Capturing and Composing Parallel Patterns with Intel CnC

Ryan Newton, Frank Schlimbach, Mark Hampton, Kathleen Knobe

Intel Corporation

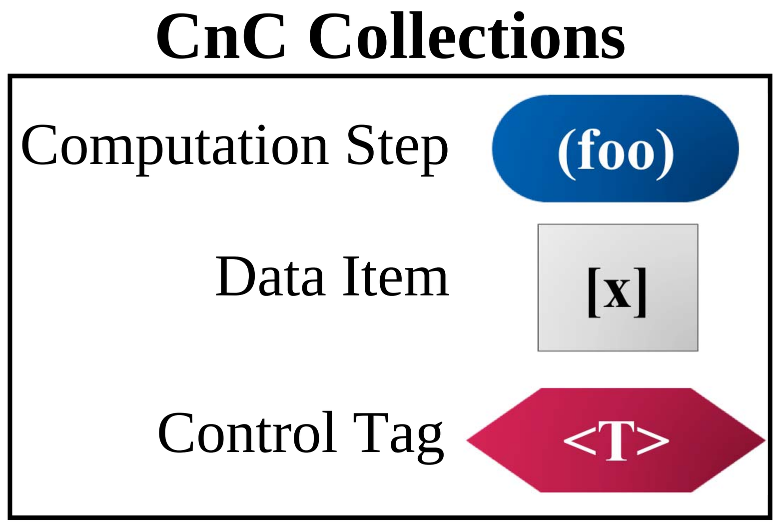
1. Introduction

- Programmer productivity can be improved by encapsulating structured, well-understood parallel algorithms, i.e. *parallel patterns*
- We believe it is important to support these parallel patterns within a high-level framework that can deliver semantic guarantees such as determinism while still providing flexibility for performance tuning
- In this work, we present Intel CnC as a candidate substrate for capturing and combining parallel patterns

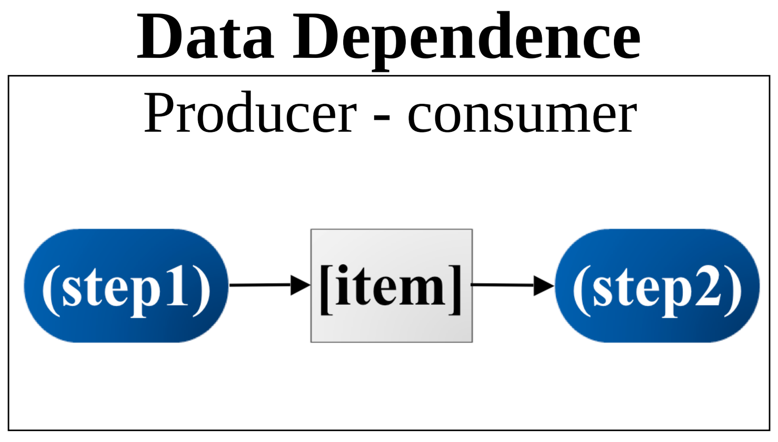
2. What is CnC?

- Intel Concurrent Collections (CnC) is a deterministic parallel programming model that supports task and data parallelism
  - It does not explicitly specify the parallel execution of operations
  - Only an application's semantic ordering constraints are specified
- There is a separation of concerns between the *domain* expert—who focuses on the semantic constraints—and the *tuning* expert—who maps the application to the target platform

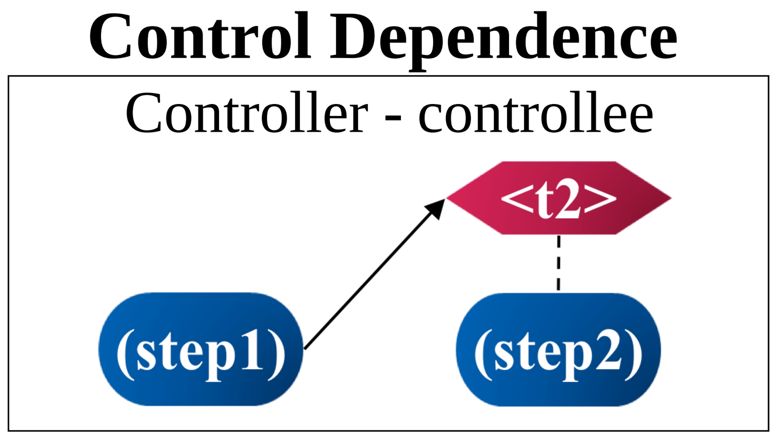
CnC Collections



Data Dependence

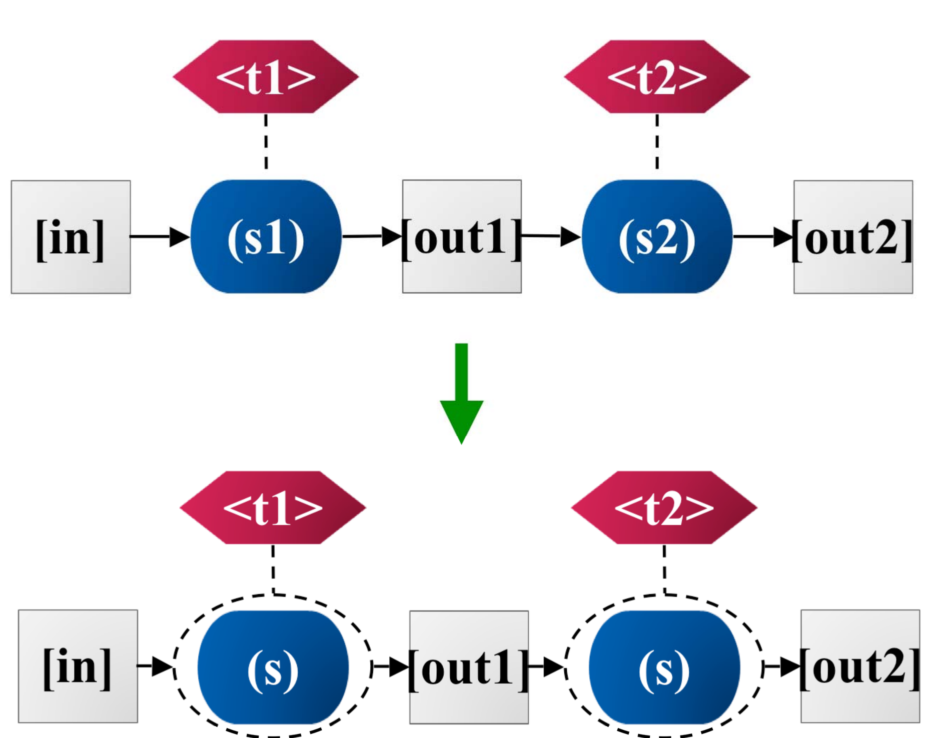


Control Dependence



- CnC provides three types of static collections:
  - Computation steps are high-level operations ordered according to their semantic constraints
  - Data items are the data produced and consumed by computation steps
  - Control tags *prescribe* steps, i.e. cause them to execute
- Collections are connected via data and control dependences that specify the program's ordering constraints
- For each static collection, a set of dynamic instances is generated at runtime; each data item instance is uniquely tagged, supporting determinism
- The execution of the CnC graph is invoked by the *environment*, which can produce and consume data items and control tags

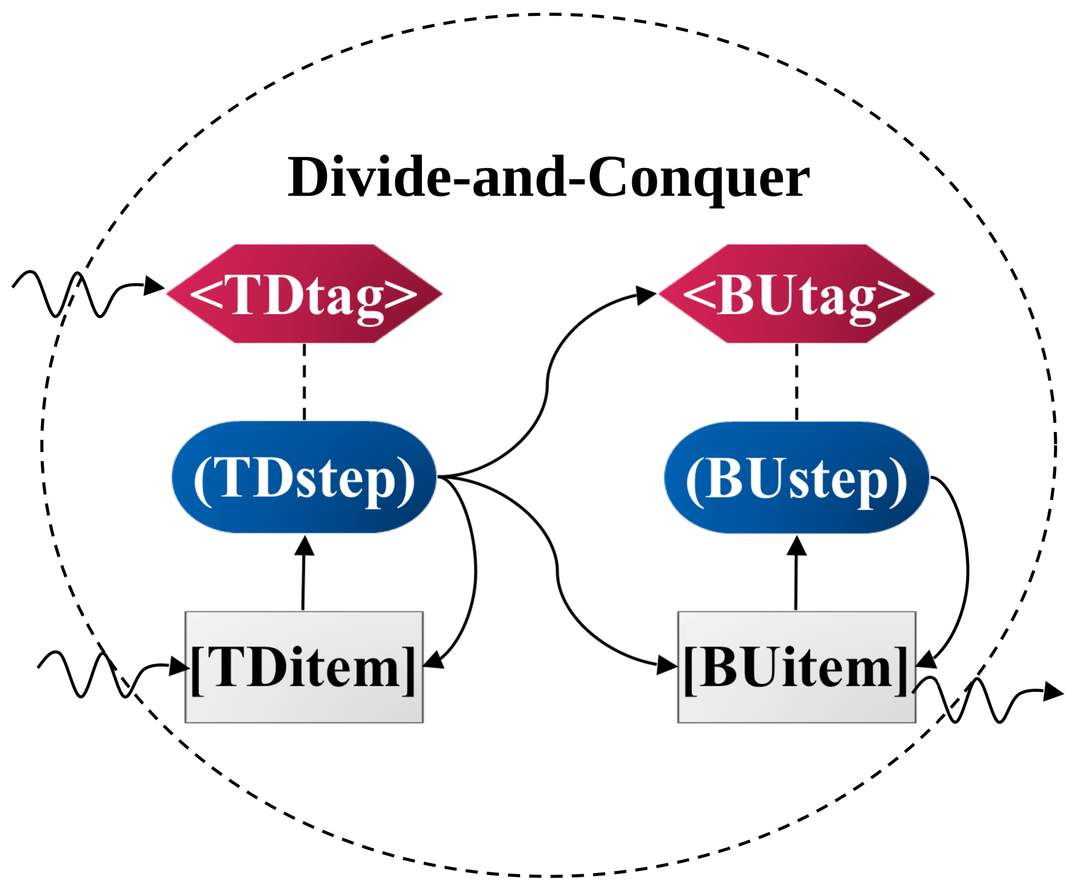
3. Using Modules in CnC



- Previously, all CnC graphs were flat, and there was no code reuse, so even if steps *s1* and *s2* performed identical computations, the programmer had to write the same code twice
- By abstracting the step as a single module *s*, the programmer only needs to write the computation code once, allowing for code reuse
- A module takes arguments at its instantiation point (resembling a function) and generates a subgraph as a result
- In addition to code reuse, our module system provides the following benefits:
  - A *scoping* mechanism for unsafe features
  - An *isolation* mechanism to reason about patterns' invariants separately from the larger environment

4. In-Place Memory Operations with CnC--

- CnC data items are single-assignment, enabling determinism, but preventing the implementation of in-place parallel algorithms
- We address this issue by using a lower-level CnC layer, CnC--
  - CnC-- can be used by modules which internally violate the rules of CnC
  - The module system safely isolates the portion of the code that contains in-place memory operations, maintaining determinism for the entire program
- Consider the following module which defines a divide-and-conquer pattern (the squiggly lines indicate input from or output to the module's external environment, i.e. the module arguments):



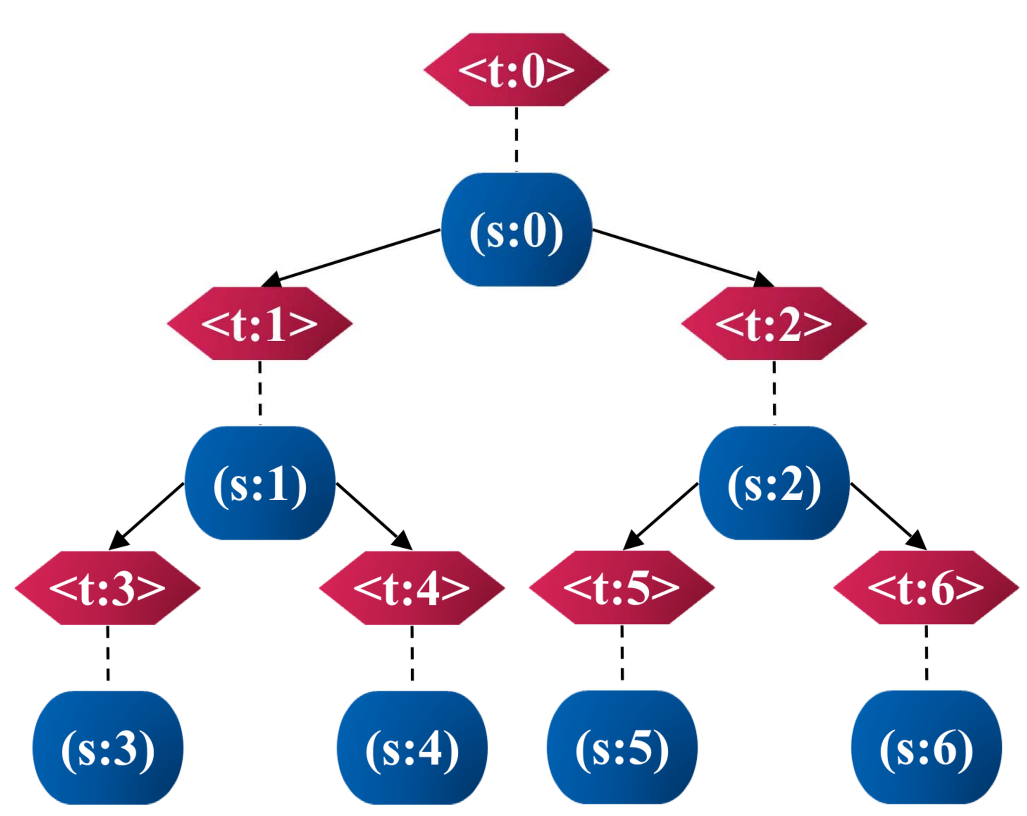
Except for the initial input and final output, the *TDitem* and *BUitem* data are completely private to the module, and can be safely operated on in-place

- The module receives an initial *TDitem* instance
- *TDstep* will descend the tree, dividing its *TDitem* input data into smaller chunks
- When the threshold size is reached, *TDstep* will work on the chunk and a *BUitem* instance will be generated
- *BUstep* combines the *BUitem* instances as it progresses back up the tree
- The final *BUstep* will output the finished data to the environment

5. Step Scheduling Controls in CnC--

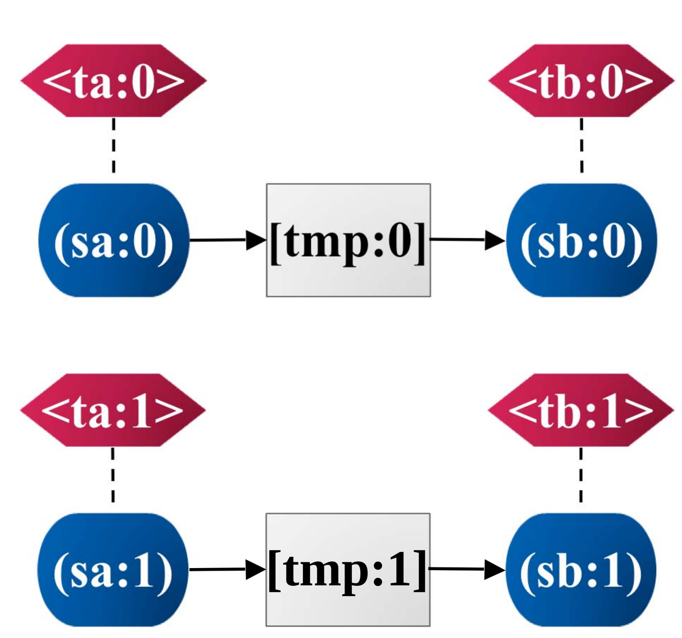
- CnC-- can also be used to provide low-level scheduling control, facilitating performance tuning for a wide range of patterns
- The scheduling controls of CnC-- include priorities, ordering constraints, dynamic chaining, and affinity
- Scheduling controls are composable and are represented as declarative functions on tags, making them amenable to static analysis
- We illustrate the application of two scheduling controls below

Priorities



- A partial dynamic instance graph is shown to the left—each step instance in the collection *s* generates tag instances in the collection *t* for its left and right children
- If we want to achieve a parallel breadth-first schedule, we can specify that the step instance with the lowest-numbered tag should have highest priority

Dynamic Chaining



- The partial dynamic instance graph to the left represents independent iterations of a loop that performs a computation step *sa* and a dependent step *sb*
- By chaining *sa:i* with *sb:i*, we can improve memory locality by forcing each consumer to execute immediately after its producer