Declarative Aspects of Memory Management in the Concurrent Collections Parallel Programming Model

Zoran Budimlić
Rice University
zoran@rice.edu

Aparna Chandramowlishwaran
Georgia Institute of Technology
aparna@cc.gatech.edu

Kathleen Knobe
Intel Corporation
kath.knobe@intel.com

Geoff Lowney
Intel Corporation
geoff.lowney@intel.com

Vivek Sarkar
Rice University
vsarkar@rice.edu

Leo Treggiari
Intel Corporation
leo.treggiari@intel.com

Abstract

Concurrent Collections (CnC)[8] is a declarative parallel language that allows the application developer to express their parallel application as a collection of high-level computations called *steps* that communicate via single-assignment data structures called *items*.

A CnC program is specified in two levels. At the bottom level, an existing imperative language implements the computations within the individual computation steps. At the top level, CnC describes the relationships (ordering constraints) among the steps. The memory management mechanism of the existing imperative language manages data whose lifetime is within a computation step. A key limitation in the use of CnC for long-running programs is the lack of memory management and garbage collection for data items with lifetimes that are longer than a single computation step. Although the goal here is the same as that of classical garbage collection, the nature of problem and therefore nature of the solution is distinct. The focus of this paper is the memory management problem for these data items in CnC.

We introduce a new declarative *slicing annotation* for CnC that can be transformed into a reference counting procedure for memory management. Preliminary experimental results obtained from a Cholesky example show that our memory management approach can result in space reductions for CnC data items of up to 28× relative to the baseline case of standard CnC without memory management.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Languages

Keywords Concurrent Collections, Reference Counts

1. Introduction

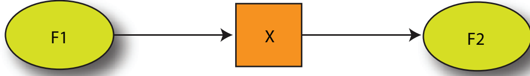
Parallel computing has become mainstream due to the rapid increase in the adoption of multicore processors. Unlike previous generations of hardware evolution, this trend will have a major impact on existing and future software. A highly desirable solution to the multicore software productivity problem is to introduce high-level declarative programming models that are accessible to developers who are experts in different domains but lack deep experience with imperative parallel programming. In previous work [1], we described the Concurrent Collections (CnC) programming model, which builds on past work on TStreams [9]. In the CnC model, the parallel structure of a program is described declaratively in terms of computation *steps* that communicate via data *items* that satisfy the *dynamic single assignment* property. A complete parallel program can be obtained by using a sequential or parallel imperative language to implement individual computation steps; however, the interaction among computation steps still remains side-effect-free thereby making parallelization and fault tolerance easy to support.

In [1], we presented parallel performance results for two different runtime implementations of CnC, one based on C++ and the Intel Threading Building Blocks [4] and the other based on Java and X10 [3]. Though the results demonstrated the feasibility of obtaining scalable parallel speedup with the CnC approach, they also highlighted a key limitation of CnC implementations, viz., *the lack of automatic memory management and garbage collection for data items*. This limitation arises from the use of data items as a single-assignment data interface among computation steps. In particular, in standard CnC when a data item is made available to other computation steps, it is unclear how many computation steps in the future may need to use that data item. In fact, some of these computation steps may not yet exist. Therefore it is unclear when the data item becomes ready for garbage collection. This is a serious limitation for long-running programs because the space requirement for data items can grow unboundedly as a CnC program executes.

Standard garbage collection approaches are insufficient to solve the memory management problem in the CnC programming model, because references (tags) to data items can be created dynamically by future computation steps performing `get` operations. Unless the runtime system is absolutely certain that no future steps will create a reference to the data item, it cannot free the memory occupied by it.

The approach proposed in this paper is to introduce declarative annotations in CnC that indicate which computation step will read a specific data item. These are checked at runtime for potential violations. These *slicing annotations* are converted into reference-counting procedures that can be used to identify dead data items. A single data item may be a scalar or may consist of a large data

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAMP'09, January 20, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-419-5/09/01...$5.00



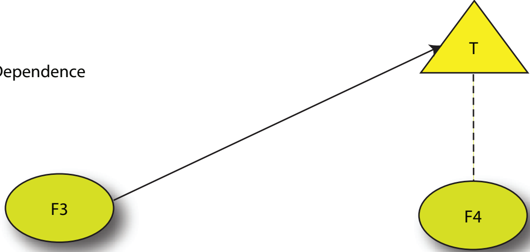


Figure 1. Data and Control dependences in a CnC program

structure. However, the only references permitted to a data item are through *get* operations, thus making the problem of identifying dead CnC data items different from the classical garbage collection problem. Our preliminary experimental results show that our memory management approach can result in space reductions for data items of up to 28× relative to the baseline case of standard CnC without memory management, for the Cholesky example studied in this paper.

The rest of the paper is organized as follows. Section 2 summarizes the CnC model from past work. Section 3 introduces a new declarative *slicing annotation* that serves as a key input for automatic memory management. Section 4 defines the Memory Management problem statement for CnC, and Section 5 describes our solution. Section 6 contains preliminary experimental results, and Section 7 contains our conclusions.

2. Concurrent Collections Programming Model

The goal of the CnC language design is to provide a way to express the semantically required ordering constraints but avoid all the arbitrary ones. This simplifies the mapping to parallel hardware and also makes it more effective. There is a required ordering between two computation steps if one produces data consumed by the other (data dependence) or one determines that the other will execute (control dependence). Arbitrary orderings often appear in other languages in several forms. Programs written using modifiable variables unnecessarily limit the interleavings of distinct assignments and uses. Our system is based on dynamic single assignment to avoid this ordering constraint. Serial languages (and parallel languages embedded within serial languages) often exhibit cases where two code segments are serialized arbitrarily. It is also common to tightly bind the question of *if* a fragment will execute with *when* it will execute. For example, code determines the loop indices for the next body to execute and then *immediately* executes it. Control flow arrives at the point of a call, determines *that* the call will execute and the code *immediately* executes it.

The three constructs in this model are computation *steps*, data *items*, and control *tags*. Statically, each of these constructs is a *collection* representing a set of dynamic *instances*. Step instances are the unit of distribution and scheduling. Item instances are the unit of synchronization and communication. Item instances are the unit of control.

The program is represented as a graph. The computation step, data item and control tag collections are represented as circles, boxes and triangles respectively (See Figure 1). We represent the graph in textual form using `()` to suggest circles for computation steps, `[]` to suggest boxes for data items and `<>` to suggest triangles for control tags.

The edges in the graph specify the partial ordering constraints required by the semantics. One type of ordering constraint arises from a data dependence. This relationship is shown in Figure 1 where an instance of step `(F1)` produces a an instance of item `[X]` consumed by an instance of step `(F2)`. Clearly the producing step instance must occur before the consuming step instance.

Another type of ordering constraint arises from a control dependence, where one computation step determines if another computation step will execute. Since we explicitly do not require that the controlled computation step execute immediately, we need a way of referring to it. For this purpose we introduce the notion of control tags to distinguish among instances of computation steps. These typically have some meaning within the application. They might be loop indices, for example, or they might identify a recursion instance. The control dependence relationship is also shown in Figure 1. Here an instance of computation step `(F3)` produces a control tag instance in `<T>` that controls an instance of computation step `(F4)`. The controller step controls the controllee step. The controller step puts a control tag in the tag collection. The meaning of this is that (not necessarily immediately but) *at some time* the controllee step *with that same tag*, e.g., with the same loop indices, will execute. The controlled computation step has access to its controlling tag much like the body of a loop has access to its loop indices. These will be used to determine which data item instances to consume and which to produce much like a loop body uses the loop indices for array references. If there is a control tag in a tag collection with tag value `k`, then a step with tag `k` will execute. That step might consume items tagged `k-1`, `k` and `k+1`. It might produce a tag value `k+1`.

As opposed to languages that embed parallelism within serial code, in CnC there is no overwriting of the items and no arbitrary serialization among the steps. The data in items is accessed by value, not by location. The items are tagged and obey dynamic single assignment constraint. The steps themselves are implemented in a serial language and are viewed as atomic operations in the model. They are functional and have no side-effects.

2.1 Creating a graph specification

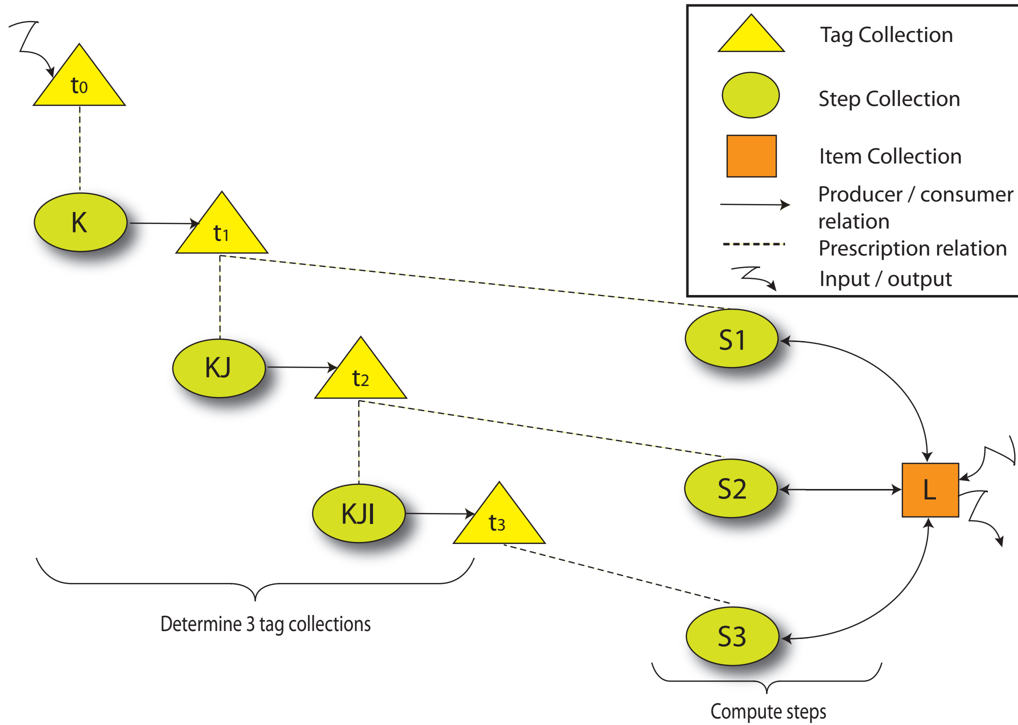
We will introduce the graph specification by showing the process of creating a graph for a specific application. This discussion refers to Figure 2 which shows a simplified graphical representation of a Cholesky factorization example.

Cholesky factorization [2] takes a symmetric positive definite matrix as an input, and factors it into a lower triangular matrix and its transpose. It can be derived by equating corresponding entries of A and LL^T and generating them in order. The input is a symmetric positive definite matrix A and A_{ij} represents a block of size bxb where $b = n/p$ and i ∈ [0,1,...,p-1], j ∈ [0,...,i].

The process below describes how to put an application into CnC form.

2.1.1 Step collections:

The computation is partitioned into high-level operations or step collections. In this application, there are six step collections: three step collections `(k)`, `(kj)` and `(kji)` produce tags `<K>`, `<KJ>` and `<KJI>` for the computation steps. The computation can be broken down into three step collections. The step `(s1)` performs unblocked Cholesky factorization of the input symmetric positive definite tile producing a lower triangular matrix tile. Step `(s2)` applies a triangular system solve on the result of the step `(s1)`. Finally the step `(s3)` is used to update the underlying matrix via a matrix-matrix multiplication.




```
// Declarations
[double** L: int, int, int]; // The in/out matrix
// p and b are global read-only values, not items
global int p;           // Tile loop end value
global int b;         // Tile size

// Tags
<singleton>;
<K>;       //Step 1 indices [k = 0...p-1]
<KJ: int,int>; //Step 2 indices [j = k+1...p-1]
<KJI: int,int,int>;//Step 3 indices [i = k+1...j]

// Step Prescriptions
<singleton> :: (k);
<K> :: (s1),(kj);
<KJ> :: (s2),(kji);
<KJI> :: (s3);

// Input: tile pointers, tile size and loop end value
env -> [L], p, b, <singleton>;
```

```
// Step execution
// The k step produces ’k’ loop indices
p -> (k);
(k) -> <K>;
// The kj step produces ’j’ loop indices
p -> (kj);
(kj) -> <KJ>;
// The kji step produces ’i’ loop indices
p -> (kji);
(kji) -> <KJI>;
// Step 1 Executions
[L: k, k, k], b -> (s1);
(s1) -> [L: k, k, k+1];
// Step 2 Executions
[L: j, k, k], [L: k, k, k+1], b -> (s2);
(s2) -> [L: j, k, k+1];
// Step 3 Executions
[L: j, i, k], [L: j, k, k+1],
[L: i, k, k+1], b -> (s3);
(s3) -> [L: j, i, k+1];
// Return value
[L: k, k, k+1], [L: j, k, k+1] -> env;
```

Figure 2. Graphical and Textual CnC Representations for Cholesky Factorization Example

2.1.2 Item collections and producer-consumer relations:

The data is partitioned into data structures or item collections. In this application there is one item collection, `[L]`, corresponding to the input/output matrix tile. The producer and consumer relationships between step collections and item collections are represented as directed edges between steps and items as shown in Figure 2.

The environment (the code that invokes the graph) may produce and consume items and tags. These relationships are represented by directed squiggly edges in the graphical form and by producer and consumer relations with `env` in the text form. In our application, for example, `env` produces `[L]` input tile items and consumes `[L]` output tile items.

After completing these first two phases the domain expert has a description that is similar to how people communicate informally about their application on a whiteboard. The next two phases are required to make this description precise enough to execute.

2.1.3 Tag components:

The typical computation steps in CnC are not long-lived computations that continually consume input and produce output. Rather, as each step instance is scheduled it consumes item instances, executes, produces item instances and terminates.

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

In our example, the instances of the `[L]` item collection are distinguished by `k`, `j` and `i`.

2.1.4 Tag Collections and Prescriptions:

The specification of the tag components that distinguish among instances is still not precise enough to execute. Knowing that we distinguish instances of `(s1)` steps by values of `k` does not tell us if a `(s1)` step is to be executed for `k = 3`. This control is the role of tag collections.

Tag collections are sets of tag instances. There are three tag collections in our example graph. A tag in `<K>` identifies the tiles step `(s1)` computes on. A tag in `<KJ>` identifies the tiles step `(s2)` computes on and a tag in `<KJI>` identifies the tiles step `(s3)` computes on. A *prescriptive relation* may exist between a tag collection, (`<K>` for example) and a step collection (`(s1)` for example). The meaning of such a relationship is this: if a tag instance `t`, say `k=3`, is in `<K>`, then the step instance `s` in `(s1)` with tag value `k=3`, is guaranteed to (eventually) execute. Notice that the prescription relation mechanism determines *if* a step will execute. *When* and *where* it executes is up to a subsequent scheduler or the tuning expert. A prescriptive relation is shown as a dotted edge between a tag collection and a step collection. The form of the tags for a step collection is identical to the form of the tags of its prescribing tag collection, e.g., instances of the tag collection `<K>` and the step collection `(s1)` are both distinguished by `k`. The task of `(s2)` and `(s3)` steps is to perform some processing on each of the tiles generated. A `(s2)` step will execute for each `k` and `j`. A `(s3)` step will execute for each `k`, `j` and `i`.

Now we consider how the tag collections are produced. The tags in `<K>` are produced by the step `(k)`. The tags in `<KJ>` are produced by the step `(kj)` and the tags in `<KJI>` are produced by the step `(kji)`.

Step `(k)` and step `(s1)` form a control dependence mediated by `<t1>`. This approach allows us maximal flexibility in scheduling. The specific constraint represented is that the step instance of `(k)` producing *a given tag value* must execute before the step `(s1)` *with that same tag value*.

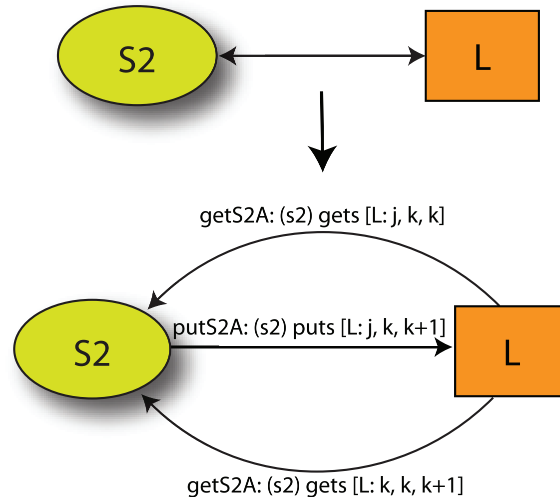


Figure 4. Step – Item relationship

In this example, the instances in the collections of `<K>` tags, `[L]` items and `(s1)` steps do not correspond exactly to each other. For example, step `(s1: k)` produces items `[L: k, k, k+1]`. These relationships are allowed to be complex, involving nearest neighbor computations, top-down or bottom-up tree processing or a wide variety of other relationships. Tags make the Concurrent Collections programming model more flexible and more general than streaming programming models. The relationship in Figure 2 really represents the more complex relationship as shown in Figure 4. The step code in Figure 3 illustrates how we can identify the puts and gets with these labels.

2.2 Textual Representation

A full textual representation of the graph includes one statement for each relation in the graph. Arrows are used for the producer and consumer relations. The symbol `::` is used for the prescription relation. Declarations indicate the tag components for item and tag collections. (Recall that tag components for step collections are derived from the tag components of their prescribing tag collections.) The resulting graph for Cholesky factorization in textual form is shown in Figure 2.

2.3 The serial code

In addition to specifying the graph, we need to code the steps and the environment in an imperative language. Figure 3 is the step code for the step S2. The step has access to the values of its tag components. It uses `get` operations to consume items and `put` operations to produce items and tags.

2.4 Optimizing and Tuning a Concurrent Collections specification

In this section, we consider the potential parallelism in the example from the perspective of an optimizing compiler or a tuning expert that has no knowledge of the internals of the steps or items. While the specification determines *if* a step will execute, the output of an optimizing compiler or tuning-expert will in general determine *when* and *where* each step will execute.

What can we tell just by looking at the textual representation in Figure 2? We will consider each of the three step collections de-

```
StepReturnValue_t S2_compute(cholesky_graph_t& graph, const
Tag_t& S2_compute_Tag) {

    double **A_block;
    double **Li_block;
    double **Lo_block;
    double temp;
    const int b = graph.b.Get(Tag_t(0));
    const int k = (S2_compute_Tag[0]);
    const int j = (S2_compute_Tag[1]);

    // Get the input tile.
    A_block  = graph.Lkji.Get(Tag_t(j,k,k));

    // Get the 2nd input tile (Output of previous step).
    Li_block = graph.Lkji.Get(Tag_t(k, k, k+1));

    // Allocate memory for the output tile.
    Lo_block = (double **) malloc(b * sizeof(double*));
    for(int i = 0; i < b; i++) {
        Lo_block[i] = (double *) malloc(b * sizeof(double));
    }

    for(int k_b = 0; k_b < b; k_b++) {
        for(int i_b = 0; i_b < b; i_b++) {
         Lo_block[i_b][k_b] = A_block[i_b][k_b]/Li_block[k_b][k_b];
        }
        for( int j_b = k_b+1; j_b < b; j_b++) {
            for( int i_b = 0; i_b < b; i_b++) {
                A_block[i_b][j_b] = A_block[i_b][j_b]
                - (Li_block[j_b][k_b] * Lo_block[i_b][k_b]);
            }
        }
    }

    // Write the output tile at the next time step.
    graph.Lkji.Put(Tag_t(j, k, k+1),Lo_block);
    return CNC_Success;
}
```

Figure 3. Perform triangular solve on the input tile

scribing the computation of the algorithm in turn. Since the step collection `(s1)` is prescribed by the tags of `<K>` and consumes only items of `[L: k, k, k]` and since the producer of this collection in the first iteration alone is `env`, the `(s1: k, k, k)` step instance is *enabled* at the start of execution. The producer of this collection for subsequent iterations is step `(s3)`. Steps in `(s2)` is prescribed by tags in `<KJ>` and consumes items of `[L: j, k, k]` and `[L: k, k, k+1]`. The producer of this collection is both the `env` and step `(s1)` in the first iteration. For all subsequent iterations the producer of this collection are both steps `(s1)` and `(s3)`. Steps in `(s3)` is prescribed by tags in `<KJI>` and consumes items of `[L: j, i, k]`, `[L: j, k, k+1]` and/or `[L: i, k, k+1]`. The producer of this collection is both the `env` and step `(s2)` in the first iteration. For all subsequent iterations the producer of this collection are both steps `(s2)` and `(s3)`. It might appear that we have to wait for the `(s2)` steps to complete before beginning any `(s3)` steps but let us examine the specification a bit more closely with a focus on the tag components. The scope of a tag component name is a single statement. If the same name, e.g., `j`, `k` is used on both sides of an arrow, it has the same value. For example, `(s2: j, k) -> [L: j, k, k+1];` means that the `[L]` item instance produced has the same `j`, `k` as the step that produced it. `[L: j, i, k] -> ( s3: j, i, k);` means that the `[L]` item instance consumed has the same `i`, `j`, `k` as the instance of `(s3)` step that consumed it. This means that there is a data dependence, and therefore an ordering constraint, between a step instance of `(s2: j, k)` and any step instance of `(s3: j, i, k)` with the same `j` and `k`. For similar reasons, there is a control dependence between a step `(s2: j, k)` and any step `(s3: j, i, k)` with the same j and k. This control dependence is via the tag collection `<KJ: j, k>`. In this particular application the control and data dependencies leads to exactly the same ordering constraint. Notice that none of this reasoning has anything to do with the code within the step or the data structures in the items. In other words, the tuning expert does not need to have any knowledge of the domain.

The current implementation does not support arithmetic operators. Instead we allow user-supplied tag functions such as `northwest(i, j)` or `parent(nodeID)`.

3. Slicing Annotation

The Concurrent Collections Programming Model described in the previous section allows expression of very general forms of dynamic parallelism and dynamic data accesses, while still preserving important invariants such as determinism and data race freedom. In this section, we introduce a declarative *slicing annotation* for CnC programs. Slicing annotations can be used by the programmer to restrict which step instances can read a specific item instance. The annotations can be used for additional correctness checking by (say) enabling the CnC runtime to throw an exception if a *get* operation is attempted by a step instance in violation of a slicing constraint. In addition, the annotation can be used by the CnC run-

time to perform optimizations such as garbage collection of dead items and update-in-place transformations [7, 5]. The later sections of this paper focus on the use of slicing annotations of this paper for optimized memory management and garbage collection of dead items.

Consider an item instance in collection `[C: T]`, and an instance of step `(S: I)`. The annotation,

$$(\texttt{S: I}) \subseteq \textit{readers}([\texttt{C: T}]), \textit{constraints}(\texttt{I}, \texttt{T})$$

indicates that step instance `(S: I)` may perform a get operation on item `[C: T]` if *constraints*(I, T) = *true*. In the absence of any annotations for `[C: T]`, the default semantics is that *readers*(`[C: T]`) $= *$ *i.e.,* any step may perform a get operation on (`[C: T]`). In general, compiler analysis can be used to further refine the slicing annotations (if any) provided by the user. However, if an annotation is provided for (`[C: T]`), we assume that *readers*(`[C: T]`)will exactly equal the union of the `(S: I)`'s specified in all slicing annotations for which *constraints* = *true*. In this paper, we will restrict our attention to constraints that include conjunctions and disjunctions of affine equalities and inequalities involving tag components and literal or symbolic global constants. We refer to this annotation as a *slicing* annotation because it specifies the slice (set) of step instances that reads a given item, akin to an *iteration slice* [10].

As an example, consider collection `[Lijk]` in the Cholesky CnC program discussed in Section 2. Items in this collection are indexed by a three-dimensional tag, T = `t1,t2,t3`. The slicing annotations for `[Lijk]` are shown in Figure 5. The constraints in these annotations include conjunctions and disjunctions of affine equalities and inequalities[1] involving tag components and global constants. The naming convention for labels is used to distinguish among different get operations in the same step e.g., *getS3A*, *getS3B*, and *getS3C* refer to three different gets in step *S3*.

4. Memory Management Problem

This section addresses the use of memory for application data. We show the various types of user data and show the transitions among these types to motivate the memory management issues addressed in the remainder of the paper.

4.1 Possible states for regions of memory

There are several types of user data available in Concurrent Collections.

Global read-only memory: This data is produced by the environment of the CnC graph and not modified by the CnC execution. The environment is responsible for managing this storage.

Locals: A local within a step might be a stack local or a heap local. In either case, the lifetime of a local is within the life of a single step instance. It may be shorter. Stack locals are managed by the serial language. The step code is responsible for deallocating any heap allocated local.

Contents of items: The lifetime of an item may span many step executions. It is alive from the time it is produced until all the steps that get it have executed. This lifetime may vary from execution to execution as the schedule varies but the contents of an item is guaranteed to be alive for any get. For a statically scheduled execution, the guarantee is a requirement of a valid static schedule. For a dynamically scheduled execution, it is the responsibility of the runtime to schedule the steps to ensure that this is true.

[1] Note that $i \neq j$ is translated to (`i<j` $\vee$ `i>j`) in annotation 1.

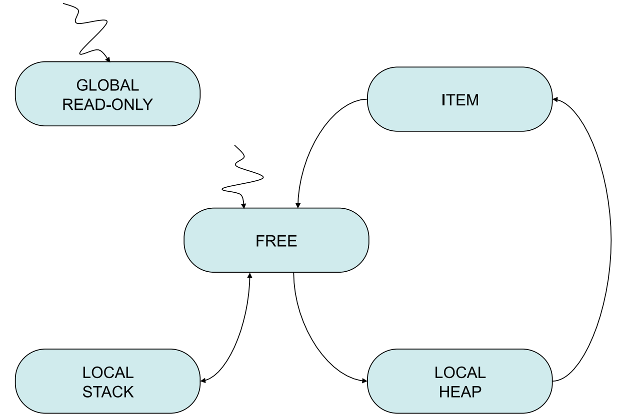


Figure 6. State Transitions in Global Memory

4.2 Memory state transitions

As the program executes, memory undergoes state transitions. The possible states of a memory location are the states of user data presented above and one additional state: free.

Free: Memory is free before it becomes the contents of an item or a local. It is free at the end of the lifetime of an item containing it. It becomes free at the end of the lifetime of a local. It stops being free when it becomes part of a local, either stack or heap.

From the perspective of the Concurrent Collections code, memory for user data starts out as free or as global read-only memory. Global read-only memory remains in that state through out. Memory that starts as free may go through the following state transitions:

- Free to Stack Local: This transition occurs when a local is allocated on the stack (upon entering a step or a block within a step).
- Free to Heap Local: This transition occurs when a local is on the heap (by explicit allocation).
- Stack Local to Free: This transition occurs when a stack local is deallocated from the stack (upon exiting a step or a block within a step).
- Heap Local to Free: This transition occurs when a heap local is explicitly deallocated within a step.
- Heap Local to Item: This transition occurs when heap allocated local memory is put as the contents of an item.
- Item to Free: This transition is instigated by the runtime system when dead item analysis has determined that the item is dead, i.e., its last use has executed.

Notice that there is no transition from free directly to item. The assumption is that for a step to put the contents as an item the contents had to be heap local first.

To ensure clean semantics regardless of the schedule and limit the transitions to those shown above we impose the following limitations on what a step instance s may do.

- s may read read-only global data without performing a get. It may not write into read-only global data.
- s may create local values and it may write and read them. It may not read its own locals before it has written them. (This is

$$\begin{array}{rcl}
\textit{getS1A:}\ (s1:k) \subseteq \textit{readers}([Lijk:t1,t2,t3]) & , & t1=t2 \ \wedge\ t2=t3 \ \wedge\ t1=k \\
\textit{getS2A:}\ (s2:k,j) \subseteq \textit{readers}([Lijk:t1,t2,t3]) & , & t2=t3 \ \wedge\ t3=k \ \wedge\ t1=j \\
\textit{getS2B:}\ (s2:k,j) \subseteq \textit{readers}([Lijk:t1,t2,t3]) & , & t1=t2 \ \wedge\ t2=k \ \wedge\ t3=k+1 \\
\textit{getS3A:}\ (s3:k,j,i) \subseteq \textit{readers}([Lijk:t1,t2,t3]) & , & t1=j \ \wedge\ t2=i \ \wedge\ t3=k \\
\textit{getS3B:}\ (s3:k,j,i) \subseteq \textit{readers}([Lijk:t1,t2,t3]) & , & t1=j \ \wedge\ t2=k \ \wedge\ t3=k+1 \ \wedge\ i=j \\
\textit{getS3C:}\ (s3:k,j,i) \subseteq \textit{readers}([Lijk:t1,t2,t3]) & , & t1=i \ \wedge\ t2=k \ \wedge\ t3=k+1 \ \wedge\ (i<j \ \vee\ i>j)
\end{array}$$

Figure 5. Get Operations and Slicing Annotations for Collection $Lijk$ from Cholesky example

a requirement of any serial language.) It may not read or write locals of other step instances.

- s may put an item with name n and tag t in accordance with single assignment rules. It may not put an item with name n and tag t if a distinct instance has put an item with the same name and tag but a different value.
- s may read the contents of an item if s has performed a get on that item. The contents of an item may include a tag, a pointer to elsewhere within the same item, or a pointer to read-only memory. It may use these for a get, for traversing its own contents or to find a place in read-only memory. It may not read the contents of an item instance it has not gotten. It may not write into the contents of an item instance regardless of whether it has or has not gotten that item. It may not contain a pointer to memory that is local, contents of other items or free.

Note: It is the responsibility of the runtime system (not the CnC specification) to ensure that an item is put before it is gotten.

5. Memory Management via the Slicing Annotation

There are no new memory management issues within steps or within the environment. These are handled by the language of the steps and the environment. However, Concurrent Collections has introduced the new concept of items. Taken all together, these items may occupy more memory than is available. But, while the lifetime of an item is longer than a single step, it is typically much shorter than the execution of the whole graph. It is both possible and critical to remove items when they will no longer be referenced. The language semantics include the single assignment rule, so the semantics are at the level of values, not storage. This frees the user from worrying about the management of the memory. This task is left to the runtime. Notice that, in fact, the determination of when an item becomes dead can not be part of the CnC specification because that is very dependent on the schedule which is not known to the specification.

Dead item analysis is similar to garbage collection in that we are interested in determining when values are dead so their storage can be reclaimed. But it is different from standard GC in several ways: First, for CnC, we have built a wide range of runtimes. They vary according to whether the grain-size, the schedule and the distribution among processors is determined statically or dynamically and is performed by a programmer or automatically. In this paper we discuss how to determine when an item is dead regardless of the style of runtime. GC, on the other hand, is typically part of a dynamic managed runtime. In addition, dead item analysis is not about detecting if a pointer points to that storage. We have no pointers to an item. We have to prove that no future step (even if the step is not yet prescribed and it does not yet have any of its inputs available) can later execute a get on that item.

5.1 Attributes and their Propagation

The problem may sound impossible. Let us look at what information we have to work with. We have access to the information in the static CnC graph. In addition, we have access to the dynamic attributes in the execution frontier and the cover sets described in [9].

In the discussion below, some of the expressions are static properties of the graph (e.g., `<T>:: (S)`). Some expressions are dynamic (e.g., `[I: i].available`.

1. The primitive attributes

   The primitive attributes are `.executed` for steps and `.available` for items and tags. These are generated by the runtime directly.

2. Propagation of attributes

   These propagated attributes are based on the primitive attributes and other propagated attributes.

   - inputs-available

```
(S:s).inputs-available =
   // indicates when all the inputs
   // for a step are available
   For all [I:i] s.t. (S:s) gets [I:i]
      [I: i].available
```

   - prescribed

```
(S:s).prescribed =
   // indicates when it is known
   // that the step will execute
   <T> :: (S) and <T:t>.available
```

   - enabled

```
(S:s).enabled =
   // indicates when the step is ready to execute
   (S:s).inputs-available and (S:s).prescribed
```

   - complete

```
(S:s).complete =
   // indicates when it is known that the
   // step will not execute in the future
   (S:s).executed
   or ( <T>:: (S) and !<T:s>.available )
```

   - !< ... >

```
!<T:t>.available =
   // indicates when known that the tag will never
   // be available. This attribute can be put by
   // a step directly and is also propagated
   not(<T:t>.available) and
   for each (S) s.t. (S) -> <T>
      for each s in writers((S),<T>)
         (S:s).complete
```

- dead

  One thing to note in the definition below is that the expression (`!<T:t>.available`) is not, in general, monotonic. However, the whole definition is monotonic because, once the second expression in the conjunction is true, all the producers are complete so <T:t> can not subsequently become available.

```
[I:i].dead =
   // indicates when an item
   // will not be used in the future
   for each (S) s.t. [I] -> (S)
      for each s in readers((S), [I])
         (S:s).complete
```

5.2 Transform declarations into reference count functions

We consider each item collection in turn. The goal is to produce a function that will convert the tag of an instance of an item to the number of gets that will occur on that instance. We will call that function at the time an item is produced and set its reference count accordingly. Each time the item is gotten the count will be decremented. The item can be removed when the reference count goes to zero.

The process is shown below by applying it to the Cholesky example introduced in Section 2 and then by a dynamic application, face detection.

5.2.1 Cholesky

In the discussion below, p and b are global read-only data. k, j and i are tag components.

1. tag spaces:

   `<K: k>` is `k = 0, p-1`
   `<KJ: k, j>` is `k = 0, p-1; j = k+1, p-1`
   `<KJI: k, j, i>` is `k = 0, p-1; j = k+1, p-1; i = k+1, j`

2. From the perspective of each get in each step

   The three compute steps are prescribed as follows:

```
(s1: k) :: <K: k>
(s2: k, j) :: <KJ: k, j>
(s3: k, j, i) :: <KJI: k, j, i>
```

3. The distinct get operations are identified below. We give them names here to help in following the example. The names are not part of the specification. getS1A is in step `(s1)`, getS2A and getS2B are distinct static get operations within step `(s2)`. These two get operations get distinct instances of `[L]` as shown by the distinct tag components below. Similarly for the three get operations in `(s3)`.

   getS1A in `(s1: k)` gets `[L: k, k, k]`

   getS2A in `(s2: k, j)` gets `[L: j, k, k]`
   getS2B in `(s2: k, j)` gets `[L: k, k, k+1]`

   getS3A in `(s3: k, j, i)` gets `[L: j, i, k]`
   getS3B in `(s3: k, j, i)` gets `[L: j, k, k+1]` if `i != j`
   getS3C in `(s3: k, j, i)` gets `[L: i, k, k+1]`

4. From the perspective of an item

We have to transform the information from the perspective of a step that does a get, to the perspective of an item that is gotten. Remember that the ref count function we are generating will be call for an item instance when that instance is put.

For a given item instance, say `[L: t1, t2, t3]`, we determine the conditions under which each get in each step of its consuming slice will execute and how many of these there will be in total. We need to consider two distinct constraints:

- slice constraint: Is this item one that might be gotten by any conceivable instance of the step?
- tag constraint: Is that step instance actually going to execute?

We then determine a simplified version of the constraint and from that we compute the number of such instances.

In this example, a clean expression of these instances can be determined statically. In general, for more dynamic computations, we will use a more conservative notion of `completed` rather than `executed`, where the initial ref count and the number of completed steps must be identical.

- getS1A
  from step: `(s1: k)` gets `[L: k, k, k]`
  from item: `[L: t1, t2, t3]` is gotten by `(s1: k)`
  slice constraint: $t1 = t2 \wedge t2 = t3 \wedge t1 = k$
  tag constraint: $0 \leq t1 \leq p-1$;
  $\Longrightarrow$ simplified constraint: $t1 = t2 = t3$
  number of instances: 1

- getS2A
  from step: `(s2: k, j)` gets `[L: j, k, k]`
  from item: `[L: t1, t2, t3]` is gotten by `(s2: k, j)`
  slice constraint: $t2 = t3 \wedge t3 = k \wedge t1 = j$
  tag constraint: $0 \leq t3 \leq p-1 \wedge t3+1 \leq t1 \leq p-1$
  $\Longrightarrow$ simplified constraint: $t2 = t3 \wedge t1 > t3$
  number of instances: 1

- getS2B
  from step: `(s2: k, j)` gets `[L: k, k, k+1]`
  from item: `[L: t1, t2, t3]` is gotten by `(s2: k, j)`
  slice constraint: $t1 = t2 \wedge t2 = k \wedge t3 = k+1$
  tag constraint: $0 \leq t2 \leq p-1 \wedge t2+1 \leq t1 \leq p-1$
  $\Longrightarrow$ simplified constraint: $t1 = t2 \wedge t3 = t2 + 1$
  number of instances: `p-t2-1`

- getS3A
  from step: `(s3: k, j, i)` gets `[L: j, i, k]`
  from item: `[L: t1, t2, t3]` is gotten by `(s3: k, j, i)`
  slice constraint:
  $t1 = j \wedge t2 = i \wedge t3 = k$
  tag constraint: $0 \leq t3 \leq p-1 \wedge t3 \leq t1 \leq p-1 \wedge t3+1 \leq t2 \leq t1$
  $\Longrightarrow$ simplified constraint: $t1 > t3 \wedge t2 > t3$
  number of instances: 1

- getS3B
  from step: `(s3: k, j, i)` gets `[L: j, k, k+1]`
  from item: `[L: t1, t2, t3]` is gotten by `(s3: k, j, i)`
  slice constraint: $t1 = j \wedge t2 = k \wedge t3 = k+1$
  tag constraint: $0 \leq t2 \leq p-1 \wedge t2+1 \leq t1 \leq p-1$
  $\Longrightarrow$ simplified constraint: $t3 = t2 + 1 \wedge t1 > t2$

number of instances: 1

- getS3C
  from step: if i != j (s3: k, j, i) gets [L: i, k, k+1]
  from item: [L: t1, t2, t3] is gotten by (s3: k, j, i)
  slice constraint: $t1 = i \wedge t2 = k \wedge t3 = t2+1 \wedge (i \leq j \vee i \leq j)$
  tag constraint: $0 <= t2 <= p-1 \wedge t2+1 <= i$
  $\Longrightarrow$ simplified constraint: $t3 = t2 + 1 \wedge t1 > t2$
  number of instances: p-t2-2

5. A ref count function can be built directly from this as follows:

```
rc = 0;
// getS1A
if(t1 == t2 && t2 == t3)
  // adjustment for (s1:k) where t1 = k
  rc = rc + 1;
// getS2A
if(t2 == t3 && t1>t3)
  // (S2:k, j) where t3 = k and t1 = j
  rc = rc + 1;
// getS2B
if(t1 == t2 && t3 == t2+1)
  // (s2: k, j) step constraints t2 = k for all j
  // range of j: k+1, p-1 = (p-1)-(k+1)+1 = p-k-1
  rc = rc + p - t2 - 1 ;
// getS3A
if(t1>t3 && t2>t3)
  // step constraints t1 = j and t2 = i and t3 = k
  rc = rc + 1;
// getS3B
if(t2 +1 == t3 && t1 > t2)
  // step constraints t1 = j and t2 = k for all i = j
  rc = rc + 1;
// getS3C
if(t2 + 1 == t3 && t1 > t2)
  // step constraints t1 = i and t2 = k for all i != j
  // range of j : k+1, p-1 = (p-1)-(k+1)+1 = p-k-1
  rc = rc + p - t2 - 2;
```

Notice that the approach used above uses the fact that an expression describing each tag space is known statically. If the expression might not be known at the time of the put the method will not reliably convert the tag constraint to a constant number of references. Below we show one way of extending the class of applications reliably handled by the approach. Even with this extension, there may be applications for which the approach does not apply. The approach is conservative in that nothing will be removed that is not actually garbage but it may neglect to remove actual garbage. As future work, we plan to better characterize the class of applications handled and, in addition, to extend the approach to expand the class.

5.2.2 Face Detection

Here we consider an application that is much more dynamic than Cholesky. It is an abstraction of a cascade face detector used in the computer vision. It includes a cascade of classifiers. The first classifier, (c1) operates on images. It is controlled by a tag collection, called ¡tc1¿. This tag collection contains tags specifying the image. If the first classifier determines that an image might be a face, it produces that same tag value but into a different tag collection, ¡tc2¿ that controls the second classifier. This tag collection will be a subset of the first tag collection. An image may fail on any classifier but if it makes its way to the end, it is deemed to be a face. If an image passes the last of the classifiers, it is identified in the item collection [face]. The classifiers are determined by machine learning, but you might think of them as looking for eyes, nose, mouth, etc. The tag w indicates a window in the image. The windows are square, of all sizes, in all positions in the frame.

- tag spaces:

  // tag collections for classifiers
  ```
  <tc1: w>, <tc2: w>, <tc3: w>;
  ```
  // detected faces
  ```
  <face: w>;
  ```
  // detected as not faces
  ```
  !<tc2: w>, !<tc3: w>, !<face: w>;
  ```

- From the perspective of each get in each step

  Examine each get operation in each step.
  ```
  env -> <tc1>;
  <face> -> env;
  ```

- prescriptions
  ```
  (c1: w) :: <tc1: w>;
  (c2: w) :: <tc2: w>;
  (c4: w) :: <tc3: w>;
  ```

- produces < ...> or !< ...>;
  ```
  [window: w] -> (c1: w) -> <tc2: w>, !<tc2: w>;
  [window: w] -> (c2: w) -> <tc3: w>, !<tc3: w>;
  [window: w] -> (c3: w) -> <face: w>, !<face: w>;
  ```

- from the perspective of the steps

  get in (c1: w) gets [window: w] if <tc1:w>.available
  get in (c2: w) gets [window: w] if <tc2:w>.available
  get in (c3: w) gets [window: w] if <tc3:w>.available

- From the perspective of an item

  Consider an item instance [window: w]. Under what conditions is get g in step s complete? Each of these will count as one reference.

  (c1: w).complete =
  (c1: w).executed $\vee$!<tc1: w>.available
  (c2: w).complete =
  (c2: w).executed $\vee$!<tc2: w>.available
  (c3: w).complete =
  (c3: w).executed $\vee$!<tc3: w>.available

  Notice that this case is dynamic and the Cholesky application was static. We deal with the dynamism by considering both < ...> and !< ...> for each tag. This means that for any w the ref count for [window: w] is a constant 3.

6. Preliminary Experimental Results

We have implemented the Cholesky factorization example [2] using the CnC programming model on two platforms. One is Intel® Concurrent Collections [8] for C/C++ based on Intel® Threading Building Blocks [4]. The other one is an X10/Java-based implementation using the Habanero Multicore Software Research project at Rice University [6].

We have also implemented by hand a simulation of the memory management technique using slicing annotation described in this paper within our two Cholesky implementations. We have ran experiments for a matrix size of 2000x2000 and 1000x1000 and different tile sizes, on a 2-core Intel MacBook Pro with 4GB of memory, and on an Intel 8-core (two quad-core processors) Vista machine with 3.2GB of memory.

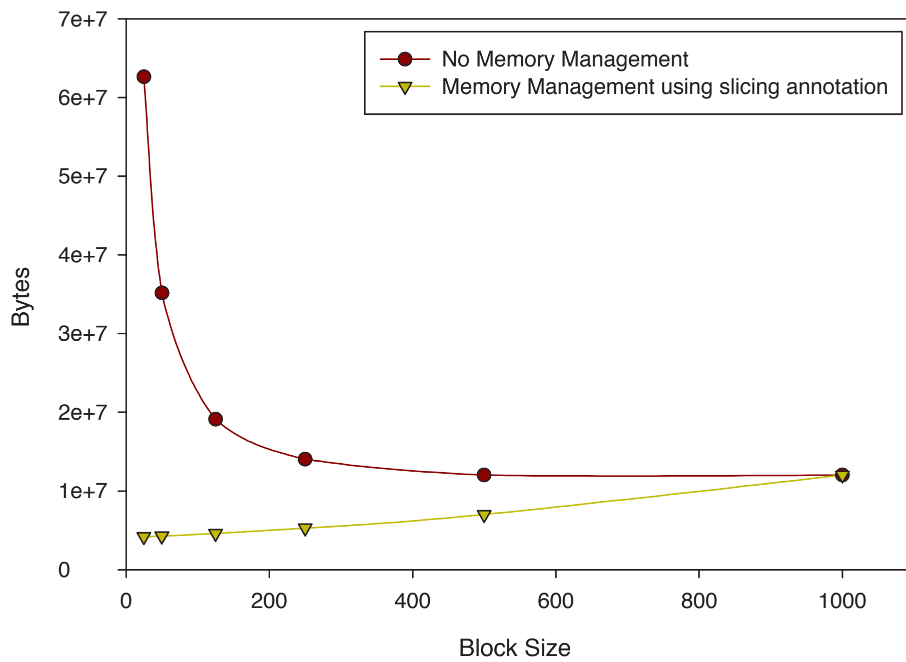


Figure 7. Memory requirements for Cholesky factorization using X10 for a 1000x1000 matrix

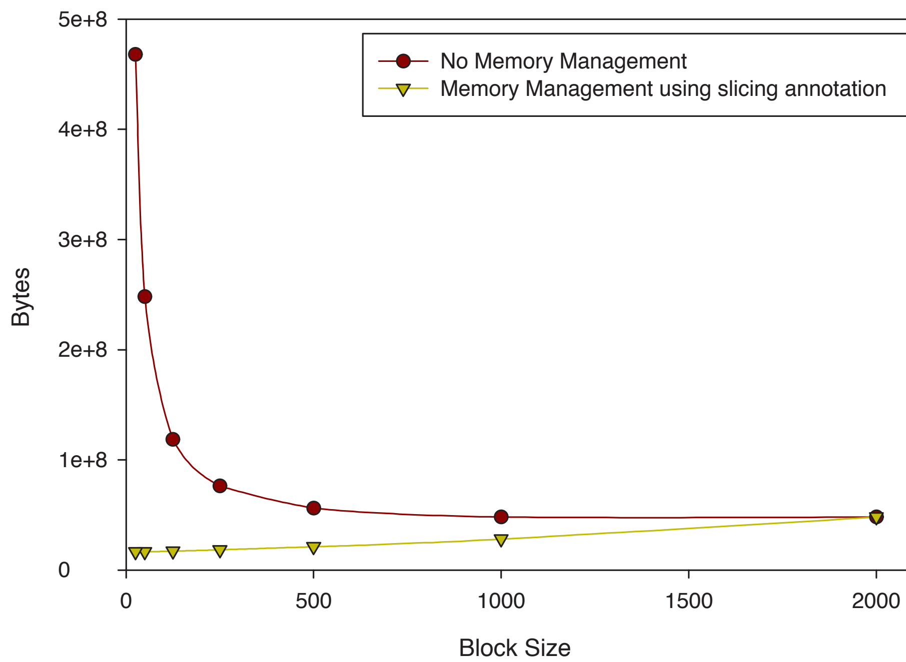


Figure 8. Memory requirements for Cholesky factorization using X10 for a 2000x2000 matrix

An interesting and perhaps surprising result was that the memory requirements in this particular application only depended on the problem size and tile size, and did *not* vary with number of cores, number of threads or whether a C/C++ or X10/Java implementation was used. We believe that this is an inherent property of the parallel algorithm used in this example, and that the memory usage will depend on the amount of parallelism available and exploited in an application.

The results of our experiments are shown on graphs on Figures 7 and 8. The numbers indicate the maximum space used by CnC for items collections, without and with our optimized memory management. Space used for other user data and space used for the runtime are not considered.

We can conclude that the memory management technique presented in this paper has a significant impact on the memory footprint of the Cholesky factorization example. Memory requirements for a problem with a small tile size were lowered up to 28 times using our memory management technique. The optimal tile sizes in terms of running times of the program were 125 in the X10/Java implementation and 50 in the C/C++ implementation [1]. In these two cases, the memory savings of our memory management technique were a factor of 7 and 14 respectively.

While it is customary in similar studies to compare the proposed technique with the state of the art, and comparing our results against no memory management at all may seem to be unjustly favoring the technique we are proposing, we are unaware of the existence of any memory management technique that is applicable to item collections in the Concurrent Collections programming model.

7. Conclusions and Future Work

In this paper, we addressed the memory management problem for item collections in the Concurrent Collections (CnC) programming model. We introduced a new declarative *slicing annotation* for CnC that can be automatically transformed into a reference counting function, which can be further used for memory management. Preliminary experimental results obtained from a Cholesky factorization example show that our memory management approach can result in space reductions for item collections of up to 28× relative to the baseline case of standard CnC without memory management.

There are many opportunities for future work to further extend the memory management approach described in this paper. We plan to extend the approach to handle a broader class of CnC applications. We will explore ways of compactly indicating that a step instance (with a specific tag) will never be executed in the future; that information can be used to improve the precision of the garbage collection approach presented in this paper. A hierarchical approach to specifying CnC programs can be explored that enables all sub-nodes within a hierarchical node to be collected, whenever a super-node's reference count becomes zero. Finally, we intend to explore other applications of the slicing annotation, such as copy elimination [7, 5], to further improver the performance of CnC programs.

References

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

[2] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Lapack working Note 191*, abs/0709.1272, 2007.

[3] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[4] Intel Corporation. Thread building blocks. http://www.threadingbuildingblocks.org/.

[5] K. Gharachorloo, V. Sarkar, and J. L. Hennessy. Efficient Implementation of Single Assignment Languages. *ACM Conference on Lisp and Functional Programming*, pages 259–268, July 1988.

[6] Habanero multicore software research project web page. http://habanero.rice.edu.

[7] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. *Proceedings of the Twelfth Annual ACM Conference on the Principles of Programming Languages*, pages 300–313, January 1985.

[8] Intel (r) concurrent collections for c/c++. http://softwarecommunity.intel.com/articles/eng/3862.htm.

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

[10] William Pugh and Evan Rosser. Iteration space slicing and its application to communication optimization. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 221–228, New York, NY, USA, 1997. ACM.