CnC: A Dependence Programming Model

Zoran Budimlić[1] and Kathleen Knobe[2]

Abstract— **Application tuning is the one of the major hurdles on the road to exascale computing. Tuning is often directed at a specific architecture or towards some specific tuning goal. As currently practiced, the tuning activity requires serious expertise in the application domain, target architecture and tuning goals. Keeping all these (sometimes conflicting) concerns in mind at the same time while developing a program is very difficult and error prone.**

Dependence programming is a class of dataflow programming in which both data and control flow are explicit, are distinguished and are given equal weight. This paper gives an overview of CnC, a dataflow dependence programming model, from the perspective of the needs of exascale computing and shows how CnC addresses those needs through a *separation of concerns*. The goal of CnC is to enable a process that is easier, less error-prone and more effective, by separating these concerns into independent activities. Rather than proposing yet another approach to tuning, CnC provides a way to specify the application in a way that a) hides details not relevant to tuning, b) includes as much detail as possible to support the analysis and tuning process, and c) does not assume any specific architecture, style of tuning or tuning goals.

This results in a specification of the application that can be applied to any existing or future architectures and can use any existing or future tuning approach. At the same time it can be more efficient in human time spent on creating the application and more effective in finding the best tuning.

I. Motivation

In this section we first identify the needs of exascale computing (and large-scale computing in general), then we identify some existing approaches for addressing those needs together with their strengths and weaknesses. Finally we identify how CnC addresses the needs of large-scale computing through its unique features.

A. Needs of exascale computing

The challenges for Exascale computing generally fall under two basic categories: software engineering (ease of development) and tuning (ability to meet performance goals).

1) Software engineering: Software engineering support typically involves the separation of various distinct activities and different ways of thinking. Increasing the separation of concerns relating to the application algorithm (requiring expertise in the application domain) from tuning concerns (requiring expertice in the target architecture and the tuning goals) allows each of these activities to be accomplished more efficiently and with more effective results. This separation also facilitates the reuse, for example, of the application specification for different target platforms or tuning goals. Specifically, for software engineering reasons the following separations are beneficial:

- *The computation from its use.* Separating the definition of the computation from the usage supports reuse of the same computation in different applications and in different places in the same application.
- *The specification of the application from its tuning.* This supports reuse of the application spec for distinct platforms and for distinct tuning approaches.
- *Development of independent distinct architecture targets within the same application.* This supports reuse of tuning approaches for other applications.
- *Documentation for the domain expert from that for the tuning expert.* This supports the reuse of the documentation of the domain spec for distinct tunings.
- *The application specification from its support for resilience.* This supports the reuse of the resilience support across applications.

2) Tuning activity: In the context of large-scale (and especially exascale) computing, tuning the program consumes the bulk of the time and energy. Tuning, as we refer to it here, includes both static and dynamic approaches (and therefore includes various runtime approaches). For software engineering reasons alone, one would like to separate the application development from its tuning. In the context of large-scale computation tuning might consider: a wide range of architectures (some of them yet unknown), targets with faulty components (changing targets), heterogeneous targets, a variety of distinct goals (time, energy, memory usage, etc.) and trade-offs between the ease of achieving good performance vs. the possibility of achieving great performance.

A goal of a successful programming model for exascale is to support and simplify this wide range of tuning approaches. Furthermore, making the tuning process easier enables exploration of more possibilities with fewer resources.

The distinction between software engineering activities and tuning activities is roughly consistent with the distinction between the domain expert (physicist, economist, bioengineer etc.) and the tuning expert (computer scientist with a focus on performance and parallel computing).

B. Some relevant current approaches

Here we identify the pros and cons of several existing approaches to program development and tuning.

*This work was supported in part by the DOE X-Stack Program and in part by the Mayo Clinic

[1]Zoran Budimlić is a Senior Research Scientist in the Computer Science Department, Rice University, 6100 Main Street, Houston, TX 77584, USA zoran at rice.edu

[2]Kathleen Knobe is a Senior Research Scientist in the Computer Science Department, Rice University, 6100 Main Street, Houston, TX 77584, USA kath.knobe at rice.edu

1) Starting with a serial or explicitly parallel program: When creating the first version of the program, the programmer must know (and think about) all the program dependencies to get to a correct serial or parallel app

Every further modification of the program starts with the current version. This version might be designed for some different architecture or optimized for some different goal. When making each further modification, the programmer performs a variety of tasks. One is to figure out (again) which dependencies are actually required and which were introduced arbitrarily (to enable previous tuning, for example). Another is to make sure the proposed change is legal. As part of this process the programmer must determine if some overwriting of data is getting in the way of an otherwise legal code orderings.

These decisions take time and effort (cognitive load). Further, they are opportunities to make mistakes. If a tuning approach is too optimistic it might create an error in the program, but if it is too pessimistic some tuning opportunities might be lost.

2) Dependence analysis generates an internal representation of dependence graph: Since a generated dependence graph is not directly accessible the user it is not very helpful for detecting errors. Further it may have to be too conservative since it has no domain knowledge.

3) White board drawing: The evidence that the white board drawing is close to how we think about applications is that its how we explain our applications to one another. The dependencies are explicit. One does not have to deduce them from the code when one wants to modify the application. One serious drawback, of course, of the white board drawing approach is that it is not executable.

Each of these three approaches has their advantages, but each also has real problems.

II. CnC

Application tuning, while critical for performance, is time consuming, error prone and not portable. The goal of CnC is to make this process simpler and more effective. In this section we introduce the CnC concepts showing how they support the needs of exascale shown in the previous section. These concepts are discussed here at an abstract level. In the next section we show, through an example, how those concepts are used in programming CnC.

First, we present the CnC characteristics that support application development and then those that support tuning.

Application development is centered on the CnC graph. We refer to this graph as the *domain spec*. It is developed from the perspective of the domain expert. It includes exactly and only the constructs required to specify the meaning of the application and the semantically required constraints on tuning without including any tuning or arbitrary decisions. The goal of this language is to create an ideal starting point for analysis and optimizations (the focus of the tuning expert).

Tuning might be static (compiler-based) or dynamic (runtime-based). It might focus on a specific target architecture and/or on a specific optimization criteria (energy, memory usage, time, etc.). There are two aspects by which CnC supports tuning: software engineering and analyzability.

a) Software engineering support makes tuning simpler: CnC separates the ordering requirements of the application from the tuning decisions, separates the details of the data structures and the computation code from the ordering requirements, and avoids polluting the application with arbitrary ordering decisions.

b) Some CnC language features make tuning more effective: CnC provides the information needed to perform analysis, optimization and tuning of any style by explicitly defining the relationships between pieces of data (that we will refer to as *data items* from now on) and pieces of computation (that we will refor to as *computation steps* from now on) in the program.

For some applications and platforms the CnC domain spec results in acceptable performance on its own. In general, a domain spec will be paired with a tuning for a specific architecture and a specific tuning goal. However, there is no CnC-specific runtime style and there is no CnC-specific tuning style, and a wide variety of runtime approaches and tuning approaches have been built by the CnC community. Some of these are presented in Section IV. We now introduce the concepts of CnC by showing how they support the needs of large scale computing. We will refer to our example graph here (shown later in Figure 1 to present the concepts. The actual application will be covered in Section III.

A. Represent the ordering requirements but no arbitrary orderings.

CnC makes the required orderings among the computation steps explicit. There are exactly two types of required orderings: if one computation step produces data that another consumes, the producer must execute before the consumer; if one computation step determines if (or which) other computation steps will execute, the controller must execute before the controllee. These ordering constraints indicate the minimal coordination that must be maintained for correctness.

The ordering requirements in the application are represented as a graph with nodes corresponding to computation steps (e.g., `(T)`), data items (e.g., `[U]`) and control (e.g., `<IR>`) with edges among them representing the control and data dependencies. There are no arbitrary orderings.

B. Separate the details of the computations and the data structures from the ordering requirements (dependences) among them

The details of the computation steps (e.g., `(T)`) and data items `[U]`) are hidden within these graph nodes. We currently support various programming languages for implementing the computation including C, C++, Java, Haskell, Scala, Babel and Python. The granularity of the computation steps is arbitrary and depends on the domain expert defining the domain spec.

A computation step in CnC must be able to execute by getting its input, executing and then putting is output, i.e. it

should behave as a pure function. Internally it can choose to interleave inputs and outputs but the results can not require that interleaving. In other words, CnC computation steps cannot be co-routines. One implication of this is that the ordering requirments among the computation steps allow for serial execution.

Hierarchical dependence graphs can be used to express multiple distinct granularities in the same program. In the hierarchical representation a coarse grain computaton step is decomposed into a finer grain CnC graph and a coarse grain data item is decomposed into finer grain data items.

This separation of the computation steps from the dependences among them has software engineering benefits. For example, the computation step code can be modified without modifying the dependence graph. As long as the producer/consumer and controller/controllee relationships are maintained, static analysis and optimization on the dependence graph is unaffected by modifications to the code within the individual computation steps.

A computation step may consume and produce multiple data items, e.g. `(U)` consumes `[T]` and `[U]` and produces `[U]`. It may produce multiple control tags. A computation step is controlled by exactly one control tag (more on control later).

The inputs and outputs to the CnC graph are represented as producer and consumer relationships. Input to the whole application is produced by the *environment* of the graph (e.g., the environment produces `[U]`). Output of the whole program is consumed by the environment (e.g., the environment consumes `[T]`). This view simplifies reuse, for example, in libraries and supports separate development of graphs that are later merged in a larger application.

Domain spec defines the application as a graph consisting of the computation steps and data items and the explicit dependencies among them.

C. Separate the tuning of the application from the ordering requirements of the untuned application.

Tuning a given application and modifying the semantics of that application should be two separate and distinct activities. One shouldn't be forced to weed through one in order to modify the other. Distinct tunings of a given application should be isolated from the specification of the application and then applied as is appropriate.

Keeping the tuning separate from the application makes it much simpler to evolve the application itself without introducing bugs caused by assumptions made by the various tunings.

Since the domain spec is a separate and distinct untuned specification of the application it facilitates the communication about the algorithm among humans and serves as an excellent documentation tool. Since the domain spec captures the constraints of the algorithm implemented by the application and *only* those constraints, it represents the algorithm at a high level and is much easier to communicate and understand than the whole program with all its low level details.

D. Support the ability to identify, place, schedule and track individual instances

In CnC, all instances of data, computation and control are identified by a unique identifier, called a *tag*. These tags will often have meaning in the application. In our example an integer tuple (row, column, iteration) is used to identify the specific computation step `(U: row, col, iter)` that is processing a specific row and column in a matrix, during a specific iteration.

The ability to identify the individual computation steps, individual data items, and the relationships among them, greatly helps in the tuning of CnC programs. For example, the knowledge that a computation step tagged with (row, column, iteration) reads data item for (row-1, column, iteration) and produces data item for (row+1, column, iteration+1) can help the tuner (a person or a compiler) decide where to place the data items and how to schedule the computation steps in order to take advantage of the memory hierarchy.

Dynamically, these tag components take on distinct values so, for example, [itemName: row, col, iter] corresponds to a set of instances called a *collection*. The nodes in our dependence graph are computation step collections, data item collections and control tag collections.

E. Maximize analyzibility

a) Dynamic single assignment (DSA): By default the core language assumes that the data is identified at the level of values and not memory (dynamic single assignment DSA). This does not imply that there is memory storage associated with each value. It does mean that the specification is at a higher level, the level of identifying values rather than storage locations.

Reuse of memory is a vital aspect of tuning but it is not part of the semantics of the application. For example, the best reuse of memory may depend on the target architecture and the goals for tuning.

DSA form makes analyzing and optimizing the use of memory both easier and more effective. For example, it removes anti-dependencies that either constrain reordering or require analysis to undo the overwriting.

The association of a tag with a unique value facilitates debugging as well. Since CnC programs are serializable and there are no race conditions, it is easy to replay the execution and reproduce the exact error one is searching for.

It is important to point out that non-DSA data is also supported in CnC.

b) Deterministic computation: CnC is deterministic by default [5]. This also makes the analyses and transformations simpler and more effective. For example it is always legal to execute a deterministic computation multiple times, which greatly simplifies debugging. There are some extensions to CnC that support some inherently non-deterministic computations as well.

c) Data and control dependencies are represented explicitly and at the same level: They both imply ordering constraints but they have different implications and are used differently in a generic runtime and in the context of analysis

and optimization. For example, control dependencies might indicate that distinct computation steps occur exactly under the same conditions. Data dependencies might indicate that distinct computation steps input the same data items.

Another use of this distinction supports speculative execution. If the data items for a computation step have arrived but its control has not, then the step can be speculatively executed.

d) Optional dependence functions: Some functions indicate what data items a computation step consumes and produces. Others indicate what computation steps a data item is produced-by and consumed-by. These have many uses. For example, backward propagation through these dependence functions supports demand-driven execution. Also analysis of the control and data dependence with the dependence functions can identify when a data item becomes garbage.

The exact features that support analysis and optimization also support the development process. For example, DSA item where a given tag is associated with a given data value, simplifies debugging.

These analyses and optimizations can be applied statically or dynamically, making them particularly useful in the large-scale environments. Explicit functions may also allow us to statically check that they are accurate (either code or tag function might be the culprit) and dynamically check those that are not statically checkable. The domain spec graph that includes the optional dependence functions is highly analyzable. It separates the details of the computation and data structures from the dependence graph indicating the ordering constraints. It separates the ordering constraints from use of those constraints for any tuning.

III. The meaning of a CnC app via an example

Here we show the details of a CnC via an example, Cholesky factorization. The graph is small and easy to follow but the dependence are complicated enough that tuning this application is not at all trivial [6]. Then we discuss the semantics of a CnC program.

A. Cholesky factorization in CnC

Figure 1 shows a graphical version of the domain specification for Cholesky factorization. Figure 2 illustrates the data dependencies in the Cholesky factorization.

A CnC graph has three kinds of nodes
Computation: `(name: ...)`
Data: `[name: ...]`
Control: `<name: ...>`

Each name corresponds to a static collection of dynamic instance, e.g.,`[x: j]` is a static collection of data items. `[x: 3]` is a dynamic instance in that collection. These tags are most often meaningful within the application, e.g., `(foo: row, column, iteration)` or `[employees: department, year, employeeID]`.

A collection is simply a set of instances distinguished by the values of the tags. CnC tags are the analog of array indices or database keys. The application needs to determine what instances of each of these computations will execute. Each computation step collection is controlled by exactly one control tag collection. A control tag collection is an analog of an iteration space, but there is no assumed ordering among the tags in a control collection. A specific ordering might unfold as part of the execution.

The Cholesky application works on a 2-D NxN matrix of tiles over N iterations. A given iteration performs a lower triangular computation. In CnC, the distinct instances of the computation steps are distinguished by row, column and iteration. Iteration I, works on a lower triangle of tiles with columns I to N and rows I to N.

There are three distinct computations in our application: cholesky, trisolve, and update. These three distinct computations are represented in CnC by three distinct computation step collections: `(C)`, `(T)` and `(U)`.

For each iteration, I, we execute the cholesky computation for the tile where row = I, col = I, iter = I. The different instances of `(cholesky)` are distinguished only by iter, i.e., `(c: iter)`. For a given interation there is exactly one instance of `(cholesky)`.

For each tile in that same column and iteration, we execute an instance of trisolve. Since the column and the iteration are identical it is represented as `(t: iter, row)`.

The rest of the lower triangular matrix for this iteration is processed by update. An update computation, identified by all three tag components is represented as `(U: iter, row, col)`.

This version of the application distinguishes among three distinct data item collections `[C]`, `[T]` and `[U]`. `[U]` is used as input and for intermediate values. `[C]` and `[T]` are outputs.

We also have three distinct control collections, `<I: iter>`, `<IR: iter, row>` and `<IRC: iter, row, col>`. These act as control flow by indicating which instances of each computation will be executed. This is part of the meaning of the application. The control tags do not indicate *when* the computations will occur. That is a tuning concern.

A CnC graph has three kinds of relationships among the nodes: consumer, producer and controller. The producer relationships in Cholesky are:

`(C: iter)` → `[c: iter]`
`(T: iter, row)` → `[T: iter, row]`
`(U: iter, row, col)` → `[U: iter, row, col]`

While the producer relationships are trivial in this particular example, the consumer relationships are a bit more interesting:

`[C: iter-1]` → `(C: iter)`
`[T: iter-1, row]` → `(T: iter, row)`
`[C: iter]` → `(t: iter, row)`
`[U: iter-1, row, col]` → `(U: iter, row, col)`
`[T: iter, row]` → `(U: iter, row, col)`
`[T: iter, col]` → `(U: iter, row, col)`

In addition to producers and consumers among graph nodes, we view input and output as a producer and consumer

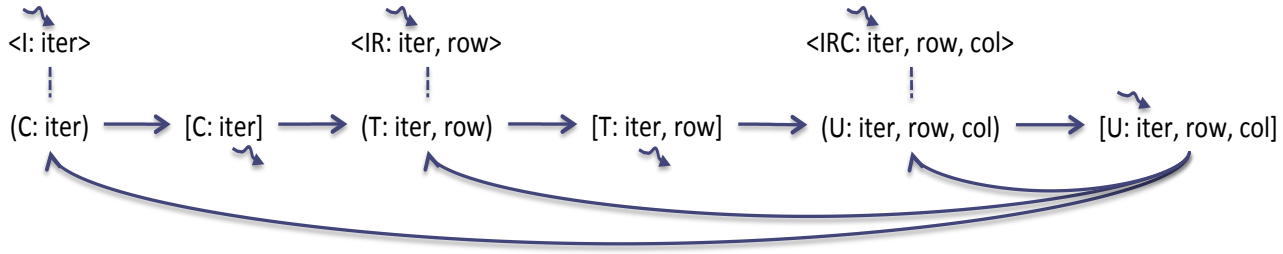

Fig. 1. Domain specification for Cholesky factorization.

relationships where the program environment (a special concept indicating the world outside of the program specified by the CnC graph) is the producer of input and the environment is the consumer of output. This approach makes it easier to compose distinct graphs. Figure 1 shows inputs and outputs for Cholesky as squiggly arrows.

So far we have expressed dependencies among collections but CnC allows us to express explicitly the dependencies between individual instances of data items and computation steps. For example, consider the consumer relationships in Cholesky :

`[U: iter-1, row, col]` → `(U: iter, row, col)`
`[T: iter, row]` → `(U: iter, row, col)`
`[T: iter, col]` → `(U: iter, row, col)`

These specify that the CnC step `(U: iter, row, col)` produces the item `[U: iter, row, col]` and consumes the values of items `[U: iter-1, row, col]`, `[T: iter, row]` and `[T: iter, col]`.

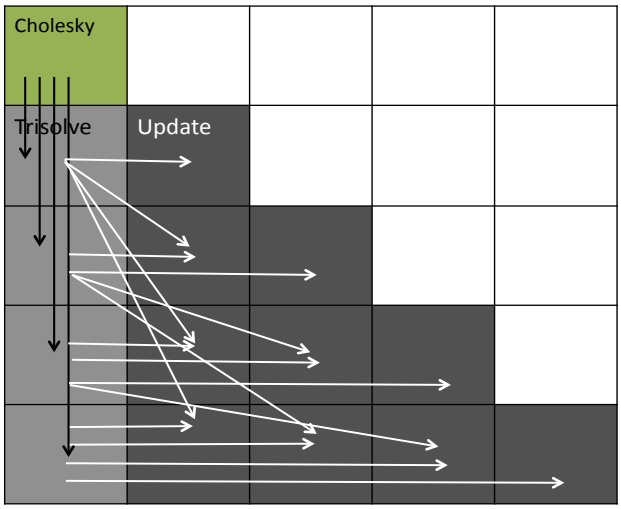

Fig. 2. Data dependencies in the Cholesky factorization.

In addition to the dependencies shown in Figure 2, each of the three computations `((Cholesky), (Trisolve), (Update))`, consumes the value from iter-1 (with the same row and col).

A control tag collection is a set of tag instances. The meaning of the relationship between a computation step collection and its control tag collection is that for each tag instance in the control collection, an instance in the computation step collection with that exact same tag value will execute. This is very similar to an iteration space but the order that the instances are put into the control collection is constrained only by the dynamic order in which they are produced and the order that the corresponding computation will execute is constrained by when the control is produced and when the inputs are also available. In our example, if `<C: 5>` is put into the control collection `<C: t>`hen `(C: 5)` will execute sometime after after this control tag becomes available and after its input data item is also available. In this example, each control collection controls exactly one computation step collection. This is not a language restriction. `<tag: J, K>` might control both `(foo: J, K)` and `(bar: J, K)`. In this case the arrival of `<tag: 12, 7>` would mean that `(foo: 12, 7)` and `(bar: 12, 7)` will execute. When exactly will each be executed is limited by when its input data items are available. In this Cholsky application, `<C: iter>` controls `(C: iter)`. `<IR: iter, row>` controls `(IR: iter, row)` and `<IRC: iter, row, col>` controls `(IRC: iter, row, col)`. The control is known from the beginning for this application as it is a function of N. So these tag collections are simply input to the application. It is common for a tag instance to be produced by a step instance. This way a step can determine if another will execute by conditionally putting its tag (the analog of an IF statement). A step can determine which other steps will execute by computing a set of tag values to put based on its input data (the analog of generating an iteration space). A step can determine which control collection to put tags into (the analog of an If-THEN-ELSE). These are analogs, not precise equivalents. The CnC versions are less constrained. There are no assumptions about which tags will be generated in what order and further there is no implied association between the order in which the tags are generated and the order in which the controlled step executes. The computation that produces the tag and the computation controlled by that tag are each constrained by their dependences. In an extreme case the application might be required to produce all the tags in forward order but the controlled computation might be required to execute in exactly the reverse order.

There is exactly one instance of the cholesky computation for each iteration. For each iteration, I, we execute cholesky with row = I, col = I, iter = I. So, in CnC, the different instances of `(cholesky)` are distinguished only by iter,

i.e., `(cholesky: iter)`.

For each tile in that same column and iteration, we execute an instance of trisolve. So trisolve is represented as `(trisolve: iter, row)`.

The rest of the lower triangular matrix for this iteration is processed by `(update: iter, row, col)`.

B. An abstract execution model for CnC

Here we discuss the semantics of a CnC domain spec. These address the required orderings. We will address tuning in section IV. Tunings are critical to exascale computing but they are isolated from the semantics of the application in CnC and are not addressed here.

As data item instances and control tag instances are produced we consider them as *available*. Notice that if no control tags are produced by the environment, no computation steps will execute.

When a control tag instance and the data item input instances for a specific step instance are *available*, that step instance is considered *ready*. When a computation step instance is *ready* it can execute. In the process of executing, a computation step may create more data item and control tag instances. These will then become *available*. Once a step instance completes it is considered *executed*.

A CnC graph terminates when

- No steps are executing
- No unexecuted steps are *ready*
- No more input will arrive from the environment

We can distinguish between two distinct states upon termination. The normal case is that all *ready* steps are also *executed*. It may be the case that the input for some *ready* has not arrived. This might be considered an error or not but it is a distinct termination state.

There are some interesting implications of this very simple semantics. For example, it means that by tracking the available item and tag instances (with the contents of the items) and tracking which step instances have executed we have the entire relevant state of the application. The forward progress of an application can be tracked by these attributes. Depending on the execution model, we might or might not want to actually maintain this state during execution. If we do maintain the state, we can clean up irrelevant state details by removing items and tags that are *d*ead and steps that have *e*xecuted. This small, simple state supports simple development for debugging, tracing, performance analysis and continuous, asynchronous, automatic checkpointing.

IV. Tuning approaches

The crux of CnC is that the ordering requirements are specified explicitly as a dependence graph that is totally isolated from any tuning. This allows for serious flexibility. There is no CnC-specific runtime or CnC-specific tuning. In this section we refer to both as tuning. The tuning may vary dramatically with the architecture, the application class, the target platform and the tuning goals. This has serious ramifications for both productivity and performance.

There is only one rule for a valid tuning: The execution must obey the dependences of the domain spec. A tuning expert can tune the next application by using a combination of these existing techniques or by developing new ones crafted for a very specific architecture, application class or a tuning goal.

Here we present some CnC tuning approaches that have already been built, some that are under development and some ideas for future tuning directions.

The ”tuning expert” focuses on analysis and optimization. This might be a person or an automatic analyzer/optimizer. The tuning might be static or dynamic. The focus might be on mapping DSA values to memory locations, scheduling computation in time, mapping computation or data to the platform. The platform might be shared memory, distributed memory, or a heterogeneous mix of CPUs, GPUs and FPGAs. The goals might be to minimize time, minimize energy, minimize the memory footprint, support resilience or some yet undiscovered concern. The starting point is always the CnC domain spec that describes application's constraints in terms of data and control dependences. The schedule and placement of computation and the mapping of data to memory might each be determined either statically or dynamically. A full tuning system might involve some combination of static and dynamic approaches.

A. Static tunings

There are several examples of static systems. Some focus on mapping the DSA data items onto non-DSA memory. Others focus on scheduling the work across time and/or placing the work across the platform.

A CnC runtime can be completely based on a static distribution and schedule. Each rank executes a program paramaterized by the worker ID. The program encodes the distribution and schedule. It waits to receive the input data items the next step will consume. It executes that step. It send the data items it produces to the ranks that will execute the consumers of that data. The runtime overhead is extremely low, making this approach be very effective for applications with predictable schedules. Static distribution and scheduling is not an appropriate choice for applications with unpredictable behavior.

One example is polyhedral tiling, which uses a constrained version of CnC, in which it must be statically known which data items will be input to each tiled polyhedral step [14], [16], [17]. Another project involves investigating automatic conversion of element-level stencil applications to tiled versions.

For Distributed memory systems, the tuning expert can provide functions [8] that map data items to nodes in a cluster as a function of their tags and then determines where the computation is executed based on this data distribution. In addition the tuning expert can provide functions that map the computation steps to a node in the cluster based on tags and then determines where the data is needed based on this computation distribution. These functions cam have a serious impact the performance of the application. In this system the

distribution of the computation or data among the distributed memory nodes (where) is static but the schedule of execution within a node is dynamic.

B. Dynamic tunings

Here we briefly describe a few of the dynamic mechanisms that have been developed for CnC. A straightforward dynamic CnC runtime for shared memory will keep a track of all the steps that are ready to execute, and keep a pool of worker threads to execute available tasks.

We currently have dynamic runtimes that are based on the Open Community Runtime (OCR) [4], Thread Building Blocks (TBB), Babel, and Haskell.

The basic dynamic runtimes addresses load balance in that any worker can execute any available task. One example of an enhancements to this basic runtime takes advantage of the cache hierarchy with a scheduler that can prioritize execution of the steps that access the same data on the same worker thread [8].

In the tuning described above that provides static mapping of the work and/or the data across a distributed memory system, the current placement (space and time) within a shared memory node is still dynamic.

Another version of dynamic tuning CnC programs is designed for heterogeneous hardware platforms [18]. It supports dynamic decisions that determine which type hardware a comptuation step should use based on input from the tuning expert. This input specifies for each computation step collection which platforms it can run (CPU, GPU, FPGA), what is the preference (i.e. performance) of that step for running on each of these platforms, and provides an appropriate variant of the step code for each platform. The runtime will dynamically attempt to make the best match among the available computation steps, available computation resources, and overall program execution balance.

One tuning approach attempts to maximize locality and reuse in CnC programs by providing support for representing a hierarchy of affinities for a given CnC domain-spec [13]. A tuning spec in this language is declarative, and very similar in syntax and semantics to the CnC spec language. The tuning expert specifies a hierarchical affinity grouping of the computation steps based on their access to common data items. Intuitively, a low level affinity group will include a set of computation steps that exibit significant reuse of the same data items. Lower level affinity groups are merged at a higher level if they reuse some of the same data items. The runtime then executes the domain spec obeying its required ordering constraints but also guided by this affinity-based tuning spec. In this approach the affinity groups are available statically but the current implementation uses it to dynamically guide the placement across both the platform and time.

Since CnC is a dynamically a single-assignment language, tuning for memory usage is one of the most important aspects of tuning. One approach for reusing memory locations in streaming applications is to use the tag functions to determine when the specific items are no longer needed and then reuse the memory that was occupied by those items [20]. Another approach involves an inspector phase execution of the CnC graph to determine the memory usage patters in the program for the given input, then to create a schedule that minimizes the memory usage [19].

While the tunings above aim to reduce the execution time, consumed energy or the amount of storage used for the application itself, we can also reduce the runtime overhead for a specific application. Some activities that the runtime must do in general are not required for a given application. Consider the attributes available, ready and executed described as part of the CnC abstract execution model in Section II. A runtime can track these changes in state to know when a step can be scheduled. Lowering the application graph to make these state changes explicit allows this lowered graph to then be optimized resulting in application-specific runtime with lower overhead [12].

C. Future tunings

1) Out-of-core computation: Consider a hierarchical CnC app where the computation, data and control at different levels in the hierarchy are of different grains. The domain spec itself doesn't specify if the data input is coming from a register, cache, memory, another node in a cluster, etc. One option is that it's out-of-core. In this case the code to bring out-of-core data into core and the reverse is just part of the dynamic runtime. Just as the runtime now manages the communcation among nodes in a cluster with no involvement by the domain-expert, the out-of-core runtime would manage this out-of-core communication. Just as the tuning expert for the distributed system might provide distribution, the tuning expert might indicate the level in the hierarchy at which this data movement ought to occur but.

2) Checkpoint-continue: Checkpointing, as is usually implemented in today's systems, saves the state of the data values at specific points during execution. In CnC, in addition to these values, the system records a few execution attributes, such as ready, executed etc. This enables an automatic, continuous, asychronous checkpoint-continue system for a non-hierarchical CnC spec. [21]. For future work we plan a hierarchical version, in which the granularity of the checkpoints varies with the hierarchical level. At any fault in the platform at any level, the current checkpoint for that node can be moved without stoping the rest of the application. This might be used not only for failure recovery but for also for dynamic remapping of the application for other reasons.

3) Collaborating runtimes: For some applications different parts of the application have different requirements. CnC can allow combining different parts of an application or even multiple CnC programs running on distinct runtimes together to form a single larger heterogeneous application. Different runtimes might have different representations for collections while the combined application will know how to access them for each of the runtimes.

4) Inspector-executor: A system the performs dynamic distribution and scheduling based on inspection of previous execution costs [19] is also under consideration.

5) Demand-driven and speculative execution: This can be achieved using on a few additional execution attributes, for example, a step might be demanded if its result is needed, or it might be speculatively executed if its data is ready but has not been prescribed yet.

V. Related Work

A. CnC as a dataflow language

In contrast to the traditional dataflow languages [9], CnC isolates specification of the flow (required orderings) from the actual computation and from any tuning. While there are fine-grain dataflow and macro-dataflow, CnC is closer to macro dataflow not because of the granularity but because it isolates the computation from the ordering constraints. CnC also adds unique tags to computation, data and control instances to maximize analyzability.

B. CnC and task-based languages

Task-based parallel programming languages such as Habanero-C [1], Habanero-Java [2], Cilk [10] and OCR [4] are typically harder to analyze even than serial programs and harder still than CnC. CnC computation steps are at a higher level. The dynamic runtime or static tuning is responsible for tracking when a task is ready, not the domain expert. Because a CnC computation step just puts control tags (no spawning of a specific other step) it is easier to put together new apps from existing pieces since connection between tasks is not explicit in the code.

C. CnC and serial or high performance parallel languages

Many of the high-performance parallel languages [2], [1], [15] can be used as a language for the computation within CnC computation steps, as long as the code within the step respects the step-like behavior. Some high-performance parallel languages [7] make good tuning languages. The computation code would be replaced by CnC computation step invocations. The aspects of the language that addresses placement and scheduling would be used to tune the application. Other models such as Legion [3] and Charm++ [11] go a long way towards separating the algorithm dependencies from tuning decisions but they still do not completely and explicitly separate the two as CnC does.

VI. Conclusions

In this paper we give an overview of CnC, a dataflow dependence programming model that is very attractive as the programming model for exascale systems. CnC separates program dependence specification from tuning the program for a specific architecture or for specific tuning goals. CnC is deterministic and race-free, its data has the dynamic single assignment property, and all its computation and data is tagged with unique identifiers with optional functions between tags that greatly simplify analysis and optimization. CnC does not assume a specific target platform, style of parallelism, computation language or granularity.

In addition to simplifying the tuning process, CnC also has significant software engineering benefits as CnC programs are easy to analyze, maintain, debug, port and evolve.

Acknowledgments

There is a large and growing CnC community that contributes to the development of the programming model and its implementations. We would especially like to thank Michael Burke, Sanjay Chatterjee, Aparna Chandramowlishwaran, Milind Kulkarni, Ryan Newton, Vivek Sarkar, Alina Sbirlea, Dragos Sbirlea, Frank Schlimbach, Sagnak Tasirlar and Nick Vrvilo for making CnC what it is today.

References

[1] *Habanero-C++: A Compiler-free Work-Stealing Library*. http://habanero-rice.github.io/hclib/.

[2] R. Barik, Z. Budimlić, V. Cavé, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar. The Habanero multicore software research project. In *OOPSLA '09*, pages 735–736, New York, NY, USA, 2009. ACM.

[3] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. SC '12, Los Alamitos, CA, USA.

[4] Z. Budimlić, V. Cavé, S. Chatterjee, R. Cledat, V. Sarkar, B. Seshasayee, R. Surendran, and N. Vrvilo. Characterizing application execution using the open community runtime. In *RESPA 15*, 2015.

[5] Z. Budimlić et al. Concurrent collections. *Scientific Programming*, 18:203–217, August 2010.

[6] A. Chandramowlishwaran, K. Knobe, and R. Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *IPDPS*, pages 1–12. IEEE, 2010.

[7] S. Chatterjee, S. Tasirlar, Z. Budimlić, V. Cavé, M. Chabbi, M. Grossman, Y. Yan, and V. Sarkar. Integrating Asynchronous Task Parallelism with MPI. In *IPDPS '13*.

[8] S. Chatterjee, N. Vrvilo, Z. Budimlić, K. Knobe, and V. S. r. Declarative Tuning for Locality in Parallel Programs. In *ICPP'16*, Aug 2016.

[9] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[11] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[12] K. Knobe and Z. Budimlić. Compiler Optimization of an Application-specific Runtime. In *CPC '13*, Jul 2013.

[13] K. Knobe and M. Burke. The Tuning Language for Concurrent Collections. In *CPC '12, Proceedings of the 2012 Workshop on Compilers for Parallel Computing*, Jan 2012.

[14] M. Kong, L.-N. Pouchet, P. Sadayppan, and V. Sarkar. PIPES: A language and compiler for task-based programming on distributed-memory clusters. SC '16. IEEE, 2016.

[15] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 4.0*, July 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[16] A. Sbirlea, L.-N. Pouchet, and V. Sarkar. Dfgr an intermediate graph representation for macro-dataflow programs. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on*, pages 38–45. IEEE, 2014.

[17] A. Sbîrlea, J. Shirako, L.-N. Pouchet, and V. Sarkar. Polyhedral optimizations for a data-flow graph language. In *LCPC*, Sept 2015.

[18] A. Sbîrlea, Y. Zou, Z. Budimlić, J. Cong, and V. Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. LCTES '12, pages 61–70, New York, NY, USA, 2012. ACM.

[19] D. Sbîrlea, Z. Budimlić, and V. Sarkar. Bounded memory scheduling of dynamic task graphs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 343–356, New York, NY, USA, 2014. ACM.

[20] D. Sbîrlea, K. Knobe, and V. Sarkar. Folding of tagged single assignment values for memory-efficient parallelism. Euro-Par'12, pages 601–613, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] N. Vrvilo. Asynchronous Checkpoint/Restart for the Concurrent Collections Model. Master's thesis, Rice University, Aug 2014.