Multi-core Implementations of the Concurrent Collections Programming Model

Zoran Budimlić[†], Aparna Chandramowlishwaran[‡§], Kathleen Knobe[‡], Geoff Lowney[‡], Vivek Sarkar[†], and Leo Treggiari[†]

[†]Department of Computer Science, Rice University
[‡]Intel Corporation
[§]Georgia Institute of Technology
zoran@rice.edu, aparna@cc.gatech.edu, kath.knobe@intel.com, geoff.lowney@intel.com, vsarkar@rice.edu, leo.treggiari@intel.com

Abstract. In this paper we introduce the Concurrent Collections programming model, which builds on past work on TStreams [8]. In this model, programs are written in terms of high-level application-specific operations. These operations are partially ordered according to only their semantic constraints. These partial orderings correspond to data flow and control flow.

This approach supports an important separation of concerns. There are two roles involved in implementing a parallel program. One is the role of a domain expert, the developer whose interest and expertise is in the application domain, such as finance, genomics, or numerical analysis. The other is the tuning expert, whose interest and expertise is in performance, including performance on a particular platform. These may be distinct individuals or the same individual at different stages in application development. The tuning expert may in fact be software (such as a static or dynamic optimizing compiler). The Concurrent Collections programming model separates the work of the domain expert (the expression of the semantics of the computation) from the work of the tuning expert (selection and mapping of actual parallelism to a specific architecture). This separation simplifies the task of the domain expert. Writing in this language does not require any reasoning about parallelism or any understanding of the target architecture. The domain expert is concerned only with his or her area of expertise (the semantics of the application). This separation also simplifies the work of the 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 any understanding of the domain (as is often the case for compilers).

We describe two implementations of the Concurrent Collections programming model. One is Intel® Concurrent Collections for C/C++ based on Intel® Threaded Building Blocks. The other is an X10-based implementation from the Habanero project at Rice University. We compare the implementations by showing the results achieved on multi-core SMP machines when executing the same Concurrent Collections application, Cholesky factorization, in both these approaches.

1 Introduction

It is now well established that parallel computing is moving into the mainstream with a rapid increase in the adoption of multicore processors. Unlike previous generations of mainstream hardware evolution, this shift will have a major impact on existing and future software. A highly desirable solution to the multicore software productivity problem is to develop high-level programming models that are accessible to developers who are experts in different domains but lack deep experience with parallel programming. In this paper we introduce the Concurrent Collections (CnC) programming model [7], which builds on past work on TStreams [8]. In this model, programs are written in terms of high-level application-specific operations. These operations are partially ordered according to only their semantic constraints.

The rest of the paper is organized as follows. Section 2 describes the key concepts of the CnC model. Section 3 summarizes the runtime semantics of CnC, and describes two implementations of this model for multicore SMP's. Section 4 presents preliminary results for a Cholesky factorization computation expressed in CnC and implemented in both runtimes, and Section 5 concludes.

2 Key Concepts of the Concurrent Collections Model (Vivek and Zoran)

CnC programs are written in terms of high-level application-specific operations. These operations are partially ordered according to only their semantic constraints. The three constructs in this model are *step collections*, *item collections*, and *tag collections*. Each static collection represents a set of dynamic *instances*. Step instances are the unit of distribution and scheduling. Item instances are the unit of synchronization or communication. The model includes three relations among these constructs — *producer*, *consumer* and *prescription* relations. Items (data) are produced and consumed by steps (computation). This constitutes data flow. Tags are produced by steps. They are also used to *prescribe* steps, that is, to specify exactly which steps will execute. This CnC analog of control flow determines *if* a step will execute but not *when* it will execute. The Concurrent Collections model of an application is represented as a graph where the nodes can be either step, item or tag collections. The edges can be producer, consumer or prescription relations.

Many languages for expressing parallelism embed parallelism within serial code. Serial code is often written in terms of locations which can be overwritten. In addition, serial code requires a serial ordering. If there is no semantically required ordering, an arbitrary ordering must be specified. Execution of the code in parallel can require complex analysis to determine if reordering is possible. These two characteristics of serial code, overwriting and over-serialization, combine to make it difficult to perform valid compiler transformations. For these reasons, embedding parallelism in serial code can limit both the language's effectiveness and its ease of use. In CnC, there is no overwriting of the items and no arbitrary serialization among the steps. The data in items are 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

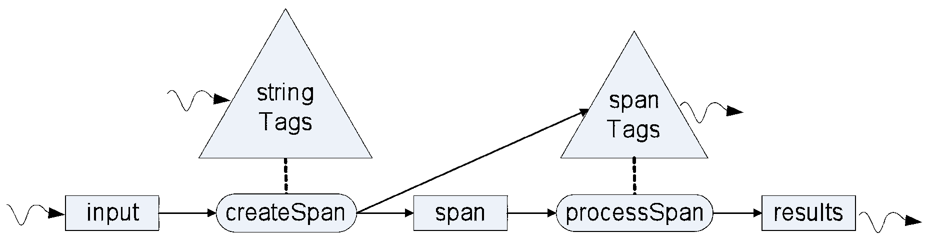


Fig. 1. Graphical Representation for an Example program

We will introduce the graph specification by showing the process to create a graph of a specific application. This discussion refers to Figure 1 which shows a simplified graphical representation of the application. The example is a trivial one. Both computation and data are too fine-grained to be realistic but the example illustrates all aspects of the model and is easy to follow.

The program reads a set of strings. For each input string it generates a collection of strings that partitions the input into substrings that contain the same character. For example, if an instance of an input string contains "aaaffqqqmmmmmmm", then it creates four substrings with values: "aaa", "ff", "qqq", "mmmmmmm". These instances are further processed.

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

Step collections: The computation is partitioned into high-level operations called *step collections*. Step collections are represented in the graphical form as ovals and in the textual form as paired parentheses. In this application, there are two step collections: (createSpan) converts each string to a set of substrings. (processSpan) processes each of the substrings.

Item collections and producer-consumer relations: The data is partitioned into data structures called *item collections*. Item collections are represented by rectangles in the graphical form and by paired square brackets in the textual form. In this application there three item collections, [input], [span] and [results]. These correspond to the input strings, the created substring spans and the results of processing these substring spans, respectively. The producer and consumer relationships between step collections and item collections are represented as directed edges between steps and items as shown in Figure 1.

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 [input] items and consumes [results] 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.

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, employee number and year or maybe xAxis, yAxis, and iterationNum.

In our example, the instances of the [input] item collection are distinguished by stringID. The (createSpan) step instances are also distinguished by stringID. The instances of the [span] item collection, the (processSpan) step collection and the [results] item collection are distinguished by both a stringID and a spanID within the string.

Tag Collections and Prescriptions: A specification that includes the tag components that distinguish among instances is still not precise enough to execute. Knowing that we distinguish instances of (createSpan) steps by values of stringID doesnt tell us if a (createSpan) step is to be executed for stringID 58. This control is the role of *tag collections*.

Tag collections are shown as triangles in the graphical form and paired angle brackets in the textual form. They are sets of tag instances. There are two tag collections in our example graph. A tag in <stringTags> identifies the set of strings. A tag in <spanTags> identifies the set of substrings. A *prescriptive relation* may exist between a tag collection, (<stringTags> for example) and a step collection ((createSpan) for example). The meaning of such a relationship is this: if a tag instance t, say stringID 58, is in <stringTags>, then the step instance s in (createSpan) with tag value stringID 58, will execute. Notice that the prescription relation mechanism determines *if* a step will execute. *When* it executes is up to a subsequent scheduler. 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 <stringTags> and the step collection (createSpan) are both distinguished by stringID. The work of (processSpan) steps is to perform some processing on each of the substrings generated. A (processSpan) step will execute for each substring. We require a tag collection, <spanTags> that identifies a stringID and a spanID for each substring to be processed. <spanTags> tags will prescribe (processSpan) steps.

Now we consider how the tag collections are produced. The tags in <stringTags> are produced by the environment which also produces [input] items. The step (createSpan) not only produces items in [span] but also produces the tags in <tagSpan> that identifies them.

In this example, the instances in the collections of <spanTags> tags, [span] items and (processSpan) steps correspond exactly to each other. These relationships are allowed to be much more complex, involving nearest neighbor computations, top-down or bottom-up tree processing or a wide variety of other relationships. Tags make this language more flexible and more general than a streaming language.

2.2 Textual Representation

We have already introduced the textual version of step, item and tag collections. 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 in textual from is shown below.

```
// declarations
<stringTags: int stringID>;
<spanTags: int stringID, int spanID>;
<results: int stringID, int spanID>;
[input: int stringID];
[span: int stringID, int spanID];

// prescriptions
<stringTags> :: (createSpan);
<spanTags> :: (processSpan);

// program inputs and outputs
env -> [input], <stringTags>;
[results], <spanTags> -> env;

// producer/consumer relations
[input: stringID] -> (createSpan: stringID);
(createSpan: stringID) -> <spanTags: stringID, spanID>;
(createSpan: stringID) -> [span: stringID, spanID];
[span: stringID, spanID] -> (processSpan: stringID, spanID);
(processSpan: stringID, spanID) -> [result: stringID,spanID];
```

In addition to specifying the graph, we need to code the steps and the environment in a serial language. 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.

2.3 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* each step will execute.

What can we tell just by looking at the textual representation in Section 2.2? We will consider each of the two step collections in turn. Since the step collection (createSpan) is prescribed by the tags of <stringTags> and consumes only items of [input] and since the only producer of these two collections is env, all the (createSpan) step instances are *enabled* at the start of execution. Steps in (processSpan) are prescribed by tags in <spanTags> and consume [span] items. The only producer of these two collections is the step collection (createSpan). It might appear that we have to wait for all the (createSpan) steps to complete before beginning any (processSpan) steps but lets 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., stringID is used on both sides of an arrow, it has the same value. For example, (createSpan: stringID) → [span: stringID, spanID]; means that the [span] item instance produced has the same stringID as the step that produced it. [span: stringID, spanID] → (processSpan: stringID, spanID); means that the [span] item instance consumed has the same stringID and spanID as the instance of (processSpan) step that consumed it. This means that there is a data dependence, and therefore an ordering constraint, between a step instance of (createSpan: stringID) and any step instance of (processSpan: stringID, spanID) with the same stringID. For similar reasons, there is a control dependence between a step (createSpan: stringID) and any step (processSpan: stringID, spanID) with the same stringID. This control dependence is via the tag collection <spanTags: stringID, spanID>. 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 any knowledge of the domain.

3 Runtime Semantics and Implementations

3.1 Runtime Semantics

As the program executes, instances of tags, items, and steps monotonically accumulate attributes indicating their state. The set of attributes for instances of the tags, items and steps and the partial ordering in which an instance can acquire these attributes is shown in Figure 2.

We will not discuss garbage collection here except to say that an item or tag that has been determined to be garbage is attributed as *dead*. It is a requirement of any garbage collection algorithm that the semantics remain unchanged if dead objects are removed.

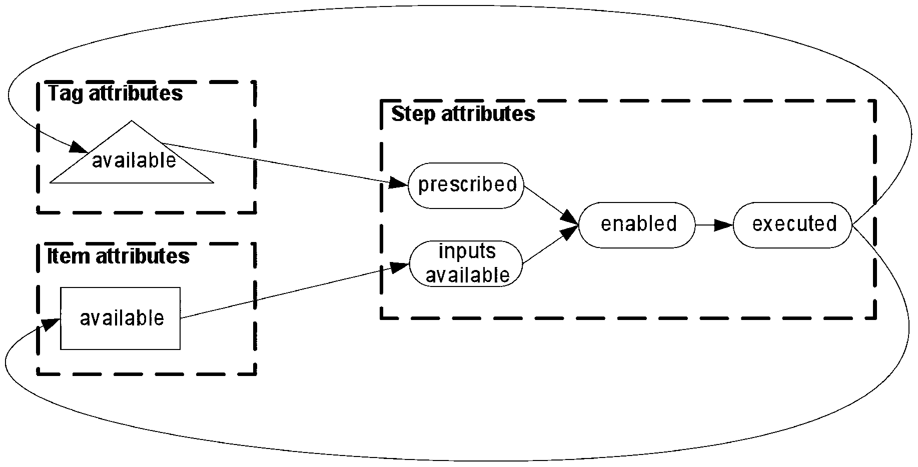


Fig. 2. Tag, Item, and Step Attributes

The *execution frontier* is the set of instances that are of any interest at some particular time, i.e., the set of instances that have any attribute but are not yet *dead* (for items and tags) or*executed* (for steps). The execution frontier evolves during execution.

Program termination occurs when no step is currently executing and no unexecuted step is currently *enabled. Valid program termination* occurs when a program terminates and all prescribed steps have executed. The program is deterministic and produces the same results regardless of the schedule within or the distribution among processors. It is possible to write an invalid program, one that stops with steps that are prescribed but whose input is not available. However, altering the schedule or distribution will not change this result. Note that the semantics allow but do not imply parallel execution. This makes it easier to develop and debug an application on a uniprocessor.

The Concurrent Collections execution model can be seen as a natural convergence of the data flow and program dependence graph execution models. The producer-consumer relations established via item collections support a general data flow execution model, whereas tag collections and prescriptions can be used to enforce control dependences and correspond to region nodes in program dependence graphs.

3.2 Runtime System Design Space

The CnC semantics outlined in the previous section allow for a wide variety of runtime systems. They vary along several dimensions: The target architecture determines the types of parallelism supported and whether the target has shared or distributed memory. The 3 basic decisions when mapping a given program to a given architecture include: choice of grain, distribution among computational resources, scheduling within a computational resource. The runtimes vary depending on whether these decisions are made statically or dynamically. In addition, a runtime might include additional capabilities such as checkpoint/restart, speculative execution, demand-driven execution, auto-tuning, etc.

We have explored the following points in the CnC design space:

– distributed memory, static grain, static distribution among address spaces, dynamic schedule within an address space, on MPI (HP)
– distributed memory, static grain, static distribution among address spaces, dynamic schedule within an address space, on MPI, with a checkpoint/restart capability (HP)
– distributed memory, static grain, static distribution among address spaces, static schedule within and address space, on MPI (HP)
– shared memory, dynamic grain, dynamic distribution among cores, dynamic schedule within cores, on MPI (GaTech [9])
– shared memory, static grain, dynamic distribution among cores, dynamic schedule within cores, in X10 (Rice)
– shared memory, static grain, dynamic distribution among cores, dynamic schedule within cores, on TBB (Intel)

Of the runtimes listed above, the last two represent recent work for targeting multicore SMPs and are the focus of this paper.

The runtimes above operate on distinct representations. For each representation, a translator can be used to convert the CnC textual form to that representation. We have built a translator for the C/C++ runtime that converts to the appropriate classes described below. In addition, for each step collection, it creates a hint, i.e., a template of the step based on information in the graph specification. The hint for a step specifies the form for the step tag, as well as the form for its gets and puts. The user fills in the local step computation. For now, the X10 forms are hand-generated but a similar translator is planned.

3.3 Runtime based on C++ and Threaded Building Blocks

The Concurrent Collection model allows for a wide variety of runtime systems as described above. The implementations discussed in this paper target shared memory multi-core processors, and are characterized by static grain choice, dynamic schedule and dynamic distribution. It consists of a small C++ class library built on the Intel® Threading Building Blocks (TBB) [2]. TBB controls the scheduling and execution of the program. There are several advantages in implementing Concurrent Collections on TBB. TBB supports fine-grain grain parallelism with tasks. TBB tasks are user-level function objects which are scheduled by a work-stealing scheduler. A work-stealing scheduler provides provably good performance for well-structured programs; the TBB scheduler is inspired by Cilk [1]. TBB also supports a set of concurrent containers, including vectors and hash-maps. These provide functionality similar to the containers in the Standard Template Library [12], and they also permit fast concurrent access. Finally, TBB provides a scalable concurrent memory allocator [6], which addresses a common bottleneck in parallel systems.

Classes represent the Concurrent Collection objects. A Graph class represents the program; it contains objects that represent the steps, items, and tags and their relationships. A step is represented as a user-written C++ function wrapped in a function object. When a tag that prescribes a step is created, an instance of the step functor is created and mapped to a TBB task. Get and Put APIs enable the user function to read and write item instances. The runtime provides classes to represent the three types of collections in a Concurrent Collections program (TagCollection, StepCollection, and ItemCollection). A TagCollection maintains a list of all the StepCollections that it prescribes. When a TagCollection Put method is called with a tag, a StepInstance is created for each the prescribed StepCollections. These StepInstances are mapped into TBB tasks and scheduled. Data items are created by calling an ItemCollection Put method with a value and an associated tag. Items are retrieved by calling an ItemCollection Get method with a tag. The data items for each ItemCollection are maintained in a TBB hash-map, accessed by tag.

We schedule StepInstances speculatively, as soon as they are prescribed by a tag. When a StepInstance calls an ItemCollection Get method with a tag, the value associated with the tag may have not yet been created, i.e., it may not have the attribute *inputs-available* and therefore may not yet have the attribute *enabled*. If this is the case, we queue the StepInstance on a local queue associated with the tag, and release it to the TBB scheduler when the value associated with the tag is Put by another step. When we re-schedule the StepInstance, we restart it from the beginning; here we exploit the functional nature of Concurrent Collection steps. Notice that each StepInstance attempts to run at most once per input item.

3.4 Runtime based on Habanero-Java

CnC construct	Translation to HJ
Tag	*point* object (unchanged from X10)
Prescription	*async* or *delayed async*
Item Collection	`java.util.concurrent.ConcurrentHashMap`
put() on Item Collection	Nonblocking *put()* on `ConcurrentHashMap`
get() on Item Collection	Blocking or nonblocking *get()* on `ConcurrentHashMap`

Table 1. Summary of mapping from CnC primitives to HJ primitives

In this section, we discuss our experiences with an implementation of the CnC programming model in the Java-based Habanero-Java (HJ)[1] programming language being developed in the Habanero Multicore Software Research project at Rice University [5] which aims to facilitate multicore software enablement through language extensions, compiler technologies, runtime management, concurrency libraries and tools. The HJ language is an extension to v0.41 of the X10 language described in [3]. The X10 language was developed at IBM as part of the

[1] Habanero-C++, a C++ based variant of this language, is also in development.

PERCS project in DARPA's High Productivity Computing Systems program. The initial versions of the X10 language (up to v1.5) used Java as its underlying sequential language, but future versions of X10 starting with v1.7 will move to a Scala-based syntax [13] which is quite different from Java.

Some key differences between HJ and v0.41 of X10 are as follows:

- HJ includes the *phasers* construct [11], which generalizes X10's clocks to unify point-to-point and collective synchronization.
- HJ includes a new *delayed async* construct, which is described below in more detail.
- HJ permits dynamic allocation of *places*.
- HJ extends X10's *atomic* construct so as to enable enforcement of mutual exclusion across multiple places.

Since HJ is based on Java, the use of certain primitives from the Java Concurrency Utilities [10] is also permitted in HJ programs, most notably operations on Java Concurrent Collections such as `java.util.concurrent.ConcurrentHashMap` and on Java Atomic Variables.

One of the attractive properties of the CnC model is that it has a very small number of primitives. We were pleasantly surprised to see how straightforward it has been to map CnC primitives to HJ, as summarized in Table 1. The following subsections provide additional details for this mapping.

Tags We used the X10 *point* construct to implement Tags. A *point* in X10 is an integer tuple, that can be declared with an unspecified rank. A multidimensional tag is implemented by a multidimensional point.

Prescriptions We have optimized away all prescription tags in the HJ implementation. When a step needs to put a prescription tag in the tag collection, we perform a normal *async* or a *delayed async* for each step prescribed by that tag. The normal async statement, *async* $\langle stmt \rangle$, is derived from X10 and causes the parent activity to create a new child activity to execute $\langle stmt \rangle$. Execution of the async statement returns immediately i.e., the parent activity can proceed immediately to its next statement.

The delayed async statement, *async* $(\langle cond \rangle)$ $\langle stmt \rangle$, is similar to a normal async except that execution of $\langle stmt \rangle$ is guaranteed to be delayed until after the boolean condition, $\langle cond \rangle$, evaluates to true. Section 3.4 outlines how delayed async's can be used to obtain more efficient implementations of tag prescriptions, compared to normal async's.

Item Collections We use the `java.util.concurrent.ConcurrentHashMap` class to implement item collections. Our HJ implementation of item collections supports the following operations:

- `new ItemCollection(String` *name*`)`: create and return a new item collection. The string parameter, *name*, is used only for diagnostic purposes.
- C`.put(point` p`, Object` O`)`: insert item O with tag p into collection C. Throw an exception if C already contains an item with tag p.

- C.`awaitAndGet(point` p`)`: return item in collection C with tag p. If necessary, the caller blocks until item becomes available.
- C.`containsTag(point` p`)`: return true if collection C contains an item with tag p, false otherwise.
- C.`get(point` p`)`: return item in collection C with tag p if present; return `null` otherwise. The HJ implementation of CnC ensures that this operation is only performed when tag p is present *i.e.,* when C.`containsTag(point` p`)` = `true`. Unlike `awaitAndGet()`, a `get()` operation is guaranteed to always be nonblocking.

Put and Get Operations A CnC `put` operation is directly translated to a `put` operation on an HJ item collection, but implementing `get` operations can be more complicated. A naive approach is to translate a CnC `get` operation to an `awaitAndGet` operation on an HJ item collection. However, this approach does not scale well when there are a large number of steps blocked on `get` operations, since each blocked activity in the current X10 runtime system gets bound to a separate Java thread. A Java thread has a larger memory footprint than a newly created async operation. Typically, a single heavyweight Java thread executes multiple lightweight async's; however, when an async blocks on an `awaitAndGet` operation it also blocks the Java thread, thereby causing additional Java threads to be allocated in the thread pool[4]. In some scenarios, this can result in thousands of Java threads getting created and then immediately blocking on `awaitAndGet` operations.

This observation lead to some interesting compiler optimization opportunities of `get` operations using delayed asyncs. Consider a CnC step S that performs two `get` operations followed by a `put` operation as follows (where T_x, T_y, T_z are distinct tags):

$$S\text{: } \{\ x := C.\texttt{get}(T_x);\ y := C.\texttt{get}(T_y);\ z := F(x, y);\ C.\texttt{put}(T_z, z);\ \}$$

Instead of implementing a prescription of step S with tag T_S as a normal async like "`async` $S(T_S)$", a compiler can implement it using a delayed async of the form "`async await(`C`.containsTag(`T_x`) &&` C`.containsTag(`T_y`))` $S(T_S)$". With this boolean condition, we are guaranteed that execution of the step will not begin until items with tags T_x and T_y are available in collection C.

4 Case Study: Cholesky (Kath and Aparna)

4.1 Cholesky factorization algorithm

In this case study we describe a numerical algorithm, Cholesky factorization on CnC and discuss in detail the various challenges involved in exposing the potential parallelism in the application to the CnC runtime for effective mapping and scheduling on the target architecture. Cholesky factorization takes in a symmetric positive definite matrix as input which is factorized into a lower triangular matrix and its transpose.

The Cholesky factorization algorithm can be divided into six steps and the operations on the elements inside each step is independent of each other and this maps well onto the CnC model. We also explore optimizations such as tiling to improve performance.

Algorithm 1 is the pseudo-code for tiled Cholesky factorization. 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 $b = n * p$. We iterate over the outer loop from 0 to p-1. The computation can be broken down into three steps. The step (S1) performs unblocked Cholesky factorization of the symmetric positive definite tile A_{kk} of size bxb producing a lower triangular matrix tile L_{kk}. Step (S2) is used to apply the transformation computed by the step (S1) on the tile A_{jk} by means of a triangular system solve. Finally the step (S3) is used to update the trailing matrix by means of a matrix-matrix multiply.

Algorithm 1: Tiled Cholesky Factorization algorithm pseudo-code

Input: Input matrix: A, Matrix size: nxn where $n = p * b$ for some b which denotes the size of the tile

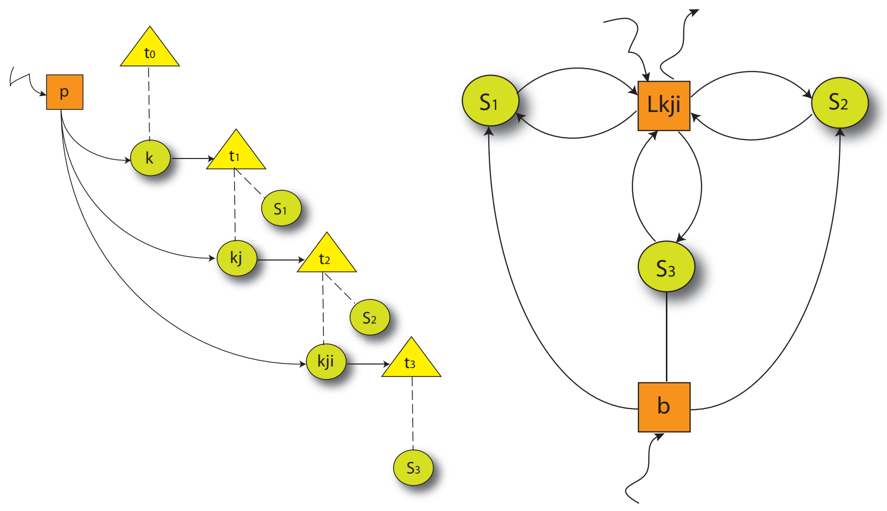
Output: Lower triangular matrix: L

```
1 for $k = 0$ to $p - 1$ do
2     S1($A_{kk}, L_{kk}$);
3     for $j = k + 1$ to $p - 1$ do
4         S2($L_{kk}, A_{jk}, L_{jk}$);
5     for $j = k + 1$ to $p - 1$ do
6         for $i = k + 1$ to $j$ do
7             S3($L_{jk}, L_{ik}, A_{ij}$);
```

4.2 CnC graphical notation

Figure 3 represents how the Cholesky factorization algorithm maps onto the CnC graphical notation. (The actual graph is connected. The graph is shown in two parts for easier reading.) Figure 3(a) represents the control flow of the algorithm. The three steps, (S1), (S2) and (S3), discussed in Alg. 1 map onto the three steps denoted as ovals. Item, [n] represents the matrix size and [n] is input from the user environment. The tag t_0 is a singleton tag to start the computation. Step (k) generates the tag collection <t_1> required for step (S1). Step (kj) converts tag collection <t_1> to tag collection <t_2>. Step (kji) converts Tag collection <t_2> to tag collection <t_3>. The tag collections <t_1>, <t_2> and <t_3> prescribe steps (kj), (S1); (kji), (S2); and (S3) respectively.

Figure 3(b) represents the data flow in the algorithm. We take in item [Lkji] at k=0 as input from the user environment. We then perform Cholesky factor-



(a) Control flow notation (b) Data flow notation

Fig. 3. CnC graphical notation of control and data flow in Cholesky factorization

ization and the output lower triangular matrix is again written back into the user environment represented as outward pointing arrows.

4.3 Preliminary Experimental Results

The graph in figures 3(a) and 3(b) represent a static graph and the user reasons only about the sequential flow of control and data in the application. Cholesky is an interesting example because its performance is impacted by both parallelism (number of cores used) and locality (tile size), thereby illustrating that both considerations can be taken into account when tuning a CnC program. We implemented the Cholesky example using the TBB-based runtime described in Section 3.3. Figure 4 shows the relative speedup obtained by using multiple cores on an 8-way Intel® dual Xeon Harpertown SMP system.

We also implemented the Cholesky example using the Habanero-Java runtime described in Section 3.4. Figure 5 shows the relative speedup obtained by using multiple cores on the same SMP system as in Figure 4. Though the HJ runtime is implemented in Java, the baseline sequential runtime for this version is only 14% slower than that of the TBB version. Further analysis of JVM overheads, such as dynamic compilation and garbage collection, is needed to determine why the scalability of the HJ version is poorer than the TBB version for this example. However, these preliminary results demonstrate that the CnC model is amenable to portable implementation on different parallel runtime systems.

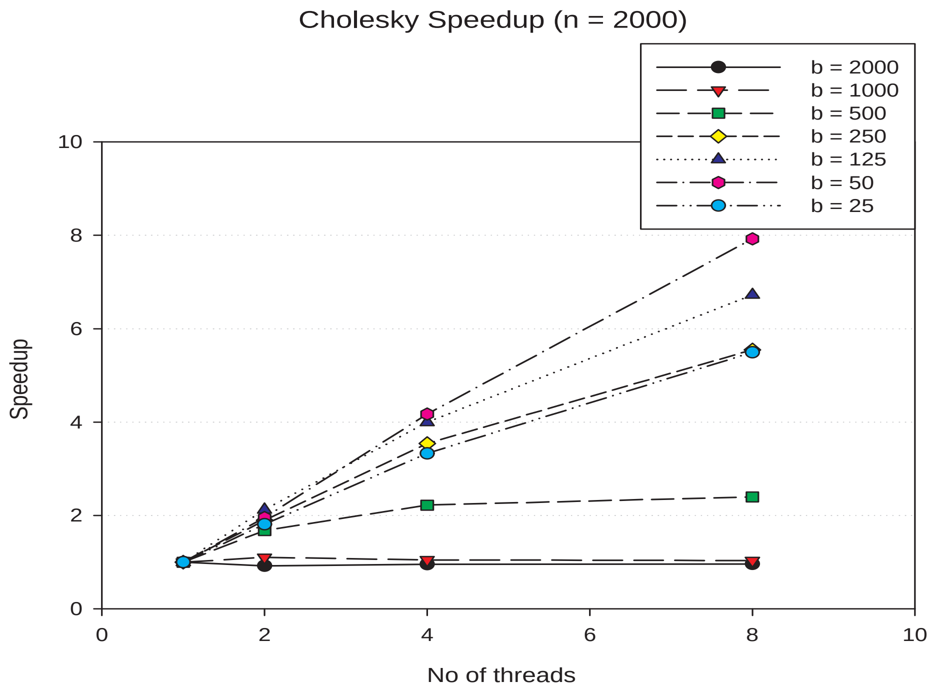


Fig. 4. Speedup results for TBB implementation of 2000×2000 Cholesky Factorization CnC program on an 8-way (2p4c) Intel$^{®}$ dual Xeon Harpertown SMP system. The running time for the baseline (1 thread, tile size 2000) was 24.911 seconds

5 Conclusions and Future Work

In this paper, we introduced the Concurrent Collections programming model, which builds on past work on TStreams [8] and discussed its advantages in separation of concerns between the domain expert, and the automatic/human tuning expert which may in fact be realized in the form of a static or dynamic optimizing compiler. We described two implementations of the Concurrent Collections programming model. One is Intel$^{®}$ Concurrent Collections for C/C++ based on Intel$^{®}$ Threaded Building Blocks. The other is an X10-based implementation from the Habanero project at Rice University. We compared the implementations by showing the results achieved on multi-core SMP machines when executing the same Concurrent Collections application, Cholesky factorization, in both these approaches. The scalability of these early results attest to the robustness of the CnC model and its ability to be mapped efficiently to multiple runtime systems.

References

1. Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work-stealing. In *Proceedins of the 35th Annual IEEE Conference on Foundations of Computer Science*, 1994.

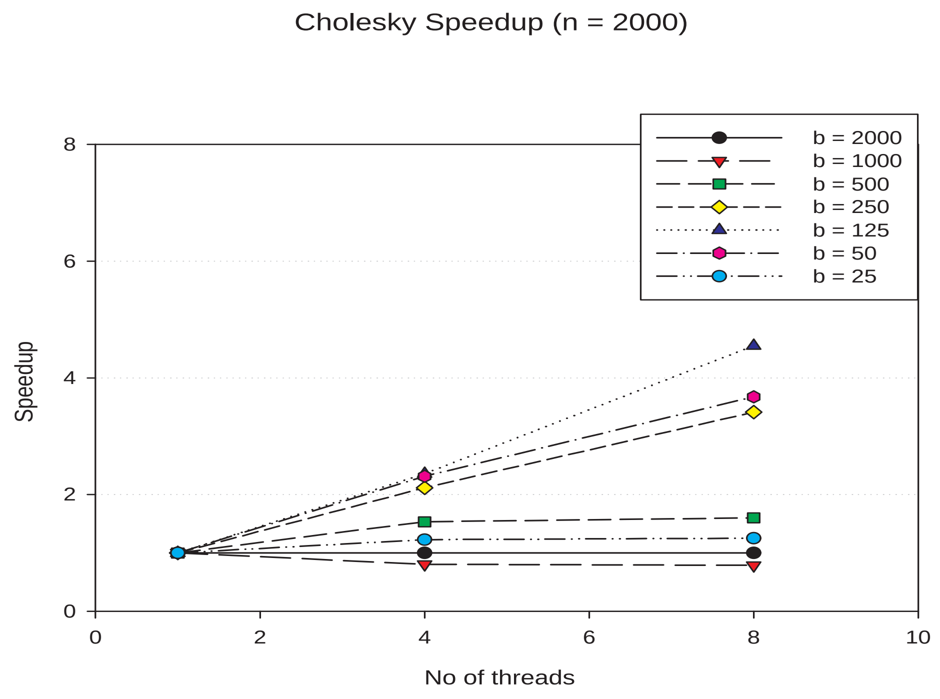


Fig. 5. Speedup results for Habanero-Java implementation of 2000×2000 Cholesky Factorization CnC program on an 8-way (2p4c) Intel® dual Xeon Harpertown SMP system. The running time for the baseline (1 thread, tile size 2000) was 28.346 seconds

2. Intel Corporation. Thread building blocks. http://www.threadingbuildingblocks.org/.
3. P.Charles et al. 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. R.Barik et al. Experiences with an smp implementation for x10 based on the java concurrency utilities. In *Workshop on Programming Models for Ubiquitous Parallelism (PMUP), held in conjunction with PACT 2006*, September 2006.
5. Habanero multicore software research project web page. http://habanero.rice.edu.
6. Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. Mcrt-malloc a scalable transactional memory allocator. In *Proceedings of the 2006 ACM SIGPLAN International Symposium on Memory Management*, pages 74–83, June 2006.
7. Intel (r) concurrent collections for c/c++. http://softwarecommunity.intel.com/articles/eng/3862.htm.
8. Kathleen Knobe and Carl D. Offner. Tstreams: A model of parallel computation (preliminary report). Technical Report HPL-2004-78, HP Labs, 2004.
9. Hasnain Mandviwala. *Capsules: Expressing composable computations in a parallel programming model*. PhD thesis, Georgia Institute of Technology, June 2008.
10. Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
11. Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization.

In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.
12. Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.
13. X10 v1.7 language specification. http://x10.sourceforge.net/docs/x10-170.pdf.