Capturing and Composing Parallel Patterns with Intel CnC

Ryan Newton Frank Schlimbach Mark Hampton Kathleen Knobe
Intel
{ryan.r.newton, frank.schlimbach, mark.hampton, kath.knobe}@intel.com

Abstract

The most accessible and successful parallel tools today are those that ask programmers to write only isolated serial kernels, hiding parallelism behind a library interface. Examples include Google's Map-Reduce [5], CUDA [13], and STAPL [12]. This encapsulation approach applies to a wide range of structured, well-understood algorithms, which we call *parallel patterns*. Today's high-level systems tend to encapsulate only a single pattern. Thus we explore the use of Intel CnC as a single framework for capturing and composing multiple patterns.

1 Introduction

The computing version of Alfred North Whitehead's maxim[1] is that software advances in proportion to the functionality that a programmer can access without thinking about it. Libraries and frameworks make this possible, but when it comes to parallel programs, composing new software from old is more problematic.

Today, parallel software is built on top of scheduling systems that assume control over machine resources and are not designed to cooperate [3, 1, 15]. Cooperation requires conventions, which can be established anywhere from the hardware layer to the programming language.

Where to put them? High-level conventions can mean more changes to legacy code. Further, we agree with previous authors [8] that a single high-level tasking framework with a *global* scheduler cannot be a universal solution. On the other hand, a low-level or unstructured combination of parallel codes cannot deliver *semantic* guarantees, such as deterministic parallelism. Therefore we believe that reusable parallel software should, wherever possible, be integrated into a high-level framework *semantically*, irrespective of how it is implemented.

Those functions which we strive to package and reuse range from fixed-function library routines (MKL [10]) to programming paradigms in their own right (MapReduce [5]). We are concerned with the more flexible end of this spectrum and will call our targets parallel *patterns*. Our goal is a framework for capturing patterns—individually well understood—and combining them.

In this paper, we consider Intel Concurrent Collections (CnC)[9]. as a candidate substrate for patterns. CnC is a parallel model of computation that supports flexible combinations of task and data parallelism while retaining determinism. CnC is implicitly parallel, with the user providing serial functions called *computation steps*, which, together with their semantic ordering constraints, form a CnC graph.

One feature of CnC is a separation of concerns between the *domain* expert and the *tuning* expert. The domain expert focuses on the semantics of the application as a graph with declarative constraints, whereas the tuning expert, obeying those constraints, maps the graph onto a target platform. Consistent with this philosophy, the mechanisms in this paper allow the domain expert to choose among common reusable patterns (based only on their semantics and interfaces) such as tree-based algorithms, pipelines, stencils, and other patterns specific to a domain of interest. The tuning expert can then choose an implementation, such as depth-first tree traversal, wavefront stencil, etc. Our proposal for supporting this separation consists of:

1. A module system for encapsulating patterns.

2. An unsafe API for step code, controlling execution details such as memory management. This API can only be used within modules that are certified to provide safe external interfaces.

3. A language of tuning constraints for expressing scheduling strategies for each pattern.

The benefits of the these mechanisms are two-fold: the domain expert can more easily put together applications,

[1] "Civilization advances by extending the number of important operations which we can perform without thinking about them."

drawing from a library of parallel patterns, and the library writer can implement patterns more easily by composing and re-tuning existing modules. In this paper we describe our experience in a series of case studies, examining parallel patterns and their scheduling constraints.

2 Using CnC: Now with modules

To develop an application in CnC, the domain expert identifies the data and control dependencies in the application and captures them as related collections of **computation steps**, **data items** and **control tags**. We say that tags *control* steps (cause them to execute) which produce items and more tags. Statements in the CnC specification language correspond to edges in a graph, and use the following notation:

```
<tags> :: (step); // prescribe step
(step) → <tags2>, [items];
```

The bracketing characters in the syntax are inspired by the CnC graphical notation [9]. In this paper, types are omitted from these specifications for brevity.

Computation steps take a tag as argument and may read and write item collections (which are single-assignment associative containers). Normally, computation steps are written externally in a host programming language (C++, Java and Haskell are supported), but here we will write steps directly within a CnC spec using a micro-subset of C++:

```
step mystep(i) {
  x = items1.get(i);
  y = items2.get(i);
  items3.put(i, x+y);
}
```

A CnC specification file is processed by the *translator* which generates the code necessary for creating and executing a graph. For the domain expert, that's all there is to it. All parallelism is implicit. Next, the tuning expert uses the exposed knobs to control parallel execution and increase performance. For example, she can control the relative priorities and processor affinities of different steps. A contribution of this work is to add new and more powerful means of scheduling control to CnC (Section 3.2).

2.1 Module System

To date, CnC programs have been flat graphs, transcribed directly in the specification file with no means of "graph reuse". For example, a subgraph describing a matrix multiplication would have to appear twice in order to perform two matrix multiplications in an application. Clearly this is a limitation.

In this paper, we propose a module system that abstracts subgraphs in a reusable way. Further, we get mileage out of our module system by using it as: (1) a *scoping* mechanism for unsafe features, and (2) an *isolation* mechanism to reason about patterns' invariants separately from the larger environment.

CnC modules take arguments at their instantiation point and generate subgraphs as results. Module arguments might be item collections, tag collections, serial functions (defined externally), and even other modules. As a simple example, to use a pipeline module, the user provides two stages (themselves modules) along with input and output item collections.

```
// Instantiate a module by prescription:
<MyTags> :: Pipeline2( [in], Mod1, Mod2, [out] );
```

The `Pipeline2` module is controlled by (invoked by) `<MyTags>`. Thus modules closely resemble computation steps, being controlled by a single tag collection. This convention is assumed without loss of generality because multiple tag collections can always be unioned. Further, the convention allows a form of self-similarity that encourages hierarchical design.

Defining modules is as simple as writing normal graph specifications but with certain pieces abstracted and placed in the argument list. (Indeed, this looks much like a function definition.) For example, to abstract a step from the item collections and tag collections it is "wired" to, we would write the following:

```
// Wrap an instance of our step as a module:
module Mod1<tag>( [in], [out] )
{
    <tag> :: (my_step1);
    [in] -> (my_step1) -> [out];
}
// ... Same for Mod2 ...
```

Two steps, referring to the same serial computation, and referred to by the same name, are distinct *instances* if they fall into different module scopes. Given steps abstracted as modules, the definition of our `Pipeline2` module is as follows:

```
module Pipeline2<tag>( [in], (M1), (M2), [out] )
{
    [tmp]; // Fresh item collection
    <tag> :: M1( [in], [tmp] );
    <tag> :: M2( [tmp], [out] );
}
```

3 Low-level CnC: *CnC--*

This paper presents the first design for a lower-level CnC layer to be used by library-writers providing modules that, while internally breaking the rules of CnC, are externally safe to use by general CnC programmers (i.e. they don't compromise determinism or memory safety). We call this layer *CnC--*, read "CnC minus minus". CnC-- is our answer to the question "Why can't I do X in CnC?" (a question asked of all high-level programming models). Thus, the answer is "Yes, you can—if the violating portions are isolated safely."

```
module DivideConquer<TDtag>
        (bottom, split, merge, work, [in], [out])
 {
   // Build subgraph, introduce new steps/items/tags
   <TDtag> :: (TDstep);
   <BUtag> :: (BUstep);

   // All data dependencies:
   [TDitem] -> (TDstep) -> [TDitem], [BUitem];
   [BUitem] -> (BUstep) -> [BUitem];

   // Skeleton definitions for the new steps:
   step TDstep(node) {
     // Copy first input from [in]:
     if (root(node))
          x = in.get(node)
     else x = TDitem.getRW(node)

     work(x); // e.g. sort in place

     // If at bottom start going back up:
     if ( bottom(node, x) )
       BUitem.put(node, x);
     else {
       l, r = split(x, param);
       TDitem.put(node.left,  l);
       TDitem.put(node.right, r);
       BUtag.put(node);
     }
   }
   step BUstep(node) {
     r = BUitem.getRW(node.right);
     l = BUitem.getRW(node.left);
     lr = merge(L, R);
     // Copy final output to [out]:
     if (root(node))
          out.put(node, lr)
     else BUitem.put(node, lr);
   }
 }
```

Figure 1: A divide-and-conquer pattern as a CnC module. The contents of the module includes the introduction and wiring of new collections, as well as *skeletons* for new serial steps.

There are three capabilities provided by CnC--:

- Stateful step computations (not treated here)
- Control over in-place memory operations
- Control over scheduling

The first two extend the API used by step code and the third extends the graph specification language.

3.1 Controlling Memory Management

CnC item collections are single-assignment and contain only immutable data. This is key to achieving a deterministic parallel model. Unfortunately, in-place parallel algorithms then become impossible. To fix this limitation, CnC-- includes unsafe memory operations such as `getRW` on item collections (get to modify). But then does a single use of `getRW` spoil the determinism of an entire CnC program? No, because modules provide *isolation*.

Figure 1 contains a module definition for a divide-and-conquer pattern, which, if provided the appropriate serial functions as arguments, would implement Quicksort. The pattern is implemented using in-place operations, including two unsafe operations in addition to `getRW`: `split` and `merge`. These act on a memory buffer in-place, returning pointers into subranges of the buffer and recombining them, respectively.

The collections `[TDitem]` and `[BUitem]` are effectively *private* to the module. All puts and gets to them are local and thus can be confirmed to never destructively modify memory without exclusive access. Such a module is safe for general use—similar to how native calls in Java are hidden behind library interfaces, or side-effecting functions in Haskell can be encapsulated to provide a purely functional interface.

For Quicksort, enabling in-place operation increases performance by 3.39x on an 8-core Nehalem with hyperthreading. CnC-- can take credit for this speedup, by allowing CnC to *safely* do what it could not before. Today, however, patterns must be certified by hand. In the future, there are many existing techniques that could be automatically applied, including linear type systems [17].

3.2 Four Controls for Step Scheduling

There is an extensive literature on parallel scheduling of directed task graphs. With CnC-- we provide a framework for navigating this space, tailoring scheduling behavior to individual parallel patterns.

It is important in CnC to make both the domain expert and tuning expert's jobs easier, especially as the proliferation of new multicore architectures makes re-tuning more frequent. The power of CnC--'s scheduling controls lies in their separation from the code itself[2], their composability (highlighted in Section 4), and their representation as declarative functions on tags, making them amenable to static analysis (future work).

Priorities: A pre-existing tuning mechanism of CnC, *priorities* allow the tuning expert to provide each step collection with a function that, given a tag as input, computes a priority for the step. For example, given a tree-shaped computation of a known size, the tag (tree index) is sufficient to compute the step's order within a depth-first tree traversal. Interpreting this number as a priority yields a *parallel depth first* scheduling of the computation [2].

```
(myTreeStep : ti | priority = indInDF(ti))
```

The tag `ti` (tree index) is used to set the priority by computing `ti`'s index in a depth-first traversal.

[2]The snippets of tuning specifications in this section would be part of a CnC specification but relegated to a separate file from the application logic.

Ordering Constraints: The control and data dependencies within an application provide semantic constraints on the order steps are executed. The tuning expert adds additional constraints to further control the execution.

Ordering constraints are declarative rules that constrain step orderings. For example, if a computation `b` depends on a computation `a` and both are indexed by a one dimensional iteration space, `i`, then the following would ensure that `b` is executed before `a` moves on to the next iteration:

```
(b : i) before (a : i+1);
```

Dynamic Chaining:

An important aspect of implementing parallel algorithms is deciding which computations are serialized (and therefore run without returning control to the scheduler). To this end, CnC-- allows *chaining* steps together when two steps match a chaining rule specified by the tuning expert. Chaining is dangerous in general due to the potential for an infinite chain to starve other steps, but can be used in a controlled way inside a pattern implementation.

Chaining is permitted in two situations: (1) producer / consumer relationships and (2) sibling steps that respond to the same collection of tags. The translator generates code to catch chained invocations in the step where tags are produced. In a simple scenario, if we wanted to serialize the invocation of `a` and `b` from the previous example, within each iteration, we would write:

```
(a : i) chainwith (b : i)
```

Affinity: Whereas chaining helps achieve temporal locality within consecutive tasks on a single thread, to maximize the overlap in working sets between non-consecutive tasks and between different hardware threads (constructive cache sharing [2]), CnC-- provides an *affinity* mechanism. Each a step collection may have an additional tag function to compute an integer *affinity group*. These integers are interpreted as binary tree indices. Therefore affinity numbers sharing longer binary prefixes are "closer" and should be mapped nearer to one another in the machine hierarchy (e.g. the same socket, or hardware threads in the same core).

By setting affinities, the tuning expert can control data distribution, for example, implementing a row-oriented decomposition of 2D data as follows:

```
(myStep : i,j | affinity = i % NUMPROCS)
```

Discussion: Implementation on TBB

In constructing the scheduler interface, we wanted to avoid discrete choices between schedulers in favor of composable controls. But this requires a runtime system with the flexibility to simultaneously support the above four mechanisms. Intel's TBB (Thread Building Blocks) [15] is well suited to this task and is the basis for our current CnC/C++ implementation. (Other recent systems, like PFUNC [11] would also make good candidates.)

Of the four controls above TBB makes two of them easy to implement. TBB already supports its own form of affinity. Also, TBB supports a form of "chaining" wherein one task returns another task-pointer to be invoked immediately. Ordering constraints can be implemented simply by introducing fake data dependencies in the code generated by the translator. Finally, priorities needed to build into CnC ourselves and are the only notable feature we would like to see added to TBB.

4 Case Studies

Both at Intel and through our collaborators, we have applied CnC to a variety of parallel applications. In this section we describe a subset of these applications, the parallel patterns they represent, and the lessons learned about scheduling.

Graphics: In one of our graphics-related applications we deal with the problem of parallelizing a directed graph of mesh computations, containing both data-parallelism (within the mesh) and task parallelism. In tuning this application we found that data parallelism was more important than task parallelism. In particular, we found that by *chaining* together the execution of mesh transformers on a given piece of data increased scalability—increasing the maximum parallel speedup (on an 8-core Nehalem machine with hyper-threading) from 4.2x to 8.12x.

If the opposite were true, and it were advantageous to complete each transformer on all data before proceeding to the next, we would instead have used ordering constraints to effect barriers between the transformers.

Data Deduplication: Data deduplication is a form of compression that eliminates duplicate copies of identical data. This is relevant, for example, in large email systems such as Gmail. In CnC, our deduplication application consists of a pipeline of three stages: Produce, Filter, and Compress. The first stage splits raw data up into blocks, computes fingerprints, finds anchor-points, and further splits the data into small chunks. The Filter stage uses SHA1 hashes to eliminate duplicates. Finally, the Compress stage further compresses unique items.

In spite of the simple pipeline structure of this CnC graph, there are significant per-application tuning decisions in terms of scheduling. With any pipeline in CnC, there is a trade-off between task-parallelism between the pipeline stages and data-parallelism. Depending on memory constraints and other factors one or the other may be desirable. In the deduplication application disk IO plays a substantial role. The domain expert, how-

ever, does not worry about these considerations when selecting the `Pipeline` pattern.

In tuning this application, we tested six scheduler configurations. Rather than simply chaining together the pipeline stages, the solution in this case was to assign priorities to the stages such that (1) later stages were more likely to execute, and (2) data elements near the beginning of the stream were more likely to be processed. This strategy encouraged flushing results to disk early, and limiting the amount of work in flight. It performed 60.1% faster than our baseline scheduler (TBB with default task scheduling).

Stencil Computations: Typical stencil computations repeatedly update the value in each position in a matrix based on the values of neighboring positions. Frequently the computation at each position is lightweight (e.g. the game of life) and the problem becomes memory bound. Cache affinity becomes important. In a *wavefront* stencil, each position depends on its north and west neighbors from the same iteration, and computation can proceed one diagonal at a time. Therefore, one could exploit locality by, for example, assigning contiguous pieces of the diagonal to different processors:

```
(stencilStep : i,j |
 affinity = (i-j) / (i+j) * NUMPROCS)
```

More complex strategies could also be encoded. For example, some implementations give special treatment to the early and late diagonals. These are too short to parallelize; thus they should be serialized (chaining). Further, the tail-end of one iteration can be explicitly overlapped with the start of the next iteration (ordering constraints).

5 Related Work

There has been substantial effort invested in characterizing parallel patterns and their taxonomies. Patterns may also go by the name "algorithmic skeleton" [14, 6, 4].

Recent years have seen major improvements in the selection of widely accessible libraries providing task-schedulers and concurrent data structures. These include the Microsoft Parallel Patterns Library (PPL), the `java.util.concurrent` library, and Intel's Thread Building Blocks (TBB). TBB, along with other recent systems such as PFUNC [11] and Manticore [7] provide highly parameterized or extensible schedulers. Lithe [8] excels in this respect, composing completely unrelated schedulers based on rationing machine resources. These systems, however, focus on low level parallel building blocks and have not aimed to capture higher level parallel patterns. Thus our effort is complimentary to this line of work, and indeed, our flagship CnC implementation is built on top of TBB.

The particular focus of CnC on separating the role of domain-expert and tuning-expert has precedents. For example, in the context of Haskell, parallel *strategies* [16] refers to a technique for physically separating algorithm and execution strategy in the code.

6 Conclusion

We have begun to collect parallel patterns in CnC. Rather than change the implementation for each new pattern that doesn't strictly fit the CnC model, we introduced a well-defined lower layer called CnC--. Once implemented through CnC--, many important patterns are deterministic and can be integrated with the CnC framework.

In this paper we reported on some of the patterns we integrated. For a sample of applications that did not achieve peak performance using CnC's default scheduler (and many do), significant improvements were made using the scheduling mechanisms exposed by CnC--.

7 Acknowledgments

We would like to thank Sagnak Tasirlar and Alex Wells for their contributions to deduplication and the graphics applications, respectively.

References

[1] ARB, O. The OpenMP API specification for parallel programming. `http://openmp.org/wp/about-openmp/`, 1997.

[2] BLELLOCH, G. E., GIBBONS, P. B., AND MATIAS, Y. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM 46*, 2 (1999), 281–321.

[3] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not. 30*, 8 (1995), 207–216.

[4] CAMPBELL, D. K. G. Towards the classification of algorithmic skeletons. Tech. rep., University of York, 1996.

[5] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2004), USENIX Association, pp. 10–10.

[6] FALCOU, J., SROT, J., CHATEAU, T., AND LAPREST, J. Quaff: efficient c++ design for parallel skeletons. *Parallel Computing 32*, 7-8 (2006), 604 – 615. Algorithmic Skeletons.

[7] FLUET, M., RAINEY, M., AND REPPY, J. A scheduling framework for general-purpose parallel languages. *SIGPLAN Not. 43*, 9 (2008), 241–252.

[8] HEIDI PAN, BENJAMIN HINDMAN, K. A. Lithe: Enabling efficient composition of parallel libraries. In *HotPar'09: Hot Topics in Parallelism* (2009).

[9] Intel (r) concurrent collections for c/c++. http://softwarecommunity.intel.com/articles/eng/3862.htm.

[10] INTEL. Intel Math Kernel Library (MKL). `http://www.intel.com/software/products/mkl`, 2004.

[11] KAMBADUR, P., GUPTA, A., GHOTING, A., AVRON, H., AND LUMSDAINE, A. Pfunc: modern task parallelism for modern high performance computing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), ACM, pp. 1–11.

[12] LAWRENCE RAUCHWERGER, FRANCISCO ARZU, K. O. Standard Templates Adaptive Parallel Library (STAPL). `http://parasol.tamu.edu/stapl/`, 1998.

[13] NVIDIA. Compute Unified Device Architecture (CUDA), 2007. `http://www.nvidia.com/object/cuda_home.html`.

[14] P. CIECHANOWICZ, M. POLDNER, H. K. The mnster skeleton library muesli - a comprehensive overview. ERCIS Working Paper No. 7, ISSN 1614-7448, 2009.

[15] REINDERS, J. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'REILLY, 2007.

[16] TRINDER, P., HAMMOND, K., LOIDL, H., L, S., AND JONES, S. P. Algorithm + strategy = parallelism. *Journal of Functional Programming 8* (1998), 23–60.

[17] WADLER, P. Linear types can change the world! In *Programming Concepts and Methods* (1990), North.