Deterministic Reductions in an Asynchronous Parallel Language

Zoran Budimlić[†], Michael Burke[†], Kathleen Knobe[‡]
Ryan Newton[‡], David Peixotto[†], Vivek Sarkar[†], Edwin Westbrook[†]
[†]*Rice University,* [‡]*Intel Corporation*

***Abstract*—Reduction operations are a common and important feature in many parallel programming models. In this paper, we present a new reduction construct for Concurrent Collections (CnC). CnC is a deterministic, asynchronous parallel programming model in which data production and reduction can overlap. While reductions are most frequently incorporated in synchronous contexts where all data is available before parallel reduction begins, our solution works for the asynchronous CnC model. We retain the determinism of the CnC parallel programming model while providing an efficient high-level construct for specifying reductions.**

***Keywords*-reductions; declarative languages; single-assignment languages; determinism; parallelism.**

I. Introduction

With the proliferation of multicore processors, parallel computing has gone mainstream. Unfortunately, the predominant parallel programming model (serial languages with explicit threading) is still acceptable only to a relatively small number of expert programmers. This disparity has spurred development of many higher-level parallel programming models that aim to simplify parallelism and parallel programming to a level that would be adequate for most of today's mainstream programmers.

One attractive programming model is Concurrent Collections (CnC) [1]. CnC is in the family of dataflow and stream-processing languages—a program is a graph of kernels, communicating with one another. In CnC, these kernels are called *steps*, and are related by control and data dependences. The benefit of this approach is that CnC allows the application domain expert to focus on specifying the data and control dependence constraints between computation steps—rather than on synchronization, messaging, or data races—and grants them the freedom to implement the steps in a sequential (or even parallel) language of their choice.[1] At the same time, CnC allows a *tuning expert* to focus on fine-tuning the platform-dependent aspects of the application to achieve higher performance. CnC is suited for many applications that exploit task, data, loop, pipeline or tree parallelism. Even though it exposes many different kinds of parallelism, CnC is provably deterministic [1]. An excellent example of the power of CnC are the CnC implementations of dense linear algebra algorithms that outperform the multithreaded vendor-tuned codes by a factor of up to 2.6x [2].

However, one powerful and naturally parallel construct that CnC does not currently support is reduction. A *reduction* refers to applying a binary operator repeatedly to a collection of data items (for example, summing a list of numbers is a reduction with the binary operator "+"). Reductions have a well-understood theory [3] and a long history in both sequential [4] and parallel [5], [6], [7], [8] programming languages. In addition, they are found in nearly all recent multicore programming libraries, such as Intel's Threading Building Blocks [9], OpenMP [10], and Microsoft's Parallel Pattern Library [11], as well as distributed programming models such as MapReduce [12]. The key difficulty in adding reductions to a data-driven asynchronous execution language such as CnC is determining when all inputs to a reduction are available. This in turn makes it difficult to know when a reduction has computed a final (as opposed to an intermediate) value, which is ready to be passed to the next step of execution.

In this paper, we show how reductions can be added to CnC. This represents the first language design we are aware of that combines reductions with a data-driven asynchronous execution. The key is an algorithm for computing *doneness* that indicates when all the inputs to a reduction have been given. Doneness can only be computed for CnC programs where reductions do not occur in cycles in the program graph (i.e. when reductions are not part of any iterative loop in a CnC program). Intuitively, this restriction is necessary in order to prohibit erroneous programs where the output of a reduction is fed back as in input into the same reduction. In order to allow programs that use reductions in a cycle in a well-defined way, we introduce a hierarchical verison of CnC, where a single CnC step can contain an entire CnC sub-program. Using hierarchy, a CnC program that contains a reduction inside an iterative loop can be decomposed into two pieces, an outer program that contains the loop and an inner program that contains the reduction.

The rest of this paper is organized as follows. Section II-A introduces CnC and briefly discusses the previous work on reductions in higher-level languages. Section III presents the CnC language extensions for specifying reductions. Section IV introduces hierarchical decomposition of CnC programs and describes how the hierarchy is used to determine when reductions are complete. Section V elaborates

[1]Currently, CnC implementations support Java, Habanero Java, C, C++, F#, C# and Haskell as the step implementation languages, with Python and Scala in development.

on how reductions are implemented in CnC, followed by a section with conclusions and directions for future work.

II. Background

A. The CnC Programming Model

The three main constructs in CnC are *step collections*, *data collections*, and *control collections*. While these collections and their relationships are defined statically, a set of dynamic *instances* is generated at runtime for each collection.

A step collection is analogous to a specific computation (a procedure), and its instances are analogous 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 causes corresponding step instance to eventually execute. The invoked step may add instances to other control collections, causing other step instances to execute, and so on.

Steps dynamically read and write data. All the data, control and step instances in CnC can only be assigned once in the execution of the program. This dynamic single assignment rule is essential in enabling some key features of CnC such as determinism, race-freedom, and exposure of many types of parallelism. If a step might touch data within a data collection, then a (static) dependence exists between those step and data collections. The execution order of step instances is constrained only by their data and control dependencies.

Thus a complete CnC specification is a graph where the nodes can be step, data, or control collections, and the edges represent *producer*, *consumer* and *prescription* dependencies. Programs using CnC also have another component, the *environment*, which is the user code outside of CnC. This code processes the inputs, initializes the CnC program, produces initial data and control instances and consumes data instances produced by the execution of the CnC graph. The following is an example snippet of the textual form of CnC specification (where bracket types distinguish the three types of collections):

```
env → <myCtrl>;
<myCtrl> :: (myStep);
[myData] → (myStep) → <myCtrl>, [myData];
[myData] → env;
```

Below is the graphical form of the code above. By convention, in the graphical notation specific shapes correspond to control, data, and step collections. Dotted edges represent prescription (control/step relations), and arrows represent production and consumption of instances. Squiggly edges represent communication with the environment. We will use the graphical notation for CnC code in the rest of the paper.

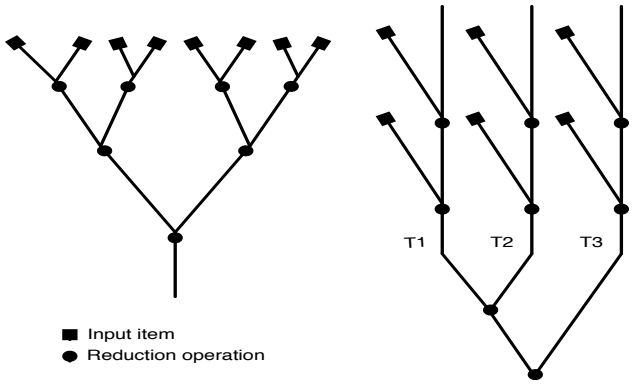

Figure 1. Reductions are defined by their inputs (leaves) and can take on any binary tree structure. Many systems construct balanced binary trees (left). The reduction mechanism proposed in this paper instead produces deep trees (right) with width equal to the number of system threads (T1, T2, T3). That is, each thread maintains a reduction state and reduces a stream of inputs. This same strategy is used by Cilk Hyperobjects.

For each step, such as `myStep` above, the programmer provides an implementation in a separate programming language. A compiler generates code from the CnC specification and links it with the user code and runtime system. A complete CnC program includes the specification, the step code, and the environment code.

Inside each type of collection the control, data, and step instances are all identified by a unique *tag*. Tags can serve simply as unique identifiers, but they are not opaque and generally have some meaning within the application. They may be tuples of integers modeling an iteration space, points in non-grid spaces, or any other values that are useful for the application.

The CnC specification can also specify the relationship between the tags of the specific data and step instances. For example:

```
[myData: i] → (myStep: i) → [myData: i+1];
```

This line of the code specifies that the step instance of step collection `myStep` with the tag `i` reads the data instance from the data collection `myData` with the tag `i`, and produces a data instance in the same data collection `myData` with the tag `i+1`.

B. Reductions

A reduction (or *fold*) applies a binary operator repeatedly to reduce a collection of data to a single value. Operationally, we view a reduction as a *tree* of applications of its binary operator (Figure 1). Independent branches of the tree can be processed separately, yielding a natural opportunity for parallelism. Given an associative and commutative binary operator, a reduction is defined only by its *leaves* (input values)—*any* binary tree with that leaf set is a valid implementation of the reduction. As a result, implementations vary in whether the structure of reduction trees is known

statically[2], dynamically, or varies non-deterministically.

In implementing parallel reduction, it is critical to know when all inputs are available and have been incorporated – i.e. when the reduction is *done*. Many formulations of reduction require that the input domain for a reduction be known in advance of its invocation. For example, in TBB[9] the reduction domain can be specified as the elements of an array. When the domain is known in advance, detecting that a reduction is done becomes trivial, but the production of inputs cannot overlap with their reduction—an unnecessary constraint on the parallel schedule. The Cilk language [13] does better, allowing production and reduction to overlap using a construct called *hyperobjects* which resemble accumulation variables that can be used in parallel. Still, the strictly nested parallel model in Cilk requires that the production of all inputs for a reducer reside within a subcomputation forked from a single point in the program. A barrier synchronization ensures that the subcomputation (and therefore the reduction) is complete before the reduced value can be used.

III. Adding Reductions to CnC

In contrast to Cilk, CnC is a much more asynchronous parallel programming model and provides a uniquely challenging (but rewarding!) context for reductions. The challenge, addressed later in this section, is how to determine when reductions are done.

But, first, we present the API for reductions in CnC. A CnC *reduction object* is initialized with a binary operator and an initial value (identity element) at construction. It exposes three methods:

- `put(v)` a value as input to the reduction,
- `done()` to signal that no further `puts` will occur, and
- `get()` to retrieve the result of reduction.

Thus a regular expression that describes the allowable sequence of operations on a reduction object is:

```
((put)+ done (get)*) | ((put)* (done)?)
```

In this design a reduction has a similar interface to a CnC item collection (`put` and `get`). Consumers must block on a `get` until the value is available. The differences is that multiple `puts` are allowed for a reduction.

To follow the analogy further, items in CnC exist within *item collections* indexed by tags while reductions exist within *reduction collections* indexed by tags. A single reduction collection represents a dynamic set of ongoing and completed reductions. This is similar to handling reductions in the MapReduce framework. MapReduce includes a shuffle phase between the map and reduce phases, where a key produced by the mapper indicates to which reduction the value should be sent.

But when and how should `done()` be called for a reduction? It turns out that there are very few CnC programs in which the user could easily signal `done()` manually. In any program that produces reduction inputs in parallel it would be necessary to establish a barrier that ensures completion of all relevant parallel computations. In principle, the user could have each parallel producer also output an additional dummy item (providing synchronization via the blocking `get`), but this would be both tedious and inefficient. CnC (unlike Cilk) is not natively suited to synchronizing on the completion of parallel subcomputations. The solution is to signal `done()` for reductions automatically.

Signaling Completion of Reductions in CnC

First, note that an alternative to the above synchronous approach to the producer-reducer relationship is for reduction inputs to be provided in a *stream* (partially or totally ordered). An end-of-stream token is then required to signal that reductions are done. This formulation is a good fit with stream programming models. Alas, CnC is not a stream programming model! Values produced by CnC steps are not ordered and whether or not more values will be produced for a reduction is a non-local property.

Thus the key challenge to overcome in equipping CnC with reductions is to determine when end-of-stream tokens can be issued for a particular reduction. The essence of our approach is to automatically track in-flight instances of all step collections in the CnC graph. When all upstream computations that might produce values for a reduction (i.e. share an edge in the CnC specification) are done, then the reduction itself is done and downstream computations may access the reduced value.

Our approach is similar to Cilk's (a coarse grained synchronization on upstream producer computations), except that inputs to a reduction can come from arbitrarily disparate parts of the program rather than a specific nested subcomputation. This distinction makes a significant software engineering difference, removing strict nesting requirements.

The algorithm works as follows. Each collection is mapped onto a *partition*; if there are cycles in the graph, all nodes within a cycle are mapped onto the same partition, otherwise each collection receives its own partition. An atomic counter is allocated for each partition to represent outstanding obligations, including both upstream step collections (which may push more instances into the collection) as well as steps which have been enqueued for execution but not yet executed. Each counter is initialized to the (statically known) number of upstream partitions and incremented only when new steps are prescribed. When a counter reaches zero, the corresponding collection(s) are done. For a step collection, this means that downstream counters are decremented;

[2]Before the incorporation of the features described in this paper the CnC user could still program reductions with fixed trees by constructing those trees of tasks *manually*.

for reduction collections, this means `done()` can be signaled for all contained reduction objects.

Notice that step and item/reduction collections play different roles with respect to done propagation. Step collections may generate new instances downstream, but item/reduction collections do not. The result is that upstream collections other than step collections need not be counted as "obligations" and conversely, because these passive collections are not tracked, when they become done they need not decrement their own downstream counters. This is true for reduction collections as well as item collections in spite of the fact that reductions execute code in their binary operators. These reduction operators are not allowed to perform `puts` or `gets` into the rest of the graph. The restriction exists to maintain determinism in CnC programs; the intermediate values observed by the binary operator are non-deterministic, and must not "leak" into the larger program.

The above treatment of cycles is insufficient for cycles that contain reductions—for example, an iterative program that tests the result of a reduction to determine whether another iteration is required. A cycle becomes done all at once, whereas a reduction requires that upstream step collections become done before it. We do not believe it is possible to directly solve this problem. However, it is possible to refactor many such programs to allow automatic done propagation. Our approach to refactoring is to employ *hierarchy* in CnC graphs, as described in the next section.

IV. Hierarchical CnC with Reductions

In this section we introduce hierarchical CnC and show how to propagate *done* status through hierarchical programs in general. Hierarchy provides a mechanism for isolating reductions from cycles. This isolation allows us to rely on the propagation techniques shown earlier for propagating doneness through DAGs and reduction-free cycles. Hierarchy therefore increases the set of CnC programs that can use reductions.

The key idea in hierarchical CnC is that a step may internally contain another CnC graph, called a *subprogram*. There are many design choices that determine how the inner graph interacts with the outer one. In our design a hierarchical CnC program has the following characteristics:

- The subprogram corresponds to a *logical step*. Internally, the subprogram can get and put data from the outer program, just as a step would.
- The subprogram is invoked when a control tag prescribes the logical step. A subset of steps in the subprogram are wired to the parent graph in such a way so that this control tag creates a single instance of those steps (with the control tag as argument). These steps initiate subprogram execution.
- References to the collections themselves inside the subprogram *may not escape* to the outer program. They are strictly scoped.

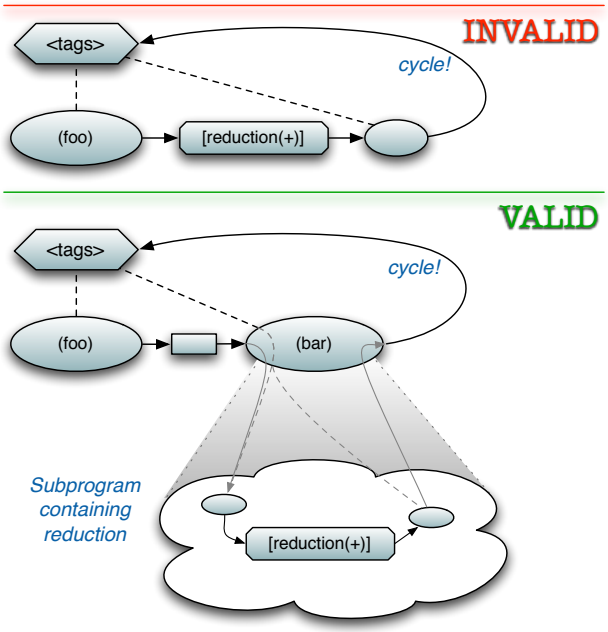

Figure 2. Reductions inside subprograms will complete with the subprogram, thus posing no problem to detecting the completion of enclosing cycles.

The final restriction enables us to ensure that subprograms appear atomic from the outside. When the subprogram is *done* according to the normal rules for DAGs, so is the logical step instance to which it corresponds. Doneness of that step can now propagate normally within the higher level graph.

Consider the case in which the higher level contains a cycle. In Figure 2 there is a cycle that starts and ends with the `<tags>` control collection. Because `(foo)` may (indirectly) call itself through the control collection, there is never a point in time that the `[reduction(+)]` node can be sure that it will not receive further inputs. In contrast, the lower portion of the figure shows a modification of the program to push the reduction down inside a subprogram `(bar)`, which makes the graph amenable to done-propagation—because the reduction is signaled as done when the subprogram completes, irrespective of whether the outer cycle has iterations remaining.

Thus we have shown a graph containing a reduction within a cycle that we can now handle by isolating the reduction from the cycles using hierarchy.

A. Determinism

An important property of CnC is its determinism [1]. This both helps the programmer—who need not reason about a combinatorial explosion in potential program interleavings—and it also frees up the CnC implementation to execute parallel tasks in whatever schedule it deems most efficient. A complete proof of determinism is beyond the scope of this

document, but we give a brief intuition here of why adding hierarchy and reductions to CnC preserves determinism.

Essentially, the reason standard CnC is deterministic is because the item collections are single assignment and because step functions are sequential, deterministic programs. Thus we can see that execution of any two step functions commutes, meaning the result of running both of them is the same no matter which happens first: the only way they could not commute would be by having the two step functions write different values for the same tag in the same item collection, which is not allowed in CnC. By the well-known diamond lemma, we thus have *confluence*, implying that all quiescent states reachable from an initial state are identical.

By an inductive argument, we can see that adding hierarchy cannot destroy determinism, since each hierarchical step is guaranteed to be deterministic by the inductive hypothesis. When we add reductions, we must additionally consider reductions and marking doneness, and whether these commute with each other and with steps. The only issue for commutativity would be that marking a reduction as done does not commute with performing a step of that reduction, but again, this is not allowed in CnC. Thus by the diamond lemma we have determinism of CnC with hierarchy and reductions.

V. Implementation using Intel Thread-Building-Blocks

There are two major parallel reduction implementation alternatives that we describe as *eager* and *lazy*.

- Eager: in an eager reduction calling `put` on a reduction object immediately invokes the reduction operator on the current thread. This means that producers of input data must be parallel in order to achieve parallel reduction. There is no additional parallelism added by reduction, but neither is their additional overhead introduced by task creation upon input to a reduction.
- Lazy: Alternatively, a `put` may store its input into a data structure. One or more reduction tasks may concurrently draw inputs from the data structure and perform reduction in parallel. Mechanisms for regulating the number of reduction workers can grow complex. But if a reduction operation is heavyweight, and the upstream data producer is serial, then the lazy strategy will be beneficial.

We believe that most programs work well with eager reduction. Further, the eager strategy can also result in significantly better space usage by bounding the number of stored reduction states. Thus we have created a prototype eager reduction implementation for Intel CnC using Threading-Building-Blocks (TBB).

In our implementation, we use thread-local storage to store a maximum of one reduction state per thread. The key reduction operations are implemented as follows:

- `put(v)`: Check for existing thread local storage (TLS) on the current thread using a key unique to the reduction object. If absent, allocate TLS and initialize with `v`. If present, apply the binary reduction operator to the TLS value and `v`, writing the result back to TLS. No data races are possible because TLS is, by definition, per-thread. Newly allocated TLS values are added to a list kept by the reduction object.
- `done()`: Because it is guaranteed at the start of `done()` that all `puts` have completed—and `puts` eagerly evaluate the reduction operation—we know that all values in TLS are final. Thus, `done()` iterates over the list of per-thread TLS values, performing one final round of reduction to yield a single value[3]. Finally, the reduction reuses CnC's existing *item collection* data-type to store the final values.
- `get()`: Because item collections have built in synchronization for readers, a reduction's `get()` is nothing more than a `get()` on the internal item collection which stores the final reduced values.

VI. Conclusions and Future Work

In this paper, we have shown that adding reductions to a deterministic, asynchronous parallel language is not only possible, but also results in an elegant high level construct that can be used in many parallel programming patterns.

In order to overlap the production of data with its reduction, we have shown an algorithm for computing *doneness* for step and data instances in a restricted form of CnC programs, where reductions do not appear on cycles. This doneness information allows the runtime system to know when a reduction is complete and its output can be consumed by other steps.

Using *hierarchical* CnC, we have extended this algorithm to handle a wider set of CnC programs, allowing reductions to appear in CnC subprograms, which can in turn appear on cycles in the enclosing CnC program. We have described a preliminary implementation of CnC with reductions based on the Intel's Thread Building Blocks.

In the future, we plan to apply a more flexible *partitioning discipline* to CnC collections for the purpose of refining done detection. Rather than tracking each collection with an atomic counter, subsets of a collection could be tracked independently provided that the CnC specification contains enough metadata to determine how one collection's partitions relate to the other partitions. This refined approach will improve performance in some cases and enable continuously running programs that use reduction.

[3]In our current implementation this final reduction is done serially to optimize for lightweight reduction operations (e.g. $+$) and modest numbers of threads. At a certain scale it would be profitable to exploit extra parallelism in this final phase.

REFERENCES

[1] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşırlar, "The CnC Programming Model," *SIAM PP10, Special Issue on Scientific Programming*, 2010.

[2] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of Concurrent Collections on high-performance multicore computing systems," in *IPDPS '10: International Parallel and Distributed Processing Symposium*, April 2010.

[3] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," in *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1991, vol. 523, pp. 124–144.

[4] K. E. Iverson, *A Programming Language*. Wiley, 1962.

[5] G. L. Steele, *Common LISP: The Language*. Bedford, MA: Digital Press, 1990.

[6] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, pp. 85–97, March 1996.

[7] L. Snyder, "The design and development of zpl," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007, pp. 8.1–8.37.

[8] "High performance fortran language specification version 2.0," 1997, http://wotug.org/parallel/standards/hpf/.

[9] "Threading Building Blocks," http://www.threadingbuildingblocks.org/.

[10] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared memory programming," *IEEE Computational Science & Engineering*, 1998.

[11] Microsoft Corporation, "Parallel Pattern Library (PPL)," http://msdn.microsoft.com/en-us/library/dd492418.aspx.

[12] J. Dean and S. Ghernawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI '04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[13] M. Frigo *et al.*, "Reducers and other cilk++ hyperobjects," in *Proceedings of SPAA '09*, 2009, pp. 79–90.