Concurrent Collections

Zoran Budimlić[1] **Michael Burke**[1] **Vincent Cavé**[1] **Kathleen Knobe**[2]
Geoff Lowney[2] **Ryan Newton**[2] **Jens Palsberg**[3] **David Peixotto**[1]
Vivek Sarkar[1] **Frank Schlimbach**[2] **Sağnak Taşırlar**[1]

[1]Rice University [2]Intel Corporation [3]UCLA

Abstract

We introduce the Concurrent Collections (CnC) programming model. CnC supports flexible combinations of task and data parallelism while retaining determinism. CnC is implicitly parallel, with the user providing high-level operations along with semantic ordering constraints that together form a CnC graph.

We formally describe the execution semantics of CnC and prove that the model guarantees deterministic computation. We evaluate the performance of CnC implementations on several applications and show that CnC offers performance and scalability equivalent to or better than that offered by lower-level parallel programming models.

1 Introduction

With multicore processors, parallel computing is going mainstream. Yet most software is still written in traditional serial languages with explicit threading. High-level parallel programming models, after four decades of proposals, have still not seen widespread adoption. This is beginning to change. Systems like MapReduce are succeeding based on implicit parallelism. Other systems like Nvidia CUDA are partway there, providing a restricted programming model to the user but also exposing too many of the hardware details. The payoff for a high-level programming model is clear—it can provide semantic guarantees and can simplify the understanding, debugging, and testing of a parallel program.

In this paper we introduce the Concurrent Collections (CnC) programming model, built on past work on TStreams [13]. CnC falls into the same family as dataflow and stream-processing languages—a program is a graph of kernels, communicating with one another. In CnC, those computations are called *steps*, and are related by control and data dependences. CnC is provably deterministic. This limits CnC's scope, but compared to its more narrow counterparts (StreamIT, NP-Click, etc), CnC is suited for many applications—incorporating static and dynamic forms of task, data, loop, pipeline, and tree parallelism.

Truly mainstream parallelism will require reaching the large community of non-professional programmers—scientists, animators, and financial analysts—but reaching them requires a separation of concerns between application logic and parallel implementation. We say that the former is the concern of the *domain expert* and the latter of the performance *tuning expert*. The tuning expert is given the maximum possible freedom to map the computation onto the target architecture and is not required to have an understanding of the domain. A strength of CnC is that it is simultaneously a dataflow-like parallel model

and a simple specification language that facilitates communication between the domain and tuning experts.

We have implemented CnC for C++, Java, .NET, and Haskell, but in this paper we will primarily focus on the Java and C++ implementations. The contributions of this paper include: (1) a formal description of an execution semantics for CnC with a proof of determinism and (2) experimental results demonstrating that CnC can effectively exploit several different kinds of parallelism and offer performance and scalability equivalent to or better than what is offered by lower-level parallel programming models.

2 What is CnC?

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

A step collection corresponds to a specific computation (a 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. If a step might touch data within a collection, then a (static) dependence exists between the step and data collections. The execution order of step instances is constrained only by their data and control dependencies. 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* dependencies. The following is an example snippet of a CnC specification (where bracket types distinguish the three types of collections):

```
// control relationship: myCtrl prescribes instances of step
<myCtrl> :: (myStep);
// consume from myData, produce to myCtrl, myData
[myData] → (myStep) → <myCtrl>, [myData];
```

For each step, like `myStep` above, the domain expert provides an implementation in a separate programming language and assembles the steps using a CnC specification. (In this sense CnC is a coordination language.) The domain expert says nothing about how operations are scheduled, which depends on the target architecture. The tuning expert 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. In the case where operations are scheduled to execute on a parallel architecture, the domain expert in effect prepares the program for parallelism. This differs from the more common approach of embedding parallelism constructs within serial code.

A whole CnC program includes the specification, the step code, and the environment. Step code implements the computations within individual graph nodes, whereas the environment is the external user code that invokes and interacts with the CnC graph while it executes. The environment can produce

data and control instances, and consume data instances.

Inside each collection, control, data, and step instances are all identified by a unique *tag*. These tags generally have meaning within the application. For example, they may be tuples of integers modeling an iteration space. They can also be points in non-grid spaces—nodes in a tree, in an irregular mesh, elements of a set, etc. In CnC, tags are arbitrary values that support an equality test and hash function. 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 (dynamic single assignment). The immutability of entries within a data collection is necessary for determinism. An instance in data collection x with tag "i, j" is denoted $[x : i, j]$.

The colon notation above can also be used to specify *tag functions* in CnC. These are declarative contracts that constrain the data access patterns of steps. For example, a step indexed by an integer i which promises to read data at i and produce $i + 1$ would be written as "`[x: i]` → `(f: i)` → `[x: i+1]`".

Because control 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 read and write operations on collections as *gets* and *puts*.)

2.1 Simple Example

The following simple example illustrates the task and data parallel capabilities of CnC. This application takes a set (or stream) of strings as input. Each string is split into words (separated by spaces). Each word then passes through a second phase of processing that, in this case, puts it in uppercase form.

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

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

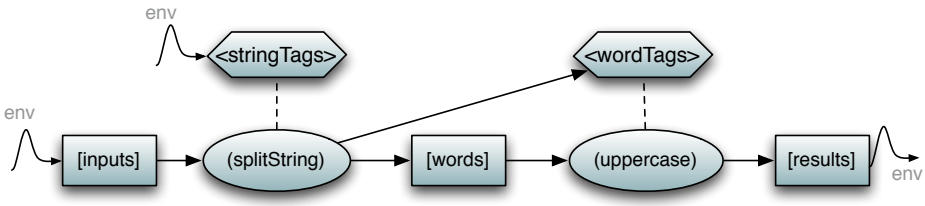

Figure 1: A CnC graph as described by a CnC specification. By convention, in the graphical notation specific shapes correspond to control, data, and step collections. Dotted edges represent prescription (control/step relations), and arrows represent production and consumption of data. Squiggly edges represent communication with the environment (the program outside of CnC)

The only keyword in the CnC specification language is `env`, which refers to the *environment*—the world outside CnC, for example, other threads or processes written in a serial language. The strings passed into CnC from the environment are placed into `[inputs]` using any unique identifier as a tag. The elements of `[inputs]` may be provided in any order or in parallel. Each string, when split, produces an arbitrary number of words. These per-string outputs can be numbered 1 through N—a pair containing this number and the original string ID serves as a globally unique tag for all output words. Thus, in the specification we could annotate the collections with tag components indicating the pair structure of word tags: e.g. `(uppercase: stringID, wordNum)`.

The step implementations (user-written code for `splitString` and `uppercase` steps, ommitted here due to space constraints), specification file, and code for the environment together make up a complete CnC application. The CnC graph can be specified in a text file using the syntax described above, or it can be constructed using a graphical programming tool[1]. It can also be conveyed in the host language code itself through an API, such as has been done for the Intel® CnC implementation.

3 Formal Semantics

In this section, we introduce Featherweight CnC and describe its semantics, which is used to prove its determinism. Featherweight CnC simplifies the full CnC model without reducing its power. A given CnC program can be translated to Featherweight CnC by: (1) combining its data collections into a single, merged collection (differentiated by an added field in the tags); (2) enumerating all tag data types in the program (which assumes countable sets) and thereby mapping them to integers; and (3) translating serial step code to the simple (but Turing complete) step language embedded in Featherweight CnC.

[1]Since the 0.5.0 release, Intel® has introduced a preliminary graphical programming tool for creating CnC programs in Windows

3.1 Syntax

Featherweight CnC combines the language for specifying data and step collections with the base language for writing the step bodies. A Featherweight CnC program is of the form:

```
f1(int a) {d1 s1}
f2(int a) {d2 s2}
 ...
fn(int a) {dn sn}
```

Combining the languages into one grammar allows us to give a complete semantics for the execution of a Featherweight CnC program. The full grammar is shown below. We use c to range over integer constants and n to range over variable names.

$$
\begin{array}{lrcll}
\textit{Program}: & p & ::= & f_i(\textit{int } a)\{d_i\ s_i\}, i \in 1..n & \\
\textit{Declaration}: & d & ::= & n = \textit{data.get}(e); d & \\
 & & | & \epsilon & \\
\textit{Statement}: & s & ::= & \textit{skip} & \\
 & & | & \textit{if } (e > 0)\ s_1\ \textit{else}\ s_2 & \\
 & & | & \textit{data.put}(e_1, e_2); s & \\
 & & | & \textit{prescribe } f_i(e); s & \\
\textit{Expression}: & e & ::= & c & \text{(integer constant)} \\
 & & | & n & \text{(local name)} \\
 & & | & a & \text{(formal parameter name)} \\
 & & | & e_1 + e_2 &
\end{array}
$$

Each f_i is a step, the a is the tag passed to a step instance, and the d_i and s_i make up the step body. The d_i are the initial *get*s for the data needed by the step and the s_i are the statements in the step bodies. Writing a data item to a data collection is represented by the $\textit{data.put}(e1, e2)$ production. The expression $e1$ is used to compute the tag and the expression $e2$ is used to compute the value. Control collections have been removed in favor of directly starting a new step using the *prescribe* statement. A computation step consists of a series of zero or more declarations followed by one or more statements. These declarations introduce names local to the step and correspond to performing a *get* on a data collection. A step must finish all of its *get*s before beginning any computation. The computation in a step is limited to simple arithmetic, writing to a data collection, and starting new computation steps.

3.2 Semantics

For an expression e in which no names occur, we use $[\![e]\!]$ to denote the integer to which e evaluates. Further, we use A to denote the state of the array *data*, a partial function whose domain and range are integers. If $A[i]$ is undefined, we say that $A[i] = \perp$, and we use $A[n_1 := n_2]$ to extend A. We use A_0 to denote the empty mapping, where $dom(A) = \{\}$. We define an ordering $\sqsubseteq$ on array mappings such that $A \sqsubseteq A'$ if and only if $dom(A) \subseteq dom(A')$ and for all $n \in dom(A) : A(n) = A'(n)$.

We will now define a small-step operational semantics for the language. Our main semantic structure is a tree defined by the following grammar:

$$Tree: \quad T \quad ::= \quad T \parallel T \quad | \quad (d\ s)$$

We assert that $\parallel$ is associative and commutative, that is

$$\begin{aligned} T_1 \parallel (T_2 \parallel T_3) &= (T_1 \parallel T_2) \parallel T_3 \\ T_1 \parallel T_2 &= T_2 \parallel T_1 \end{aligned}$$

A state in the semantics is a pair (A, T) or error. We use σ to range over states. We define an ordering $\leq$ on states such that $\sigma \leq \sigma'$ if and only if either $\sigma' = \mathsf{error}$, or if $\sigma = (A, T)$ and $\sigma' = (A', T)$, then $A \sqsubseteq A'$.

We will define the semantics via a binary relation on states, written $\sigma \to \sigma'$. The initial state of an execution of s is (A_0, s). A final state of the semantics is either of the form $(A, skip)$, of the form error, or of the form (A, T) in which every occurrence in T of $(d\ s)$ has that property that d is of the form $n = data.get(e); d'$ and $A[\![e]\!] = \bot$ (e.g. a "blocked" read).

We now show the rules that define $\to$.

$$(A, skip \parallel T_2) \to (A, T_2) \qquad (1) \qquad\qquad (A, T_1 \parallel skip) \to (A, T_1) \qquad (2)$$

$$\frac{(A, T_1) \to \mathsf{error}}{(A, T_1 \parallel T_2) \to \mathsf{error}} \qquad (3) \qquad\qquad \frac{(A, T_2) \to \mathsf{error}}{(A, T_1 \parallel T_2) \to \mathsf{error}} \qquad (4)$$

$$\frac{(A, T_1) \to (A', T_1')}{(A, T_1 \parallel T_2) \to (A', T_1' \parallel T_2)} \qquad (5) \qquad\qquad \frac{(A, T_2) \to (A', T_2')}{(A, T_1 \parallel T_2) \to (A', T_1 \parallel T_2')} \qquad (6)$$

$$(A, if\ (e > 0)\ s_1\ else\ s_2) \to (A, s_1) \qquad (\text{if } [\![e]\!] > 0) \qquad (7)$$

$$(A, if\ (e > 0)\ s_1\ else\ s_2) \to (A, s_2) \qquad (\text{if } [\![e]\!] \leq 0) \qquad (8)$$

$$(A, data.put(e_1, e_2); s) \to (A[[\![e_1]\!] := [\![e_2]\!]], s) \quad (\text{if } A[[\![e_1]\!]] = \bot) \qquad (9)$$

$$(A, data.put(e_1, e_2); s) \to \mathsf{error} \quad (\text{if } A[[\![e_1]\!]] \neq \bot) \qquad (10)$$

$$(A, prescribe\ f_i(e); s) \to (A, ((d_i\ s_i)[a := [\![e]\!]]) \parallel s) \quad (\text{the body of } f_i \text{ is } (d_i\ s_i)) \qquad (11)$$

$$(A, n = data.get(e); d\ s) \to (A, (d\ s)[n := A[[\![e]\!]]]) \quad (\text{if } A[[\![e]\!]] \neq \bot) \qquad (12)$$

Notice that because Rules (11) and (12) perform substitution on step bodies, in a well-formed program all evaluated expressions, $[\![e]\!]$, are closed. Now, given the semantics above, we formally state the desired property of determinism:

If $\sigma \to^* \sigma'$ and $\sigma \to^* \sigma''$, and σ', σ'' are both final states, then $\sigma' = \sigma''$.

3.2.1 Proof of Determinism

Lemma 3.1. (Error Preservation) *If* $(A, T) \rightarrow \mathsf{error}$ *and* $A \sqsubseteq A'$*, then* $(A', T) \rightarrow \mathsf{error}$*.*

Proof. Straightforward by induction on the derivation of $(A, T) \rightarrow \mathsf{error}$; we omit the details. □

Lemma 3.2. (Monotonicity) *If* $\sigma \rightarrow \sigma'$*, then* $\sigma \leq \sigma'$*.*

Proof. Straightforward by induction on the derivation of $\sigma \rightarrow \sigma'$. The interesting case is for Rule (9) which is where σ can change and the single-assignment side-condition plays an essential role. We omit the details. □

Lemma 3.3. (Clash) *If* $(A, T) \rightarrow (A', T')$ *and* $A[c] = \bot$ *and* $A'[c] \neq \bot$ *and* $A_d[c] \neq \bot$ *and then* $(A_d, T) \rightarrow \mathsf{error}$*.*

Proof. Straightforward by induction on the derivation of $(A, T) \rightarrow (A', T')$; we omit the details. □

Lemma 3.4. (Independence) *If* $(A, T) \rightarrow (A', T')$ *and* $A'[c] = \bot$*, then* $(A[c := c'], T) \rightarrow (A'[c := c'], T')$*. ,*

Proof. From $(A, T) \rightarrow (A', T')$ and Lemma 3.2 we have $A \sqsubseteq A'$. From $A \sqsubseteq A'$ and and $A'[c] = \bot$, we have $A[c] = \bot$. The proof is now straightforward by induction on the derivation of $(A, T) \rightarrow (A', T')$; we omit the details. □

Lemma 3.5. (Diamond) *If* $(A, T_a) \rightarrow (A', T'_a)$ *and* $(A, T_b) \rightarrow (A'', T''_b)$*, then there exists* σ_c *such that* $(A', T'_a \parallel T_b) \rightarrow \sigma_c$ *and* $(A'', T_a \parallel T''_b) \rightarrow \sigma_c$*.*

Proof. We proceed by induction on the derivation of $(A, T_a) \rightarrow (A', T'_a)$. We have twelve cases depending on the last rule used to derive $(A, T_a) \rightarrow (A', T'_a)$.

- Rule (1). In this case we have $T_a = (\mathit{skip} \parallel T_2)$ and $A' = A$ and $T'_a = T_2$. So we can pick $\sigma_c = (A'', T_2 \parallel T''_b)$ because $(A', T'_a \parallel T_b) = (A, T_2 \parallel T_b)$ and from $(A, T_b) \rightarrow (A'', T''_b)$ and Rule (6) we have $(A, T_2 \parallel T_b) \rightarrow (A'', T_2 \parallel T''_b)$, and because $(A'', T_a \parallel T''_b) = (A'', (\mathit{skip} \parallel T_2) \parallel T''_b)$ and from Rule (1) we have $(A'', (\mathit{skip} \parallel T_2)) \rightarrow (A'', T_2)$, and finally from $(A'', (\mathit{skip} \parallel T_2)) \rightarrow (A'', T_2)$ and Rule (5) we have $(A'', (\mathit{skip} \parallel T_2) \parallel T''_b)) \rightarrow (A'', T_2 \parallel T''_b)$.
- Rule (2). This case is similar to the previous case; we omit the details.
- Rules (3)–(4). Both cases are impossible.
- Rule (5). In this case we have $T_a = T_1 \parallel T_2$ and $T'_a = T'_1 \parallel T_2$ and $(A, T_1) \rightarrow (A', T'_1)$. From $(A, T_1) \rightarrow (A', T'_1)$ and $(A, T_b) \rightarrow (A'', T''_b)$ and the induction hypothesis, we have σ'_c such that $(A', T'_1 \parallel T_b) \rightarrow \sigma'_c$ and $(A'', T_1 \parallel T''_b) \rightarrow \sigma'_c$. Let us show that we can pick σ_c such that $(A', (T'_1 \parallel T_b) \parallel T_2) \rightarrow \sigma_c$ and $(A'', (T_1 \parallel T''_b) \parallel T_2) \rightarrow \sigma_c$. We have two cases:
  - If $\sigma'_c = \mathsf{error}$, then we use Rule (3) to pick $\sigma_c = \mathsf{error}$.
  - If $\sigma'_c = (A_c, T_c)$, then then we use Rule (5) to pick $(A_c, T_c \parallel T_2)$.

From $(A', (T_1' \parallel T_b) \parallel T_2) \to \sigma_c$ and $(A'', (T_1 \parallel T_b'') \parallel T_2) \to \sigma_c$, the result then follows from $\parallel$ being associative and commutative.

- Rules (6)–(8). All three cases are similar to the case of Rule (1); we omit the details.
- Rule (9). In this case we have $T_a = (item.put(e_1, e_2); s)$ and $A' = A[[\![e_1]\!] := [\![e_2]\!]]$ and $T_a' = s$ and $(A[[\![e_1]\!]] = \bot)$. Let us do a case analysis of the last rule used to derive $(A, T_b) \to (A'', T_b'')$.
  - Rule (1). In this case we have $T_b = skip \parallel T_2$ and $A'' = A$ and $T_b'' = T_2$. So we can pick $\sigma_c = (A', s \parallel T_2)$ because $(A', T_a' \parallel T_b) = (A', s \parallel (skip \parallel T_2))$ and from Rule (6) and Rule (1) we have $(A', s \parallel (skip \parallel T_2)) \to (A', s \parallel T_2) = \sigma_c$, and because $(A'', T_a \parallel T_b'') = (A, T_a \parallel T_2)$ and from Rule (5) we have $(A, T_a \parallel T_2) \to (A', s \parallel T_2) = \sigma_c$.
  - Rule (2). This case is similar to the previous case; we omit the details.
  - Rule (3)–(4). Both cases are impossible.
  - Rule (5). In this case we have $T_b = T_1 \parallel T_2$ and $T_b'' = T_1'' \parallel T_2$ and $(A, T_1) \to (A'', T_1'')$. We have two cases:
    - If $[\![e_1]\!] \in dom(A'')$, then we can pick $\sigma_c = \mathsf{error}$ because from $(A''[[\![e_1]\!]] \neq \bot)$ and Rule (10) and Rule (5) we have $(A'', T_a \parallel T_b'') \to \sigma_c$, and because from $(A, T_b) \to (A'', T_b'')$ and $(A[[\![e_1]\!]] = \bot)$ and $(A''[[\![e_1]\!]] \neq \bot)$ and $(A'[[\![e_1]\!]] \neq \bot)$ and Lemma 3.3, we have $(A', T_b) \to \mathsf{error}$, and so from Rule (4) we have $(A', T_a' \parallel T_b) \to \sigma_c$.
    - If $[\![e_1]\!] \notin dom(A'')$, then we define $A_c = A''[[\![e_1]\!] := [\![e_2]\!]]$ and we pick $\sigma_c = (A_c, T_a' \parallel (T_1'' \parallel T_2))$. From $(A[[\![e_1]\!]] = \bot)$ and $(A, T_b) \to (A'', T_b'')$ and $[\![e_1]\!] \notin dom(A'')$ and Lemma 3.4, we have $(A', T_b) \to (A_c, T_b'')$, and then from Rule (6) we have $(A', T_a' \parallel T_b) \to (A_c, T_a' \parallel T_b'') = \sigma_c$. From Rule (6) and Rule (9) and $[\![e_1]\!] \notin dom(A'')$, we have $(A'', T_a \parallel T_b'') \to \sigma_c$.
  - Rule (6). This case is similar to the previous case; we omit the details.
  - Rule (7)–(8). Both cases are similar to the case of Rule (1); we omit the details.
  - Rule (9). In this case we have $T_b = (item.put(e_1', e_2'); s')$ and $A'' = A[[\![e_1']\!] := [\![e_2']\!]]$ and $T_b'' = s'$ and $(A[[\![e_1']\!]] = \bot)$. We have two cases:
    - If $[\![e_1]\!] = [\![e_1']\!]$, then we can pick $\sigma_c = \mathsf{error}$ because $(A'[[\![e_1]\!]] \neq \bot)$ and $(A''[[\![e_1]\!]] \neq \bot)$ and from Rule (10) we have both $(A', T_a' \parallel T_b) \to \sigma_c$ and $(A'', T_a \parallel T_b'') \to \sigma_c$.
    - If $[\![e_1]\!] \neq [\![e_1']\!]$, then we define $A_c = A[[\![e_1]\!] := [\![e_2]\!]][[\![e_1']\!] := [\![e_2']\!]]$ and we pick $\sigma_c = (A_c, s \parallel s')$. From $(A[[\![e_1']\!]] = \bot)$ and $[\![e_1]\!] \neq [\![e_1']\!]$, we have $(A'[[\![e_1']\!]] = \bot)$. From Rule (6) and Rule (9) and $(A'[[\![e_1']\!]] = \bot)$ we have $(A', T_a' \parallel T_b) \to \sigma_c$. From $(A[[\![e_1]\!]] = \bot)$ and $[\![e_1]\!] \neq [\![e_1']\!]$, we have $(A''[[\![e_1]\!]] = \bot)$. From Rule (5) and Rule (9) and $(A''[[\![e_1]\!]] = \bot)$

we have $(A'', T_a \parallel T_b'') \to \sigma_c$.

- Rule (10). This case is impossible.
- Rule (11)–(12). Both cases are similar to the case of Rule (1); we omit the details.

- Rule (10). This case is impossible.
- Rules (11)–(12). Both of cases are similar to the case of Rule (1); we omit the details.

□

The standard notion of Local Confluence says that: if $\sigma \to \sigma'$ and $\sigma \to \sigma''$, then there exists σ_c such that $\sigma' \to^* \sigma_c$ and $\sigma'' \to^* \sigma_c$. We will prove a stronger property that we call Strong Local Confluence.

Lemma 3.6. (Strong Local Confluence) *If $\sigma \to \sigma'$ and $\sigma \to \sigma''$, then there exists σ_c, i, j such that $\sigma' \to^i \sigma_c$ and $\sigma'' \to^j \sigma_c$ and $i \leq 1$ and $j \leq 1$.*

Proof. We proceed by induction on the derivation of $\sigma \to \sigma'$. We have twelve cases depending on the last rule use to derive $\sigma \to \sigma'$.

- Rule (1). We have $\sigma = (A, \mathit{skip} \parallel T_2)$ and $\sigma' = (A, T_2)$. Let us do a case analysis of the last rule used to derive $\sigma \to \sigma''$.
  - Rule (1). In this case, $\sigma' = \sigma''$, and we can then pick $\sigma_c = \sigma'$ and $i = 0$, and $j = 0$.
  - Rule (2). In this case we must have $T_2 = \mathit{skip}$ and $\sigma'' = (A, \mathit{skip})$. So $\sigma' = \sigma''$ and we can pick $\sigma_c = \sigma'$ and $i = 0$, and $j = 0$..
  - Rule (3). This case is impossible because it requires (A, skip) to take a step.
  - Rule (4). In this case we have $\sigma'' = \mathsf{error}$ and $(A, T_2) \to \mathsf{error}$. So we can pick $\sigma_c = \mathsf{error}$ and $i = 1$ and $j = 0$ because $(A, T_2) \to \mathsf{error}$ is the same as $\sigma' \to \sigma_c$ and $\sigma'' = \sigma_c$.
  - Rule (5). This case is impossible because it requires *skip* to take a step.
  - Rule (6). In this case we have $\sigma'' = (A', \mathit{skip} \parallel T_2')$ and $(A, T_2) \to (A', T_2')$. So we can pick $\sigma_c = (A', T_2')$ and $i = 1$ and $j = 1$ because from Rule (1) we have $\sigma'' \to \sigma_c$, and we also have that $(A, T_2) \to (A', T_2')$ is the same as $\sigma' \to \sigma_c$.
  - Rules (7)–(12). Each of these is impossible because $T = \mathit{skip} \parallel T_2$.
- Rule (2). This case is similar to the previous case; we omit the details.
- Rule (3). We have $\sigma = (A, T_1 \parallel T_2)$ and $\sigma' = \mathsf{error}$ and $(A, T_1) \to \mathsf{error}$. Let us do a case analysis of the last rule used to derive $\sigma \to \sigma''$.
  - Rule (1). This case impossible because it requires $T_1 = \mathit{skip}$ which contradicts $(A, T_1) \to \mathsf{error}$.
  - Rule (2). In this case we have $T_2 = \mathit{skip}$ and $\sigma'' = (A, T_1)$. So we can

pick $\sigma_c = \mathsf{error}$ and $i = 0$ and $j = 1$ because $\sigma' = \sigma_c$ and $(A, T_1) \rightarrow \mathsf{error}$ is the same as $\sigma'' \rightarrow \sigma_c$.

- Rule (3). In this case, $\sigma' = \sigma''$, and we can then pick $\sigma_c = \sigma'$ and $i = 0$ and $j = 0$.
- Rule (4). In this case, $\sigma' = \sigma''$, and we can then pick $\sigma_c = \sigma'$ and $i = 0$ and $j = 0$.
- Rule (5). In this case we have $(A, T_1) \rightarrow (A', T_1')$ and $\sigma'' = (A', T_1' \parallel T_2)$. From the induction hypothesis we have that there exists σ_c' and $i' \leq 1$ and $j' \leq 1$ such that $\mathsf{error} \rightarrow^{i'} \sigma_c'$ and $(A', T_1') \rightarrow^{j'} \sigma_c'$. Given that error has no outgoing transitions, we must have $\sigma_c' = \mathsf{error}$ and $i' = 0$. Additionally, given that $(A', T_1') \neq \mathsf{error}$ we must have $j' = 1$. So we can pick $\sigma_c = \mathsf{error}$ and $i = 0$ and $j = 1$ because $\sigma' = \sigma_c$ and because from Rule (3) and $(A', T_1') \rightarrow^{j'} \sigma_c'$ and $j' = 1$, we have $\sigma'' \rightarrow \mathsf{error}$.
- Rule (6). In this case we have $(A, T_2) \rightarrow (A', T_2')$ and $\sigma'' = (A', T_1 \parallel T_2')$. In this case we can pick $\sigma_c = \mathsf{error}$ and $i = 0$ and $j = 1$ because $\sigma' = \sigma_c$ and because from $(A, T_1) \rightarrow \mathsf{error}$ and $(A, T_2) \rightarrow (A', T_2')$ and Lemma 3.2 and Lemma 3.1 we have $(A', T_1) \rightarrow \mathsf{error}$, so from Rule (3) we have we have $\sigma'' \rightarrow \sigma_c$.
- Rules (7)–(12). Each of these is impossible because $T = T_1 \parallel T_2$.

- Rule (4). This case is similar to the previous case; we omit the details.
- Rule (5). We have $\sigma = (A, T_1 \parallel T_2)$ and $\sigma' = (A', T_1' \parallel T_2)$ and $(A, T_1) \rightarrow (A', T_1')$. Let us do a case analysis of the last rule used to derive $\sigma \rightarrow \sigma''$.
  - Rule (1). This case is impossible because for Rule (1) to apply we must have $T_1 = \mathit{skip}$, but for Rule (5) to apply we must have that (A, T_1) can take a step, a contradiction.
  - Rule (2). In this case, we must have $T_2 = \mathit{skip}$ and $\sigma'' = (A, T_1)$. So we can pick $\sigma_c = (A', T_1')$ and $i = 1$ and $j = 1$ because from Rule (2) we have $\sigma' \rightarrow \sigma_c$, and we also have that $(A, T_1) \rightarrow (A', T_1')$ is the same as $\sigma'' \rightarrow \sigma_c$.
  - Rule (3). In this case we have $(A, T_1) \rightarrow \mathsf{error}$ and $\sigma'' = \mathsf{error}$. From the induction hypothesis we have that there exists σ_c' and $i' \leq 1$ and $j' \leq 1$ such that $\mathsf{error} \rightarrow^{i'} \sigma_c'$ and $(A', T_1') \rightarrow^{j'} \sigma_c'$. Given that error has no outgoing transitions, we must have $\sigma_c' = \mathsf{error}$ and $i' = 0$. Additionally, given that $(A', T_1') \neq \mathsf{error}$ we must have $j' = 1$. So we can pick $\sigma_c = \mathsf{error}$ and $i = 1$ and $j = 0$ because $\sigma'' = \sigma_c$ and because from Rule (3) and $(A', T_1') \rightarrow^{j'} \sigma_c'$ and $j' = 1$, we have $\sigma' \rightarrow \mathsf{error}$.
  - Rule (4). In this case we have $(A, T_2) \rightarrow \mathsf{error}$ and $\sigma'' = \mathsf{error}$. So we can pick $\sigma_c = \mathsf{error}$ and $i = 1$ and $j = 0$ because $\sigma'' = \mathsf{error}$ and because from $(A, T_2) \rightarrow \mathsf{error}$ and $(A, T_1) \rightarrow (A', T_1')$ and Lemma 3.2 and Lemma 3.1 we have $(A', T_2) \rightarrow \mathsf{error}$, so from Rule (4) we have $\sigma' \rightarrow \mathsf{error}$.
  - Rule (5). In this case we must have $\sigma'' = (A'', T_1'' \parallel T_2)$ and $(A, T_1) \rightarrow$

(A'', T_1''). From $(A, T_1) \to (A', T_1')$ and $(A, T_1) \to (A'', T_1'')$, and the induction hypothesis, we have that there exists σ_c' and $i' \leq 1$ and $j' \leq 1$ such that $(A', T_1') \to^{i'} \sigma_c'$ and $(A'', T_1'') \to^{j'} \sigma_c'$. We have two cases.

1. If $\sigma_c' = \mathsf{error}$, then we can pick $\sigma_c = \mathsf{error}$ and $i = 1$ and $j = 1$ because from $(A', T_1') \to^{i'} \sigma_c'$ and $i' \leq 1$ we must have $i' = 1$ so from $(A', T_1') \to^{i'} \sigma_c'$ and $i' = 1$ and Rule (3) we have $\sigma' \to \sigma_c$, and because from $(A'', T_1'') \to^{j'} \sigma_c'$ and $j' \leq 1$ we must have $j' = 1$ so from $(A'', T_1'') \to^{j'} \sigma_c'$ and $j' = 1$ and Rule (3) we have $\sigma'' \to \sigma_c$.
2. If $\sigma_c' = (A_c, T_c)$, then we can pick $\sigma_c = (A_c, T_c \parallel T_2)$ and $i = i'$ and $j = j'$ because from $(A', T_1') \to^{i'} \sigma_c'$ and Rule (5) we have $\sigma' \to^{i} \sigma_c$, and because from $(A'', T_1'') \to^{j'} \sigma_c'$ and Rule (5) we have $\sigma'' \to^{i} \sigma_c$.

- Rule (6). In this case we must have $\sigma'' = (A'', T_1 \parallel T_2')$ and $(A, T_2) \to (A'', T_2')$. From $(A, T_1) \to (A', T_1')$ and $(A, T_2) \to (A'', T_2')$ and Lemma 3.5, we have that there exists σ_c such that $(A', T_1' \parallel T_2) \to \sigma_c$ and $(A'', T_1 \parallel T_2') \to \sigma_c$, that is, $\sigma' \to \sigma_c$ and $\sigma'' \to \sigma_c$. Thus we pick $i = 1$ and $j = 1$.
- Rules (7)–(12). Each of these is impossible because $T = T_1 \parallel T_2$.

- Rule (6). This case is similar to the previous case; we omit the details.
- Rules (7)–(12). In each of these cases, only one step from σ is possible so $\sigma' = \sigma''$ and we can then pick $\sigma_c = \sigma'$ and $i = 0$ and $j = 0$.

□

Lemma 3.7. (Strong One-Sided Confluence) *If $\sigma \to \sigma'$ and $\sigma \to^m \sigma''$, where $1 \leq m$, then there exists σ_c, i, j such that $\sigma' \to^i \sigma_c$ and $\sigma'' \to^j \sigma_c$ and $i \leq m$ and $j \leq 1$.*

Proof. We proceed by induction on m. In the base case of $m = 1$, then result is immediate from Lemma 3.6. In the induction step, suppose $\sigma \to^m \sigma'' \to \sigma'''$ and suppose the lemma holds for m. From the induction hypothesis, we have there exists σ_c', i', j' such that $\sigma' \to^{i'} \sigma_c'$ and $\sigma'' \to^{j'} \sigma_c'$ and $i' \leq m$ and $j' \leq 1$. We have two cases.

- If $j' = 0$, then $\sigma'' = \sigma_c'$. We can then pick $\sigma_c = \sigma'''$ and $i = i' + 1$ and $j = 0$.
- If $j' = 1$, then from $\sigma'' \to \sigma'''$ and $\sigma'' \to^{j'} \sigma_c'$ and Lemma 3.6, we have σ_c'' and i'' and j'' such that $\sigma''' \to^{i''} \sigma_c''$ and $\sigma_c' \to^{j''} \sigma_c''$ and $i'' \leq 1$ and $j'' \leq 1$. So we also have $\sigma' \to^{i'} \sigma_c' \to^{j''} \sigma_c''$. In summary we pick $\sigma_c = \sigma_c''$ and $i = i' + j''$ and $j = i''$, which is sufficient because $i = i' + j'' \leq m + 1$ and $j = i'' \leq 1$.

□

Lemma 3.8. (Strong Confluence) *If $\sigma \to^n \sigma'$ and $\sigma \to^m \sigma''$, where $1 \leq n$ and $1 \leq m$, then there exists σ_c, i, j such that $\sigma' \to^i \sigma_c$ and $\sigma'' \to^j \sigma_c$ and $i \leq m$ and $j \leq n$.*

Proof. We proceed by induction on n. In the base case of $n = 1$, then result is immediate from Lemma 3.7. In the induction step, suppose $\sigma \rightarrow^n \sigma' \rightarrow \sigma'''$ and suppose the lemma holds for n. From the induction hypothesis, we have there exists σ'_c, i', j' such that $\sigma' \rightarrow^{i'} \sigma'_c$ and $\sigma'' \rightarrow^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq n$. We have two cases.

- If $i' = 0$, then $\sigma' = \sigma'_c$. We can then pick $\sigma_c = \sigma'''$ and $i = 0$ and $j = j' + 1$.
- If $i' \geq 1$, then from $\sigma' \rightarrow \sigma'''$ and $\sigma' \rightarrow^{i'} \sigma'_c$ and Lemma 3.7, we have σ''_c and i'' and j'' such that $\sigma''' \rightarrow^{i''} \sigma''_c$ and $\sigma'_c \rightarrow^{j''} \sigma''_c$ and $i'' \leq i'$ and $j'' \leq 1$. So we also have $\sigma'' \rightarrow^{j'} \sigma'_c \rightarrow^{j''} \sigma''_c$. In summary we pick $\sigma_c = \sigma''_c$ and $i = i''$ and $j = j' + j''$, which is sufficient because $i = i'' \leq i' \leq m$ and $j = j' + j'' \leq n + 1$.

□

Lemma 3.9. (Confluence) *if $\sigma \rightarrow^* \sigma'$ and $\sigma \rightarrow^* \sigma''$, then there exists σ_c such that $\sigma' \rightarrow^* \sigma_c$ and $\sigma'' \rightarrow^* \sigma_c$.*

Proof. Strong Confluence (Lemma 3.8) implies Confluence. □

Theorem 1. (Determinism) *If $\sigma \rightarrow^* \sigma'$ and $\sigma \rightarrow^* \sigma''$, and σ', σ'' are both final states, then $\sigma' = \sigma''$.*

Proof. We have from Lemma 3.9 that there exists σ_c such that $\sigma' \rightarrow^* \sigma_c$ and $\sigma'' \rightarrow^* \sigma_c$. Given that neither σ' or σ'' have any outgoing transitions, we must have $\sigma' = \sigma_c$ and $\sigma'' = \sigma_c$, hence $\sigma' = \sigma''$. □

The key language feature that enables determinism is the single assignment condition. The single assignment condition guarantees monotonicity of the data collection A. We view A as a partial function from integers to integers and the single assignment condition guarantees that we can establish an ordering based on the non-decreasing domain of A.

3.3 Discussion

Three key features of CnC are represented directly in the semantics for Featherweight CnC. First, the *single assignment* property only allows one write to a data collection for a given data tag. This property shows up as the side condition of Rules (9) and (10). Second, the *data dependence* property says a step cannot execute until all of the data it needs is available. This property shows up as the side condition of Rule (12). Third, the *control dependence* property, captured by Rule (11), queues a step for execution without saying *when* it will execute.

Turing completeness: We argue that the language provided for writing step bodies is powerful enough to encode the set of all *while* programs, which are known to be Turing complete. While programs have a very simple grammar consisting of a while loop, a single variable x, assigning zero to x, and incrementing x by one. We can write a translator that will convert a while program

to a Featherweight CnC program by using recursive *prescribe* statements to encode the while loop. The value of x can be tracked by explicitly writing the value to a new location in the *data* array at each step and passing the tag for the current location of x to the next step.

Deadlock: We do not claim any freedom from deadlocks in Featherweight CnC. We can see from the semantics that a final state can be one in which there are still steps left to run, but none can make progress because the required data is unavailable. We say the computation has reached a *quiescent* state when this happens. In practice, deadlock is a problem when it happens nondeterministically because it makes errors difficult to detect and correct. Because we have proved that Featherweight CnC is deterministic, any computation that reaches a quiescent final state will always reach that same final state. Therefore, deadlocks become straightforward to detect and correct.

4 Implementing CnC on Different Platforms

Implementations of CnC need to provide a translator and a runtime. The translator uses the CnC specification to generate code for a runtime API in the target language. We have implemented CnC for C++, Java, .NET, and Haskell, but in this paper, we primarily focus on the Java and C++ implementations.

C++: uses Intel®'s Threading Building Blocks (TBB) and provides several schedulers, most of which are based on TBB schedulers, which use work stealing [7]. The C++ runtime is provided as a template library, allowing control and data instances to be of any type.

Java: uses Habanero-Java (HJ)[10], an extension of the X10 language described in [6], as well as primitives from the Java Concurrency Utilities [14], such as `ConcurrentHashMap` and Java *atomic* variables. For a detailed description of the runtime mapping and the code translator from CnC to HJ see [2].

.NET: takes full advantage of language generics to implement type-safe *put* and *get* operations on data and control collections. The runtime and code generator are written in F#, the step bodies can be written in any .NET language (F#, C#, VB.NET, IronPython, etc).

Haskell: uses the work stealing features of the Glasgow Haskell Compiler to implement CnC, and provides an Haskell-embedded CnC sub-language. Haskell both enforces that steps are pure (effect-free, deterministic) and allows complete CnC graphs to be used within pure Haskell functions.

Following are some of the key design decisions made in the above implementations.

Step Execution and Data Puts and Gets: In all implementations, step prescription involves creation of an internal data structure representing the step to be executed. Parallel tasks can be spawned eagerly upon prescription, or delayed until the data needed by the task is ready. The *get* operations on a data collection could be blocking (in cases when the task executing the step is spawned

before all the inputs for the step are available) or non-blocking (the runtime guarantees that the data is available when *get* is executed). Both the C++ and Java implementations have a roll back and replay policy, which aborts the step performing a *get* on an unavailable data item and puts the step in a separate list associated with the failed *get*. When a corresponding *put* gets executed, all the steps in a list waiting on that item are restarted. Both Java and C++ implementations can delay execution of a step until the items are actually available.

Initialization and Shutdown: All implementations require some code for initialization of the CnC graph: creating step objects and a graph object, as well as performing the initial *puts* into the data and control collections. In the C++ implementation, ensuring that all the steps in the graph have finished execution is done by calling the `run()` method on the graph object, which blocks until all runnable steps in the program have completed. In the Java implementation, ensuring that all the steps in the graph have completed is done by enclosing all the control collection *puts* from the environment in an Habanero-Java `finish` construct [10], which ensures that all transitively spawned tasks have completed.

Safety properties: Different CnC implementations are more or less *safe* in terms of enforcing the CnC specification. The C++ implementation performs runtime checks of the single assignment rule, while the Java and .NET implementations also ensure tag immutability and check for CnC *graph conformance* (e.g., a step cannot perform a *put* into an data collection if that relationship is not specified in the CnC graph). Finally, CnC guarantees determinism as long as steps are themselves deterministic—a contract strictly enforceable only in Haskell.

Memory reuse: Releasing data instances is a separate problem from traditional garbage collection. We have two approaches to determine when data instances are dead and can safely be released (without breaking determinism). First, [3] introduces a declarative *slicing annotation* for CnC that can be transformed into a reference counting procedure for memory management. Second, our C++ implementation provides a mechanism for specifying *use counts* for data instances, which are discarded after their last use. (These can sometimes be computed from tag functions, and otherwise are set by the tuning expert.) Irrespective of which of these mechanisms is used, data collections can be released after a graph is finished running. Frequently, an application uses CnC for finite computations inside a serial outer loop, reclaiming all memory between iterations.

Distributed memory: All the above implementations assume a shared memory platform for execution. In addition, Intel®'s C++ implementation provides a prototype of a distributed runtime, which requires only minimal additions to a standard CnC program [17]. The runtime intercepts *puts* and *gets* to tag and item collections. The runtime uses a simple default algorithm to determine the host on which to execute the *put* or *get*, or the user can provide a custom partitioning. Each item has its home on the process which produces it. If a process finds an item unavailable when issuing a *get*, it requests it from all other processes. The programmer can also specify the home of the item, in which case the runtime requests the item only from the home process. The

owner sends the item as soon as it becomes (or is) available. This strategy is generic and does not require additional information from the programmer, such as step and item partitioning or mapping. Still, it is close to optimal for cases with good locality, i.e., those that are good candidates for distributed memory.

5 Experimental Results

In this section, we present experimental results obtained using the C++ and Java implementations of CnC outlined in the previous section.

5.1 CnC-C++ Implementation

We have ported Dedup, a benchmark from the PARSEC [1] benchmark suite, to the TBB-based C++ implementation of CnC. The Dedup kernel compresses a data stream with a combination of global and local compression. The kernel uses a pipelined programming model to mimic real-world implementations.

Figure 2 shows execution time as a function of the number of worker threads used, both for our implementation and a standard pthreads version obtained from the PARSEC site. The figure shows that CnC has superior performance to the pthreads implementation of Dedup. With a fixed number of threads per stage, load imbalance between the stages limits the parallelism in the pthreads implementation. The CnC implementation does not have this issue, as all threads can work on all stages. With pthreads, data is never local to a thread between stages. CnC's depth-first scheduler keeps the data local, and in the FIFO case locality also occurs in our experiment. The use of conditional variables in pthreads is expensive.

We have experimented with two CnC scheduling policies: TBB_TASK and TBB_QUEUE. TBB_TASK wraps a step instance in a `tbb::task object` and eagerly spawns it, thereby delegating the scheduling to TBB without much overhead. TBB_QUEUE provides a global FIFO task queue, populated with scheduled steps that are consumed by multiple threads. This policy is a good match for a pipelined algorithm such as Dedup.

In addition to improved performance, the CnC version also simplifies the work of the programmer, who simply performs *put*s and *get*s without the need to think about lower-level parallelism mechanisms such as explicit threads, mutexes and conditional variables.

The second CnC-C++ example that we evaluated was a Cholesky Factorization [8] of size 2000×2000. The tiled Cholesky algorithm consists of three steps: the conventional sequential Cholesky, triangular solve, and the symmetric rank-k update. These steps can be overlapped with one another after initial factorization of a single block, resulting in an asynchronous-parallel approach. There is also abundant data parallelism within each of these steps.

This is an interesting example because its performance is impacted by both parallelism (number of cores used) and locality (tile size), thereby illustrating how these properties can be taken into account when tuning a CnC program. Figure 3 shows the speedup relative to a sequential version on an 8-way Intel® dual Xeon Harpertown SMP system.

Finally, we have also tested CnC microbenchmarks such as NQueens and

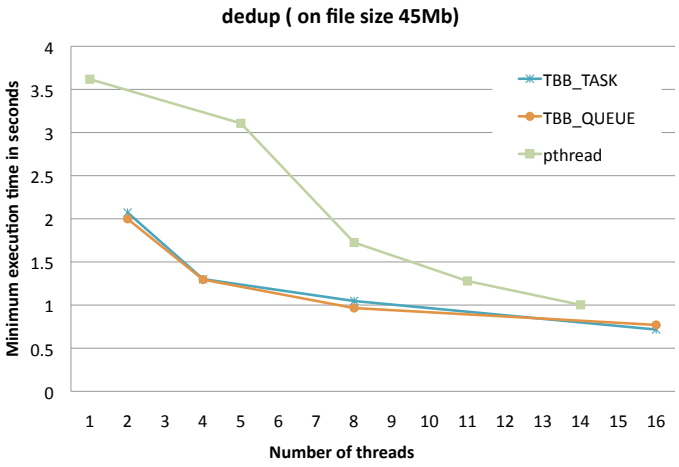

Figure 2: Execution time for pthreads and CnC-C++ implementations of the PARSEC Dedup benchmark with 45MB input size on a 16-core Intel® Caneland SMP, as a function of number of worker threads used.

the Game of Life and observed that the CnC-C++ implementation matched the scalability of the TBB and OpenMP implementations. We omit the detailed performance results for these two benchmarks, but still discuss their behavior below.

NQueens We compared three parallel implementations of the same NQueens algorithm in C++: OpenMP, TBB (`parallel_for`) and CnC. We used the CnC implementation with the default `tbb::parallel_while` scheduler. CnC performed similarly to OpenMP and TBB. TBB and particularly OpenMP provide a convenient method to achieve parallelization with limited scalability. CnC's straightforward specification of NQueens allows extreme scalability, but the fine grain of such steps prevents an efficient implementation. To create sufficiently coarse-grain steps, we used a technique which unrolls the tree to a certain level. We found that implementing this in CnC was more intuitive than with OpenMP and TBB, since with CnC we could express the cutoff semantics in a single place while TBB and OpenMP required additional and artificial constructs at several places.

Game of Life When implementing Game of Life with CnC, grain size was an issue again. With a straightforward specification, a CnC step would work on a cell, but again such a step is too fine grained. For sufficiently coarse granularity, the CnC steps work on tiles rather than on cells. Interestingly, even our OpenMP version shows better performance when applied to the tiled algorithm. CnC performs similarly with OpenMP and TBB(`parallel_for`) on the same algorithm. CnC's potential to concurrently execute across generations results in good scalability.

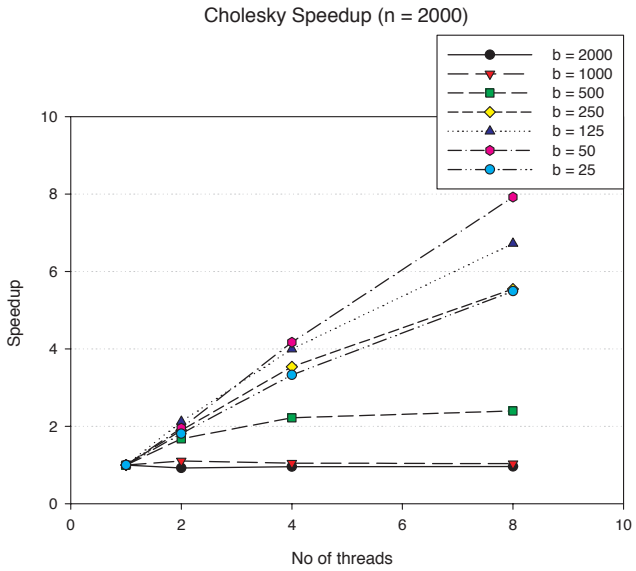

Figure 3: Speedup results for C++ implementation of 2000×2000 Cholesky Factorization CnC program on an 8-way (2p4c) Intel® dual Xeon Harpertown SMP system. The running time for the baseline (1 thread, tile size 2000) was 24.9 seconds

5.2 CnC-Java Implementation

The Blackscholes [1] application is an Intel® RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation. Figure 4 shows the speedup of the HJ CnC Cholesky factorization and Blackscholes implementations on a 16-way Intel® Xeon SMP, using four different strategies for data synchronization when performing *put*s and *get*s on the data collection: (1) coarse-grain blocking on the whole data collection, (2) fine-grain blocking on individual items in the data collection, (3) data-driven computation using a roll-back and replay, and (4) a non-blocking strategy using the `delayed async` construct. The speedups reported in Figure 4 are relative to the sequential Cholesky and Blackscholes implementations in Java. For Cholesky, we observe that the CnC infrastructure adds moderate overhead (28%-41%, depending on the synchronization strategy) in a single thread case, while Blackscholes only shows minimal (1%-2%) overhead. Data-driven and non-blocking strategies scale very well, while we can see the negative effects of coarse-grain and even fine-grain blocking beyond 8 processors. For Blackscholes, since all of the data is available to begin with, and we can evenly partition the data, we do not see much difference between implementation strategies.

Figure 5 shows the speedup for the 64-threads case for Cholesky and Blackscholes on the UltraSPARC T2 (8 cores with 8 threads each). We see very good scaling for Blackscholes using 64-threads but we can start to see the negative effects of the coarse-grain blocking strategy for both applications. Even though

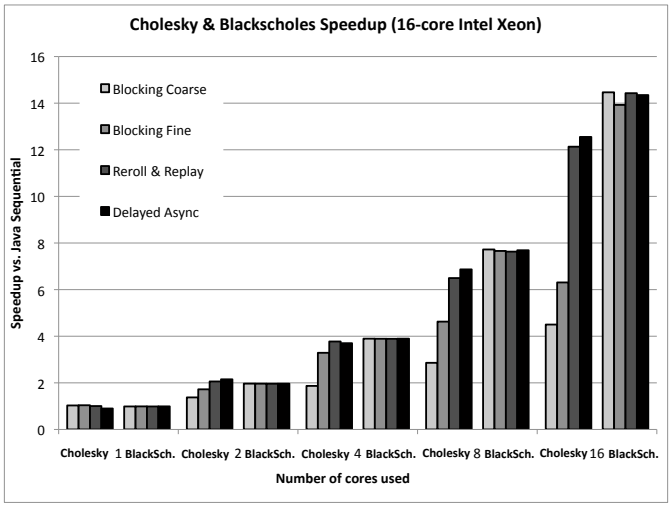

Figure 4: Speedup results for CnC-Java implementation of 2000×2000 Cholesky Factorization CnC program with tile size 100 and Blackscholes CnC program on a 16-way (4p4c) Intel® Xeon SMP system.

the data is partitioned among the workers, the coarse-grain blocking causes contention on data collections which results in worse scalability than the fine-grained and non-blocking versions. Cholesky, which is mostly floating point computation, achieves a better than 8x speedup for 64 threads, which is a very reasonable result considering that the machine has only 8 floating point units.

6 Related Work

We use Table 1 to guide the discussion in this section. This table classifies programming models according to their attributes in three dimensions: *Declarative*, *Deterministic* and *Efficient*. For convenience, we include a few representative examples for each distinct set of attributes, and trust that the reader can extrapolate this discussion to other programming models with similar attributes in these three dimensions.

A number of lower-level programming models in use today — *e.g.,* Intel® TBB [15], .Net Task Parallel Library, Cilk, OpenMP [5], Nvidia CUDA, Java Concurrency [14] — are non-declarative, non-deterministic, and efficient[2]. Deterministic Parallel Java [16] is an interesting variant of Java; it includes a subset that is provably deterministic, as well as constructs that explicitly indicate when determinism cannot be guaranteed for certain code regions, which is why it contains a "hybrid" entry in the *Deterministic* column.

The next three languages in the table — High Performance Fortran (HPF)

[2]We call a programming model efficient if there are known implementations that deliver competitive performance for a reasonably broad set of programs.

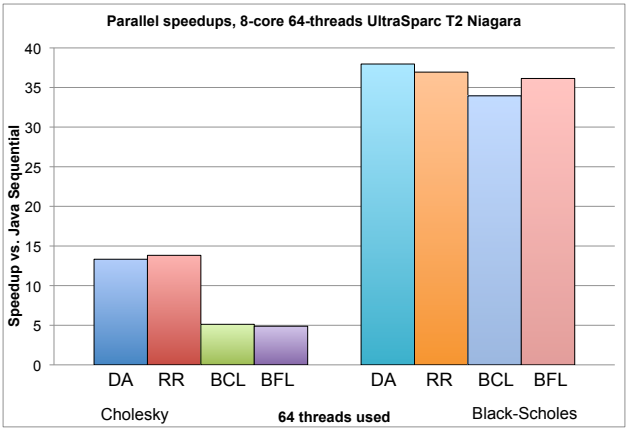

Figure 5: Speedup results for CnC-Java implementation of 2000×2000 Cholesky Factorization CnC program with tile size 80 and Blackscholes CnC program on a 8 core 64 thread UltraSPARC T2 Sun Niagara system. The acronyms stand for Blocking Coarse-Locking (BCL), Blocking Fine-Locking (BFL), Rollback and Replay (RR), and Delayed Async (DA).

[12], X10 [6], Linda [9] — contain hybrid combinations of imperative and declarative programming in different ways. HPF combines a declarative language for data distribution and data parallelism with imperative (procedural) statements, X10 contains a functional subset that supports declarative parallelism, and Linda is a coordination language in which a thread's interactions with the tuple space is declarative. Linda was a major influence on the CnC design, but CnC also differs from Linda in many ways. For example, an `in()` 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.

The last four programming models in the table are both declarative and deterministic. Asynchronous Sequential Processes [4] is a recent model with a clean semantics, but without any efficient implementations. In contrast, the remaining three entries are efficient as well. StreamIt is representative of a modern streaming language, and LabVIEW [18] is representative of a modern dataflow language. Both streaming and dataflow languages have had major influence on the CnC design.

The CnC semantic model is based on dataflow in that steps are functional and execution can proceed whenever data is ready, without unnecessary serialization.

Parallel prog. model	Declarative	Deterministic	Efficient Impl.
Intel TBB [15]	No	No	Yes
.Net Task Par. Lib.	No	No	Yes
Cilk	No	No	Yes
OpenMP [5]	No	No	Yes
CUDA	No	No	Yes
Java Concurrency [14]	No	No	Yes
Det. Parallel Java [16]	No	Hybrid	Yes
High Perf. Fortran [12]	Hybrid	No	Yes
X10 [6]	Hybrid	No	Yes
Linda [9]	Hybrid	No	Yes
Asynch. Seq. Processes [4]	Yes	Yes	No
StreamIt	Yes	Yes	Yes
LabVIEW [18]	Yes	Yes	Yes
CnC [this paper]	Yes	Yes	Yes

Table 1: Comparison of several parallel programming models.

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, data 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 (such as Cholesky factorization) that would not be considered to be dataflow or streaming applications. In summary, CnC has benefited from influences in past work, but we're 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.

For completeness, we also include a brief comparison with graphical coordination languages in a distributed system, using Dryad [11] as an exemplar in that space. Dryad is a general-purpose distributed execution engine for coarse-grain data-parallel applications. It combines sequential vertices with directed communication channels to form an acyclic dataflow graph. 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-controllee relationships and tagged data collections. Also, the CnC implementations described in this paper are focused on multicore rather than distributed systems.

7 Conclusions

This paper presents a programming model for parallel computation. A computation is written as a CnC graph, which is a high-level, declarative representation of semantic constraints. This representation can serve as a bridge between domain and tuning experts, facilitating their communication by hiding information that is not relevant to both parties. We prove deterministic execution of the model. Deterministic execution simplifies program analysis and understanding, and reduces the complexity of compiler optimization, testing, debugging, and tuning for parallel architectures. We also present a set of experiments that show several implementations of CnC with distinct base languages and distinct runtime systems. The experiments confirm that the CnC model can express and exploit a variety of types of parallelism at runtime. When compared to the state of the art lower-level parallel programming models, our experiments indicate that the CnC programming model implementations deliver competitive raw performance and equal or better scalability.

Acknowledgments

We would like to thank our research colleagues at Intel, Rice University, and UCLA for valuable discussions related to this work. We thank Aparna Chandramowlishwaran and Richard Vuduc at Georgia Tech for providing some of the benchmarks evaluated in this paper, and Philippe Charles at Rice for implementing the CnC parser. This research was partially supported by the Center for Domain- Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127 and by a sponsored research agreement between Intel and Rice University.

References

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[2] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. Cnc programming model (extended version). Technical Report TR10-5, Rice University, February 2010.

[3] Zoran Budimlić, Aparna M. Chandramowlishwaran, Kathleen Knobe, Geoff N. Lowney, Vivek Sarkar, and Leo Treggiari. Declarative aspects of memory management in the concurrent collections parallel programming model. In *DAMP '09: the workshop on Declarative Aspects of Multicore Programming*, pages 47–58. ACM, 2008.

[4] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous sequential processes. *Information and Computation*, 207(4):459–495, 2009.

[5] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Programming in OpenMP*. Academic Press, 2001.

[6] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vivek Sarkar, and Christoph Von Praun. X10: An object-oriented

approach to non-uniform cluster computing. In *Proceedings of OOPSLA'05, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 519–538, 2005.

[7] Intel Corporation. Intel(R) Threading Building Blocks reference manual. Document Number 315415-002US, 2009.

[8] A.Chandramowlishwaran et al. Performance evaluation of Concurrent Collections on high-performance multicore computing systems. In *IPDPS '10: International Parallel and Distributed Processing Symposium (To Appear)*, April 2010.

[9] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[10] Habanero multicore software research project. http://habanero.rice.edu.

[11] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

[12] Ken Kennedy, Charles Koelbel, and Hans P. Zima. The rise and fall of High Performance Fortran. In *Proceedings of HOPL'07, Third ACM SIGPLAN History of Programming Languages Conference*, pages 1–22, 2007.

[13] Kathleen Knobe and Carl D. Offner. Tstreams: A model of parallel computation (preliminary report). Technical Report HPL-2004-78, HP Labs, 2004.

[14] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[15] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[16] Jr. Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for Deterministic Parallel Java. In *Proceedings of OOPSLA'09, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 97–116, 2009.

[17] Frank Schlimbach. Distributed CnC for C++. In *Second Annual Workshop on Concurrent Collections*. October 2010. Held in conjunction with LCPC 2010.

[18] Jeffrey Travis and Jim Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun*. Prentice Hall, 2006. 3rd Edition.