: finding the total number of occurrences of each word, and finding the maximum occurrence for each word. These reductions are done by the `sumReduction` and `maxReduction` reduction steps respectively. The output of each reduction is written to its own item collection.

```
// Declarations
[int wordCounts: string word]
[int maxOccurs: string word];
[string documents: string docName]
<mapTask: string docName>
(countWords: string docName);

// Reductions
[[int counts: string word]]
((sumReduction: string word));
(maxReduction: string word);

// Prescriptions
<mapTask: string docName> :: (countWords);

// Relations
[documents] -> (countWord) -> [[counts]];
[[counts]] -> ((sumReduction)) -> [wordCounts];
[[counts]] -> ((maxReduction)) -> [maxOccurs];
```

4. CnC-Hadoop Implementation

This section described the translation of the CnC primitives described in Section 3 to their corresponding Hadoop implementation.

4.1 Graph Partitioning

A MapReduce framework like Hadoop inherently executes in a sequence of bulk-synchronous stages. Thus a key requirement in mapping a graphical coordination language like CnC onto Hadoop is to partition a CnC graph such that each dynamic instance of a step is mapped to a unique stage. As we will see, it will be possible to map multiple steps to the same stage so long as they don’t include a dependence chain that contains more than one reduction step. To formalize this problem, we define a CnC Step Dependence Graph (SDG) in which data collections and control collections are omitted and replaced by data and control dependences between steps.

```
Figure 2. An example Step Dependence Graph and its partitioning into two stages (step S2 is a reduction step)
```

There are two key constraints that must be satisfied when partitioning an SDG into Hadoop stages:

1. Contractability: The output of a reduction step $R$ cannot be placed in the same stage as any step reachable from $R$ in the SDG. We model this by labeling all outgoing edges from $R$ as non-contractable, denoted by an “X” in Figures 2 and 3. The source and sink of a non-contractable edge must be placed in distinct stages.

2. Acyclicity: The set of inter-stage SDG edges must be acyclic. This is akin to the acyclicity constraint for loop fusion [14].

When an SDG partition is obtained that satisfies these two constraints, a topological sort can be performed on the inter-stage SDG edges to obtain a legal sequencing of the Hadoop stages. Note that there may be more than one way to partition an SDG into distinct stages. For instance, Figures 2 and 3 show the two possible ways to partition the graph. (Step S2 is enclosed in a double circle because it is a reduction step.) Either map step S4 can be executed in stage 1 alongside step S1, or it can be executed in stage 2 alongside S3.

These constraints formalize the criteria for legal partitions. Selecting the criteria for an optimal partition is a subject for future research.

4.2 Mapper

Each map step is translated to a class which implements the `map()` method of the `Mapper` interface described in Section 2.2.2. Hadoop requires a MapReduce job to have only a single input and a single output. Hence we restrict a map step to read from only a single collection. Similarly, a map step is allowed to only write to one accumulator collection. For example the `countWords` step in word count program is translated as follows:

```
public class countWords implements
```

```
```
Mapper<LongWritable, Text, Text, IntWritable>
{
    public void
    map(LongWritable k1,
          Text v1,
          OutputCollector<Text, IntWritable> counts,
          Reporter reporter) throws IOException
    {
        // code for countWords
        // generate output k2:string, v2:int;
        counts.collect(k2, v2);
    }
}

Note that in the translation of countWords, we do not use the data types of key and value of the documents input collection. This is because the current implementation only supports the default TextInputFormat provided by Hadoop for map inputs where key is the byte offset of a line of text within the file which is of type LongWritable and the value is the line of text itself which is of type Text. LongWritable, IntWritable and Text are the Hadoop implementations of long, int and String data types respectively.

OutputCollector is a class in Hadoop used to store the output result of the map task. It is analogous to the accumulator collections in CnC. Reporter is a class defined by Hadoop used to access custom metrics about the task.

4.3 Reducer

Each reduction step is translated to a class which implements the reduce() method of the Reducer interface described in Section 2.2.2. Similar to map, we also restrict a reduction step to read from only one accumulator collection and write to a single output collection. For example the reduction step sumReduction step in word count program is translated as follows:

```
{[int counts: string]} -> ((sumReduction));
((sumReduction)) -> [int wordCounts: string];
```

becomes

```java
public class sumReduction implements Reducer<Text, IntWritable, Text, IntWritable>
{
    public void
    reduce(Text key,
          Iterator<IntWritable> values,
          OutputCollector<Text, IntWritable> wordCounts,
          Reporter reporter) throws IOException
    {
        int sum = values[0];
        for (int value : values[1] to values[n])
            // n is the number of values
            sum = sumReduction(sum, value);
        wordCounts.collect(key, sum);
    }
    public int sumReduction(int v1, int v2)
    {
        // code for sumReduction
    }
}
```

The inputs key and value come from the counts collection which is the output of the map task. Hadoop runtime only supports lazy reduction i.e. it waits for all the values to be accumulated and then carries out the reduction.

4.4 Accumulator Collections

By design, MapReduce requires a barrier between the map and reduce phases i.e. a reduce task cannot be started until all the map tasks are complete. In other words a get on an accumulator collection should block until all the puts to the accumulator are complete.

In CnC, we implement this by associating a complete attribute with each accumulator. When the environment initiates map, it sets the complete attribute of the accumulator that the map phase writes to. Each map task created by the environment is registered to it. Hence the environment can keep a count of the number of map tasks that are running. All the map tasks eventually complete, write their outputs to the accumulator and then deregister with the environment decrementing its count of tasks. In the meantime any get operation on the accumulator will block if the complete attribute is set. Once the count of processes reaches zero, it is certain that there will be no more puts to the accumulator. At this point environment resets the complete attribute in turn allowing any pending gets to go through.

The Hadoop translation of an accumulator requires no extra coding as the implementation of the Map-Reduce barrier is enforced by the Hadoop runtime itself. We only translate accumulator collections to the output of map and the input of reduce functions as explained in Sections 4.2 and 4.3. The co-ordination of data from output of map task to the input of the reduce task is managed by the Hadoop runtime with the help of the HDFS.

4.5 Item Collections

Item collections are translated to HDFS file locations. For every item collection, a pair of input and output directories is created. Input directories provide input data to the map tasks and output directories are used to store the outputs of reduce tasks. By default the input and output directories are created in the users’ home directory. Thus, for example, the collection [wordCounts] in the word count program will translate to the locations

```
[wordCounts]
    -> /user/home/wc/input // input directory
    -> /user/home/wc/output // output directory
```

This default setting can be overridden by annotating the input collection declaration with a user-defined path. It is the responsibility of the user to ensure that the location exists in the HDFS.

As mentioned above, all HDFS files are in a specific pre-defined format called as TextInputFormat. All puts to the collection are done by concatenating the key and the string representation of the value separated by the string. For example, if a reduce step executes the following put on a collection [string C : int]

C.put(10, "Text");

the Hadoop translation will be

```
C.collect(10, "Text");
```

and it will produce the following line of text in the output file

```
10 Text
```

The user code should ensure that the value Text is correctly interpreted in any subsequent map step which consumes this output file.

5. Preliminary Experimental Results

The preliminary experimental results were obtained by manually translating CnC-Hadoop programs to an equivalent Hadoop representation and running them on a small-scale distributed Hadoop cluster consisting of 4 nodes with 4 cores per node. The framework
used HDFS as the underlying file system. The evaluation described below was performed in the context of expressing certain forms of queries in CnC, and comparing them with approaches used by standard query languages such as Pig from Yahoo, Jaql from IBM, and Hive from Facebook. For this evaluation, we used an input table expressed as a text file of size 2 GB. All experimental results are preliminary and highlight different possible implementation strategies for the CnC-Hadoop model\(^3\).

As a simple database query example, consider two data tables 

\begin{align*}
\text{CityData}: \text{(CityName, Region, PopulationBracket)}
\text{and SalesData}: \text{(CityName, Sales, Date)}.
\end{align*}

Suppose we want to group the overall total sales according to both Region and PopulationBracket. This requires first doing a join on CityData and SalesData based on CityName and later grouping the results of the join according to the desired category. Each of these is translated to a MapReduce job. A baseline Hadoop implementation for this query derived from a standard query language will perform the join twice as shown in Figure 4, which results in a sequence of four MapReduce jobs. However, CnC-Hadoop can instead use the optimized structure shown in Figure 5, in which the join is performed only once\(^4\) resulting in a sequence of three MapReduce jobs. Finally, the CnC graph corresponding to Figure 5 is also amenable to exploiting thread-level parallelism as shown in Figure 6\(^5\). In this case, the two group-by operations are spawned as parallel MapReduce jobs, thereby resulting in a sequence of one MapReduce job followed by two parallel MapReduce jobs.

Table 2 shows the performance measured for the three versions described above, when the join-multigroup query is performed on a table of size 2GB. All results were obtained by setting the \(m\) and \(r\) parameters to 2 (the default settings mentioned in Section 2.2.2). The 1.82\( \times\) speedup of the Figure 5 version relative to Figure 4 can be attributed to the fact that the join operation is performed once in Figure 5 and twice in Figure 4. In fact, the execution time for the join operation can be estimated by computing the difference between the two execution times, 407.219 - 224.193 \(\approx\) 183 seconds. The 2.04\( \times\) speedup of the Figure 6 version can be attributed to the fact that the two group-by MapReduce jobs are executed in parallel by launching them in parallel threads. Since the 183s time for the join operation is a fixed overhead, the speedup due to parallelism in the time spent on the group-by jobs can be estimated as \((224 - 183) / (199 - 183) \approx 2.6\times\). Since each MapReduce job was already configured to use 2 cores (with \(m = r = 2\)) this 2.6\( \times\) speedup in the time spent on group-by jobs on 4-core nodes represents a super-linear speedup thereby showing that exploitation of thread-level parallelism can be very beneficial in the MapReduce context.

In summary, CnC’s graph representation can offer performance improvement opportunities relative to baseline Hadoop by elimination of redundant operations and exploitation of thread-level parallelism.

6. Related Work

We discuss related work according to their attributes in three dimensions: Declarative, Deterministic and Efficient. A number of lower-level programming models in use today for intra-node parallelism — e.g., Intel’s Threading Building Blocks, Microsoft’s .Net Task Parallel Library, OpenMP, Nvidia CUDA, Java Concurrency — are non-declarative, non-deterministic, and inefficient\(^6\). Deterministic Parallel Java \([4]\) is an interesting variant of Java; though imperative (non-declarative), it includes a subset that is provably deterministic, as well as constructs that explicitly indicate when determinism cannot be guaranteed for certain code regions.

Linda is a coordination language in which a thread’s interactions with the tuple space is declarative \([7]\). It was a major influence on the CnC design, but CnC also differs from Linda in many ways. For example, an init() operation in Linda atomically removes the tuple from the tuple space, but a CnC get() operation does not remove the item from the collection. This is a key reason why Linda programs can be non-deterministic in general, and why CnC programs are provably deterministic. Further, there is no separation between tags and values in a Linda tuple; instead, the choice of tag is implicit in the use of wildcards. In CnC, there is a separation between tags and values, and control tags are first class constructs like data items.

\(^3\)The final version of the paper will include results on a larger cluster and for additional CnC programs.

\(^4\)While some relational database query optimizers may be able to also achieve the optimized query structure shown in Figure 5, such optimizers are not available for Hadoop-like middleware and cannot handle more complex graphs that can be expressed in CnC.

\(^5\)For simplicity, these figures just show the graphical relationship among steps by eliding intervening CnC item collections and control collections.

\(^6\)We call a programming model efficient if there are known implementations that deliver competitive performance for a reasonably broad set of programs.
Execution times in seconds, and speedup of relative to Baseline Hadoop implementation of join-multigroup query executed on a table of size 2 GB with $m = r = 2$. 

Both streaming and dataflow languages have also had influences on the CnC approach. The CnC semantic model is based on dataflow in that steps are functional and execution can proceed whenever data is ready, without unnecessary serialization. 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. We observe that CnC’s dynamic put/get operations on data and control collections is a general model that can be used to express many kinds of applications (including Word-Count and database queries) that would not be considered to be dataflow or streaming applications.

Dryad [11] is a notable effort in the area of graphical coordination language for distributed systems. It is a general-purpose distributed execution engine for coarse-grain data-parallel applications that combines sequential vertices with directed communication channels to form an acyclic dataflow graph. As described in [11], channels contain full support for distributed systems and are implemented using TCP, files, or shared memory pipes as appropriate. The Dryad graph is specified by an embedded language (in C++) using a combination of operator overloading and API calls. The main difference with CnC is that CnC can support cyclic graphs with first-class tagged controller-controlee relationships and tagged item collections.

In summary, CnC has benefited from influences in past work, but we are not aware of any other parallel programming model that shares CnC’s fundamental properties as a coordination language, a declarative language, a deterministic language, and a language amenable to efficient implementation. To the best of our knowledge, this is the first experience with mapping the CnC model on to a framework for cloud computing such as Hadoop.

### 7. Conclusions

In this paper, we extended past work on Intel’s Concurrent Collections (CrC) programming model to address the multiscale programming challenge using a model called CrC-Hadoop. We address this problem by extending CnC with reduction steps and accumulator items, and by proposing the introduction of a partitioning phase in the tool chain that maps CnC steps to bulk-synchronous stages in a Hadoop framework while allowing for multiregion execution of steps within each stage. Preliminary proof-of-concept results show that CnC-Hadoop can offer performance improvement opportunities relative to baseline Hadoop by elimination of redundant operations and exploitation of thread-level parallelism. There are a number of promising directions for future work in addressing the challenge of distributed multiscale parallelism with CnC-Hadoop including optimization functions for graph partitioning, experimentation with larger examples and clusters, and integration with richer forms of intra-node parallelism.

### Acknowledgments

The authors would like to thank our research colleagues at Rice University and Intel for valuable discussions related to this work. The authors also acknowledge the support of the Multiscale Systems Center, one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

### References


