Programming Models for Parallel Computing

Edited by Pavan Balaji

The MIT Press
Cambridge, Massachusetts
London, England

Contents

1	**Concurrent Collections**—*K. Knobe, M. Burke, and F. Schlimbach*		1
1.1	Introduction		1
1.2	Motivation		2
	1.2.1	Foundational hypotheses	2
1.3	CnC Domain Language		3
	1.3.1	Description	3
	1.3.2	Characteristics	6
	1.3.3	Example	7
	1.3.4	Execution Semantics	10
	1.3.5	Programming in CnC	11
	1.3.6	Futures	17
1.4	CnC Tuning Language		18
	1.4.1	Description	19
	1.4.2	Characteristics	23
	1.4.3	Examples	23
	1.4.4	Execution Model	26
	1.4.5	Futures	29
1.5	Current Status		30
1.6	Related Work		30
	1.6.1	CnC Domain Language	31
	1.6.2	CnC Tuning Language	32
1.7	Conclusions		32

1 Concurrent Collections

Kath Knobe, Rice University
Michael G. Burke, Rice University
Frank Schlimbach, Intel

1.1 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 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 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 analysis, debugging, and testing of a parallel program.

The CnC programming model is quite different from most other parallel programming models in several important ways. It is a programming model specifically for coordinating among potentially parallel chunks of computation and data. As such it is a coordination language, and so must be paired with a separate language for computation. In addition, CnC is declarative. It specifies the required orderings among chunks of computation code, but does not in any way indicate how those requirements are to be met. In particular, it is not some syntax for connecting with a specific runtime. CnC runtimes might be characterized by how they determine: the grain of data and computation; the placement of data and computation across the platform; the schedule across time within a component of the platform. Instances of CnC implementations have determined all three dynamically [hasnain], all three statically [hp] and various combinations. Most of the current implementations fix the grain statically but allow for dynamic choice of mapping across the platform and across time. We will address the current implementations in more detail in Section 1.5.

The CnC programming model is a high level, declarative model built on past work with TStreams [8]. CnC falls into the same family as dataflow and stream-processing languages—a program is a graph of kernels, communicating with one another.

A CnC graph can execute along with its environment, a program written in a sequential or parallel language. A CnC graph indicates what input it expects from the environment and what output it returns to the environment. This is all the CnC graph needs to know about the environment. But a CnC graph may play fundamentally different roles in different applications, depending on the relationship of the graph to its environment. Sometimes we think of the environment of a CnC graph as basically main. It is written in some computation language. Its entire purpose might be to manage a CnC graph execution. In this case, it creates a graph (see Section 1.3.1), inputs data to the graph, and indicates that it is done putting input. (In finitely executing CnC graphs, the program cannot know when it is finished unless it knows that there will be no more input.) The environment waits until the

graph execution completes and then gets output data from the graph. Above we describe a single execution of a single CnC graph. The environment might be much more complex. In particular, most of the application might be in the environment. A single instance of the graph as described above might, for example, be in a loop. Then the same graph might execute multiple times with different inputs. In addition, the environment may have many distinct CnC graphs at separate places in the code. The environment itself might be a parallel program. This would allow for the distinct dynamic instances of a given graph and/or instances of distinct graphs to execute in parallel.

1.2 Motivation

Explicitly parallel languages and explicitly serial languages are each over-constrained, though in different ways. Concurrent Collections (CnC) avoids both forms of unnecessary constraints. A CnC program is an implicitly parallel program that avoids unnecessary constraints and thereby maximizes the scheduling freedom in executing the program for a given target (efficiency) and also among distinct targets (portability).

The domain expert writing a CnC program focuses on the meaning of the application, not on how to schedule it. CnC isolates the work of the domain expert (interested in finance, chemistry, gaming) from the tuning expert (interested in load balance, locality, scalability). This isolation minimizes the need for the domain expert to think about all the complications of parallel systems. In the CnC domain language, no particular category of machine is assumed. This isolation also minimizes the need for the tuning expert to know about the application domain.

"Old World" languages impair communication between the programmer and the compiler. They introduce arbitrary serial constraints due to the serial ordering of statements and to arbitrary overwriting of variables (resulting in complicated anti-dependences and output dependences). Program analysis and transformations are then required to expose the true dependences. In these languages the domain and tuning code are intertwined. One has to be aware of one when modifying the other, complicating both tasks. Also the programmer writing the domain code has to commit to the form of parallelism that the application will use when targeted to a parallel architecture. CnC is neutral to the form of parallelism, supporting different forms of parallel applications.

1.2.1 Foundational hypotheses

1. User specification of the semantically required constraints, rather than the parallelism, is simpler and leads to comparable if not better performance.

2. Separation of computation from coordination simplifies these activities and supports reuse.

3. Separating the specification of the semantics from the specification of the tuning simplifies both activities.

4. Starting with too much asynchronous parallelism and providing a separate, simple and effective way to control it is easier and more effective than starting with a serial program and adding parallelism.

5. Dynamic single assignment (DSA)/determinism eliminates race conditions, making programs less error prone and easier to reason about and debug. The apparent increase in memory usage can be adequately addressed.

CnC is broader than other parallel programming models along one dimension and narrower along another. CnC is provably deterministic with respect to a program's output (not the schedule). Although this makes it easier to use, it also limits the scope of applications it supports. (We are experimenting with adding controlled nondeterminism to extend the range of applications supported without losing the ease-of-use advantages.) Some applications are inherently nondeterministic. On the other hand it allows a wider range of parallel execution styles than other systems. It supports both static and dynamic forms of task, data, loop, pipeline, and tree parallelism.

1.3 CnC Domain Language

1.3.1 Description

The CnC domain specification indicates computations of a program, and the control and data dependences among these computations. These relationships impose constraints on the execution order. These are the only constraints imposed by the domain specification. There is no arbitrary serial ordering of statements, only the partial ordering based on the dependences. These constraints are based on the application logic, and are independent of the target architecture.

The CnC domain language coordinates among computation steps. These computation steps are written in a sequential or parallel programming language. For example, Intel ® Concurrent Collections for C++ supports C++ programs. Other existing systems have Java, C with OpenMP, Scala, Haskell, Python, Habanero Java, and a subset of MATLAB as the computation language. The data model is based on tuple spaces.

A computation step instance that produces a data item must execute before the computation step instance that consumes that data. A computation step instance that produces a

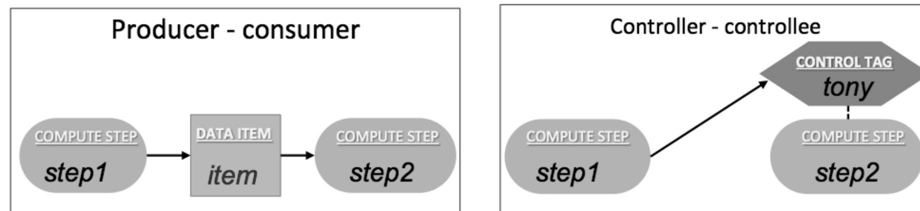

Figure 1.1: Ordering requirements

control tag must execute before the computation step instance controlled by that control tag. These entities and relationships form the nodes and edges of a graph as illustrated in Figure 1.1.

This graphical description includes the computation steps (in circles), the data items (in squares), the control tags (in hexagons) and the producer/consumer relations among them (arrows, dotted in the case of control dependences). The inputs and outputs are shown as data items produced by the environment and data items consumed by the environment. This facilitates composability of graphs. The graph may be cyclic.

In the remainder of the chapter we use our textual notation instead of the graph notation described above. We represent step, items and tag collections using syntax $(stepName)$, $[itemName]$ and $< tagName >$. Arrows are used for producer and consumer relations. The control relation represented as a dotted line in the graph becomes `::` in the text. More than one relation can appear in a statement. Statements end in a `;`.

CnC specification graph. The three main constructs in a CnC specification graph 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 created as the program executes.

A step collection corresponds to a specific computation, and its instances correspond to invocations of that computation with different input tags. A control collection is said to *control* a step collection—adding an instance to the control collection *prescribes* one or more step instances i.e., causes the step instances to eventually execute when their inputs become available. The invoked step may enable other step executions by adding instances to other control collections, and so on.

Steps also dynamically read (`get`) and write (`put`) data instances. The execution order of step instances is constrained only by their producer and consumer relationships, including control relations. 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 control relationships.

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. It can consume data instances and use control instances to prescribe conditional execution.

Collections indexed by *tags*. Within each collection, control, data, and step instances are each identified by a unique *tag*. Tags may be of any data type that supports an equality test and hash function. Typically, tags have a specific meaning within the application. For example, they may be tuples of integers modeling an iteration space (i.e., the iterations of a nested loop structure). Tags can also be points in nongrid spaces—nodes in a tree, in an irregular mesh, elements of a set, etc. Collections use tags as follows:

- A data collection is an associative container indexed by tags. The contents indexed by a tag `i`, once written, cannot be overwritten (dynamic single assignment). In a specification file a data collection is referred to with square-bracket syntax: `[x:i,j]`.

- A step begins execution with the tag indexing that step instance. The tag provides access to (optional) input data. The next input tag may be a function of the data found in the first input tag, and so on. So the first input tag serves as a seed value for computing the tags of all the step's input and output data.

  For example, in a stencil computation a tag "`i,j`" would be used to access data at positions "`i+1,j+1`", "`i-1,j-i`" and so on.

- A control tag collection specifies which step instances are to execute. Each tag in a control tag collection is a tuple which controls the execution of a corresponding instance of the controlled computation step. A *tag function* indicates which step instance corresponds to a control tag instance. Tag functions also indicate the relationships between step instances and input and output data item instances, as well as step instances and output control tags.
  Each computation step collection is controlled by exactly one control tag collection. A given control tag collection may control more than one computation step collection. A producer produces the control tags and data items. In either case, the producer might be a computation step or the environment, as shown in Figure 1.1.

Below is an example snippet of a CnC specification.

```
// control relationship : myCtrl prescribes instances of step
<myCtrl> :: (myStep);
// myStep gets items from myData, puts tags in myCtrl and items in myData
[myData] -> (myStep) -> <myCtrl>, [myData];
```

The CnC specification can indicate tag functions:

```
[myData: i] -> (myStep: i) -> <myCtrl: i+1>, [myData: i+1];
```

1.3.2 Characteristics

- **DSA** Each data instance, that is a name/tag pair, is associated with a unique value: i.e., the items obey dynamic single assignment (DSA).

- **Determinism** The step instance as a whole has no side effects and is a pure function of its input data. This combination of this and the DSA property ensures that the CnC specification is deterministic. The same specification with the same input can run on a thousand cores or on a single core, if it fits, and will produce the same results.

  Determinism in the domain specification means the code produces the same output collections on every execution. These collections are sets so the ordering is not relevant but the names, tag values and contents must be identical.

- **No false dependences** A CnC domain specification has implicit asynchronous parallelism. The only required orderings are specified semantic dependences, not arbitrary orderings.

- **Platform Independence** The domain specification is independent of the target platform. In particular, it does not make any assumptions about the memory model. Data is identified by tags and is treated as values and not as memory references. This makes it possible to use the same domain specification on shared and distributed memory without changing the step code. In many cases the separate tuning specification alone can handle the platform differences. In Section1.4 we will show how a CnC runtime can handle distributed memory efficiently.

  Based on tag functions, DSA, determinism and ordering constraints that are due only to true dependences, the CnC domain language simplifies analysis and transformations such as loop interchange, loop splitting, distribution (local vs. distributed tag component).

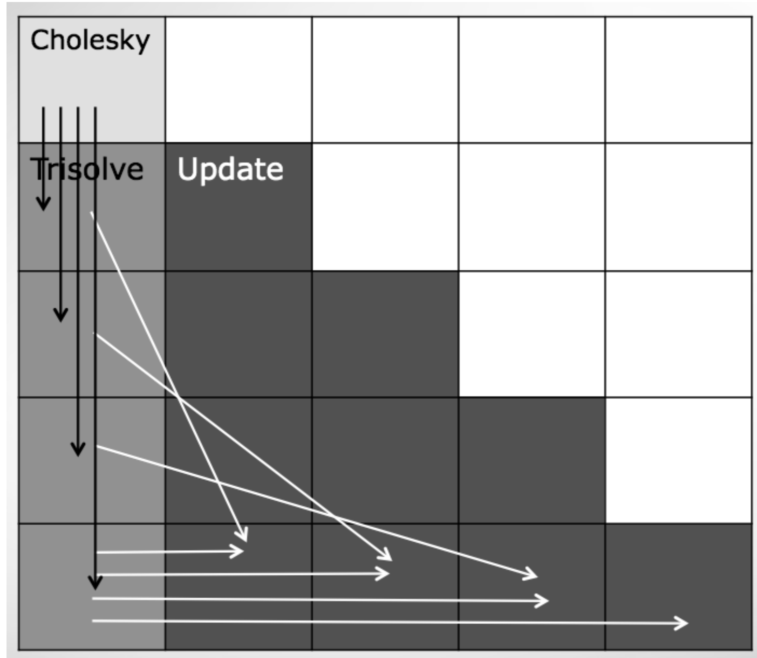

Figure 1.2: Cholesky factorization

1.3.3 Example

Cholesky factorization takes a symmetric positive definite matrix as an input and factors it into a lower triangular matrix and its transpose. The computation can be broken down into three CnC step collections. The step $(cholesky)$ performs unblocked Cholesky factorization of the input symmetric positive definite tile producing a lower triangular matrix tile. Step $(trisolve)$ applies a triangular system solve on the result of the step $(cholesky)$. Finally the step $(update)$ is used to update the underlying matrix via a matrix-matrix multiplication. Figure 1.2 shows the dependences between tiles of $(cholesky)$, $(trisolve)$, and $(update)$. For more detail, see [2].

Here we describe the four stages for developing a CnC domain specification for an application. It starts with a "whiteboard" level, which does not provide sufficient information for execution. The remaining stages provide the missing information. We use Cholesky factorization as our example.

- **Stage 1**: The whiteboard description

This stage identifies the computations, drawn in circles, and the data, drawn in boxes, as one might describe an application to a colleague at a whiteboard. It includes arrows between boxes and circles that represent the producer/consumer relations between computation and data. We call the computations *computation steps* and represent them textually in parentheses, e.g., $trisolve()$. We call the data *data items* and represent them in square brackets, e.g., $[X]$. Producer and consumer relations are represented by arrows. For example, $trisolve() - > [X]$. A full producer/consumer relationship (corresponding to a data dependence) would be represented as:

```
(trisolve) -> [X ]
[X ] -> (update)
```

In CnC, we view input/output as instances of producer/consumer relationships. The environment of the CnC graph produces and consumes data items. This is written as $env- > [X]$ or $[X]- > env$. The environment can also produce and consume control instances.

- **Stage 2**: Distinguish among the computation instances

  In the whiteboard description, a computation is processing some stream of input over an indefinite length of time. However there will be distinct instances of the computation, and each takes place somewhere in the target machine at some point in time. The programmer distinguishes among these instances by associating each instance with a distinct tag. In the case of $(trisolve)$, the instances are identified by a tag that is a $row, iter$ tuple. Thus we write $(trisolve : row, iter)$. The instances of $[X]$ are distinguished by row, column and iteration. We write $[X : col, row, iter]$.

  The tuples such as $< column, iteration >$ or $< row, col, iteration >$ are called tags. Tags are used to identify instances and distinguish among them. The term *collection* indicates that a static computation step, say $trisolve()$ is a collection of dynamic instances and a static data item, say [X], is a collection of dynamic instances.

- **Stage 3**: Identify the instances to be executed

  We can now distinguish among the instances of $(trisolve)$ by tags of the form $row, iter$. But this does not tell us if trisolve(52, 4) will be executed. Each step collection is controlled by one control collection, that determines which step instances will execute. We represent control collections in angle brackets. A control collection holds a set of tuples. In our example, $(trisolve : row, iter)$ is controlled by $< tagRowIter : row, iter >$. The relationship is represented by ::. We say $< tagRowIter : row, iter > :: (trisolve : row, iter)$. The meaning of this

relationship is that every tuple in $< tagRowIter >$ controls the execution of a corresponding instance of $(trisolve)$. During its execution it has access to the value of its tag.

In the Cholesky application, each of the three computation steps is controlled by its own control collection. In many applications, a control collection may control more than one computations step collections. Imagine an application that processes video frames. A control collection containing $frameIDs$ might control more multiple computations that are to be performed on each frame.

- **Stage 4**: Indicate how the control collections are produced

  In the case of Cholesky, the control instances are statically known. So in this case the control collections are produced by the environment. We write this as $env- >< tagIter : k >$, and similarly for the other control collections. In general, however, one computation may determine if another will execute. This is a controller/controllee relation (corresponding to a control dependence). For example, we might say:

```
foo( ) -> <barTag: tag>
<barTag: tag> :: bar( )
```

At this point we have the entire representation of the CnC coordination graph for the Cholesky application.
The CnC domain specification for Cholesky:

```
env -> [X];
[X] -> env;

<tagIter: k> :: (cholesky: iter);
<tagRowIter: row, iter > :: (triSolve: row, iter);
<tagColRowIter: col, row, Iter> :: (update: col, row, iter);

[X: iter, iter, iter] ->
   (cholesky: iter) ->
      [X: iter, iter, iter +1];
[X: iter, iter, iter+1], [X: iter, row, iter] ->
   (triSolve: row, Iter) ->
       [X: iter, row, iter +1];
[X: col, row, iter], [X: row, col, iter] ->
   (update: col, row, iter) ->
      [X: col, row, iter +1];
```

1.3.4 Execution Semantics

During execution, the state of a CnC program is defined by *attributes* of step, data, and control instances. (These attributes are not directly visible to the CnC programmer.) Data instances and control instances each have an attribute *Avail*, which has the value *true* if and only if a `put` operation has been performed on it. A data instance also has a *Value* attribute representing the value assigned to it where *Avail* is true. When the set of all data instances to be consumed by a step instance and the control instance that prescribes a step instance have *Avail* attribute value *true*, then the value of the step instance attribute *Enabled* is set to *true*. A step instance has an attribute *Done*, which has the value *true* if and only if its execution is complete.

Instances acquire attribute values monotonically during execution. For example, once an attribute assumes the value *true*, it remains *true* unless an execution error occurs, in which case all attribute values become undefined. Once the *Value* attribute of a data instance has been set to a value through a `put` operation, assigning it a subsequent value through another `put` operation produces an execution error, by the single assignment rule. The monotonic assumption of attribute values simplifies program understanding, formulating and understanding the program semantics, and is necessary for deterministic execution.

Given a complete CnC specification, the tuning expert maps the specification to a specific target architecture, creating an efficient schedule. Tag functions provide a tuning expert with additional information needed to map the application to a parallel architecture, and for static analysis they provide information needed to optimize distribution and scheduling of the application.

Strategies for Implementing the Execution Model

They key challenge for every CnC implementation is the execution model of CnC. The currently available runtimes all use dynamic strategies to schedule the execution of steps. The most interesting aspect is how a runtime detects steps to be *enabled*, in particular in absence of tag-functions. It is a trivial task to determine when a step is *prescribed* (e.g., when the tag is put), but knowing when it is fully *inputs-available* is generally impossible without tag-functions. Hence different strategies have been developed to achieve a correct step execution. All approaches have advantages and disadvantages. They make different tradeoffs between what limitations they induce, how much overhead (memory and/or computation) they imply and how visible they are to the programmer.

To avoid the loss of generality usually increased overhead is accepted with some probability. Steps can be executed “speculatively” until a needed data item is found unavailable. When this happens several actions can be taken.

- Roll-back step execution and re-play after the item became available.
  This solution implies memory overhead and depending on the implementation language the programmer must be aware of this behavior to some degree.
- Block thread until the unavailable item becomes available.
  This requires special attention to dead-lock prevention like creating a new thread. As thread (or even process) creation is a relatively costly feature the overhead in computation and memory is significant.
- Halt step execution and resume after item became available.
  Continuations require features that are not available in the common languages (yet). This solution promises to keep the computation overhead very low, but the effects on memory consumption are currently unclear.

Runtime performance can be maximized by making certain assumption about and limiting generality. The known approaches which fall into this category do not support data-dependent gets (e.g., require data-tags which depend on data and not only the step-tag). As mentioned above, full tag function availability (or equivalent information) provides the necessary information. An alternative approach limits the scope even further by requiring that the control and corresponding data are produced at the same place.

1.3.5 Programming in CnC

Expressing the domain

As a coordination language and programming model CnC must be paired with one or more programming languages to let the developer define the actual step computations. There are two approaches to connecting the rules for coordination with the actual computations. One possibility is to specify coordination and computation in the same programming language via a CnC specific API. The other is to define a new language whose only function is exactly to express the coordination and let a compiler generate the CnC glue in a computation language. Both approaches have (mostly software engineering related) pros and cons. Using a single language has two apparent advantages: it keeps the tool-chain needed to create an application short and, more importantly, it can be fully based on existing industry-standards. However, it also binds the language-independent semantics to a specific language (or even implementation). In our CnC domain language, step-collections are identified with putting their name in parenthesis `(step1)`, data-collections are put in square brackets `[data1]` and control-collections are represented in angle brackets `<control1>`. The types of data times and control tags precede the collections name. The data-tag follows the data-collection name in angle-brackets. Figure 1.3 shows side

`(step1);`	`step_collection< s1 > step1;`
`(step2);`	`step_collection< s2 > step2;`
`[double data1 <int>];`	`item_collection< int, double > data1;`
`<int control1>;`	`tag_collection< int > control1;`
`(step1) -> [data1];`	`step1.produces( data1 );`
`(step2) <- [data1];`	`step2.consumes( data1 );`
`control1 :: (step1),`	`control1.prescribes( s1 );`
`(step2);`	`control1.prescribes( s2 );`
`(step1) -> <control1>;`	`step1.controls( control1 );`

Figure 1.3: CnC semantics expressed in a CnC domain syntax and in a C++ representation

by side equivalent declarations in the domain syntax and a C++ representation, where the collections are represented as simple template classes. Both express the combined graph of Figure 1.1.

The CnC domain language expresses producer/consumer relations with arrows ``->'' and the control relation is expressed with double colons ``::''. The C++ API uses methods in the collections for the equivalent functionality. It is clear from Figure 1.3 that both representations are equivalent and that trivial transformations exist to get from one to the other.

Example: Cholesky

Domain Specification of Cholesky. As an example, the full CnC domain specification for Cholesky could look like this:

```
# step collections
(cholesky: iter);
(trisolve: iter, row);
(update: iter, row, col);
# data collection
[double X <iter, row, col>];
# control collections
<int tagIter>;
<pair tagRowIter>;
<triple tagColRowIter>;
# I/O
env -> [X: iter, row, col],
     <tagIter: iter>,
     <tagRowIter: iter, row>,
     <tagColRowIter: iter, row, col>
env <- [X: iter, row, col];
# control relations
<tagIter: iter>         :: (cholesky: iter);
<tagRowIter: iter, row> :: (triSolve: iter, row);
```

```
<tagColRowIter: iter, row, col> :: (update: iter, row, col);
# producer/consumer relations
[X: iter, iter, iter] -> (cholesky: iter) -> [X: iter, iter, iter +1];
[X: iter, row, iter], [X: iter+1, iter, iter]
   -> (triSolve: iter, row)
   -> [X: iter, row, iter+1];
[X: iter, row, col], [X: iter+1, row, iter], [X: iter+1, col, iter]
   -> (update: iter, row, col)
   -> [X: iter+1, row, col];
```

In the C++ API a CnC graph is defined within a so called `context`. For Cholesky, such a context could look like the following: [1]

```
// The context class
struct cholesky_context : public CnC::context< cholesky_context >
{
   // Step Collections
   CnC::step_collection< cholesky_step > cholesky;
   CnC::step_collection< trisolve_step > trisolve;
   CnC::step_collection< update_step > update;
   // Item collections
   CnC::item_collection< triple, tile_const_ptr_type > X;
   // Tag collections
   CnC::tag_collection< int > tagIter;
   CnC::tag_collection< pair > tagRowIter;
   CnC::tag_collection< triple > tagColRowIter;
   // The context class constructor
   cholesky_context( int _b = 0, int _p = 0, int _n = 0 )
      : cholesky( *this ),
       trisolve( *this ),
       update( * this ),
       X( *this ),
       tagIter( *this ),
       tagRowIter( *this ),
       tagColRowIter( *this )
   {
      // I/O relations
      ENV.produces( X );
      ENV.consumes( X );
      ENV.controls( tagIter );
      ENV.controls( tagRowIter );
      ENV.controls( tagColRowIter );
      // control relations
      tagIter.prescribes( cholesky, *this );
      tagRowIter.prescribes( trisolve, *this );
      tagColRowIter.prescribes( update, *this );
      // producer/consumer relations
      cholesky.produces( X );
```

[1] The current implementations are evolving. Some of the facilities described here are not available at the time of this writing.

```
      cholesky.consumes( X );
      trisolve.produces( X );
      trisolve.consumes( X );
      update.produces( X );
      update.consumes( X );

   }
};
```

There are several possibilities to express tag functions in the C++ API. One is to describe them in a separate interface, like

```
template< class T >
void tuner::depends( const triple & tag, my_context & c, T & dC ) const
{
   dC.depends(X, tag);
   if( tag.col == tag.row ) { // Diagonal tile.
      dC.depends(X, triple(tag.iter+1, tag.col, tag.iter);
   } else { // Nondiagonal tile .
      dC.depends(X, triple(tag.iter+1, tag.col, tag.iter);
      dC.depends(X, triple(tag.iter+1, tag.row, tag.iter);
   }
}
```

A sample step. Once the CnC semantics are specified, the actual computation needs to be provided. To let the runtime call the steps they have to conform to a given interface which is defined by the given CnC implementation. Here we show examples which closely follow the conventions as defined by the Intel C++ implementation. Different implementations vary in how strictly they enforce the CnC rules, such as steps having no side effects and items being immutable. Languages like C/C++ provide less capabilities to actually enforce such rules than for example Java or even Haskel. In C++ capabilities for disallowing side-effects are already exhausted with requiring steps to be "const". The input arguments to a step are its tag and the collections it has relations with. In the C++ implementation every step has access to the entire graph through the second `''context''` argument. As an illustration, the following shows the step-body of the update step of the Cholesky example:

```
// Performs symmetric rank-k update of the submatrix.
int update_step::execute(const triple & tag, cholesky_context & c) const
{
   // init local vars
   const int k = tag[0], j = tag[1], i = tag[2];
   tile_const_ptr_type A_tile, L2_tile, L1_tile;

   c.X.get(triple(k,j,i), A_tile); // Get the input tile.

   if( i==j ) { // Diagonal tile, i=j, hence both the tiles are the same.
```

```
        c.Lkji.get(triple(k+1,j,k), L1_tile);
        dsyrk( L1_tile, A_tile, ... ); // computation
    } else {    // Nondiagonal tile .
        c.X.get(triple(k+1,i,k), L2_tile); // Get the first tile.
        c.X.get(triple(k+1,j,k), L1_tile); // Get the second tile.
        dgemm( L1_tile, L2_tile, A_tile, ... ); // computation
    }

    c.X.put(triple(k+1,j,i), A_tile); // output for the next iteration
}
```

This code has no side-effects other than putting a new tile into the data-collection: it uses only local variables, it has no status and does not overwrite data. In principle, the computation follows the pattern "consume (get) input → compute → produce (put) output". More interestingly, this code illustrates that relations in CnC can be conditional. Different instances of this step consume different numbers of data items: when computing the diagonal it depends on only two input tiles, while on nondiagonals three inputs are consumed. Even though this example puts a condition only on the consumer relations, producer and controller relations can of course also be conditional.

The Environment. The environment in a CnC program is responsible for instantiating the actual CnC-graph instance and producing initial data control. The latter is done with the same interfaces as used by steps. Between consuming the output and producing input and control, the environment can do arbitrary computation. A simplified environment code for Cholesky could look like this:

```
// Create an instance of the context class which defines the graph
cholesky_context ctxt;
// produce input matrix
for(int i = 0; i < N; i++) {
    for(int j = 0; j <= i; j++) {
        ctxt.X.put( triple(0,j,i), init_tile(j,i) );
    }
}
// produce control tags
for(int k = 0; k < N; k++) {
    ctxt.tagIter.put( k );
    for(int j = k+1; j <= N; j++) {
        ctxt.tagRowIter.put( pair(k,j) );
        for(int i = k+1; i <= j; i++) {
            ctxt.tagColRowIter.put( triple(k,j,i) );
        }
    }
}
// Wait for all steps to finish (optional)
ctxt.wait();
// get result
```

```
for (int k = 0; k < N; k++) {
   for (int j = k+1; j < N; j++) {
      tile_const_ptr_type tile;
      ctxt.X.get( triple(j+1,j,i), tile );
      do_something( tile );
   }
 }
```

Distributed memory

A CnC domain specification never refers to a particular place (e.g., an address) or time (e.g., "now"). However, it explicitly identifies all entities needed to map the program execution to distributed memory: control instances identify what needs to be computed and data instances identify the data needed for the computations. Moreover, through gets and puts a CnC program has explicit hooks at all places in the code relevant for distribution and so does not require explicit message handling. The runtime knows exactly which data and control instances are needed, there is no need to infer this information indirectly like other approaches such as virtual shared memory systems or ClusterOpenMP.

Intel Concurrent Collections for C++ allows any legal CnC application to run on distributed memory. For this, only minimal additional coding is needed. Besides a trivial initialization variable, only marshaling capabilities for nonstandard data types need to be provided. The latter is a limitation of the underlying programming language C++, because it does not provide marshaling features on the language level. Still, marshaling in this system is very simple and requires the developer to provide the marshaling functionality only. The runtime will take care of using it at the appropriate places.

Distributed evaluation is available in two modes, automatic and user-guided. In nontrivial cases automatic distribution requires tag-function and their analysis to achieve acceptable efficiency. In the general case the runtime can work distributed even when no tag functions are available. This default behavior requires significant bookkeeping and overhead. When a step is executing, the runtime cannot know where (e.g., on which process) an unavailable item was or will be produced. As a consequence, it needs to synchronize data between processes, leading to broadcast-like communication patterns (or worse). Nevertheless, applications with few data-dependencies between steps can still perform well with the automatic distribution feature.

As an example for distributing the computation, the following code would partition a two-dimensional space (like a matrix) in a row-cyclic manner:

```
int tuner::compute_on( int row, int column ) {
   return row % numProcs();
}
```

The runtime will distribute rows cyclically across the processes and all computations within the same row will be computed on the same process. For column-cyclic distribution one would use something like:

```
int tuner::compute_on( int row, int column ) {
   return column % numProcs();
}
```

For convenient experiments with different distribution plans and/or to make more dynamic decisions, one could also provide several options and decide on the fly which distribution to use. Unlike with explicit message passing approaches changing the distribution plan does not require any changes to the step-codes.

Tag-functions significantly simplify changes to the distribution. The consumer-steps can be determined by the tag-functions. With this information the runtime uses `compute_on` to compute the address space where the consuming step-instances will be executed. This mechanism is very convenient because it allows changing the distribution plan only for the computation and the runtime will automatically send the data to where it is needed.

In theory tag-function analysis allows computing good distribution plans automatically and research on this is in progress. Much of the existing literature on automatic loop transformations and automatic data distributions applies. There are several other differences. For example, dynamic single assignment makes our problem easier than the general case.

1.3.6 Futures

There are several related topics for future work. The first is a hierarchical domain specification. In such a specification, what looks like a step at one level might actually be a CnC graph at the level below. The inputs/outputs from/to the environment of the lower-level graph would look like inputs/output to/from the higher level step.

Another potential area of future inclusion is support for reuse or libraries of existing subgraphs. This might involve hooking up an input of one subgraph to the output of another or it might be accomplished by using hierarchy. In this case the reused subgraph would be the lower level implementation of what looks like a step at the level above.

Commonly used abstract patterns like tree walks, reductions, joins, streams are also being investigated for future inclusion. The user would need to supply some missing information to make each abstraction concrete.

CnC is deterministic. Some applications are inherently nondeterministic, for example, any that depend on a random number or have a “choose any one” component. Support nondeterminism, but in a controlled way, is also planned. For example, we might be able to continue to support our guarantees but only for a subgraph of the graph of the whole program.

Standard static analyses based on CnC domain specifications will be implemented. They are easier and more effective in CnC. Also an application-specific CnC attribute graph can serve as a base for more aggressive static analysis.

1.4 CnC Tuning Language

The CnC tuning language is separate from the CnC domain language. The tuning specification for an application is indicated by a high-level declarative language. A CnC tuning specification and domain specification are distinct specifications written in distinct but related high-level declarative languages. The tuning specification cannot violate the constraints of the domain specification. The domain specification is written by a domain expert. This is a person with expertise in finance, graphics, chemistry, etc. The tuning specification is written by a tuning expert. This *tuning expert* might be the same person as the domain expert but at a later time, a different person with expertise in tuning, a static analyzer, an auto-tuner, etc.

Since the domain specification indicates constraints based on the application only, a single domain specification may be used for a wide range of scenarios (distinct architectures, configurations, or goals). There are two caveats. First, the appropriate grain may vary among targets. That may simply be reflected in a parameterized grain but if the appropriate grains are very different, the static graphs may have to be different. Second, if the target machines are too different, different algorithms might be needed. Even with these caveats, multiple tuning specifications may be used with the same domain specification. So conceptually they must be distinct specifications. But the tuning specification makes reference to collections in the domain specification and may add additional collections and addition relations (constraints). Together they form a single tuned CnC specification.

The domain specification exposes the potential parallelism in the application. Since it only indicates the semantic constraints, there is typically more than enough parallelism exposed. The tuned application must obey the constraints provided by the domain specification but that leaves many possible legal executions (an execution is a mapping across the platform and through time). The application will execute on a specific architecture, with a limited capacity and its own performance characteristics. The job of the tuner then, is to improve performance of the application for a specific platform by guiding execution away from or actually eliminating some semantically legal executions that result in poor performance. A primary tuning focus is to improve temporal and spatial locality.

The separation of domain and tuning concerns isolates the domain expert from the tuning facility. This isolation allows the tuning language to provide strong capabilities for control and flexibility without complicating the work of the domain expert. From the tun-

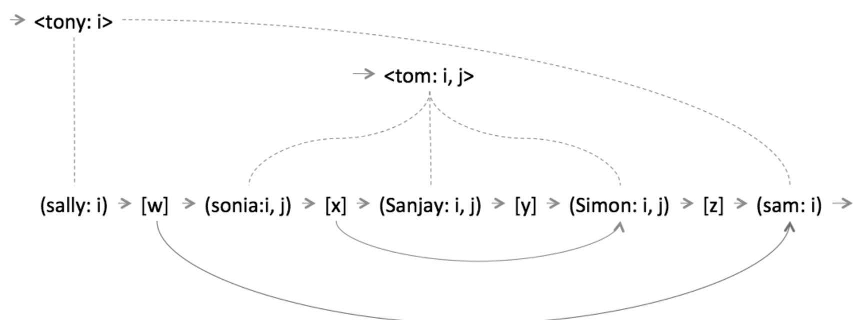

Figure 1.4: Sample CnC graph

ing expert's perspective, the value of this isolation is that he/she does not have to wade through application code.

1.4.1 Description

The tuning specification refers to the step, item, and control collections in the domain specification. Although separate specifications are written for domain and tuning, some mechanism such a build model is used to integrate the two specifications into a single specification. This integration supports the use of the domain collections and their instances in the tuning specification. The tuning specification generates tags of its own and adds steps of its own, basically extending the application.

The basic concept of the tuning language is the *affinity collection*, a set of computations (collection of steps) that the tuner suggests should be executed close in space and time. We sometimes refer to an affinity collection as a *group* of step collections. Affinity collections could be generated by static analysis. The next basic concept is a hierarchy of these collections: *hierarchical affinity collections* (HACs). Hierarchical affinity collections allow the specification of relative levels of affinity, with tighter affinity at lower levels. Computations that touch the same data will not benefit from locality if they are too far apart in space or time. Hierarchical affinity collections are the tuning mechanism for indicating computations that must be proximate in both time and space. This is the highest concern. The tuning language provides additional separate mechanisms for each of space and time.

Hierarchical Affinity Collections

Consider the CnC graph in Figure 1.4. An option for a hierarchical affinity collection for this graph is shown in Figure 1.5. Assume for our examples that all step collections

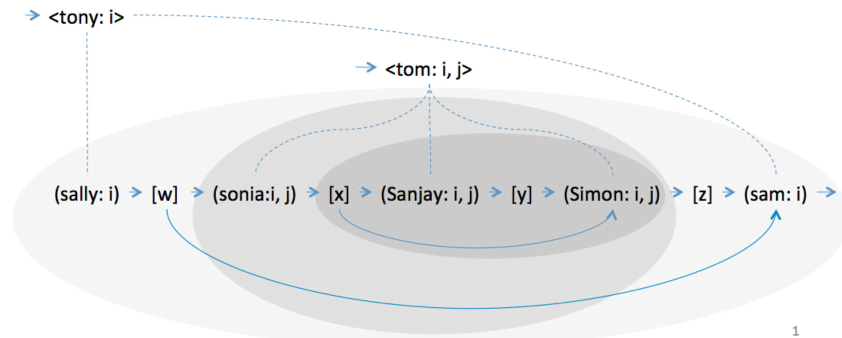

Figure 1.5: Affinity graph

in Figure 1.5 form an outer affinity collection. A second grouping option would to be have a single inner affinity collection made up of the step collections $(sonia)$, $(sanjay)$, and $(simon)$. A third option would be to have step collections $(sonia)$, $(sanjay)$, and $(simon)$ form an affinity collection within the outer affinity collection and step collections $(sonia)$ and $(sanjay)$ form an affinity collection within that one. The domain specification does not imply a particular affinity grouping. This decision is the work of the tuning expert and may depend on the target architecture, configuration, characteristics of the data set, goal (e.g., power vs speed) etc.

We will show the tuning language textual representation for the hierarchical affinity collection for Figure 1.5. We use curly brackets to show the hierarchical nesting of affinity collections:

```
{(sally)
 (sam)
   {(sonia)
      {(sanjay)
       (simon)
      }
   }
}
```

Affinity collections have names:

```
{groupa:
      (sally)
      (sam)
      {groupb:
            (sonia)
            {groupc:
```

```
                  (sanjay)
                  (simon)
            }
      }
}
```

We need a mechanism to indicate the dynamic instances of affinity collections. In the domain language, steps are used to control conditional execution of step instances. In the tuning language, these collections are prescribed to control which specific instances of the collection will exist.

```
<tony: i> ::
      {groupa: (sally: i)
              (sam: i)
                    <tom: i, j> ::
                          {groupb:   (sonia: i, j)
                                 {groupc: (sanjay: i, j)
                                       (simon: i, j)
                                 }
                          }
      }
```

The above example illustrates the prescription of affinity collections. Recall that step collections are a static construct and we indicate the instances of a step by a prescribing control collection. An instance in the prescribing control collection corresponds to a step instance that will be executed. Here an affinity collection is a static construct, and an instance in the prescribing control collection corresponds to a collection instance that will be created. So this specific statement means that for each instance of a tag $< tony : i >$ there will be a corresponding instance of the affinity collection $\{groupa : i\}$.

We will also use prescription to indicate when the tuning actions associated with an affinity collection will take place. The affinity collection action is controlled by a control collection.

Time- and space-specific mappings

The hierarchical affinity collections enable the tuning expert to express affinity within space-time. It is important to provide this general way of expressing locality without requiring a distinction between space and time. But, of course, a tuning expert may want to distinguish spatial and temporal locality. These specific controls are presented here.

Space. The tuning expert has access to the facilities in Figure 1.6 to express specific distributions in space.

For example:

Form	Example
Distribute across *level*	Distribute across sockets
Distribute across *level* via *function*	Distribute across sockets via F(j, k)
Replicate across *level*	Replicate across address_spaces

Figure 1.6: Space-specific mapping options

```
<groupTag: j> :: {groupOuter: j replicate_across address_spaces}
```

Here for each tag the in control collection $< groupTag >$, the corresponding instance of $groupOuter$ will eventually be executed on all address spaces. $replicate_across$ is a keyword. $address_spaces$ derives from the description of the platform tree.

```
<groupTag> :: {groupOuter distribute_across sockets
      {groupInnerA}
      {groupInnerB}
      {groupInnerC}
      {groupInnerD}
}
```

Here the four inner affinity collections are distributed among the nodes in the tuning tree holding sockets. $distribute_across$ is a keyword. $sockets$ derives from the description of the platform tree. In the previous example the components to be placed in space were dynamic instances of the same static affinity collection. In this example the components are statically distinct affinity collections. The distribution annotation distributes the (static or dynamic) components of the annotated affinity collection according to user functions.

```
<groupTag: j> :: {groupOuter: j distribute_ across address_spaces via f(j)}
```

Here when a tag j arrives in $< groupTag >$, the corresponding affinity instance is place on the queue associated with address space $f(j)$.

Time. The tuner may indicate that the set of components in an affinity collection should execute in an order specified by a priority or that they are unordered. The set of components of an affinity collection may be required to run one at a time (or at most N at a time). These two possibilities for each of two traits result in four possible situations as seen in Figure 1.7.

If the set of components are to be executed in an arbitrary order and they may overlap, they are not additionally constrained with respect to time. If they are to start according to a

	Ordered	Unordered
Non-overlapping	serial/barrier	exclusive
overlapping	priority	arbitrary

Figure 1.7: Time-specific mapping options

priority and they are not to overlap then they execute serially with a barrier between them. (Notice that the component may itself be an affinity collection that executes in parallel.) The other two are the interesting cases. Consider image processing where frames are entering the system. We want the processed frames to exit in order. Here we will have a priority order for starting the work on a frame but we do not want to require that there is a time consuming barrier between them. This is the ordered/overlapping case. The other interesting case, unordered but nonoverlapping might be used for components with a large memory footprint. In this case we want to run one at a time but we may not care about the specific order. This case can be generalized from one at a time to N at a time.

1.4.2 Characteristics

1. The CnC tuning specification is declarative.
2. It is isolated from the domain specification.
3. Determinism remains intact, although the tuning language can be used to specify different mappings of data and computation in time and space, in accordance with differing architectures and goals.
4. The hierarchical affinity collection mechanism guides but does not force specific orderings. It does not explicitly say what goes on in parallel, or that this computation executes before that (just that they execute close in space and time).
5. It provides time- and space-specific mechanisms that provide more control than the basic hierarchical affinity collection.

1.4.3 Examples

Here we describe the four stages for developing a CnC tuning specification for an application. We again use Cholesky factorization as our example.

- **Stage 1**: Whiteboard

  In the case of tuning, the whiteboard level is simply an indication of the affinity hierarchy. There are a variety of possibilities for tuning Cholesky. For this example we create an inner affinity group that includes $(trisolve)$ and $(update)$. An instance of $(trisolve)$ is used with many of the instances of $(update)$. We call this inner group $groupTU$. We choose for $(cholesky)$ to have a weak affinity with that inner group since an instance of $(cholesky)$ is used with many instances of it. We will call this $groupC$. In this tuning specification we have two distinct hierarchical affinity collections, $groupC$ and $groupTU$. Further we can define them as: $groupC(cholesky)$ and $groupTU(trisolve), (update)$.

- **Stage 2**: Distinguish among the computation instances

  As with steps, groups also have instances. We need to distinguish instances so we can map and schedule distinct instances independently. Instances of $groupC$ will be distinguished by the tag $iter$, so we write $groupC : iter$. Instances of $groupTU$ are distinguished by the tag $row, iter$ so we write $groupTU : row, iter$.

- **Stage 3**: Identify the instances to be executed

  These two steps have two different sets of instances, so they will require different control collections. In this case the required control collections already exist in the domain specification. There is an instance of $groupC$ for each tag in $< CholeskyTag >$ and there is an instance of $groupTU$ for each instance of $< TrisolveTag >$, so these are the two control collections.

- **Stage 4**: Indicate how the control collections are produced

  The control collections are produced in the domain specification, so there is no reason to produce them again here. These two specifications will exist together in the full CnC specification.

For this example, we have added affinity collections but the control, item and step collections all exist in the domain specification. This is often the case, but in general we can augment the domain specification with additional control, item, and step collections that are purely in support of tuning and have no semantic impact. Suppose, for example, we want to create super-tiles to form affinity groups. (The individual step instances may be tiled, of course, but this would create groups of those step tiles that would execute close in time and space.) In this case the tuning specification may input a problem size, tile size, platform configuration size, and/or a goal (power or time). From this input, a new step might generate the number of rows and columns within a super-tile and/or the number of

rows and columns of super-tiles. This process would include additional, step, item and control collections. These can then be referenced within the tuning specification. For this example, the control collection controlling an affinity collection would be a newly computed control collection that identifies the super-tiles. These new collections might have constraint relationships among them and there might be constraint relationships from the domain collections to these new tuning collections. In addition, the tuning specification might even include new constraint relationships among the domain collections. None of these would have semantic implications for the domain applications but the new orderings would be enforced as part of the tuning process.

```
// iters have no affinity with each other
   <CholeskyTag: iter> :: {GroupC: iter
         // all the work for a given iteration has a weak affinity
         (cholesky: iter)
         <TrisolveTag: row, iter> :: {GroupTU: row, iter
           // the work for a given iter and row has a strong affinity
            (trisolve: row, iter)
            (update: col = (iter+1 .. N), row, iter)}}
```

As we have seen, there can be a variety of tuning specifications for Cholesky. Let us examine the tuning specification above. At the outermost level, there is an affinity collection $\{groupC\}$ for each value of $iter$. The distinct instances of $\{groupC\}$ are not components of any collection so there is no affinity among them. But inside a single instance of $\{groupC : iter\}$, the multiple components have an affinity with each other. One is the instance of $(cholesky)$ for this value of $iter$. The others are instances of the $\{groupTU\}$ affinity collection for this value of $iter$ and for multiple values of row. The set of $row, iter$ instances is determined by tags in the control collection $< TrisolveTag >$ from the domain specification. Within an instance of the $\{groupTU : iter\}$ affinity collection there are multiple components: $(trisolve : iter)$ and $(update : col = (iter + 1..N), row, iter)$. The value of $iter$ referenced by these components is that of their parent instance $\{groupC : iter\}$. One component is $(Trisolve)$. There are multiple $(Update)$ components. The exact number is a function of the values of $iter$ and N. There is reuse here in that this instance of $(Trisolve)$ produces a result that is used by each $(Update)$ instance in the same row. Note the scoping of the tag components. Since $(cholesky : iter)$ is within $\{groupC : iter\}$, the values of $iter$ are the same. Since $\{groupTU : iter\}$ is within $\{groupC : iter\}$, the values of $iter$ are the same.

The tuning specification above for Cholesky is an example of an iterative style specification based on control collections. We anticipate that this will be a commonly used style. Another option is a recursive style. The following tuning code for Cholesky uses a recursively defined affinity collection.

```
<iter= 1> :: {OneIter: iter
          {tri-first: iter
             (cholesky: iter)
             (trisolve: iter+1, iter)
             (update: iter+1, iter+1, iter)
             // recursive definition of {OneIter}
             {OneIter: iter+1}}
          <trisolveTag: row, iter> and row > iter+1 ::
             {tri-up-rest: iter
                (trisolve: row, iter)
                (update: col = (iter+1 .. N), row, iter)}}
```

The $\{tri - first\}$ affinity collection processes the three top tiles: $(cholesky)$, the top $(trisolve)$, and the one $(update)$ to the right of that $(trisolve)$. Then it recurses to the next $iter$. The rest of the $(trisolves)$ and $(updates)$ are a second component. The first component corresponds to the critical path, along the main diagonal of the matrix. The first component cannot complete its recursions without some of the results from instances of the second components. These constraints are part of the domain specification and do not need to be repeated here. The control tags for $\{oneIter\}$ are not from the domain specification. They are defined explicitly in the tuning specification. They begin at $< iter = 1 >$ and recurse via the statement $\{OneIter : iter + 1\}$.

One more possible tuning for Cholesky would be a tiling. Most tiles would be two dimensional rectangles, a set of instances for the same iteration and for a neighborhood of rows and columns. Tiles along the diagonal would be triangular. For this tuning specification, prescriptions identify tiles. There is no concept of a tile in the domain specification. The tuning specification has to create a new control collection that identifies tile instances. This will involve new step collections that compute the tile tags. These tags, steps and items are in the language of the domain specification but are not part of the domain specification. They belong to the tuning specification, and will differ among tuning specifications. For instance they were not used in our initial version of Cholesky, nor in the recursive version. We have shown three distinct tunings for Cholesky. The first followed the loop structure of the naive code and focused on reuse of the result of the $(trisolve)$ computation. The second was recursive. The third was tiled. The domain specification remained untouched for all three.

1.4.4 Execution Model

An execution model is needed to implement the higher level tuning language. Here we describe one such execution model. Other execution models are possible.

The foundation of the execution model is a representation of the target platform. We assume only that the platform is hierarchical. A description of this hierarchy is used as

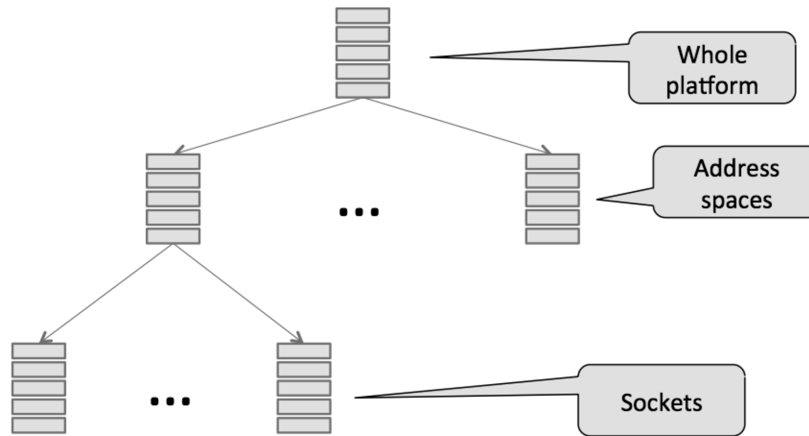

Figure 1.8: Tuning tree

the foundation of the tuning commands. The platform description names each level, for example, $Level1$, $Level2$, etc. or it might be $address_space$, $socket$, $core$, etc..

The execution model for hierarchical affinity collections is as follows. We distinguish between two components of the CnC runtime: the tuning component and the domain component. The tuning component serves as a staging area for the execution of step instances in the domain component. All tuning actions belong to the tuning component.

The tuning component consists of four parts.

- Tuning actions: one for each affinity collection. These specify the low-level processing for that collection in the tuning tree. The tuning actions control the flow of work to the domain runtime.
- Event handlers: one for each control collection in the tuning specification. These control which instances of the tuning actions take place. When a tag in the normal domain execution becomes available, the handler for that event will cause some dynamic action instances to be instantiated.
- Queue manager: one for each queue. These control when to remove items from the queue and execute them.
- The tuning tree (see Figure 1.8): same shape as the platform tree. There is a work queue associated with each node in the tuning tree. The items in the queue are either static affinity collections/steps or dynamic instances of affinity collections/steps.

```
<choleskyTag: iter> :: {groupC: iter
   (Cholesky: iter)
   <trisolveTag: row, iter> :: {groupTU: row, iter
               ...
   }
}
```

```
The action {groupC: iter} is defined as {

    insert the dynamic step instance (Cholesky: iter) on some child
    insert the static group <trisolveTag: row, iter> :: {groupTU: row, iter} on some child
}
```

Figure 1.9: Tuning action example

> Each queue contains work that is ready for an action to be performed (such as moving down the tree) and work that is not ready. An instance is *ready when its associated tag is available. The tuning runtime system selects from a queue the ready work item(s) that are nearest the head of the queue.*

Large static outer affinity collections start at the top of the tuning tree. As an affinity collection is moved down a level in the tree, it will be decomposed into its components. Since components of a collection at some node only move to children of that node (there is no work stealing), they have a tendency to remain close in the platform, in that nodes in the tuning tree correspond to nodes in the platform tree. To the extent possible, affinity collections are moved down from a node in order of their arrival, so the components of a collection have a tendency to remain close in time. Of course there is a significant opportunity here for interesting policies (not addressed here) and for the tuning expert to be more specific about when affinity collections are moved and to where.

For example, the static outer level affinity collection $Cholesky$ can have an action defined where an instance of $Cholesky$ is moved down to a child node and the static affinity collection $groupTU$ is unpacked and moved down to a child node. See Figure 1.9, which shows pseduo-API code for an action on the Cholesky static affinity collection. An action defined on a leaf node can move instances of a static affinity collection or dynamic affinity instances into the domain runtime for execution.

Control tags

In previous examples, we have used control collections from the domain specification to prescribe collections. We use tags in the execution model to determine when the actions on those instances are to be performed.

Recall that instances of steps, items and tags from the domain specification acquire attributes as the program executes. We also associate attributes, representing state changes, with affinity collection instances. Collections are similar to steps. Instances of affinity collections can be prescribed (their control tag is true). They can also be executed in the domain runtime. Although steps have an attribute $inputs_available$, affinity collections currently do not.

Strategies for Implementing the Execution Model

The key to efficient execution on distributed memory is how data and work are distributed and/or shared across address spaces. Hence the system allows the developer (e.g., the tuning expert) to specify a distribution plan which defines how data and work is mapped to the address spaces. In the spirit of separating the domain from tuning, this plan is defined in a tuning layer which is separate from the step code. In a structure called "tuner" the programmer specifies where data- and step-instances should be placed. Tuners are separate objects attached to collections when they are initialized. Whenever the runtime needs this information it will calls the respective tuning callback and takes the required actions.

1.4.5 Futures

When using prescription for steps, the prescription not only determines which instances will execute but also has some influence on when (some time after the prescribing tag is produced). For prescribing steps, the question of "when" is secondary. In the case of prescribing affinity collections, the control of "when" is a major goal of tuning. We can provide the tuning expert with more control over when affinity collection actions occur so that the tuning process can be more effective in controlling when computations are fed to the domain runtime.

The state of instances can be used to refer to points in the partial order of execution, e.g., $(foo : i).executed$, and also to identify new points in the partial order, e.g., $(foo : i).executed\ and\ [x : i + 1].available$. That is, the partial ordering on instance/attribute pairs is used to indicate a "time" within the execution of a domain specification. We can also allow the tuning specification to refer to these attributes for better control of when tuning actions should be performed. The action associated with an affinity collection can take place when the attribute expression associated with the collection holds. With this mechanism, for an affinity collection instance to be ready to execute, its attribute expression must hold.

Step instances have an attribute $inputs_available$ and cannot execute until this attribute evaluates to true. Affinity collection instances can also have an attribute $inputs_available$ as part of the mechanism to control when they execute.

Parallel prog. model	Declarative	Deterministic	Efficient
Intel TBB	No	No	Yes
.Net Task Par. Lib.	No	No	Yes
Cilk	No	No	Yes
OpenMP	No	No	Yes
CUDA	No	No	Yes
Java Concurrency	No	No	Yes
Det. Parallel Java	No	Hybrid	Yes
High Perf. Fortran	Hybrid	No	Yes
X10	Hybrid	No	Yes
Linda	Hybrid	No	Yes
Asynch. Seq. Processes [3]	Yes	Yes	No
StreamIt	Yes	Yes	Yes
LabVIEW [11]	Yes	Yes	Yes
CnC	Yes	Yes	Yes

Table 1.1: Comparison of several parallel programming models.

1.5 Current Status

CnC implementations currently include those with computation languages such as C++ (based on Intel ® Threading Building Blocks), Hababero Java (based on Java Concurrency Utilities), and .NET (based on .NET Task Parallel Library). Other existing implementations have Java, C with OpenMP, Scala, Haskell, Python, Habanero Java, and a subset of MATLAB as the computation language. An implementation of the tuning runtime described above is under development at Rice University.

1.6 Related Work

Table 1.1 is used to guide the discussion in this section. This table classifies programming models according to their attributes in three dimensions: *Declarative*, *Deterministic* and *Efficient*. A few representative examples are included for each distinct set of attributes. 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, .Net Task Parallel Library [10], Cilk, OpenMP, NVIDIA CUDA, Java Concurrency [9]—are non-declarative, nondeterministic, and efficient. Here a programming model is considered to be efficient if there are known implementations that deliver competitive performance for a

reasonably broad set of programs. Deterministic Parallel Java [1] 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) [7], X10, Linda [4]—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.

1.6.1 CnC Domain Language

Linda was a major influence on the CnC domain language design. CnC shares two important properties with Linda: both are coordination languages that specify computations and communications via a tuple/tag namespace, and both create new computations by adding new tuples/tags to the namespace. However, 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 data collection. This is a key reason why Linda programs can be nondeterministic 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 [3] is a recent model with a clean semantics, but without any efficient implementations. In contrast, the remaining three entries are efficient as well. StreamIt [5, 6] is representative of a modern streaming language, and LabVIEW [11] 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.

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. Further, CnC's dynamic `put`/`get` operations on data and control collections serves as a general model that can be used to express many kinds of applications that would not be considered to be dataflow or streaming applications.

1.6.2 CnC Tuning Language

Most of the existing parallel programming constructs address time. Data parallel constructs, both fine-grained vector constructs and the coarser Parallel For constructs such as found in OpenMP, indicate a set of operations that can occur at the same time. Fork-join constructs such as a Cilk spawn/sync or Habanaro Java's async/finish or parallel sections are all of this flavor. They indicate when the forked work can start and when the join work can proceed. Task graphs indicate a more general partial ordering.

These approaches indicate when computations take place. Some of these languages have distinct constructs for indicating where a computation is to take place, for example, Habanero Java has the Hierarchical Place Tree (HPT) [12]. But the constructs for time and for space are distinct and unrelated. HPF provides facilities, for coarse-grain and array language constructs, for defining the decomposition and placement of data for distribution across processors and address spaces.

Hierarchical affinity collections provide a mechanism that allows the programmer to specify locality, while allowing but not requiring him to distinguish between spatial and temporal locality. The programmer can optimize the space-time locality, at times trading off temporal and spatial locality. Allowing the programmer to specify space-time locality is a novel contribution.

1.7 Conclusions

Concurrent Collections is a programming model for parallel systems. Instead of providing facilities for the domain expert to describe the parallelism explicitly, it provides facilities for describing execution order constraints. Subject to these constraints, computations can potentially execute in parallel. This depends only on the application. This approach creates a partial ordering of computations and typically supplies more than ample parallelism. There is a separate language for writing CnC tuning specifications. The central component of the tuning language is hierarchical affinity collections. These provide a mechanism that allows the programmer to specify locality, while allowing but not requiring him to distinguish between spatial and temporal locality. This is a novel contribution. The programmer can optimize the space-time locality, at times trading off temporal and spatial

locality. The tuning facility enables the tuning expert to remove some of the less efficient possible mappings of the parallelism provided by the domain specification. Without the tuning language, CnC has already achieved comparable performance with other parallel models/tools.

References

[1] 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.

[2] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari. Declarative aspects of memory management in the concurrent collections parallel programming model. In *Proceedings of DAMP 2009 Workshop (Declarative Aspects of Multicore Programming)*, 2009.

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

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

[5] M. I. Gordon et al. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM.

[6] M. I. Gordon et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[7] 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.

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

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

[10] Stephen Toub. Parallel programming and the .NET Framework 4.0. http://blogs.msdn.com/pfxteam/archive/2008/10/10/8994927.aspx, 2008.

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

[12] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and date movement. In *Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing*, 2009.