The Tuning Language for Concurrent Collections

Kathleen Knobe[1] and Michael G. Burke[2]

[1] Intel Corporation
kath.knobe@intel.com
[2] Rice University
mgb2@rice.edu

Abstract. Concurrent Collections (CnC) is a programming model for parallel systems. A novel aspect of this model is that there is a clear separation of the specification of the application semantics, called the domain specification, and the tuning specification, which maps the domain spec to a platform. The domain spec is declarative specification that indicates the constraints on execution. These correspond to data dependences and control dependences. The domain spec can be written by a domain expert who does not necessarily have knowledge of the target architecture. The separation of concerns isolates the domain expert from the tuning facility. This isolation permits the tuning language to provide strong capabilities for control and flexibility for a tuning expert. In addition, the separation means that a given domain spec can have multiple tuning specs. There might be multiple targets, and within the same target there might be distinct goals, such as performance vs. power. This paper introduces the CnC language for tuning specifications in some detail and show examples of its use. The goal of the tuning language is to provide capability for mapping the parallelism implicit in the domain spec for high-performance execution on a target platform. The focus of this activity is locality. The basic concept in the tuning language is the affinity group, a set of computations that the tuner suggests executing close in time and space. Hierarchical affinity groups allow the specification of relative levels of affinity. They provide a mechanism that allows the programmer to specify locality, while allowing but not requiring him to distinguish between spatial and temporal locality. We will use Cholesky factorization to show the use of multiple tuning specs for a single domain spec. The tuning spec is under active design.

Keywords: Concurrent Collections, parallel programming, performance tuning

1 Introduction

Concurrent Collections (CnC) is a parallel programming model that supports the separation of two distinct components: the domain specification describes the application; the tuning specification indicates how the application is to be mapped to a specific target architecture [1, 3]. [3] This approach leads to the pos-

[3] This research has been funded in part by DARPA in the Ubiquitous High Performance Computing (UHPC) program, contract HR0011-10-3-0007.

sibility of a single domain spec and multiple tuning specs for a given application. The tuning specs vary with the target or the tuning goal.

The domain expert writing the domain spec does not indicate what executes in parallel. That is difficult and depends on the architecture. Instead he just indicates the semantic constraints on the ordering of computations in the application. This specification depends only on the application, not on the target architecture. This allows it to be reused for a variety of targets.

Position Many parallel languages have a serial backbone. For these languages, parallel constructs (data parallelism, fork-join, etc.) are embedded in the serial code. After the parallel construct has executed, control returns to the serial backbone. There is no serial backbone in CnC. A CnC domain specification indicates only the semantically required ordering. Here all the potential parallelism is implicitly exposed. The goal of tuning is to guide the schedule of computations through time and the mapping of the computations across the platform. Because the platform is limited, we need to reduce the parallelism. The main motivation driving the tuning decisions is typically to improve temporal and spatial locality for the given application on the given platform. The tuning language described here provides general capabilities for the tuning expert to map the application to a target architecture. The tuning language is currently under active design and implementation.

Separation of concerns 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. For example, the producer of a value must execute before the consumer of that value. These are the only constraints imposed by the domain spec. There is no arbitrary serial ordering of statements, only the partial ordering based on the dependences. These constraints are based on the application logic, not on the target architecture. The domain spec exposes the parallelism in the application, but the domain expert does not think about parallelism. For the domain expert, this separation simplifies the description of the application.

Tuning is inherently difficult. The domain expert is isolated from the tuning facility. This isolation allows the tuning language to be as complex as needed to provide strong and general capabilities for controlling the execution of computations in time and space across the target platform.

Determinism The step instance as a whole has no side effects and is a pure function of its input data items. We also require that each data item instance is associated with a unique value: items obey dynamic single assignment. This combination ensures that the CnC spec is deterministic. The same spec with the same input can run on a thousand cores or on a single core, if it fits. It will produce the same results. These properties also mean that CnC programs are serializable. These requirements on steps' external behavior do not mean that the steps need to be written in a functional language. They may, in fact, use side effects internally.

Determinism in the domain spec 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. The tuning spec cannot violate the constraints of the domain spec. So 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.

Related Work 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 are specific to time, indicating when computations take place. They say nothing about where. 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) [4]. But the constructs for time and for space are distinct and unrelated. High Performance FORTRAN (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 groups 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.

Assembly language for tuning The tuning language is fairly low level. It is intended for a scenario in which more of the burden of mapping decisions falls on the tuning expert than the runtime. The tuning expert specifies tuning actions and the control conditions under which the actions are to take place. We consider this a tuning assembly language because it is very specific about what to do when, and it could be generated by static analysis. Subsequent versions may raise the level of language but this is our starting point. The goals are:

- To gain experience with a low-level set of facilities and make early corrections to gain perspective and the right set of general facilities;
- to provide maximum control for application studies.

We assume that the application spec already exists and will not be modified. There are some additional limitations of this first design, which will be removed in subsequent designs. For now the domain spec is assumed to be flat; i.e., it has no hierarchical step collections. We are not yet concerned about the syntax of the tuning language.

In Section 2 we describe the CnC domain language. Then we describe the CnC tuning language.

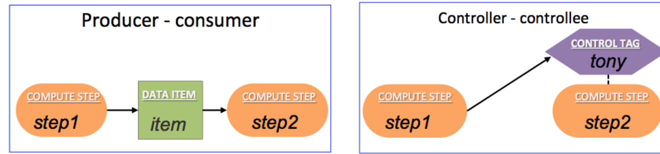

Fig. 1. Ordering requirements

2 CnC Domain Language

The CnC domain spec 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 spec. 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.

A computation step that produces a data item must execute before the computation step that consumes that data. A computation step that produces a control tag must execute before the computation step controlled by that control tag. These entities and relationships form the nodes and edges of a graph as illustrated in Figure 1. The CnC domain language coordinates among computation steps. These computation steps are written in a programming language. Currently these include: C++ (Intel), Java (Rice), and Haskell. They will soon include Chapel and Python. Support for FORTRAN and Matlab is in progress.

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). 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.
A static computation step represents a collection of distinct dynamic computation step instances. The static data item is a collection of distinct dynamic data item instances. These instances will be placed across the target platform and scheduled in time. We distinguish among the instances by a tag. A tag is a tuple, for example $< row, col >$ or $< social_Security_num, date >$. A control tag has the same status in a domain spec as a data item.

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. In this paper, we assume that all tag functions are the identity function.
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

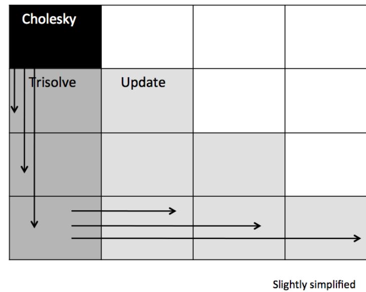

Fig. 2. Cholesky factorization

case, the producer might be a computation step or the environment, as shown in Figure 1. For more detail, see [1, 3].

In the remainder of the paper 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.

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 2 shows the dependences between tiles of $(cholesky)$, $(trisolve)$, and $(update)$. For more detail, see [2].

The simplified CnC domain spec for Cholesky:

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

<tagIter: k> :: (cholesky: Iter);
<tagRowIter: Iter, Row, > :: (triSolve: Iter, Row);
<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) ->
```

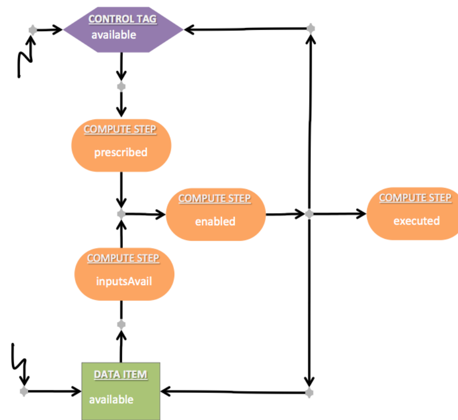

Fig. 3. CnC semantics are defined by flow of attributes among instances

```
[X:col, row, iter +1];
```

2.1 Semantics

The semantics of a CnC application are defined in terms of semantic attributes. For example, a data instance might be available. A step instance might be enabled. Expressions over these attributes can be used to indicate a time in the partial order. For example, the semantics dictate that a step cant execute until its inputs are available. In a programming model with a basically serial flow, points in that flow are used to refer to time. This indirectly ensures that the dependences are met but imposes unnecessary ordering constraints on the execution. In CnC, an attribute/instance pair denotes a time in the partial order. This can be used to directly indicate when dependences are met, avoiding unnecessary constraints.

3 CnC Tuning Language

The separation of 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.

The domain spec exposes the parallelism in the application. Because the domain spec only indicates the constraints, there is often more than enough potential parallelism. The job of tuning is to limit the parallelism to improve performance for a specific target architecture. Tuning accomplishes this by eliminating some semantically legal executions that result in poor performance. The tuner stages the work, determining when and where to release it to the normal CnC scheduler.

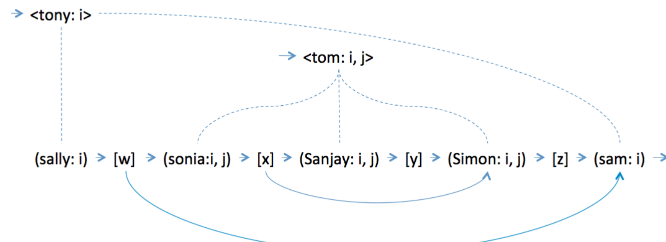

Fig. 4. Sample CnC graph

The tuning specification is written by a tuning expert. The tuning expert might be the domain expert at a later time, different programmer with expertise in tuning, a static analysis such as a polyhedral tiling facility, or even an auto-tuning facility. The language and facilities described below would be the result of the work of any of these tuning experts.

Locality is typically a major consideration in this process. The basic concept in the tuning language is the affinity group, a set of computations that the tuner suggests should be executed close in time and space. Two computations that touch the same data will not benefit from locality if they are too far apart in the platform (space) or if they are too far apart in time. The basic tuning actions map affinity groups to nodes in a hierarchical representation of the architecture. The actions are controlled by an attribute-based time at which the actions should be performed (e.g., when step instance *bar* has executed and data item x is available, map affinity group instance G to node N).

3.1 Hierarchical Affinity Groups

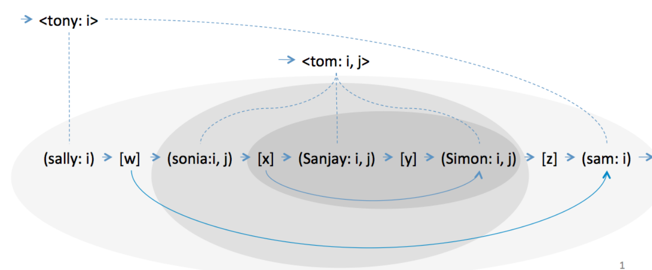

Fig. 5. Affinity graph

Hierarchical affinity groups 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. Computations that are in the same group have an affinity with each other. Hierarchical affinity groups allow the specification of relative levels of affinity, with tighter affinity at lower levels. The hierarchical affinity group mechanism is intended to be useful by itself. It is also the foundation that supports space-specific and time-specific tunings (see Section 3.1).

Consider the CnC graph in Figure 4. An option for hierarchical affinity grouping for this graph is shown in Figure 5. Assume for our examples that all step collections in Figure 5 form an outer affinity group. A second grouping option would to be have a single inner affinity group 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 group within the outer group and step collections (*sonia*) and (*sanjay*) form an affinity group within that one. The domain spec 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 grouping for Figure 5. We use curly brackets to show the hierarchical nesting of affinity groups:

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

Affinity groups have names:

```
{george:
        (sally)
        (sam)
        {greg:
                (sonia)
                {gail:
                        (sanjay)
                        (simon)
                }
        }
}
```

We need a mechanism to indicate the dynamic instances of these groups. In the domain language, steps are prescribed to control which specific instances of the step will execute. In the tuning language, groups are prescribed to control which specific instances of the group will exist.

```
<tony: i> ::
        {george:  (sally: i)
                  (sam: i)
                          <tom: i, j> ::
                                  {greg:   (sonia: i, j)
                                           {gail:  (sanjay: i, j)
                                                   (simon: i, j)
                                          }
                                  }
        }
```

The above example illustrates the prescription of groups. Recall that step collections are a static construct and we indicate the instances of a step by a prescribing tag collection. A tag instance in the prescribing tag collection corresponds to a step instance that will be executed. Here an affinity group is a static construct, and a tag instance in the prescribing tag collection corresponds to a group 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 group $\{George : i\}$.

We will also use prescription to indicate when the tuning actions associated with a group will take place. The group action is controlled by a control expression. A prescribing tag collection is the simplest control expression.

We will use the Cholesky example to show how hierarchical affinity groups work.

```
// iters have no affinity with each other
    <CholeskyTag: iter> :: {Cho-Tri-Up
            // all the work for a given iteration has  a weak affinity
           (Cholesky: iter)
           <TrisolveTag: row, iter> :: {Tri-Up
             // the work for a given iter and row has a strong affinity
              (Trisolve: row, iter)
              (Update: col = (iter+1 .. N), row, iter)}}
```

There can be a variety of tuning specs for Cholesky. Let us examine the tuning spec above. At the outermost level, there is an affinity group $\{cho-tri-up\}$ for each value of $iter$. The distinct instances of $\{cho-tri-up\}$ are not components of any group so there is no affinity among them. But inside a single instance of $\{Cho-tri-up : iter\}$, the group's 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 $\{tri-up\}$ group for this value of $iter$ and for multiple values of row. The set of $< row, iter >$ instances is determined by tags in the tag collection $< TrisolveTag >$ from the domain spec. Within an instance of the $\{tri-up : iter\}$ group there are multiple components: $(trisolve : iter)$ and $(update : col = (iter + 1N, row, iter)$. The value of $iter$ referenced by these components is that of their parent group instance $\{Cho-Tri-Up : iter\}$. 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 $\{cho - tri - up : iter\}$, the values of *iter* are the same. Since $\{Tri - Up : iter\}$ is within $\{Cho - tri - up : iter\}$, the values of *iter* are the same.

The tuning spec above for Cholesky is an example of an iterative style spec based on tag 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 group.

```
<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 $\{tri - first\}$ group 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 spec and do not need to be repeated here. The control tags for $\{oneIter\}$ are not from the domain spec. They are defined explicitly in the tuning spec. 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 spec, prescriptions identify tiles. There is no concept of a tile in the domain spec. The tuning spec has to create a new tag 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 spec but are not part of the domain spec. They belong to the tuning spec, and will differ among tuning specs. 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 nave code and focused on reuse of the result of the (*trisolve*) computation. The second was recursive. The third was tiled. The domain spec remained untouched for all three.

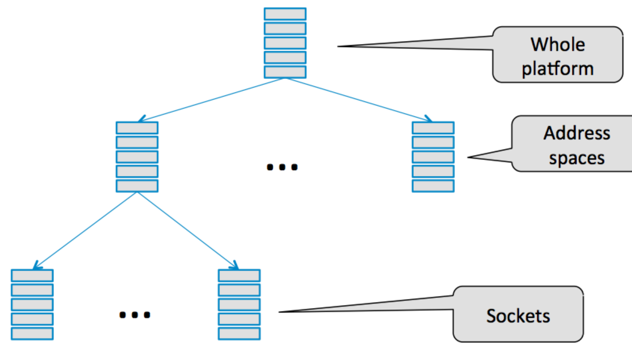

Fig. 6. Tuning tree

Execution model 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 the foundation of the tuning commands. The platform description names each level, for example, *Level*1, *Level*2, etc. or it might be *address_space*, *socket*, *core*, etc..

The execution model for hierarchical affinity groups 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 group. These specify the low-level processing for that group in the tuning tree. The tuning actions control the flow of work to the domain runtime.
- Event handlers: one for each tag collection in the tuning spec. 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 6): 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 groups/steps or dynamic instances of groups/steps. 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 *r*eady when its associated tag is available. In Section 3.1 below we generalize the property of readiness with ”control expressions.” The tuning runtime system selects from a queue the ready work item(s) that are nearest the head of the queue.

```
<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
}
```

Fig. 7. Tuning action example

Large static outer groups start at the top of the tuning tree. As a group is moved down a level in the tree, it will be decomposed into its components. Since components of a group 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, groups are moved down from a node in order of their arrival, so the components of a group 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 groups are moved and to where.

For example, the static outer level group *Cholesky* can have an action defined where an instance of *Cholesky* is moved down to a child node and the static group *groupTU* is unpacked and moved down to a child node. See Figure 7, which shows pseduo-API code for an action on the Cholesky static group. An action defined on a leaf node can move instances of a static group or dynamic group instances into the domain runtime for execution.

Control expressions 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 groups, the control of "when" is a major goal of tuning. For the most part in previous examples, we have used tag collections from the domain spec to prescribe groups. But we use tags in the execution model to determine when the actions on those instances are to be performed. We provide the tuning expert with more control over when these actions occur so that the tuning process can be effective in controlling when computations are fed to the domain runtime. Recall that instances of steps, items and tags from the domain spec acquire attributes as the program executes. 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

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

Fig. 8. Space-specific mapping options

the execution of a domain specification. We also allow the tuning spec to refer to these attributes for better control of when tuning actions should be performed. The action associated with a group can take place when the attribute expression associated with the group evaluates to true. With this mechanism, for a group instance to be ready, its attribute expression must be true. Control expressions and groups both have instances. Group instances also go through state changes. We associate attributes with these as well. Consider the attributes for groups and control expressions. Groups are similar to steps. Instances of groups can be prescribed (their control expression is true). They can also be executed in the domain runtime. Although steps have an attribute $inputs_available$, groups do not. Control expressions are similar to tags. Instances of control expressions can be available. We allow reference to these tuning attributes in controlling attribute expressions, e.g., $(foo : i).executed\ and\ (george : i - 1).executed :: garcia : i$.

Time- and space-specific mappings The hierarchical affinity groups 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, eventually a tuning expert may want to distinguish. These specific controls are presented here.

Space The tuning expert has access to the facilities in Figure 8 to express specific distributions in space.

For example:

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

Here for each tag the in tag 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 }
}
```

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

Fig. 9. Time-specific mapping options

Here the four inner groups 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 group. In this example the components are statically distinct groups. The distribution annotation applies the (static or dynamic) components of the annotated group.

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

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

Time With respect to time, CnC provides a domain mechanism for cases where a data or control dependence requires an ordering. It already allow attribute expressions that add ordering constraints. Groupings allow us to group a set of computations as being close together in space-time. There are two other kinds of tuning commands that we want to include. The tuner may indicate that the set of components in a group should execute in an order specified by a priority or maybe they are unordered. The set of components of a group 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 9.

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 priority and they are not to overlap then they execute serially with a barrier between them. (Notice that the component may itself be a group 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 dont want to require that there is a time consuming barrier between them. This is the ordered/overlapping case. The other interesting case, unordered but non-overlapping 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.

Tuning Actions The focus of this paper has been on tuning actions that stage computations for the domain runtime by using the hierarchical affinity groups to enhance locality. Below is a list of some of the actions we are considering. The first four are described above. The others are beyond the scope of this paper.

- Define a group
- Map group with space-specific mappings
- Map group with time-specific mappings
- Add steps, items and tags used only for tuning
- Include data items in affinity groups
- Set/ref tuner-defined attributes (e.g., demanded, dead)
- Set/ref tuner-defined values (e.g., optimize for power/time)
- Explicitly move data items

4 Status and Conclusions

Concurrent Collections (CnC) is 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. Implementations of the domain spec are available at http://habanero.rice.edu/cnc and at http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc. We are now developing a separate language for writing tuning specs. The central component of the tuning language is hierarchical affinity groups. 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 spec.

We have designed the tuning runtime component at a high level. The implementation is under development at Rice University as part of the Ubiquitous High Performance Computing project.

References

1. Budimlic, Z., Burke,M., Cav, V., Knobe, K., Lowney,G., Newton, R., Palsberg, J., Peixotto, D., Sarkar, V., Schlimbach, F., Tasirlar, S.: Concurrent Collections. Scientific Programming 18(3-4): 203-217 (2010)
2. Budimlic, Z., Chandramowlishwaran, A., Knobe, K., Lowney, G., Sarkar, V., Treggiari, L.: Declarative Aspects of Memory Management in the Concurrent Collections Parallel Programming Model. Proceedings of DAMP 2009 Workshop (Declarative Aspects of Multicore Programming) (2009)
3. Burke,M., Knobe,K., Newton,R., Sarkar, V.: The Concurrent Collections Programming Model. David Padua (Ed.), Encyclopedia of Parallel Computing, Springer New York (2011)
4. Yan, Y., Zhao, J., Guo, Y., Sarkar, V.: Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Date Movement. Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing (LCPC) (2009)