RICE UNIVERSITY

Enhanced Data and Task Abstractions for Extreme-scale Runtime Systems

Thesis by

Nick Vrvilo

RICE

Thesis for the Degree of Doctor of Philosophy
Department of Computer Science
Rice University (Houston, Texas) July, 2017

RICE UNIVERSITY

Enhanced Data and Task Abstractions for Extreme-scale Runtime Systems

by

Nick Vrvilo

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Vivek Sarkar

Vivek Sarkar, Chair
E.D. Butcher Chair in Engineering
Professor of Computer Science
Professor of Electrical and
Computer Engineering

Robert Cartwright

Robert Cartwright
Professor of Computer Science

[illegible]

Lin Zhong
Associate Professor of Electrical and
Computer Engineering

Houston, Texas

July, 2017

ABSTRACT

Enhanced Data and Task Abstractions for Extreme-scale Runtime Systems

by

Nick Vrvilo

Recently, we've seen a variety of emerging programming models targeting the next generation of HPC hardware, known as *extreme-scale* computing systems. Extreme-scale runtime systems need to address not only the problems presented by supporting new hardware, but also issues of scalability—whether in small-scale embedded environments or large-scale supercomputing clusters. While a runtime may present all of the necessary functionality to write software for an extreme-scale system, the runtime APIs are rarely a productive interface for application programmers. In this thesis, we present a set of abstractions, which are designed to be implemented on top of an extreme-scale runtime, that will increase programmability and productivity for software developers. These abstractions include support for blocking calls in a fine-grained task-based runtime, a data structure representation for relocatable data chunks, and a hierarchical model for design and analysis of macro-dataflow applications. We discuss and demonstrate the tradeoffs among implementation choices for these abstractions, since the specific hardware and software details of an application deployment may dictate the ideal method of implementing a given abstraction.

Acknowledgments

I thank the members of Habanero Extreme-scale Software team (both past and present) for all of their input and support. I thank the CnC community for their feedback on the CnC-OCR toolchain—and more recently on our CnC hierarchy work. I thank the members of the Extreme-scale Technology Group at Intel for their collaboration on the X-Stack Traleika Glacier project, which motivated the majority of the contents of this thesis.

I thank Gabriele Jost from Intel for her collaboration on Tempest for OCR, and for her feedback as a user of the CnC-OCR toolchain. I thank Ellen Porter from Pacific Northwest National Labs for her work on porting LULESH and HPGMG to CnC-OCR, and all the invaluable feedback she provided about the toolchain during that process.

I thank my committee for their time and input to improve the content of this thesis. I would especially like to thank my advisor, Vivek Sarkar, for all of his invaluable support, feedback, and insights during my PhD studies.

This work was supported in part by the Traleika Glacier X-Stack project (Department of Energy, Office of Science, under Award Number DE-SC0008717), and by the Data Analysis and Visualization Cyberinfrastructure funded by NSF grant OCI-0959097.

Contents

List of Algorithms

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Much of the current research focus in the field of High-performance Computing (HPC) is focused on achieving *extreme-scale computing*. The current prediction is that next-generation *extreme-scale* hardware will provide orders of magnitude more parallelism while requiring orders of magnitude less energy than current technology at a similar scale. One major target for extreme-scale computing is to provide datacenter scale machines that provide *exa-FLOPS*[1] of compute power. However, having the technology for exascale datacenters implies that we could also have petascale machines that fit in a single cabinet, or terascale machines that run with the same energy footprint as a current high-end desktop computer [2].

There are many difficult problems associated with extreme-scale computing, including energy consumption, operating system overhead, scalable algorithms, and resilience [3]. All of these issues will influence the programming model presented to application programmers on extreme-scale systems. We believe that the programmability of extreme-scale systems will play a critical role in the perceived utility and eventual adoption of next-generation hardware.

For example, due to cost and energy constraints, it is projected that extreme-scale systems may have a compute:memory ratio as small as 0.0036 bytes per FLOP [3]. Similar degradations in the compute:memory-bandwidth ratios are also expected. Based on these trends, we expect the need for a significantly different data model in extreme-scale programming models compared compared to the data models used in current runtime systems.

[1] *Exa-* is the SI prefix for 10^{18}, so an *exa-FLOP* is on the order of a *billion billion* floating-point operations per second. For comparison, the current largest supercomputer peaks at 0.093 exa-FLOPS [1].

Similarly, due to the orders-of-magnitude increase of available parallelism in extreme-scale systems, we anticipate that the overheads of blocking synchronization will become a serious problem. This will necessitate efficient support for blocking constructs in extreme-scale software runtimes in order to enable programmability without critically affecting performance.

In summary, all of the challenges associated with extreme-scale systems will have an effect on the programming model presented to the application programmers for these systems. New programming models such as Legion [4] and the Open Community Runtime (OCR) [5] have been specifically designed from the ground up to address the new issues associated with extreme-scale systems. However, if the runtimes for these systems do not provide sufficient software abstractions for programmability and productivity, then that will create a serious obstacle for the adoption and utilization of extreme-scale hardware.

Thesis Statement

We assert that runtime challenges tied to extreme-scale computing can be addressed with marginal overhead, while also limiting the burden placed on the application programmer.

Chapter 2

Efficient Encoding for Pointers in Relocatable Data Blocks

One of the fundamental concepts in the Open Community Runtime (OCR) is the relocatable *datablock* [5]. In this chapter, we propose an efficient approach for encoding C++ objects in OCR datablocks, and present a C++ library and other tools to simplify the work of the application programmer when using relocatable data blocks.

The key contributions presented in this chapter include the following.[1] We introduce a marshalled encoding for relocatable data blocks, and describe an algorithm for rewriting C++ class definitions to use our proposed encoding. We present a C++ library and other tools to simplify the work of the application programmer developing new applications or porting existing applications to emerging programming models, with our work specifically focusing on the OCR programming model [5]. Finally, we provide an experimental analysis of the overheads associated with our marshalled data encoding.

2.1 Motivation

In the past, much of the HPC software infrastructure has been focused on C or Fortran; however, there is currently an obvious shift towards the use of C++ in new programming models. With several U.S. national labs heavily investing resources in new C++-based programming models (RAJA at Lawrence Livermore [7], Kokkos at Sandia [8], and Legion at Los Alamos [4]), and the European Union investment in AllScale [9] (a C++ project built on HPX), it seems inevitable that C++ will become the de facto language for high-performance computing projects.

[1] Much of the work in this chapter was published at ISMM'17 [6].

At the same time, we are seeing a shift in programming models and runtime design as new challenges in scaling arise for resource-limited extreme-scale computing environments, ranging from exascale systems to low-energy embedded devices. Many of these challenges center around synchronization, resilience, and complex memory hierarchies [3]. One-sided communication is quickly gaining popularity as a way to decrease synchronization overheads and increased asynchrony in distributed systems [10]. We are also seeing restrictions in computation and data models, which are leveraged to provide stronger resilience guarantees [11] (e.g., allowing transparent migration of data previously located on a failed node to its neighbors). As heterogeneous hardware with dedicated accelerators become more common, runtimes also need to be able to relocate data to different memory subsystems to accommodate accelerator hardware restrictions or take advantage of locality [3]. However, the ability to transparently migrate data (whether as a form of one-sided communication, or to support resilience, locality, and other goals) can be hindered by embedded pointers bound to an object's current location in memory, which are common in object-oriented C++ software systems that make heavy use of aggregate objects.

2.2 Background

As previously discussed, we expect concepts like one-sided communication and transparent data relocation to become more important in future HPC software systems. We now discuss these ideas and their interactions with aggregate C++ objects. We also discuss why traditional serialization techniques do not apply well in this context.

2.2.1 One-sided Communication

One-sided communication is the default method of sharing data in PGAS languages like UPC [12] and models like OpenSHMEM [13]. Rather than using paired send/receive operations to transmit data between processes, data in remote memory is accessed directly via put and get operations. One-sided communication is becoming more common in popular distributed-computing paradigms, and as we move towards exascale systems (where the overhead of traditional point-to-point communication is aggravated by the scale of the machine), we anticipate that this trend to continue. Even the MPI standard, which is typically associated with two-sided send/receive-style communication, is bolstering its support for one-sided communication (originally introduced in MPI-2, but rarely used). In fact, it is now possible to implement a PGAS programming model like OpenSHMEM entirely in terms of the one-sided communication API extensions in the MPI-3 standard [14].

Currently, both one-sided and two-sided communication are only compatible with contiguous objects that do not contain internal pointers.[2] While the MPI and SHMEM APIs have support for sending non-contiguous bytes from an array or struct, there is still the underlying assumption that these bytes are read from a contiguous object in memory; however, that assumption is typically invalid when dealing with general C++ aggregate objects. To the best of our knowledge, no industry-standard HPC communication framework directly supports transfer of *aggregate objects*[3] through one-sided communication or transparently-managed data blocks.

2.2.2 Data Block Migration

Emerging parallel-computing runtimes, such as Realm [15] and OCR [11] also transparently manage data to improve scheduling and locality, e.g., when gathering inputs from remote nodes before starting a computation. Transparently supporting recovery by migrating tasks and data after a component failure, or redistributing workloads to adapt to a dynamic energy budget are other reasons the runtime might need to migrate data. Thus, the runtime needs the ability to transparently relocate objects.

In OCR, the application programmer cannot assume that a datablock will have the same base address when it is accessed by two separate tasks. For example, the runtime may move a datablock to a remote node and then move it back again, but at a new base address.

[2] We use the term *pointer* to refer to both C-style explicit pointers (e.g., `int *p`), as well as references in C++ (e.g., `int &r`).

[3] We use the term *aggregate objects* to refer to objects containing pointers to aggregated data. The aggregated data may also include aggregate objects, with nested pointers to more data.

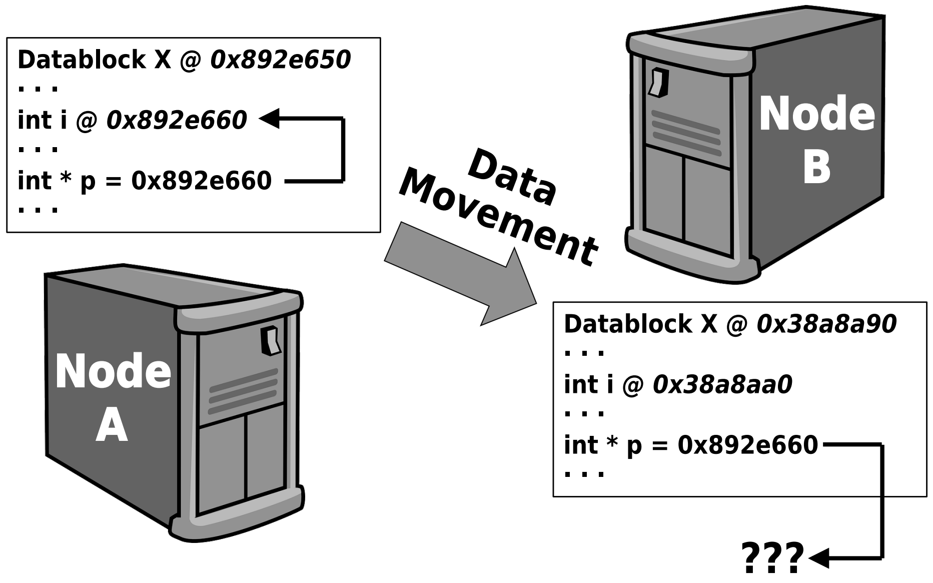


Figure 2.1: Example of pointer invalidation after migration of a datablock. The datablock *X* is initialized on node *A*, containing an integer `i` and pointer `p` that is set to the address of `i`. After the datablock *X* is migrated to node *B*, the base address of the datablock has changed, causing the absolute address stored in `p` to no longer correspond with the address of `i`.

The runtime may also choose to migrate a datablock to a different portion of the address space within a single shared-memory domain. For example, the machine might have a high-performance scratchpad, and through online profiling, the runtime may decide to migrate a heavily used datablock into the scratchpad memory. Since the base address of a datablock can only be assumed constant for the duration of the currently executing task, any pointers stored within a datablock should be considered invalid as soon as the task finishes executing.[4] Figure 2.1 illustrates an example where an error is introduced when a datablock is migrated to a remote node. When the datablock is moved, its base address changes, causing the value stored in the intra-datablock pointer to no longer correspond with the address of the target integer; instead, it now points off to an arbitrary memory position.

[4] In OCR, since *all data* that persists between tasks *must* be stored in a datablock, all valid pointers must point into datablocks. One possible exception is for function pointers, which point at *code* rather than data.

The requirement for transparently-migratable data is not limited to distributed systems, or even to runtimes with online profiling. Many hardware accelerators, such as GPUs and FPGAs, have their own dedicated memory and discrete address spaces. Naïvely copying blocks of data that contain pointers between main memory and dedicated accelerator memory may also lead to program errors.

2.2.3 Serialization

When copying aggregate C++ objects across memories, the current best-practice is to employ *serialization.* This involves packing the objects into a contiguous buffer at the source, and then unpacking (i.e., reconstructing) the objects at the destination. Note that for objects containing pointers, this typically means transitively applying serialization to all pointed-to objects. This can be problematic if an object X contains multiple references to some other object Y, as it may result in multiple copies of Y at the destination unless care is taken to track unique object pointers. Some popular serialization frameworks, such as Boost.Serialization [16], do the necessary bookkeeping to automatically handle duplicate pointers; in contrast, libraries like Cereal [17] eschew this additional bookkeeping in favor of higher throughput.

Serialization puts a burden on both the programmer (providing functions to pack and unpack[5]) and on the runtime (invoking the pack/unpack functions every time an object is relocated); nevertheless, the need to support migrating an object to another memory (e.g., a remote node or a hard disk) often makes serialization an essential feature, making the extra effort and overhead unavoidable.

However, there is a fundamental problem with using serialization in combination with one-sided communication: Due to the lack of explicit coordination with the remote

[5] Note that the burden of providing explicit implementations of serialization functions for user-defined types only exists in "classic" languages (e.g., C, C++ and Fortran) because they support neither run-time nor compile-time reflection. Languages supporting reflection can (and do) provide generic or automatically-generated serialization code. Even the modern "lower-level" languages (e.g., Rust) have support to auto-generate serialization code for most user-defined data types. However, custom serialization allows the programmer to inject semantic-aware optimizations, e.g., compression.

process, there is no straightforward way to trigger invocation of the *deserialization* code for an object at the destination. A similar problem presents in runtimes like OCR, which lack callback hooks for object pre/post migration processing, again precluding the use of traditional serialization. The purpose of this work is to enable support for C++ programs using aggregate objects within a distributed-memory OCR application.

2.3 Overview of Our Approach

In the case that traditional serialization is impractical or unavailable, we propose using a *marshalled*[6] data format—which is directly usable by the application code—as the primary representation for objects. (As discussed in section 2.2, two such motivating cases are one-sided communication and the OCR data model.) The application data is partitioned into discrete, fixed-sized *datablocks*. Intra-block pointers are encoded as relative offsets. Inter-block pointers are encoded with both a global handle for the target datablock, as well as the relative offset to the target data from the start of that datablock.

A critical requirement for our approach, of course, is correctness. We present a C++ library for creating and managing datablock-marshalled objects, and an algorithm, which we have implemented using the Clang LibTooling framework [18], for conservatively transforming persisted aggregate objects' definitions into our marshalled representation. We also present a set of possible optimizations to the output of our conservative transformation, and provide a set of run-time sanity checks to augment the correctness-checking process when applying these less-general optimizations.

Since the overarching goal of this work is to improve the productivity of an application programmer writing code for an OCR-like runtime (either directly via the OCR API or

[6] While the terms *serialize* and *marshal* are sometimes used interchangeably, we draw a traditional distinction between these two concepts. To *serialize* an object means to transform it into a contiguous byte stream, which can then be sent somewhere else, and eventually *deserialized* into an equivalent object. *Marshalling* is a more general term used for data representation transformations in memory, inclusive of (but not restricted to) serialization. A traditional example where the term *marshal* (but not *serialize*) would be appropriate is when transforming an object for compatibility with a foreign-function interface.

```
struct Node {
  int value;
  Node *left;
  Node *right;
};

struct Tree {
  Node *root;
  // ... methods ...
};
```

Listing 2.1: A simple tree data structure using native pointers.

through a programming system that targets OCR), our assumptions are based on the OCR programming model. The assumptions are as follows:

1. All *persisting data* must be stored within a runtime-managed datablock. We define *persisting data* as any data that may be accessed from more than one task.
2. The runtime is free to relocate a datablock after it is released by a task, but a datablock cannot move while currently in use by some task.
3. The contents of datablocks are opaque to the runtime.[7]
4. Each datablock has a corresponding *Globally Unique ID* (GUID), which is a valid global handle for the datablock regardless of where it currently resides in memory.

We can discuss these assumptions more concretely in terms of the simple tree data structure definition shown in listing 2.1. If a single instance of the tree data structure is accessed by more than one task, then it must be allocated within one or more datablocks. This is a consequence of #1, because between tasks, the runtime may relocate those datablocks, as per #2. Since the underlying runtime has no access to type information

[7] We do not require information on which data block entries contain pointers, as is required by *GC maps* for strongly typed languages like Java that include automatic memory management. Instead, memory is managed at the granularity of datablocks either manually or semi-automatically via reference-counting techniques.

on the contents of datablocks, as per #3, the contents of datablocks are copied byte-by-byte (as done by the standard `memcpy` function) to the destination when moved, such that the bitwise representation of our tree data structure remains the same before and after migration. However, since the destination address most likely does not match the source address, we should assume this opaque data transfer invalidates the pointer values of the fields declared on lines 3 to 8. Although the base address of the datablock containing our tree root may change several times throughout the program execution, as a consequence of #4, each task can still request access to that datablock via the datablock's GUID, since the runtime maintains a mapping from each datablock's GUID to its current location.

2.4 Pointer Usage in Tasks and Datablocks

Following the trend described in section 2.1, many simulation frameworks are being developed in C++ and are targeting exascale computing. An example of one such framework is Tempest [19, 20], a hydrodynamics simulation kernel developed entirely in C++, heavily using standard C++ idioms and aggregate data types. Many object-oriented C++ codebases, such as Tempest, use several persistent aggregate objects. Based on our past experience with a port of a subset of the Tempest framework onto OCR,[8] we know that the presence of pointers in aggregate objects (used extensively throughout the framework's API) is a major complication of porting an object-oriented C++ framework onto a datablock-based memory system, like that of OCR. More specifically, the assumptions we are making regarding memory (enumerated in section 2.3) have three non-trivial consequences for C++ applications targeting the OCR programming model:

1. Objects that persist across task boundaries cannot contain native pointer types,[9] since any pointer value is immediately invalidated if the target datablock is migrated to a new base address.

[8] Tempest is a very large simulation framework (with over 70k lines of code), and the port to OCR is still a work in progress.

[9] Note that task-local variables *are* permitted to contain native pointers, but their lifetimes are limited by task boundaries.

2. C++ code cannot make use of the built-in `new` and `delete` operators for dynamic memory management. The built-in `new` and `delete` operators in C++ simply delegate to the standard `malloc` and `free` functions for memory allocation, which would place new objects at arbitrary locations in the heap, whereas OCR requires that all persisting objects be allocated within a datablock. Instead, we require a custom allocation API for managing placement of new C++ objects within existing OCR datablocks. We do not see this as a major limitation since best practices in object-oriented programming often recommend the use of factory methods rather than using `new` directly. For example, we see a similar pattern with the usage of `std::make_shared` in C++11. Note that temporary objects that do not persist across multiple tasks need not be placed within datablocks, and thus can be allocated normally.
3. Due to the allocation descriptions already described, using C++ standard template library containers (e.g., `std::vector`) will not work unless an alternative to `std::allocator` is provided that both avoids native pointers and uses datablock-based allocation.

The base address of an acquired datablock must remain constant until it is released by the acquiring task; therefore, it is both legal and desirable to use native (position-dependent) pointers as local variables within a task, since the lifetime of that pointer is bounded by the task's lifetime. Only pointers that persist across multiple tasks must use a position-independent encoding. In the following two subsections, we describe the two additional classes of pointer use in OCR application code, and introduce a new position-independent representation for each class of pointer. These position-independent pointer objects can legally persist within OCR datablocks, and can simplify the process of porting object-oriented C++ code to OCR.

2.4.1 Intra-datablock Pointers

In the case when a pointer must address an object that resides within the same datablock as the pointer itself, then the pointer will always be at the same relative location with regard to the target object, regardless of the base address of the datablock; therefore, a *relative offset* can be used to position-independently encode that pointer. Note that this offset is calculated relative to the base address of the pointer-object itself, not the base address of the datablock.

We define the `RelPtr` template class to represent this kind of pointer. We assume that the size of a relative offset is less than or equal to that of a native pointer; hence, a `RelPtr` replacing a native pointer implies no space overhead.[10] Note that in C++ we are able to overload all operators that are typically used with native pointers, making the substitution of `RelPtr` objects for native pointers almost[11] transparent to the application programmer. A simplified version of the `RelPtr` class definition is shown in listing 2.2 for reference. While it is possible to use a `RelPtr` to address a non-persisting object (e.g., an object stored on a task's stack), it is always preferable to use a native pointer in such a case. (Since non-persisting objects are guaranteed not to move, using position-independent pointer objects is unnecessary.)

2.4.2 Inter-datablock Pointers

If the pointer object and its target object may reside in distinct datablocks, then it is not possible to find the target object using a constant offset from the pointer. This is due to the fact that either of the two datablocks may be arbitrarily moved by the runtime, changing their relative positions. Instead, we encode the position-independent pointer as a pair: the offset of the target object from the base address of its datablock, plus the GUID of that datablock. Assuming that the target datablock has been acquired by the current task, converting between the GUID and the base address is straightforward operation.[12]

[10] While a `RelPtr` introduces no additional memory footprint within a datablock, the additional template methods for the `RelPtr` class may increase the global code footprint.

[11] We say *almost* transparent because there are a few edge cases where the application programmer may need to modify existing code; e.g., when a user-defined implicit type conversion was applied to the original pointer value that the compiler will not automatically apply to the new pointer object.

[12] Translating a datablock's GUID to its base address (or vice versa) is not directly supported by the current OCR API; however, tracking this mapping for each of a task's acquired datablocks is just a matter of a little

```
template <typename T>
class RelPtr {
 public:
    constexpr RelPtr() : offset_(1) {}
    RelPtr(const RelPtr &other) { set(other); }
    RelPtr(const T *other) { set(other); }

    RelPtr<T> &operator=(const RelPtr &other) {
        set(other); return *this; }
    RelPtr<T> &operator=(const T *other) {
        set(other); return *this; }

    T &operator*() const { return *get(); }
    T *operator->() const { return get(); }
    operator T *() const { return get(); }
    bool operator!() const { return offset_ == 0; }
    bool operator==(const RelPtr &other) const {
        return get() == other.get(); }
    /* ... other pointer operators ... */

 private:
    ptrdiff_t offset_;
    ptrdiff_t base_ptr() const {
      return (ptrdiff_t)(this); }

    void set(const RelPtr &other) { set(other.get()); }
    void set(const T *other) {
        if (other == nullptr) offset_ = 0;
        else offset_ = (ptrdiff_t)(other) - base_ptr(); }
    T *get() const {
        assert(offset_ != 1);
        if (offset_ == 0) return nullptr;
        else return (T*)(base_ptr() + offset_); }
};
```

Listing 2.2: C++ definition of the `RelPtr` class, simplified for inclusion in this source listing. Please see the `ocxxr` repository for the full source code of the `RelPtr` class.

Calculating the target object's base address using the offset is trivial. Note that storing a GUID in addition to the pointer-offset within `BasedPtr` objects increases the size of all aggregate object types. Assuming that GUIDs are 128 bits, and native pointers are 64 bits, each native pointer replaces by a `BasedPtr` has a $3x$ space overhead.

We define the `BasedPtr` template class to represent this kind of pointer. As with the `RelPtr` class, we overload the relevant operators to make using `BasedPtr` objects as simple as possible. While the implementation details are not quite as simple as with `RelPtr`, it is still possible to make `BasedPtr` operations appear the same as native pointer operations in the application code—albeit with some additional overhead (both space and in computation time). A simplified version of the `BasedPtr` class definition is shown in listing 2.3 for reference.

Note that a `BasedPtr` is a valid substitute for any pointer into a datablock. This implies a simple, conservative process for taking an OCR application that illegally persists native pointers within datablocks, and correcting those violations: Replace every native pointer that is persisted in a datablock with a `BasedPtr`. The details of this process are covered in section 2.6.

2.5 Additional C++ API Support

To better facilitate the use of C++ code with OCR, we have built the constructs described in this chapter into a more general C++ library, which additionally provides C++-friendly wrappers for all existing OCR functions. We call the library `ocxxr`, which is a portmanteau of OCR and C++ (CXX). The library contains the `RelPtr` and `BasedPtr` classes described in section 2.4, as well as an API for using datablocks as the backing memory for an arena-based allocator. The library uses modern C++11 constructs. The allocation API mimics the style of the interface for allocating memory with an associated shared pointer, and

extra bookkeeping in our C++ code that wraps OCR's standard C-language API. Since OCR already tracks all of a task's acquired datablocks, extending the existing internal bookkeeping to support this translation is very straightforward.

```
template <typename T>
class BasedPtr {
 public:
    constexpr BasedPtr()
            : target_guid_(ERROR_GUID), offset_(0) {}
    BasedPtr(ocrGuid_t target, ptrdiff_t offset)
            : target_guid_(target), offset_(offset) {}
    /* ... other constructors and operators ... */

 private:
    ocrGuid_t target_guid_;
    ptrdiff_t offset_;
    ptrdiff_t base_ptr() const {
        return (ptrdiff_t)(this); }

    void set(const BasedPtr &other) {
        target_guid_ = other.target_guid_;
        offset_ = other.offset_; }
    void set(const T *other) {
        GuidOffsetForAddress(other, this,
                             &target_guid_, &offset_); }
    T *get() const {
        if (ocrGuidIsNull(target_guid_)) return nullptr;
        else return (T *)(AddressForGuid(target_guid_)
                          + offset_); } }
};
```

Listing 2.3: C++ definition of the `BasedPtr` class, simplified for inclusion in this source listing. Please see the `ocxxr` repository for the full source code of the `BasedPtr` class. The `AddressForGuid` and `GuidOffsetForAddress` routines refer to the GUID–pointer conversion operations discussed in section 2.4.2.

thus should be intuitive to C++11-savvy application programmers. E.g., the expression `new T(x,y)` can be rewritten as `arena.New<T>(x,y)` to allocate the object inside the given datablock *arena*, or `New<T>(x,y)` to use an implicit arena set via an earlier API call.

Our C++ wrappers for the C-language OCR API functions further improve the C++ integration, e.g., by adding template type parameters to eliminate C-style `void*` “generic” types and provide better static typing. One example of this is the `TaskBuilder<F>` template type, which allows construction of task instances that will run a target function, where `F` is the target function’s type signature, and all arguments passed to the task instance are checked against the argument types in `F`. We also leverage this extra type information in our pointer conversion algorithm, described in the next section.

Although `ocxxr` provides a limited set of utility classes and functions, the primary goal is to provide a foundational framework, enabling development of more complex object-oriented C++ libraries for OCR. The current version of the library is available on GitHub.[13]

2.6 Pointer Conversion Algorithm

To ease the process of porting legacy C++ code to the OCR model, we present a tool for automatic identification of native pointers that are persisted in OCR datablocks, and a process for conversion to position-independent representation. Our tool is built on Clang LibTooling [18], which provides a framework for automatic C++ source code analysis and source-to-source translation.

The transformation described here hinges on the following two key observations:

1. The `BasedPtr` class can legally replace any native pointer that addresses an object residing within a datablock.
2. Any data that persists across multiple tasks must be contained within an aggregate object that is passed as an argument to an OCR task. In other words, it is possible to

[13] https://github.com/DaoWen/ocxxr/tree/ismm17

```
1  Subroutine RewritePersistingPointers()
       Input: Source program that has been partially ported to ocxxr, but where some
              native pointers are included in persistent data.
       Result: All persisted native pointers in the input source program have been
               replaced with position-independent pointer objects.
2      Let Builders be the set of all TaskBuilder<F> type instances in the input
         program.
3      foreach $B \in Builders$ do
4          Let ArgTypes be the set of all datablock dependence types specified in the
             task function signature of F in $B$.
5          foreach $\tau \in ArgTypes$ do
6              Let $\tau'$ be the base type of $\tau$.
7              if $\tau'$ is a class type then
8                  call RewritePointersInClass($\tau'$)
9              else $\tau$ is a non-aggregate type.
10                 No rewrite is necessary for type $\tau$.
```

Algorithm 2.1: Top-level routine for whole-program persisting-pointer rewriting.

read a now-invalid pointer value if and only if that pointer is embedded within a datablock, and some task has an input dependence on that datablock.

The basic pseudocode for this transformation is given in algorithms 2.1 and 2.2. There are many other subtle details in the full algorithm that are not covered in the pseudocode. For example, it is not possible to directly replace a C++ reference type on a field with a corresponding `BasedPtr` type. Instead, a new field of type `BasedPtr` is created with a unique name, and the original field is replaced with a method that returns the original reference type. All uses of the original field are then transformed into method calls by appending a pair of empty parentheses to the original field name. In contrast, since `BasedPtr` overloads all pointer-related operators, uses of a transformed pointer-type field work transparently. Another example is handling class subtypes, which requires processing

```
1 Subroutine RewritePointersInClass()
      Input: A class type $\tau$.
      Result: The class type $\tau$ has been rewritten to $\tau'$, such that $\tau'$ contains no
              persistent-dependent pointers. This property is transitive to all
              aggregate members of $\tau'$.
      /* Calls to this subroutine must be memoized to prevent infinite
         recursion on mutually-recursive class types                          */
2     Let $Members$ be the set of all field members in $\tau$.
3     foreach $M \in Members$ do
4         Let $\phi$ be the type of $M$.
5         Let $\phi'$ be the base type of $\phi$.
          /* Rewrite pointer fields in the class                              */
6         if $\phi$ is a pointer type $\phi' *$ then
7             Rewrite $M$ from type $\phi' *$ to BasedPtr<$\phi'$>.
          /* Recursively handle nested class types                            */
8         if $\phi'$ is a class type then
9             call RewritePointersInClass($\phi'$)
```

Algorithm 2.2: Class-level routine of the persisting-pointer rewriting algorithm. Called in the inner-loop routine of algorithm 2.1. Recursively handles the rewriting of nested class definitions.

the classes in the inheritance hierarchy. For simplicity, we cover only the core concepts of the transformation in this chapter, and refer interested readers to the source code[14] for the full details of the implementation.

2.6.1 Description of the Algorithm

The top-level transformation routine is described in algorithm 2.1. We assume that the input program has already been partially translated to the OCR programming model using the `ocxxr` library; however, the input program may still store native pointers in datablocks, meaning the program likely only run in shared memory under the assumption that datablocks are never migrated.

We use only the types found in task arguments (i.e., the data types of the input dependence datablocks) as the root set for this transformation. This helps us avoid processing transient datablocks with a single-task duration (essentially being used as task-local scratch space), and avoid unnecessarily coercing the associated types into the position-independent encoding. We find and iterate through our root set of types in lines 2 to 4).

The *base type* τ' of τ (defined on line 6 of algorithm 2.1) corresponds to the target type of a pointer, or the element type of an array; e.g., the base type of `int*` is `int`, the base type of `float[10]` is `float`, and the base type of `Node*(*)[]` is `Node`. The conditional call in the inner loop (on lines 7 to 8) starts the recursive processing of each of the class types in our root set. Since only class types[15] can contain aggregate object pointers, no other types require rewriting.

The subroutine call in the inner-loop of algorithm 2.1 is described in algorithm 2.2. This subroutine transforms the specified class to remove native pointers, and it is also recursively applied to any class types referenced in the fields of that class. Lines 2 to 4 iterate through each of the target class's fields. Lines 6 to 7 transform fields with native pointer types into position-independent `BasedPtr` object types, which are safe to store

[14] https://github.com/DaoWen/ocxxr-ptr-xform

[15] We consider *class* and *struct* to be synonymous here.

within an OCR datablock. Finally, lines 8 to 9 recursively apply this subroutine to any new class types.

As described in the comment above line 2, calls to the `RewritePointersInClass` routine must be memoized in order to avoid potential infinite recursion. Since the number of `TaskBuilder` variable declarations in the input program must be finite, and the total number of class types referenced in any typeable C++ program must also be finite, we can conclude that this algorithm will always terminate. The overall computational complexity of the algorithm is linear in the *template-expanded size* of the input program. Note that if a declaration of a `TaskBuilder<F>` type appears inside of a templatized function or method, then the type `F` may be defined in terms of other type parameters, and each concrete type instance must be processed. This is analogous to running the algorithm on the fully template-expanded source code. While further optimizations to this transformation are possible, we believe the current approach is acceptable since the algorithm is relatively simple, yet the overall complexity is no worse than that of the code generation necessary to produce the application binary.

2.6.2 Example of Program Transformation

We now walk through an example of running our pointer conversion algorithm on a simple `ocxxr` program, shown in listing 2.4.

1. First, we query for all instances of the `TaskBuilder<F>` type declared in the input program. We find an instance on line 16 and another on line 26. For the first instance, `F` is the type signature of the function `SubTask`.
2. The `SubTask` function has two parameters, with types `int` and `Arena<Tree>`, respectively. The `int` type is ignored since it's a primitive type. However, `Arena` is a datablock type containing an object of type `Tree` as its root element. Since `Tree` is a class type, we need to process it.
3. The class `Tree` just one field, which has type `Node*` (line 8). Since this is a pointer type, we need to rewrite the type to `BasedPtr<Node>`. We can see this update on the

```
struct Node {
  int value;
  Node *left;
  Node *right;
};

struct Tree {
  Node *root;
  // ... methods ...
};

void SubTask(int i, Arena<Tree> tree) {
  Node *tree_root = tree->root;
  if (i < 10) {
    // ... do something with tree_root ...
    TaskBuilder<decltype(SubTask)> builder = /* ... */;
    builder.CreateTask(i+1, tree);
  } else {
    Shutdown();
  }
}

void MainTask() {
  Arena<Tree> tree = Arena<Tree>::Create(ARENA_SIZE);
  // ... set up tree ...
  TaskBuilder<decltype(SubTask)> builder = /* ... */;
  builder.CreateTask(0, tree);
}
```

Listing 2.4: Simple tree data structure in `ocxxr` using native pointers, using the tree data structure originally from listing 2.1.

same line in listing 2.5. Since the base type of the field is the class type `Node`, we also need to recursively handle that class type.

4. The class `Node` has three fields (lines 2 to 4).
   (a) The first field has primitive type `int`, so we ignore it.
   (b) The second field has pointer type `Node*`, so we rewrite the type to `BasedPtr<Node>`. However, due to memoization we see that the `Node` class has already been processed (or, rather, that it is currently being processed) so we skip recursively handling the `Node` class, and immediately return to processing the fields of `Node`.
   (c) The third field also has pointer type `Node*`, so we also rewrite its type to `BasedPtr<Node>`, and again skip recursively processing `Node` due to memoization.
5. Now that we have processed all of the fields of `Node`, we return to processing the other fields of `Tree`. However, there are no other fields in `Tree`, so we return to the top-level routine to process the next task argument type.
6. There are no more arguments to process in `SubTask`'s signature, which means that we are done processing the current `TaskBuilder<F>` instance.
7. We move on to the next `TaskBuilder<F>` instance, which is on line 26. This instance actually has the same type for `F`; however, since we would have to iterate over the whole type signature to see if it is equal to a previously processed signature, it is simpler to naïvely process the whole signature again. Again, the primitive type `int` is skipped. The second argument has type `Arena<Tree>`, which means we need to process the class `Tree`; however, we return immediately due to memoization.
8. There are no more `TaskBuilder<F>` instances to process, which means that the transformation of the input program is complete! The resulting rewritten code is shown in listing 2.5.

Notice that the type of `tree_root` on line 13 was not altered. This is because the scope of the pointer value stored in `tree_root` is limited to the currently-executing task,

```
struct Node {
  int value;
  BasedPtr<Node> left;
  BasedPtr<Node> right;
};

struct Tree {
  BasedPtr<Node> root;
  // ... methods ...
};

void SubTask(int i, Arena<Tree> tree) {
  Node *tree_root = tree->root;
  if (i < 10) {
    // ... do something with tree_root ...
    TaskBuilder<decltype(SubTask)> builder = /* ... */;
    builder.CreateTask(i+1, tree);
  } else {
    Shutdown();
  }
}

void MainTask() {
  Arena<Tree> tree = Arena<Tree>::Create(ARENA_SIZE);
  // ... set up tree ...
  TaskBuilder<decltype(SubTask)> builder = /* ... */;
  builder.CreateTask(0, tree);
}
```

Listing 2.5: Transformed tree data structure code from listing 2.4, now using `BasedPtr` objects for all persisted pointer values.

which means it does not persist across multiple tasks, and thus does not need a position-independent encoding.

Source files corresponding to listings 2.4 and 2.5 are available as examples in the `ocxxr` repository.

2.6.3 Limitations of the Algorithm

The primary purpose of this algorithm is to identify native pointer fields in aggregate objects that are persisted across multiple tasks. We do not attempt to identify pointers stored directly in global memory; i.e., we assume that all data that is accessed across multiple tasks is stored within a runtime-managed datablock. We also assume that the full set of types that may be embedded within datablocks and shared across tasks are reachable from the `TaskBuilder` definitions. This assumption means that programs that are written directly in OCR rather than using our `ocxxr` library are not analyzable using this method. It also means that if the application programmer uses explicit casts to read or write objects stored in a datablock, we will not find the type information used in the cast, and we may not correctly transform the corresponding class definitions. However, explicit casts are only problematic if they add otherwise "hidden" type information, such as `long→T*` or `void*→U*`. Valid casts up or down a class hierarchy are not problematic.

We assume that all of the source code for a program is accessible, and the entire program can be recompiled after the transformation. This can be problematic when data types from third-party libraries are used, as the user's application might just compile against a header file and then link against a pre-compiled library.

Since we transform the class definition for any objects that may be persisted in a datablock, it is possible that an objects used as temporary data will also be re-encoded using our technique. The programmer could manually create separate versions of the class definition (one for temporary objects and one for persistent objects); however, automating that process is beyond the scope of this work.

Finally, our algorithm only addresses the pointers stored in objects, assuming that all of the aggregate objects are allocated within datablocks. The ideal partitioning of aggregate objects into discrete datablocks is currently determined manually by the application programmer. The programmer must ensure that any calls to `new` associated with the transformed types are properly rewritten to use our `ocxxr` API to allocate the objects within a datablock rather than placing them directly in unmanaged heap memory; however, in our experience, manually rewriting the `new` operations after identifying and transforming the class definitions for all persisting data is much more straightforward and less error-prone than the pointer identification and transformation.

2.7 Position-independent Encoding Optimization

The automatic conversion described in section 2.6 only makes use of the more general `BasedPtr` type. While correct, programs produced by this conservative approach will obviously be outperformed by a program that utilizes the `RelPtr` type for storing intra-datablock pointers. For example, assuming the `Tree` and all of its `Nodes` are allocated within the same `Arena` datablock in listing 2.5, then it would be legal to replace the `BasedPtr<Node>` types on lines 3, 4 and 8 with `RelPtr<Node>` types.

It is possible to hook into Clang's alias analysis framework to attempt to automatically identify possible `RelPtr` candidates; however, we would require a custom alias analysis for determining if a candidate pointer and its target are always allocated within the same datablock. Traditional alias analysis algorithms check if pointer A and pointer B both alias to the same object C; in contrast, we need to know if pointer A that points to object B must reside in the same datablock as object B, for all possible targets B of the pointer A. Since we do not currently have a datablock-aware alias-analysis framework available, we propose two alternatives to help with optimization.

One option is to create a third class of pointer object that uses the relative-offset encoding for intra-datablock references, but falls back to the base-offset encoding for inter-datablock references. We name this hybrid representation `BasedDbPtr`, since it is very similar to the `BasedPtr` semantically, but we add *Db* to the name as a reminder that it must be allocated within a datablock in order to support the relative-offset encoding. It is included along with `RelPtr` and `BasedPtr` in `ocxxr`. Note that while a `BasedDbPtr` should be much more efficient than a `BasedPtr` for intra-datablock data accesses, the copy-initialization operation is much more expensive since we must check if the `BasedDbPtr` object and the target object are allocated within the same datablock (and use `RelPtr`-style encoding for intra-datablock references), rather than simply copying the target GUID and offset values.

One additional interesting feature of the `BasedDbPtr` class is that it explicitly distinguishes between inter- and intra-datablock pointers. By exposing a predicate for checking that property, we can leverage this encoding to more efficiently build and traverse data structures where the partitioning of data across datablocks is dynamically encoded within the pointers to the data.

A second method is to optionally store additional bookkeeping information in `BasedPtr` objects, and log any references to inter-datablock addresses from a specific `BasedPtr` field. The application programmer then runs the program with several inputs, producing a list of the `RelPtr` candidate fields (i.e., the complement of the set of logged fields). Programmers can then focus on a (hopefully) smaller list of candidates, and convert into `RelPtr` type just the pointers that they can guarantee must be intra-datablock references. The `BasedDbPtr` class could hypothetically be extended to perform this extra bookkeeping; however, this is not currently supported in `ocxxr`. We leave the exploration of this optimization method for future work.

2.8 Position-independent Pointer Sanity Checks

Compiling with `OCXXR_PTR_CHECKS` defined enables a set of sanity checks that should be useful during development and debugging of `ocxxr` applications. These checks help ensure that all `ocxxr` pointer objects have targets that are either `null` or are located in a datablock that is accessible from the current task. For a `RelPtr`, the check ensures that the pointer object and the target object are in fact within the same datablock, which must be one of the datablocks that was acquired for access by the current task. The `BasedPtr` and `BasedDbPtr` classes make similar checks, but without the requirement of being in the same datablock. The `RelPtr` checks are done on assignment to the pointer object, whereas the `BasedPtr` checks must be deferred until the pointer object is dereferenced in order to guarantee access to the target datablock. The `BasedDbPtr` class does checks both when assigning and when dereferencing, depending on whether the target is located intra- or inter-datablock.

We enabled these checks during the development of our benchmarks. The checks correctly flagged invalid references to objects allocated on the execution stack or in other non-datablock sections of memory. These checks also uncovered `RelPtr` objects allocated on the execution stack. While a `RelPtr` will still function correctly in this context (since the target datablock will not be moved while the current task is still accessing it), using a native pointer is a better fit for such pointers that are limited in scope to the current task. We discuss the overheads associated with these extra sanity checks in the next section.

2.9 Experimental Evaluation and Analysis

We chose to focus our experimental analysis on the additional overhead introduced by using our position-independent pointer objects in place of native pointers. However, it is important to note that while our pointer abstractions do introduce a measurable overhead, the native-pointer variants of the benchmarks violate the OCR data model restrictions discussed in section 2.4, and therefore will almost always result in errors when run on a multi-node distributed OCR configuration.

2.9.1 Benchmarks

We use a set of five benchmarks to evaluate the performance of our implementation, and specifically to measure the overhead introduced by our position-independent pointer objects when compared with using native pointers. The full source code for the benchmarks and the scripts used to run them are available in the `ocxxr` repository. A brief description of each of these benchmarks follows.

BinaryTree Performs a large number of lookups and insertions of key/value pairs stored in an unbalanced binary tree. The tree data structure is naïvely implemented in a single OCR datablock, which allows us to use the `RelPtr` pointer representation for all of the internal pointers to tree node objects (since all pointers are intra-datablock pointers). This benchmark is also the basis for the sample code shown in listings 2.1 to 2.5. The pointer type used by the tree class is parameterized, making it easy to switch among our multiple pointer representations to test a particular implementation.

Hashtable Performs a large number of lookups and insertions of key/value pairs stored in a hashtable, which is implemented as an array with a linked-list in each "bucket" to hold entries with colliding hashes. This hashtable implementation is adapted from a proof-of-concept general concurrent hashtable code included in the CnC-OCR framework. The top-level array is allocated in one datablock. The buckets are composed of fixed-sized blocks of key/value pair entries, with each of these blocks allocated in its own datablock. New entries are always added to the first block in a bucket, and a new block is inserted if the first block is full. All inter-block pointers in these internal structures use the `BasedPtr` representation.

LULESH A port of LULESH 2.0 [21] (a hydrodynamics simulation kernel) to `ocxxr`. Our implementation is based on an existing port of the code to the CnC-OCR programming model [22]. LULESH uses "indirection arrays" to represent the relationships in an unstructured hex mesh, which we implement using the `RelPtr` class. The application also includes a large aggregate data structure for storing constant data, where we again used the `RelPtr` class to encode the base pointers to the dynamically-sized arrays used to store the initial state of the mesh.

Tempest Performs climate modeling calculations on a cubed-sphere grid using a subset of the Tempest framework [20], ported to OCR using `ocxxr` constructs. Our Tempest mini-app creates a small set of patches (each covering a section of the cubed-sphere grid), and simulates 500 time-steps on the grid. Since the Tempest framework is written in idiomatic C++—making heavy use of aggregate objects in the code, including standard library containers such as `std::vector`—this mini Tempest application is a prime example for the techniques presented in this chapter. Note that, although our kernel is fairly simple, the supporting library code involves a large set of classes, making the full application code non-trivial.

UTS A port of the *Unbalanced Tree Search* benchmark [23] to `ocxxr`. Unlike many traditional implementations, which simply allocate transient tree nodes on the runtime stack during the recursive search calls, we reify the entire tree data structure with OCR datablocks. Clusters of connected nodes are allocated within discrete datablocks, and all inter-node pointers are represented using our `BasedDbPtr` class. Since the `BasedDbPtr` class can represent both intra-datablock relative offsets (as done by `RelPtr`) and inter-datablock based offsets (as done by `BasedPtr`), these objects can be used for all inter-node pointers in the tree. Furthermore, the `BasedDbPtr` provides a simple way to check if the pointer's target is local or within another datablock, which allows us to determine when to create a new task to acquire the target data when the node pointers cross datablock boundaries.

2.9.2 Experimental Setup

All experiments were run on a dedicated server with a 3.50GHz Intel Core i7 Ivy Bridge 4-core CPU (Turbo Boost disabled) and 8GiB DDR3 memory, running Ubuntu 16.04. All benchmarks were compiled with Clang v3.8. We used commit `e38167bf` of OCR,[16] running on the x86/x64 build with assertions disabled. Each reported time is the average of 100

[16] https://xstack.exascale-tech.com/git/public?p=ocr.git;a=commit;h=e38167bf260b

runs, with the error bars representing a 95% confidence interval. The workload of each benchmark was adjusted to a single-threaded execution time of about 2–10 seconds, as we found that run times shorter than 1 second often do not provide a sufficiently high signal to noise ratio, resulting in much more volatile measurements.

Since we are concerned with the overheads introduced by our pointer objects rather than the baseline performance of the benchmarks—and the pointer object usage is orthogonal to any multi-threading performance bottlenecks—we chose to run these experiments with a 1-thread worker pool. Although all of our benchmarks can be executed in parallel on multiple threads, running single-threaded helps to eliminate some schedule-related volatility. Likewise, since we chose to use native-pointer versions of our benchmarks as our performance baseline, our experiments are restricted to shared-memory runs (as the native-pointer versions will not run on distributed OCR).

2.9.3 Results and Analysis

Figure 2.2 shows the mean execution times for each of our five benchmarks. For each benchmark, we compare the performance of three different versions: (1) a baseline version using native pointers, (2) a transformed version using position-independent pointer objects, and (3) the position-independent pointer version with the additional checks described in section 2.8.

For all cases except the *BinaryTree* benchmark, the overhead incurred due to using position-independent pointer objects was very minimal. Even with the extra sanity checks enabled, the mean execution time only exceeded the baseline by a few percent at most. The reason for the significantly-higher overhead observed in the *BinaryTree* benchmark can be seen in figure 2.3; the number of operations performed on our `ocxxr` pointer objects in the *BinaryTree* benchmark is over 20*x* higher than in *Hashtable*, which is the next closest benchmark with regard to this metric.

BinaryTree can be considered as the worse-case scenario for measuring the `ocxxr` pointer object overheads, as the benchmark's computation time is dominated by creating

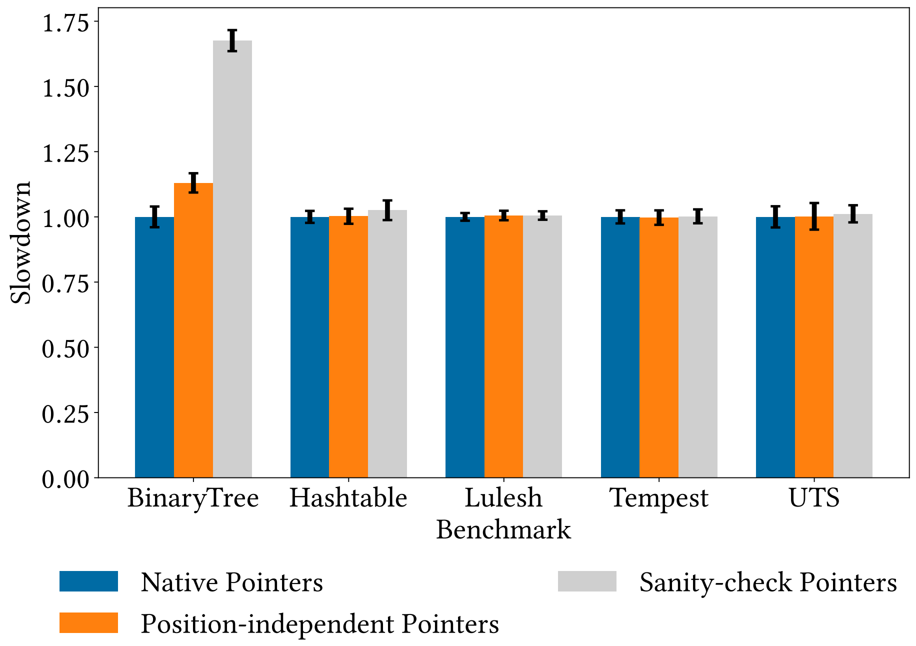


Figure 2.2: Execution times for three variants of each of our benchmarks. Times are normalized to the native pointer version of each benchmark. The sanity check variants are the same as the position-independent pointer versions, but with the addition of the debug checks discussed in section 2.8.

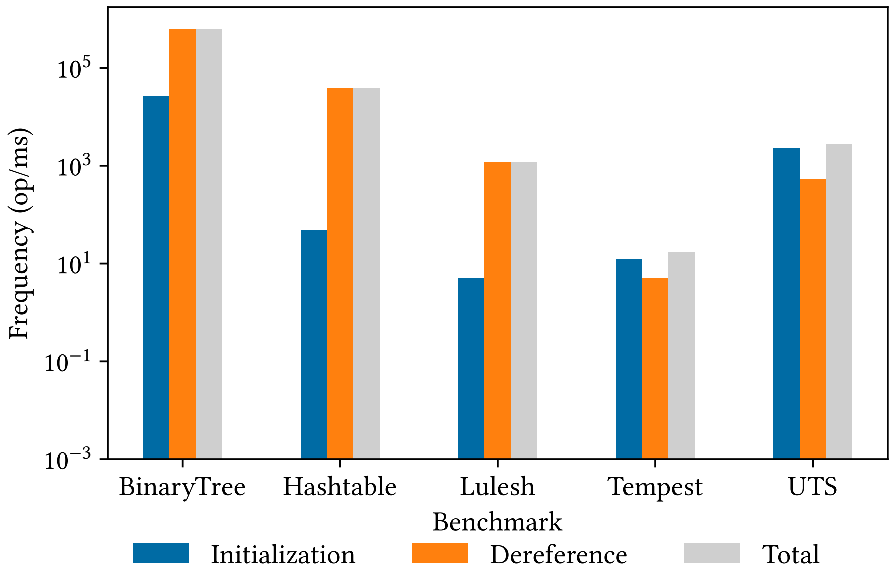


Figure 2.3: A comparison of the relative density of `ocxxr` position-independent pointer object operations in the total execution time of each benchmark. The times here correspond to the mean position-independent times from figure 2.2. Note that the y-axis uses log scale.

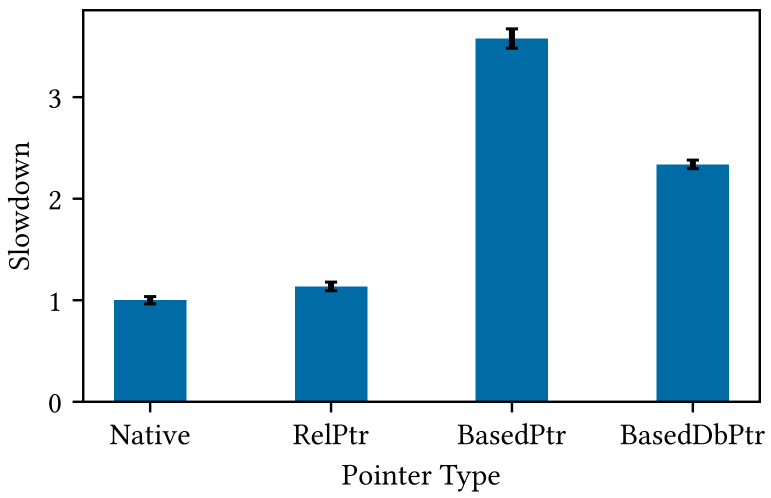


Figure 2.4: Execution time for the BinaryTree benchmark using each of the four pointer encodings discussed in this chapter. The times are normalized to the native pointer version.

and traversing the pointer objects that form the edges for the tree data structure. Even in this high-utilization scenario, we only observed about 13% slowdown when using our position-independent pointer objects compared to the native pointer baseline, and 67% total slowdown with the additional sanity checks.

In figure 2.4, we have reused the *BinaryTree* benchmark to measure the overhead of all three variants of the `ocxxr` position-independent pointer objects. As mentioned previously, this benchmark is structured very similarly to the code shown in listing 2.5. Since the entire binary tree data structure is allocated within a single datablock in this benchmark, we can choose any of our three pointer object types to encode the references from parent to child nodes.

The native and `RelPtr` times shown in figure 2.4 are identical to the first two cases shown in figure 2.2. In the case where we use `BasedPtr` objects to encode the pointers among our tree node objects, we see an overhead of about 3.5*x* the native pointer baseline. Since each pointer dereference operation requires a function call to translate the pointer target GUID into the corresponding datablock base address, it is not surprising that the overhead is significantly higher than for the `RelPtr` representation, which is able to directly compute the target address directly by using its stored relative offset. While a $3.5x$ slowdown for `BasedPtr` is non-trivial, we believe this pathological case is very unlikely in real applications. First, having the pointer operations constitute the majority of the computation in an application is unusual. Most real-world applications would also perform some heavy computation on non-pointer data (e.g., interpolations or matrix multiplies), which would offset the overall slowdown of the application. Additionally, code performing this kind of heavy pointer access would most likely include multiple objects within a single OCR datablock, meaning that the `BasedDbPtr` type is probably a better candidate. The UTS benchmark shows a more realistic example of an application that includes significant pointer-chasing, and the observed slowdown for that case is marginal.

We see that the `BasedDbPtr` variant's execution time falls about halfway between that for `RelPtr` and `BasedPtr`. This is as expected, since the `BasedDbPtr` class can be thought of as a compromise of the tradeoffs for our `RelPtr` and `BasedPtr` classes. However, note in figure 2.3 that the pointer-dereferencing operations in the *BinaryTree* benchmark are much more common than the pointer-initialization operations. The `BasedDbPtr` case benefits from this fact since it uses the relative-offset encoding for intra-datablock pointers (which is always the case here), and thus can avoid the extra function call incurred when dereferencing a `BasedPtr` object. However, the `BasedDbPtr` case still incurs the function-call overhead when initializing the pointer objects, since it must still perform the lookup for the target datablock's base address and size to determine if the `BasedDbPtr` resides within the same datablock as the target.

Both Tempest and UTS show a higher number of pointer initializations than dereferences in figure 2.3. Due to the mechanical translation of the library code and the relative simplicity of our Tempest kernel, many auxiliary data structures are created but not accessed during the patch updates, leading to this imbalance. In UTS, we instantiate the entire random tree, but the nodes are not traversed again after construction, making our UTS implementation our most initialization-focused sample.

It is worth noting that although the codebase used for our Tempest benchmark contains a large number of aggregate objects to model the many aspects of the hydrodynamics simulation—those objects were rewritten to use our `ocxxr` position-independent pointer objects to reference aggregate members—the actual density of pointer-object operations in the benchmark execution as reported in figure 2.3 is several orders of magnitude lower than the other benchmarks. This is because the Tempest code spends a significant amount of time performing floating-point operations in a loop to update the state of each grid patch. Furthermore, the absolute pointer addresses used in a particular task are computed once using the position-independent pointer objects stored in the patch's datablock, and then cached in a native pointer variable for the remainder of the task. Since the pointer-arithmetic to access the individual entries in the patches is done via that task-scoped native pointer value, the impact of the position-independent encoding on the execution time is very minimal. We would expect to see a similar trend in other compute-intensive applications.

Although one might expect our LULESH benchmark to exhibit similar properties to what we observed for Tempest, the CnC-OCR codebase that we used when porting LULESH to `ocxxr` performs element-wise updates rather than using tiles on the mesh, which means we perform only a few floating-point instructions when updating the individual element values on each iteration. While it would definitely be beneficial from a performance perspective to refactor the code to perform tiled updates, we prefer the untiled version for this study since it emphasizes the overhead of our `ocxxr` pointer objects used to encode the mesh structure, as evidenced by the higher proportion of pointer operations shown in figure 2.3 for our version of LULESH compared to our Tempest framework mini-app.

2.10 Related Work

MPI allows communication of non-contiguous data through its *derived datatypes* support [24]. OpenSHMEM currently has a much more limited functionality for puts and gets of strided array elements [25]. The functions in both MPI and OpenSHMEM for communicating non-contiguous data constitute a form of serialization support. Higher-level runtimes such as Charm++ and HPX use high-level serialization frameworks (resembling the popular Boost.Serialize API) to enable communication of user-defined data types types among compute nodes [26, 27].

A close analog in another runtime system to OCR's datablock concept is Realm's concept of *physical regions* [15]. While Realm's physical regions differ from OCR datablocks in some ways (e.g., reduction support and data-type homogeneity), they are similar in that both are discrete chunks of application data that are transparently migrated by the runtime to satisfy task dependencies. Individual elements in a region can be accessed via a *Realm pointer*. Realm pointers remain valid even after data migration because they are stored as an offset rather than an absolute address. However, in contrast to our `BasedPtr` class, Realm pointers do not store the handle of the target physical region. Other examples of programming models with datablock-like constructs include *data items* in distributed-memory Concurrent Collections (CnC) [28, 29], and distributed data-driven futures in HCMPI [30].

Most PGAS languages define a concept of a global pointer. Many PGAS languages depend on compiler support to handle global pointer accesses (e.g., X10 [31], UPC [12]). In contrast, UPC++ is a library-based solution with no specialized compiler support requirements [32]. Instead, UPC++ uses its `global_ptr` template class to handle globally-addressable data. Like our inter-datablock pointers, UPC++ global pointers are encoded as an offset into a block of data; however, the base address of each UPC++ global memory region is fixed, and only one such region exists per process, whereas in the OCR model we have multiple datablocks that may be dynamically relocated by the runtime.

Using offsets as position-independent pointers is also an established concept in some mainstream C++ libraries. Examples include `offset_ptr` from Boost.Interprocess [33] and *based pointers* in Microsoft Visual C++ [34]. The primary use case for these constructs is in data structures placed within memory-mapped files, where the file may be loaded at a different base address in each process that maps the file.

2.11 Future Research Directions

The ideas presented in this paper suggest several possible directions for future research on further optimizations and related concepts. For example, there may be many cases where it is possible to avoid the overhead of the base-address lookup for a `BasedPtr`'s target by directly supplying that value if it is already known. This optimization could be manually applied by the application programmer, or automatically applied by the compiler toolchain. As mentioned in section 2.7, a custom alias analysis in the compiler toolchain could also help identify `ocxxr` pointer objects that are provably safe to encode using the `RelPtr` class. Ideally, the transformation from native pointers to our position-independent pointer-object encoding would be handled automatically by an advanced compiler toolchain. One way to achieve that goal might be to start by making extensions to the type system, as was done with the *place types* proposal for X10 [35]. We assumed that the application programmer will manually partition the application data into discrete datablocks; however, automating the process of finding efficient data partitioning schemes would be another way to improve the application development process.

2.12 Summary

In this chapter, we presented a marshalled encoding for relocatable data blocks. We introduced `ocxxr`, a C++ library providing position-independent pointer objects and other useful classes for developing object-oriented OCR applications in C++. We also defined a conservative algorithm for rewriting native pointer types in an `ocxxr` application into our position-independent pointer types, allowing the rewritten classes to persist in datablocks that may be relocated by the runtime during a gap between task executions. We provide an implementation of this algorithm using Clang LibTooling.

To further aid in `ocxxr` application development, we outline possible optimizations for the output of our conservative rewrite algorithm, and provide a set of optional sanity checks to help maintain the correctness of the applications during the development and optimization process. We measured the overhead introduced when C++ aggregate objects are marshalled using our `ocxxr` position-independent pointer objects compared to a naïve baseline using native pointers. We found that the overhead observed in all but the most extreme case was minimal. The measured overhead compared to our baseline performance was less than $1.2x$ for the typical scenarios represented by our benchmarks, and still less than $3.6x$ for even the most extreme constructed scenario. Considering that the baseline implementations used for our benchmarks violate the OCR data model and will not correctly execute in distributed memory, we believe the tradeoff is acceptable.

Chapter 3

Practical Support for Lightweight Tasks with Blocking Constructs

While many extreme-scale runtime candidates are converging on the lightweight, non-blocking task model of execution [5, 36], the fact remains that the majority of programmers prefer to think in terms of imperative, serial control flows with blocking semantics. Since the performance impact of blocking resources (e.g., hardware threads) is expected to worsen as we move further toward extreme-scale technology, the desire of programmers to write straight-line, blocking code is directly at odds with performance-tuning best practices. Software toolchains need to address this discrepancy, and will ideally do so while adding minimal burden on the programmer.

In this chapter, we analyze the problem of supporting blocking constructs in a lightweight tasking runtime. We present several implementation strategies for supporting such constructs, including the use of threads, fibers and continuations. We evaluate these different solutions, and provide a discussion of the strengths and weaknesses of each approach. Although this chapter focuses on the Habanero-C programming model [30, 37], the concepts are transferable to other task-based runtimes.

3.1 Background

3.1.1 Continuations Support in Mainstream Languages

Support for continuations is no longer restricted to traditional functional languages (e.g., Scheme [38] or ML [39]). Many modern mainstream languages include compiler support for implicit transformations to continuation-passing style. Examples include C# [40], Scala [41, 42], Microsoft Visual C++ [43] (with proposed C++20 features [44]). Limited continuation support is also becoming more common in libraries, where support for fibers or coroutines is implemented without dedicated compiler transformations. This is especially common for supporting asynchronous I/O operations, such as in Boost.Asio [45]. This approach is also becoming more common in libraries for parallel computing, such as Sandia Qthreads [46], HPX [47], and Fibril [48] (a library-based implementation of the Cilk programming model [49]).

3.1.2 The Habanero-C Programming Model

The Habanero-C programming model [30, 50] builds off of past work in Habanero-Java [51] and X10 [31]. At the core of the Habanero-C programming model are the `async` and `finish` constructs. The `async` construct starts an asynchronous (potentially parallel) computation. Cilk's `spawn` keyword is very similar to `async`. The `finish` construct creates a new synchronization scope, such that the `finish` construct blocks awaiting the completion of all asynchronous tasks created within the *finish scope*. This differs from the behavior of Cilk's `sync` keyword, because the finish scope created by `finish` is a *dynamic scope*. Asynchronous tasks created within nested function calls, or even those created within other asynchronous tasks, must also complete before the `finish` construct completes (rather than just the asynchronous tasks created within the current function scope).

In addition to `async` and `finish`, Habanero-C model provides a number of other devices for parallelism, such as promises, futures, and *data-driven tasks*. All three of those devices are closely related to the `async_await` construct on Habanero-C. The key difference between `async` and `async_await` is that the latter allows the programmer to specify a list of *dependencies* that must be satisfied before the task is scheduled for execution. This allows the application programmer to write fully non-blocking code, using `async_await` to delay computation until all of the input data becomes available. In recent years, we have seen similar constructs appear in OpenMP [52] and HPX [47].

As a practical application of `async_await`, HCMPI [30] integrated data-driven tasks with the MPI programming model, allowing application programmers to use `async_await` with the handle returned by an asynchronous MPI function call (e.g., `MPI_Irecv`). This abstraction enables runtime-management of the overlap of communication and computation. This is typically much simpler for the application programmer than estimating how much computation is needed to cover a communication delay, and manually scheduling the computation between the non-blocking function call and a corresponding `MPI_Test` or `MPI_Wait` call.

There are currently two major implementations of the Habanero-C programming model. The first is the Habanero-C *language*, which depends on a custom source-to-source compiler to support extensions to the C language. We refer to the Habanero-C language as HC. The second is the Habanero-C library, which requires no special compiler support, but still provides a large subset of the constructs available in HC. We refer to this library implementation of Habanero-C as HClib. The HClib API supports both C and C++; however, the C++ API is usually preferred since C++11 lambdas provide programmer-friendly variable-capture support for passing lexically-scoped values to asynchronous tasks. The HClib project heavily reuses code from the earlier HC runtime, forming a single Habanero-C runtime ecosystem.

3.1.3 Deadlock

The bane of any multitasking system with blocking constructs is the possibility of *deadlock*, where two or more tasks block indefinitely due to unsatisfiable cyclic dependences. Coffman et al. [53] defined a set of four conditions that are both *necessary* and *sufficient* for deadlock to occur. The conditions are summarized by Tanenbaum [54] as follows:

1. **Mutual exclusion condition**: Each resource is either currently assigned to exactly one process or is available.
2. **Hold and wait condition**: Processes currently holding resources that were granted earlier can request new resources.

3. **No preemption condition**: Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. **Circular wait condition**: There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Note that *all four* conditions *must* be present in order for deadlock to be possible in a system. In other words, if we can guarantee that any one of these conditions is *not* present in a given system, then that system *must* be deadlock-free.

3.2 Overview of Our Approach

We examine the difficulties tied to supporting blocking constructs in a lightweight tasking runtime, specifically in the context of the Habanero-C programming model. We enumerate possible strategies used for scheduling tasks in the runtime in existing runtimes, paying particular attention to the property of deadlock-freedom. We select six strategies to implement within HC and HClib, including thread-based and fiber-based solutions within the runtime, compiler-supported continuation-passing-style (CPS) transformed code, as well as solutions where the application code is manually transformed to use only non-blocking constructs (i.e., manual CPS transformation, making heavy use of futures and data-driven tasks). Finally, we evaluate our six strategies implemented in the Habanero-C programming model (including HC and HClib), with regard to usability, performance, and resilience.

To the best of our knowledge, this is the first work to evaluate a wide variety of blocking construct support options (threads, fibers, continuations, and fully non-blocking) within the context of a single runtime ecosystem.

3.3 Deadlock Scenarios in the Habanero-C Runtime

Since the Habanero-C programming model provides blocking constructs (e.g., `finish`), it is important to ensure that programs using these constructs are deadlock-free— i.e., we don't want our software to hang. The possibility that the dependences among program tasks may introduce a dependence cycle is a problem, but there are well established techniques to avoid these problems; e.g., you can sort resources to get a global acquisition order, thus avoiding hold-and-wait cycles. We would expect programmers writing concurrent software to either use these techniques, or to use a deadlock-free subset of the available concurrency constructs.

3.3.1 Simple Task Scheduling for Async/Finish Programs

The core Habanero programming model, including only the `async` and `finish` constructs, has many desirable properties, including serializability, race-freedom, determinism, and deadlock-freedom [51]. We now describe a few possible methods of scheduling `async` tasks to run on a set of threads in an application process.

One Thread per Async

The simplest way to schedule a program written with `async` and `finish` is to create a new thread for each `async`, and destroy that thread when the `async` completes. This strategy matches the semantics described in the C++11 standard [55], where a function launched with `std::async` must run "as if in a new thread of execution."[1] While simple, this strategy can easily exhaust system resources by creating thousands of parallel threads, e.g., when executing a parallel divide-and-conquer task tree. For example, current versions of OS X limit the maximum threads per process to around 2k, meaning that a naïve recursive

[1] This description applies to `std::async` only when called with the `std::launch::async` policy. If `std::launch::deferred` is used instead, then the corresponding function is run lazily (i.e., when the result value is demanded and awaited) "in the thread that called the waiting function" [55].

```
uint64_t fib(uint8_t n) {
    // Base case
    if (n < 2) return n;
    // Recursive case
    uint64_t result_left, result_right;
    HCLIB_FINISH {
        hclib::async([&, n]() { result_left = fib(n - 1); });
        hclib::async([&, n]() { result_right = fib(n - 2); });
    }
    return result_left + result_right;
}
```

Listing 3.1: Naïve recursive function for computing the *nth* Fibonacci number in parallel.

Fibonacci program—as shown in listing 3.1—will crash when computing $Fib(n)$ for $n \geq 16$ (assuming that each `async` requires a new thread, as just described).[2]

Fixed-sized Worker Thread Pools

It is almost always desirable to avoid creating a potentially unbounded number of threads for executing the asynchronous tasks in a parallel program; e.g., limiting the number of threads to the number of processor cores in the system helps to avoid oversubscription of hardware resources. A common solution for this problem is to execute all tasks on a fixed number of threads, which form a *thread pool*. Rather than destroying a thread when a task completes, the thread waits until a new task is available, and then executes the new task. These long-lived threads are referred to as *workers*, and they continue to execute asynchronous tasks (assigned through some scheduling mechanism) until the program completes and the worker thread pool is destroyed.

However, using a fixed-sized worker pool in conjunction with blocking constructs can easily introduce deadlocks into an otherwise error-free program. If all of the worker threads become blocked at the same time, then the program can no longer progress (i.e., it

[2] The naïve recursive Fibonacci function makes $2 \cdot Fib(n+1) - 2$ recursive function calls to compute $Fib(n)$, meaning $Fib(16)$ requires over 3k `async`s in this case.

deadlocks). The `finish` construct in HClib can easily cause such deadlocks if the runtime is executing with this strategy. For example, the recursive Fibonacci program shown in listing 3.1 will deadlock with high probability for $n \geq 10$ because all workers will become blocked in one of the many finish scopes (i.e., none of the workers will be doing any useful computation).

Worker Pools with Compensation for Blocking

One solution to the blocking-thread problem with a fixed-sized worker pool is to relax the constraint for a completely static worker pool. A naïve approach would be to simply create a new replacement thread every time a one of our worker threads blocks; however, more intelligent (but more complex) variants of this strategy exist in production multitasking runtimes. For example, the Java Fork/Join framework [56] supports worker thread pools that attempt to maintain a constant level of parallelism rather than a constant worker thread count. A worker thread can cooperatively signal to the pool that it is about to block, which *may* cause the thread pool to compensate for the blocked thread by creating a new worker thread [57]. The key here is *may*, not *must*. The `ForkJoinPool` uses some internal heuristics to decide when it is worthwhile to expand the current thread pool, and conversely when the "spare" worker threads should be destroyed. Another example of a runtime that uses this type of blocked-worker compensation includes with worker pools is X10 [31].

While this strategy may be a vast improvement over using one thread per `async`, it is still very easy to exhaust system resources if the given program executes many blocking tasks in parallel. For example, the Fibonacci program shown in listing 3.1—when run for sufficiently large n—could easily result in the creation of tens of thousands of threads because each non-leaf node in the recursive computation tree contains a blocking `finish` operation, resulting in the exact same situation that we described for the first case, where a new thread is created for each `async`.

Worker Threads with Helping

Another possible solution for enabling blocking tasks within our fixed-sized worker pool is to execute multiple tasks on the same thread. When a worker is suspended by a blocking call, it can decide to *help* to make progress by finding another task, and executing that task on top of the blocked task's stack frames. We call this extra work stealing behavior *global helping*, where *global* refers to the scope from which workers are allowed to steal tasks when helping. In other words, when using global helping, workers may execute any task that is ready to run.

Several runtimes use a fixed thread pool with some form of *helping* optimization. Intel Thread Building Blocks uses a restricted variant of *global helping* where only tasks deeper in the task tree than the blocked task are allowed to execute on the current worker [58]. OCR [5]—and, by proxy, `HClib` built on OCR [59]—use a fixed-sized worker pool with global helping. As per the conditions outlined in section 3.1.3, the absence of dependence cycles implies that this optimization does not introduce any new deadlocks.

Theorem 3.1. *Given that a program uses only the* `async` *and* `finish` *concurrency constructs, and that the program is free of data races, if that program is deadlock-free when run with one thread per* `async`*, then that program is also deadlock-free when run with global helping.*

Since all four of the conditions discussed in section 3.1.3 are necessary to enable deadlock, proving the absence of any one of those properties is sufficient to prove the absence of deadlocks. We will prove the absence of a *circular wait condition* (#4).

In the following proof we use the term *task-block* to refer to a "basic block" of work within the program. Such a basic block of code begins at a start-`async` or just after an end-`finish`; similarly, the basic blocks end at an end-`async` or at an end-`finish`. We reasoning about the synchronization among tasks during concurrent program's execution in terms of orderings of task-blocks.

Proof. As stated in the theorem, we assume that the original program is deadlock-free and data-race-free when executed using one thread per `async`. Since the global helping optimization is the only change in the execution model, it follows that any new deadlocks must be introduced as a result of this optimization. The global helping optimization only introduces a one type of dependence: When a running task blocks, it can be *buried* under another task that is executed by the current worker on the top of the current stack, creating a dependence from the *buried* task on the new *burying* task's completion. Or, said another way, the current worker *holds* the blocked task's execution context until the *burying* task completes execution, preventing any other worker from completing the *buried* task even when its original blocking condition is resolved. Thus, a buried-task $\xrightarrow{buried}$ burying-task dependence is introduced each time a worker *helps* rather than simply blocking.

In the `async`/`finish` execution model—ignoring task burying via global helping—all dependence edges point from older task-blocks to newer task-blocks.

Lemma 3.2. *If the completion of some task-block $Block_A$ depends on the completion of another task-block $Block_B$, then $Block_B$ must start after $Block_A$.*

$$\text{end}(Block_A) \xrightarrow{finish} \text{end}(Block_B) \implies \text{begin}(Block_A) \prec \text{begin}(Block_B)$$

This property trivially follows from the fact that all orderings in this execution model are introduced via the `finish` construct,[3] which orders all of the tasks nested within the `finish` strictly before all tasks specified after the `finish` in the program text—and the `finish`-block obviously must start before any of the tasks nested within that block. While the task-blocks following the end-`finish` in the program text also have an implied dependence, those task-blocks cannot begin until after the `finish` ends; therefore, those task-blocks cannot execute concurrently with the `finish`, and that lack of concurrency precludes all dependence edges of the form $\text{begin}(Block_X) \xrightarrow{finish} \text{end}(Block_Y)$ from being part of a circular wait condition.

Assume some $Block_X$ blocks and is subsequently *buried* by some other $Block_Y$. Since the two tasks are serialized within a single worker context, there is an implied ordering.

[3] Note that the absence of data races means that two tasks cannot communicate by spin-waiting on a volatile condition flag, or other similar behavior.

Lemma 3.3. *If $Block_Y$ buries $Block_X$ in the execution stack, then $Block_Y$ began executing after $Block_X$:* $Block_Y \xrightarrow{buried} Block_X \implies \text{begin}(Block_X) \prec \text{begin}(Block_Y)$
Or, conversely: $Block_Y \xrightarrow{buried} Block_X \implies \text{begin}(Block_Y) \nprec \text{begin}(Block_X)$

Finally, by combining lemma 3.3 with the contrapositive of lemma 3.2, we know that $\text{end}(Block_X) \xrightarrow{finish} \text{end}(Block_Y)$ would be a contradiction; i.e., following from the implied ordering of task-blocks, both types of dependences must flow in the same direction between any pair task-blocks.

The *buried* and *finish* dependence relations are obviously transitive, and the orderings on the right-hand sides of the implications in lemma 3.3 and lemma 3.2 also force all dependences of a single type between two task-blocks to flow in the same direction. Since all dependences between two task-blocks must flow in the same direction, we conclude that a circular wait condition cannot occur. ■

3.3.2 Issues Combining Global Helping with Blocking Constructs

Even a simple deviation from the core `async`/`finish` model introduces new possibilities for deadlocks. For example, the addition of *futures* into the model is sufficient to introduce possible deadlocks with global helping. This may be a surprising result since—in the absence of data races—the `async`/`finish` model plus *futures* is provably deadlock-free [51]; however, improper use of optimizations like global helping within the runtime can create new opportunities for deadlock.

Listing 3.2 illustrates a sample of HClib program snippet that uses the blocking future `wait` method, and as a result may deadlock when run with global helping. Note that the code snippet in listing 3.3, which uses `async_await` rather than the `wait` method, is functionally equivalent to that in listing 3.2. In other words, the `f0.wait()` operation yields identical behavior to nesting an `async_await` on `f0` within a `finish`.[4] Therefore,

[4] The fact that these two idioms are functionally equivalent does not imply that they have identical performance. In our implementations, the future's `wait` method is more efficient because the blocking operation is implemented internally without the need to create a new `finish` scope.

using the non-blocking `async_await` construct within a blocking `finish` scope can lead to the same deadlock scenarios as with the future's blocking `wait` method.

The potential for deadlocking is a serious issue to be aware of when contemplating applying the global helping optimization. This deadlocking issue is not limited to a specific subset of the HClib programming model. In fact, similar deadlock scenarios may also manifest in the current reference implementation of OCR when using the default distributed-memory configuration settings (see appendix A). The deadlock conditions of the global helping optimization can be more generally phrased as follows:

Theorem 3.4. *The global helping worker optimization, applied to a fixed set of worker contexts, can cause deadlocks in an otherwise correct program if and only if it is possible to have a dependence on a blocking $Task_X$ from another blocking $Task_Y$, and $Task_Y$ may start before $Task_X$.*

Proof. Three of the four necessary and sufficient conditions for deadlock are fairly simple to establish in terms of a worker thread's execution of tasks on its private stack. Only one worker at a time can execute a task, establishing *mutual exclusion* (#1). As previously discussed, a worker holds a task in its execution stack when it blocks, and if that task is buried by a newer task then the worker *holds* the buried task while *waiting* for the newer task to complete (#2). Finally, there is no mechanism for one worker thread to forcefully take over execution of a task that has begun execution on another thread. The ability to capture references to stack-allocated data prevents migration of partially-executed tasks between two thread stacks in general, since the captured references would be invalidated. Therefore, there is *no preemption* (#3).

The remainder of this proof focuses on establishing the possibility of *circular wait conditions* (#4). The reverse-ordering of the tasks' dependence relationship relative to their potential execution order is the key condition that allows a dependence cycle to form via the stack in a worker's execution context. As stated in the theorem, we assume that there are no deadlock in the absence of the global helping optimization, which means any new deadlocks must be introduced via buried-task dependences in the worker stacks.

```
// This code executes on Worker-A.
auto f0 = hclib::async_future([]() {
    HCLIB_FINISH {
        hclib::async([]() { /* . . . */ });
        // . . .
    }
});
// The future-task for f0 is stolen by Worker-C. The async task nested within f0
// is stolen by Worker-B. Worker-C blocks awaiting completion of its stolen async,
// and starts looking for more work to execute while it waits.
hclib::async([]() { f0.wait(); });
// Worker-C steals the above async task, and blocks on f0.wait(). There is now
// a dependence cycle between this blocked task and the task for f0
// (buried deeper in Worker-C's stack), resulting in deadlock.
```

Listing 3.2: Simple HClib sample code using *futures*. This program can deadlock when using *global helping*. The comments outline one possible deadlock scenario involving three workers. This code is functionally equivalent to that in listing 3.3.

```
// This code executes on Worker-A.
auto f0 = hclib::async_future([]() {
    HCLIB_FINISH {
        hclib::async([]() { /* . . . */ });
        // . . .
    }
});
// The future-task for f0 is stolen by Worker-C. The async task nested within f0
// is stolen by Worker-B. Worker-C blocks awaiting completion of its stolen async,
// and starts looking for more work to execute while it waits.
hclib::async([]() {
    HCLIB_FINISH {
        hclib::async_await([]() { /* . . . */ }, f0);
    }
});
// Worker-C steals the above async-task, and blocks awaiting the completion of the
// async_await task. There is now a dependence cycle between this blocked task and
// the task for f0 (buried deeper in Worker-C's stack), resulting in deadlock.
```

Listing 3.3: Simple HClib sample code using *futures* and *data-driven tasks*. This program can deadlock when using *global helping*. The comments outline one possible deadlock scenario involving three workers. This code is functionally equivalent to that in listing 3.2.

Lemma 3.3 establishes that the *buried* dependencies among task-blocks all flow in the same direction; specifically, from earlier-started tasks to later-started tasks. Thus, a non-*buried*-dependence edge must exist from a later-started task to an earlier-started task in order to form a dependence cycle and create deadlock. Both tasks must be held in a worker's execution stack in order for a deadlock to form, which implies that both tasks must be blocking tasks (otherwise there would be no *hold and wait*, since one of the tasks in the cycle would eventually complete, breaking the cycle and thus breaking the deadlock). Therefore, if the conditions described in theorem 3.4 are present, then a cycle can form and deadlock can occur. Conversely, if either the dependence is missing, or one of the tasks does not block, or the tasks' order flows with the dependence, then a cycle cannot form and a deadlock cannot occur. ■

3.3.3 Program Compatibility with Global Helping

As described in theorem 3.4, programs that can deadlock with the global helping optimization all involve a *blocking* task that depends on an earlier *blocking* task. Therefore, if the application programmer avoids all blocking constructs by writing all HClib code in non-blocking style, then those programs can be safely and efficiently executed using the Master/Helper scheduling strategy. Note that a single top-level `finish` construct would not be problematic since no code within the `finish` can depend on that `finish`.

We can generalize this observation to define a set of intuitive rules for identifying application logic that may lead to a deadlock when using the global helping optimization. Assume we inspect two events in an application:

1. E_C: A *consumer* event that completes only after synchronizing with an event in some other task(s). E.g., a `finish` scope that blocks awaiting for completion of all sub-tasks, or a blocking `get()` call requesting a future's value.
2. E_P: A *producer* event whose completion may be awaited by other tasks. E.g., the completion of an `async`, or the satisfaction of a promise with a value.

If $E_C \prec E_P$, then the program will deadlock regardless of how the tasks are scheduled; thus, this case has no potential to introduce new deadlocks when using the global helping optimization. If $E_P \prec E_C$, then the consuming-task has no need to block and thus no new deadlocks will be introduced by global helping. However, if $E_C \parallel E_P$,[5] then a deadlock can be introduced via global helping if the task containing E_P executes first, but blocks before E_P, and is subsequently *buried* by the task containing E_C. In other words, the global helping optimization can introduce new deadlocks into an otherwise correct program if a *consumer* task (the task containing some *consumer* event E_C) and a corresponding *producer* task (the task containing some *producer* event E_P) are unordered, and the *producer* task contains another blocking call before producing the value awaited by the *consumer*. Note that this additional blocking event in the producer task, cannot be ordered after the start of the consuming task, otherwise the producer would never be *buried* by the consumer.

3.4 Alternative Strategies for Scheduling Blocking Tasks

Although theorem 3.4 provides a clear definition of how new deadlocks are introduced by the global helping optimization, verifying that a program does not contain the condition described in may be difficult in practice. Constructing a proof of correctness—even an informal proof—may not be feasible in many real-world applications. This would be especially difficult for large software projects that are maintained by multiple developers, where changes made by two different developers in separate places may together result in the scenario from theorem 3.4. Therefore, it becomes necessary to find a way to eliminate any chance of creating a deadlock via blocking calls. We focus on such alternative strategies for the rest of this section.

3.4.1 Requirements for an Alternative Strategy

We now define what an alternative tasking strategy must provide in order to avoid the introduction of runtime-internal deadlocks (i.e., deadlocks not caused directly by errors in

[5] Note that $x \parallel y$ means that elements x and y are incomparable, i.e., $x \npreceq y \wedge y \npreceq x$ [60].

the user's code). Referring back to the necessary conditions for deadlock in section 3.1.3, the only condition that an be reasonably addressed within the runtime is #2 (hold and wait). We eliminate this condition in our runtime by providing a means for a worker to release a blocked task back to the scheduler, rather than *holding* the blocked task, executing another (potentially blocking) task on the same stack, and waiting for that task to complete before resuming the *buried* task.

In order to suspend the blocked task and return control to the scheduler, we must have a way to capture the task's current state, and also have a way for the scheduler to resume that task (using the saved state) sometime after the blocking condition is satisfied. One obvious solution would be to capture the *continuation* at the point where the task blocks, and use the continuation to restore the task after we detect that it is no longer blocked. However, full continuations (whether delimited or undelimited) are much more powerful than what we need in this situation. Additionally, most programming languages, including C and C++, lack native support for capturing continuations.

Interestingly, the requirements of our blocked tasks correspond exactly with the functionality provided by *semi-coroutines*: a *suspend* function for saving the local state and returning control back to the scheduler, and a *resume* function to restore the state and continue the computation from the previous suspend point.[6] Semi-coroutines are a limited type of continuation. More specifically, a suspended semi-coroutine are a limited type of one-shot continuation. This distinction means that when a suspended semi-coroutine is resumed, the previously-saved state is invalidated. The fact that the saved-state can be invalidated by the resume operation makes it easier for us to implement the local state capture. For example, a typical *thread* satisfies the requirements: when a thread is suspended its local state is preserved, and when a thread is resumed that local state is restored, but that state is immediately altered by continuing the thread's execution.

[6] A full coroutine differs from a semi-coroutine in that it also supports a *transfer* function, which allows the coroutine to explicitly pass control to another coroutine [61], whereas semi-coroutines can only yield control back to the scheduler [62]. Semi-coroutines are also commonly known as *generators* in Python and other programming languages [63].

Similarly, *fibers* (lightweight cooperative threads) are also sufficiently powerful to implement semi-coroutines. Note that both threads and fibers are limited types of one-shot undelimited continuations.

As discussed in section 3.3.3, it is also possible to sidestep the potential deadlock issues in the runtime by instead transforming the application code to avoid problematic constructs. In addition to eliminating the hold-and-wait condition in the runtime code, we also consider the case where the application is manually rewritten using only non-blocking constructs, and can thus safely execute using the global-helping optimization.

3.4.2 Our Selected Strategies

We present four viable tasking strategies for the Habanero-C programming model, allowing us to avoid runtime-induced deadlocks. Our strategies are as follows:

1. **Compensation with Threads**: Create a new OS thread to compensate each time a worker thread blocks. This is basically equivalent to the strategy described as "worker pools with compensation for blocking" in section 3.3.1; however, we maintain a fixed number of active threads and use work-stealing among the active threads, whereas most implementations of this strategy use work-sharing. We do this specifically for the purpose of providing a more analogous baseline for our other strategies (which all use work-stealing).
2. **Compensation with Fibers**: Similar to the thread-compensating strategy, except that rather than using OS threads we use *fibers* (i.e., lightweight, cooperative, user-space threads). Our fiber implementation is build on code from Boost.Context [64].
3. **Transformation to Semi-coroutines**: Requires HC compiler support. All application code containing blocking constructs is automatically transformed into *continuation-passing style* (CPS). There are some restrictions placed on application code to ensure the validity of the CPS transformation. For example, arrays cannot be stack-allocated in affected functions.

4. **Fully Non-blocking Rewrite**: Requires that the application code be written in a non-blocking style. This entails avoiding `finish` and blocking future calls, instead using promises and futures with data-driven tasks for all synchronization in the application.

Additionally, we present two variant strategies based on an optimization that we call *finish-helping*, which is a restriction on the global-helping strategy where only sub-tasks belonging to the finish scope that caused a task to block are allowed to *bury* that task. The variant strategies based on the finish-helping optimization are as follows:

1. **Compensation with Threads + Finish-Helping**: This is the *compensation with threads* strategy with one additional optimization. When blocking at the end of a finish scope, we first peek at the top of our local task deque to see if that task belongs to the current finish scope. If so, we execute that task on the current worker's stack in the same manner as global-helping.
2. **Compensation with Fibers + Finish-Helping**: This is the *compensation with threads* strategy with the addition of the finish-helping optimization.

Unlike the problematic global-helping optimization discussed earlier, the finish-helping optimization that we apply in these two strategy variants is guaranteed not to introduce new deadlocks into the execution of an otherwise correct Habanero-C programs.

Theorem 3.5. *The finish-helping optimization cannot introduce a new deadlock scenario into a race-free Habanero-C program.*

Proof. Lemma 3.3 establishes that the *buried* dependencies among task-blocks all flow in the same direction: from earlier-started tasks to later-started tasks. If some $Block_A$ is buried by $Block_B$, then $Block_A \xrightarrow{finish} Block_B$ by definition of the finish-helping optimization; therefore, the *burying* dependence edge introduced via finish-helping is redundant because the existing *finish* edge already expresses the same dependence. Thus, we conclude that all *burying* dependences introduced by finish-helping are redundant with existing *finish* dependence edges. Since there are no new (non-redundant) dependence edges introduced by this optimization, finish-helping cannot introduce new deadlocks. ■

3.5 Evaluation of Selected Strategies

We now evaluate our selected tasking strategies to assess the strengths and weaknesses of each approach in a range of potential applications. First, we include a performance evaluation comparing all of the strategies, using the global-helping strategy on a fixed worker-thread pool as the performance baseline (despite the fact that it is not safe in the general case). Second, we present the tradeoffs of each strategy in terms of various usability properties (e.g., programmability and ease of debugging). Third, we analyze the implications of each strategy with regard to resilience, and rank the strategies accordingly.

Note that we don't expect to have a single, clear-cut winner among the strategies that we are evaluating. Rather, we expect the result to be a classification of the strategies based on their strengths and weaknesses, such that the reader can choose the best solution for the particular properties of their application.

3.5.1 Benchmarks

This evaluation includes a variety of performance benchmarks that test different representative use cases of the Habanero-C concurrency API. Our benchmarks are briefly described in the following paragraphs, but the full source code is also available online.[7]

Fibonacci A naïve parallel implementation of the recursive Fibonacci function, structurally identical to the function shown in listing 3.1. We consider this benchmark to represent the worst-case scenario for `async`/`finish`-style computation with regard to blocking, since a huge number of blocking `finish` scopes are created, but the work-to-`finish` ratio is extremely low (just a couple of function calls and integer additions per `finish` scope). When computing $Fib(35)$—which is the target value we use for our benchmarks—the

[7] https://github.com/DaoWen/hc-blocking-benchmarks

algorithm creates over 14 million `finish` scopes.[8] Unlike our other benchmarks, this application was written exclusively using the C API, meaning that we don't use C++11 closures in this benchmark. The explicit nature of the C API makes it easier to pinpoint the sources of overhead, which is especially informative in this worst-case usage of the blocking `finish` construct.

Cilk-sort A parallel sorting algorithm, based on mergesort, ported from the standard Cilk benchmarks [49]. The algorithm follows a traditional recursive divide-and-conquer structure, with multiple `finish` scopes at each level of recursive sorting to ensure proper synchronization among the recursive calls (i.e., the `async` tasks that sort smaller sub-ranges of the array). For our benchmark, we sort an array of 9 million deterministically-generated random 64-bit integer values (i.e., the random number generator's seed is constant). This benchmark is representative of a typical divide-and-conquer algorithm with a non-trivial workload for each task (this is in contrast with our Fibonacci benchmark, where the work for each `async` is almost nil).

Cholesky This parallel implementation of the Cholesky factorization kernel uses a tiled, iterative algorithm with barrier-like utilization of `finish` scopes for synchronization. The *sequential* Cholesky step in each iteration is followed by a `finish` scope for computing new values for the *trisolve* tiles, and then another after for new values for the remaining *update* tiles. For our benchmarks, we use a 500×500 matrix with tiles of size 5×5. This benchmark is representative of a typical OpenMP-style iterative computation (using barriers for synchronization between computation steps) being translated directly to the Habanero-C programming model.

Needleman-Wunsch A gene sequence alignment kernel using the standard Needleman-Wunsch algorithm, which performs an edit-distance computation using dynamic program-

[8] The general formula for the number of `finish` scopes created when computing $Fib(n)$ is simply $Fib(n+1)$; therefore, computing $Fib(35)$ creates $Fib(36)$ or 14,930,352 `finish` scopes.

ming. To allow maximal wavefront parallelism, we use futures to manage task dependences on input score tiles. For our benchmarks, we use input sequences of length 18,560 and 19,200, and tiles of size 232×240. The result is a dynamic-programming scoring matrix composed of 80×80 tiles total. This kernel is representative of applications that use future-based synchronization with non-trivial task workloads.

Quicksort A standard recursive parallel quicksort implementation. Quicksort is interesting in that it only requires a single point of synchronization at the very top level for a correct parallel implementation; i.e., it is *embarrassingly parallel* since there is no synchronization required among the parallel tasks. Since there is only a single top-level `finish` scope in this program, it acts as a sanity-check to demonstrate that our strategies for handling blocking constructs do not introduce excessive overhead when no blocking constructs (or just a few) are used in an application.

Unbalanced Tree Search (UTS) A port of the standard UTS benchmark [23] into the Habanero-C programming model. This app creates a random, unbalanced tree, and performs a parallel search. For our benchmarks, we use the default random tree configuration with a seed value of zero. This benchmark is similar to our Quicksort benchmark with regard to a general lack of fine-grained synchronization; however, the greater complexity of the code relative to Quicksort accentuates some differences between our HC and HClib implementations (described in the performance analysis discussion).

3.5.2 Experimental Setup

All of the experiments in this section were run on a dual-socket 18-core Haswell system (36 cores total) at 2.30GHz, running RHEL7 with 128GB DDR3 memory. To avoid cross-socket communication overheads, we configured the Habanero-C runtime to use only 18 workers, and thread-pinning is enabled by default. To reduce bottlenecks in the default allocator, all benchmarks were run using TCmalloc [65], built from the gperftools v2.6.1 source release. We also disabled TurboBoost in the OS to improve performance predictability. Finally, we manually set the stack size to 256KiB via `ulimit`—which is the same as our default fiber stack size—in order to reduce the overhead of allocating new kernel threads. We believe using a consistent stack size provides a better comparison between using kernel threads and user-level fibers.

However, there was one case where this reduced stack size was not sufficient: The global-helping version of the Fibonacci benchmark required a larger stack,[9] so in that case we left the stack size as the system default (8MiB). Since the global-helping variant runs with a fixed-sized thread pool, no new stacks are allocated during the benchmark timing phase. The larger stack sizes likely slightly influence the runtime startup overhead, but this has no impact on our benchmark timing results since we only measure the elapsed time for each benchmark application's computation kernel, excluding time for runtime start-up, file I/O, etc. (This is common practice with HPC benchmarks, since fine-tuning the computation-time overheads of runtime constructs usually has a much higher impact on real-world long-running compute kernels than the time used for runtime startup.)

All reported times are the mean of 30 runs of each configuration, and the error ranges represent a 95% confidence interval. For simplicity in analyzing and rendering the results, all execution times are reported as *slowdowns* relative to the global-helping version of each benchmark (which runs in a fixed thread pool, and is potentially unsafe in the general case). All benchmarks were compiled using GCC 6.3.0 with `-O3` for optimization, with commit `c6d392b7` from the HClib repository.[10]

[9] Since the global-helping strategy can steal and execute *any* ready `async` task, the worst-case stack depth in an application is the sum of call-stack depth of all `async` tasks that can execute concurrently and block. This is usually not an issue in practical applications since the number of concurrently blocked tasks is usually small, and the probability that a large number will be stolen and executed by the same worker thread is also low; however, this problem has also come up in OCR, where excessive numbers of tasked blocked on inter-node communications may result in a stack overflow (see https://xstack.exascale-tech.com/redmine/issues/657).

[10] https://github.com/habanero-rice/hclib/tree/vrvilo2017

3.5.3 Results and Analysis

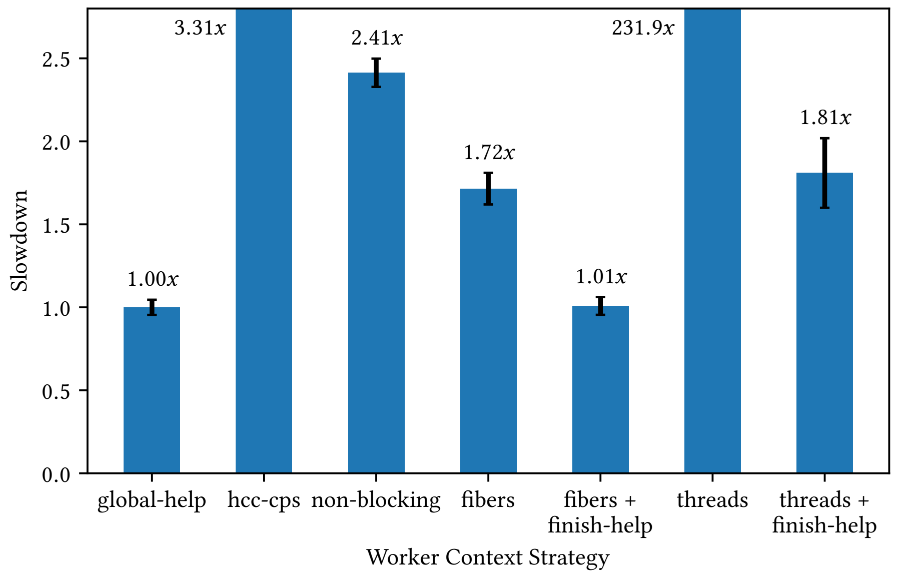
The results of our main experiments are summarized in figures 3.1(a) to 3.1(g). Note that the *global-help* label refers to our performance baseline (hence it will always have a slowdown of $1.00x$). The *hcc-cps* label refers to the HC (compiler-transformed) versions of each benchmark, and the *non-blocking* label refers to our hand-transformed versions of each benchmark that use only non-blocking constructs. The other labels should be self explanatory.

Fibonacci

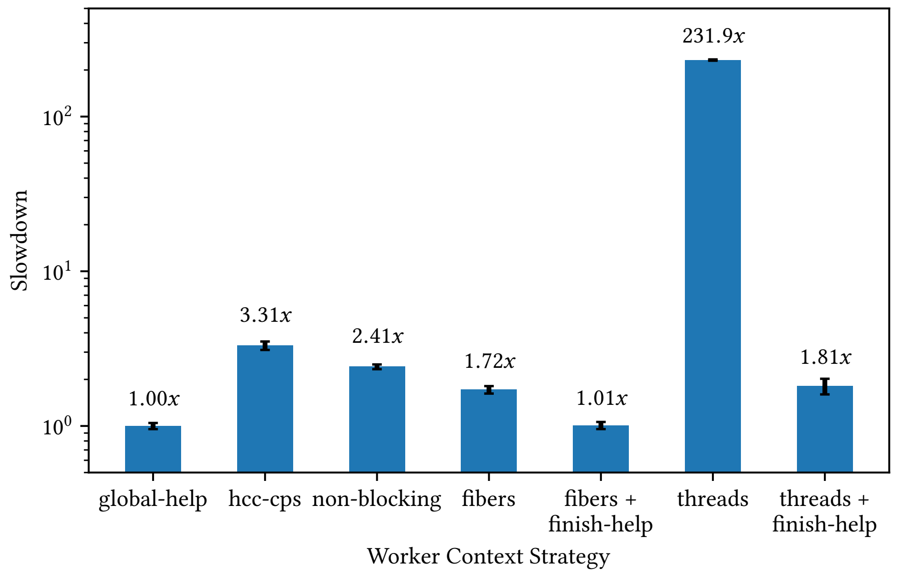
Figure 3.1(a) shows the slowdown results for our Fibonacci benchmark. Since several of the slowdowns were very high (exceeding the y-axis maximum for figure 3.1(a)), we also included a second version of the graph, figure 3.1(b), which uses log-scale on the y-axis to better accommodate all of the results. As explained in the benchmark descriptions, the Fibonacci benchmark represents an extreme scenario for blocking task overhead since there is very little computation in each task to offset the runtime overheads; therefore, we would expect the performance with each of our selected strategies to represent the worst-case performance of the overhead for the Habanero-C blocking `finish` construct. Using blocking calls on futures should have similar overhead to the `finish` construct; however, the *finish-helping* optimization obviously doesn't apply when using blocking futures rather than `finish` scopes.

The Fibonacci variant using new threads to compensate for blocked workers performs far worse than the other strategy variants of this benchmark, with a slowdown of more than $200x$ over the baseline. This result is unsurprising since invoking OS kernel services, such as creating new kernel threads, typically have a non-trivial overhead. This variant of the Fibonacci benchmark creates over *14 million* threads throughout the course of the computation, which is obviously a severe performance bottleneck.

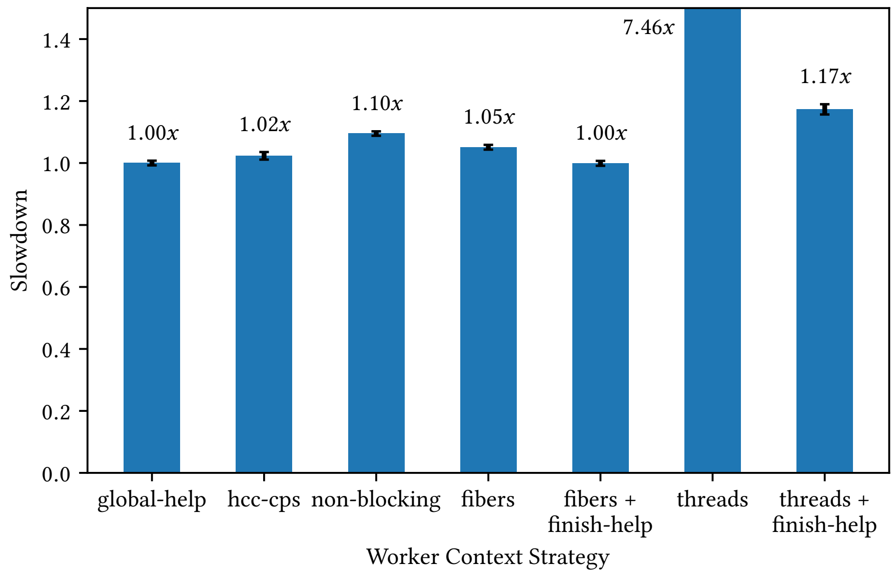
In contrast, the variant using new fibers to compensate for blocked workers only yields a $1.72x$ slowdown over the baseline. Fiber creation is much more lightweight than thread



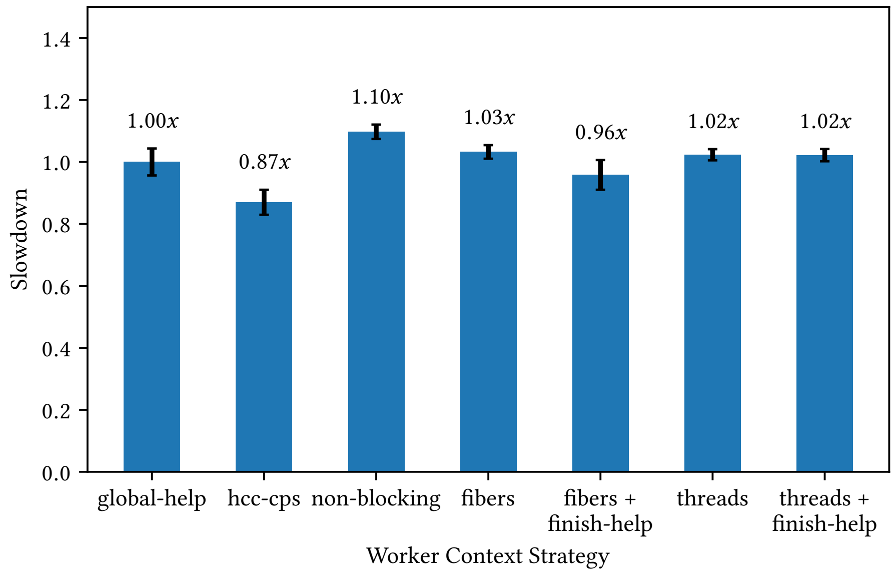
(a) Fibonacci



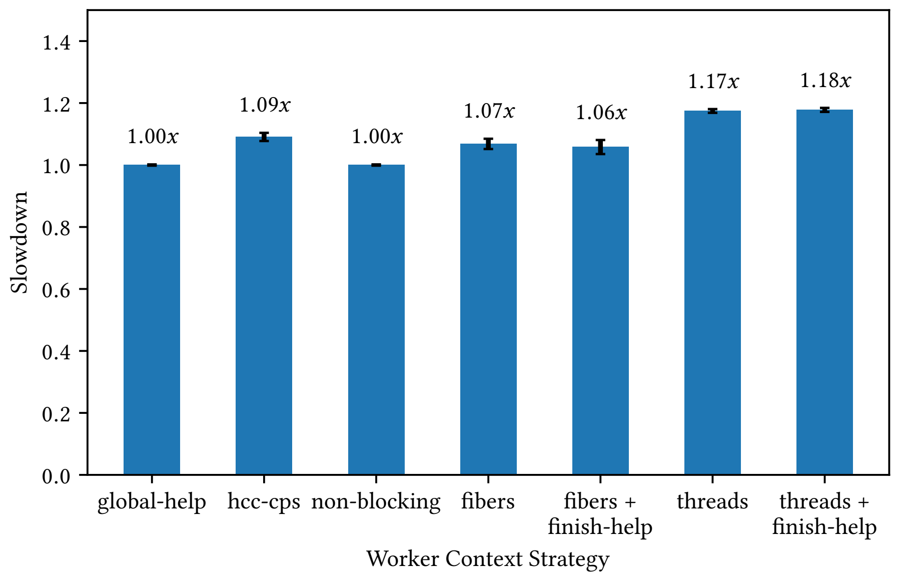
(b) Fibonacci (log scale)



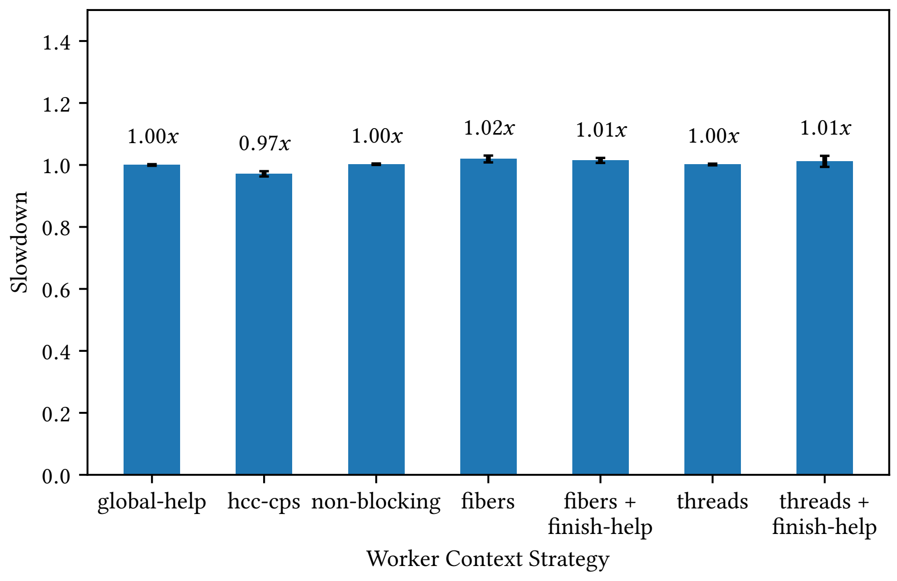
(c) Cilk-sort



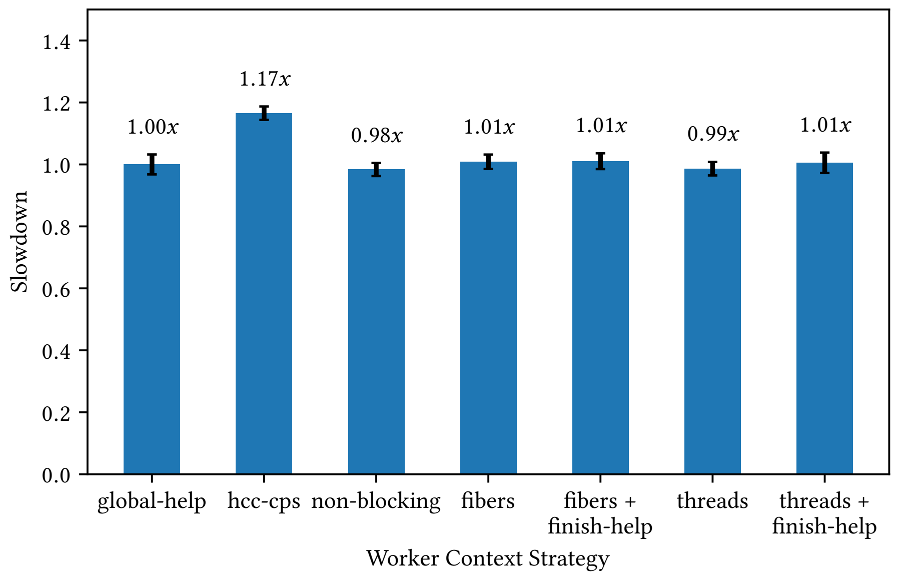
(d) Cholesky



(e) Needleman-Wunsch



(f) Quicksort



(g) Unbalanced Tree Search (UTS)

Figure 3.1: Execution overheads for our worker context strategies with five benchmarks. Slowdowns are normalized to the potentially-unsafe *global-helping* strategy's mean execution times for each benchmark. The reported times are the average of 30 runs, and error bars indicate a 95% confidence interval.

creation since the fibers are fully managed in user-space, and thus do require any services from the OS kernel. Despite needing to create over 14 million fibers for this benchmark (the same as the thread count in the thread-compensating variant), the overhead is still under $2x$. This is in part due to the fine-tuned performance of the Boost.Context library which is used to support fibers in HClib, and in part due to the use of TCmalloc for fast allocation of each new fiber's 256KiB execution stack.

Perhaps a more surprising result is that the compiler-transformed HC variant of Fibonacci had a slowdown of $3.31x$, which is worse than the variant using fibers. One might expect that a compiler toolchain that has been optimized for the Habanero-C programming paradigm would outperform a pure library-based solution; however, this is an instance of the classic time–memory tradeoff. For the Fibonacci benchmark with 16 workers, the HC variant has a maximum memory size (measured with `/usr/bin/time -v`) of about 9KiB, whereas the HC fibers variant uses over 900KiB. Whereas the HC compiler transformation copies a small portion of the execution stack to preserve the state of the blocking task (i.e., it saves the *continuation*), the HClib fibers variant simply allocates a whole new execution stack each time one of the workers blocks, avoiding the need to save or restore the local stack frame state, but at the cost of larger up-front allocations. Using a high-performance allocation library such as TCmalloc eliminates a large amount of the overhead that would normally be associated with these extra fiber stack allocations, making the creation of each new fiber relatively cheap from a computation perspective.

The *finish-helping* optimization significantly improves the performance for both the fibers and threads strategies of blocked-worker compensation. Since the Fibonacci benchmark has a significant number of tasks nested within the upper-level `finish` scopes, each worker begins work on one of the higher-level finish scopes early in the computation, and the *finish-helping* optimization allows the workers to safely execute tasks in almost the same manner as the *global-helping* baseline, resulting in very good performance. At the end of the computation, work stealing may cause `finish` scopes to block when other workers have stolen the remaining tasks in that scope, in which case a new fiber or thread must be created to avoid blocking the current worker from continuing computation. Since fiber creation is relatively cheap, and relatively few new fibers are created in this scenario, the fibers variant with finish-helping has almost identical performance to the baseline. Due to the higher overhead of kernel thread creation, the threads variant with finish-helping is a bit slower, with a $1.81x$ slowdown versus the baseline; however, this is a very significant improvement when compared with its $231.9x$ slowdown.

The only Fibonacci variant we have not discussed so far is the non-blocking version of the benchmark. While the baseline, threads and fibers variants all used identical HClib source code (i.e., the only difference was which version of the runtime was used), the non-blocking variant is written completely differently to avoid using the blocking `finish` construct. Instead, each Fibonacci task creates two *promises*, which are passed to its two sub-tasks as destinations for their result values, and then it creates an `async_await` task that depends on the two sub-task results, sums the two sub-task results, and puts the result into its parent-task's promise. The extra allocations for tasks, promises and argument structures add additional overhead compared to the other HClib versions—aside from the very slow thread-compensating variant. We will discuss the non-blocking performance in more depth later in this section.

Cilk-sort

Figure 3.1(c) shows the relative performance of our Habanero-C variants of the Cilk-sort benchmark. Since the work/task ratio is much higher for this benchmark than for Fibonacci, the overheads for the different blocking-compensating strategies are not as high as they were for Fibonacci. The HClib fibers variant and the HC compiler-transformed variant only add a few percent overhead compared to the baseline, and the fibers variant with finish-helping has no measurable overhead compared to the global-helping baseline. The HClib non-blocking variant and threads with finish-helping variant have a more noticeable overhead ($1.10x$ and $1.17x$, respectively), and these variants have more overhead for the same reasons as the corresponding variants of the Fibonacci benchmark. Lastly, the HClib threads variant has a very obvious overhead of $7.46x$. While that's not has extreme as the threads variant of the Fibonacci benchmark, the extra overhead of creating new kernel threads each time a worker blocks still becomes a major performance bottleneck in this benchmark despite its non-trivial work/task ratio and the fewer total number of `finish` scopes in the program.

Cholesky

Since the Cholesky benchmark only uses one `finish` scope per iteration, the blocking-compensating strategy overheads are not obvious for this benchmark. The results are shown in figure 3.1(d). The higher variance seen in some cases (e.g., fibers + finish-help) can be attributed to work imbalance when scheduling tasks. The variant using fibers with finish-helping appears to be particularly unpredictable, with its average execution time actually slightly below the baseline ($0.96x$); however, the error ranges for this variant and the baseline overlap, implying that the baseline still performed better on some runs. We see that the non-blocking variant has significantly more overhead than the other variants due to the necessary restructuring to avoid the blocking `finish` scopes. The HC compiler-transformed variant of Cholesky is significantly faster than the baseline ($0.87x$ slowdown). We attribute this performance difference to the overheads introduced by the C++11 closures used in this benchmark. We look at the closure-induced overhead in more detail later in this section, specifically in figure 3.2 and the corresponding discussion. The results for the other variants of our Cholesky benchmark are fairly unremarkable.

Needleman-Wunsch

The slowdown results for the Needleman-Wunsch benchmark variants are show in figure 3.1(e). In the variants that are safe for general blocking (i.e., all variants except *non-blocking* and the *global-helping* baseline), the three input tiles for each `async` task are synchronized and read using the blocking `future.get()` operation; however, since this use of blocking futures almost always causes a deadlock when using the *global-helping* strategy, we used the non-blocking variant for our baseline, hence the identical performance for those two variants. The non-blocking variant of this benchmark performs better than the blocking variants because we use the `async_await` construct to avoid scheduling a task until all of its inputs are available, whereas the blocking variants eagerly-schedule tasks and block if an input is not yet available. The use of `async_await` is more idiomatic in the Habanero-C paradigm; however, we decided to use `future.get()` calls in the other variants so that (1) blocking future operations are represented in our benchmarks, and (2) because programmers familiar with the C++11 futures library might naïvely compose code like this if they're not yet familiar with the Habanero-C `async_await` construct.

Since this benchmark does not create `async` tasks nested inside multiple `finish` scopes, we would expect the *finish-helping* optimization to have no effect on the performance, which is exactly what we observe with the *fibers* and *threads* variants. The tasks in this benchmark are only created once the "upper" input becomes available, which means that many of the tasks are not scheduled to run until all of the inputs are already available, meaning that many of the blocking `future.get()` operations do not actually have to suspend the current worker since the input data is already available. However, some of the tasks *are* scheduled early enough to require blocking, hence the difference in slowdown between the fibers and threads variants ($1.07x$ and $1.17x$, respectively). Finally, the HC compiler-transformed variant again exhibits very similar performance to the fibers variant, with a slowdown of $1.09x$.

Quicksort

Figure 3.1(f) shows the performance results for the Quicksort benchmark variants. Parallel quicksort only requires one top-level `finish` to properly synchronize among all of the divide-and-conquer `async` tasks. This means that the application is *embarrassingly parallel*, and that our blocking construct overheads should not be evident since there is only one blocking call and a significant amount of computation to offset it. As expected, none of the quicksort variants show a significant deviation from the baseline performance.

Unbalanced Tree Search (UTS)

Figure 3.1(g) shows the performance results for the UTS benchmark variants. As with quicksort, the UTS program only requires a single top-level `finish` scope, meaning that the blocking task overheads should not be evident. While most of the UTS benchmark variants do not exhibit a significant slowdown compared to the baseline performance, the HC compiler-transformed variant averaged a $1.17x$ slowdown. This slowdown isn't caused by blocking tasks, but rather by a limitation of the Habanero-C compiler (HCC). The compiler does not allow `async` tasks to capture stack-allocated arrays (since the whole array must be copied by-value). This limitation forced us to heap-allocate a fixed-sized array that was stack-allocated in all of the HClib variants of UTS. Since this additional call to `malloc` is the only notable difference between the HC code for UTS and the HClib version, we assume it is the source of the extra overhead.

Sources of Overheads in Fibonacci

Figure 3.2 shows some further analysis of the sources of overheads for a subset of our strategies on the Fibonacci benchmark (which accentuates the runtime overheads due to the low work/task ratio). Figure 3.2 uses the global-helping optimization as a baseline. Since the Fibonacci application uses only `async` and `finish`, global-helping is a valid and efficient tasking strategy. Note that unlike the previous results, we did not disable TurboBoost for these measurements; however, comparing these results with those in figure 3.1(a) shows that although the variance is higher in these results, the mean relative performance is very similar. We can draw several conclusions from these results about the sources of overhead in the Fibers and Non-blocking variants.

First, even in this pathological example, compensating with Fibers when blocking results in less than a $1.8x$ slowdown, which seems very reasonable since our baseline is a potentially unsafe optimization in the general case. Choosing a fiber stack size of 4KiB (the default in the Qthreads runtime [46]) or 256KiB (the default in HClib) does not have a

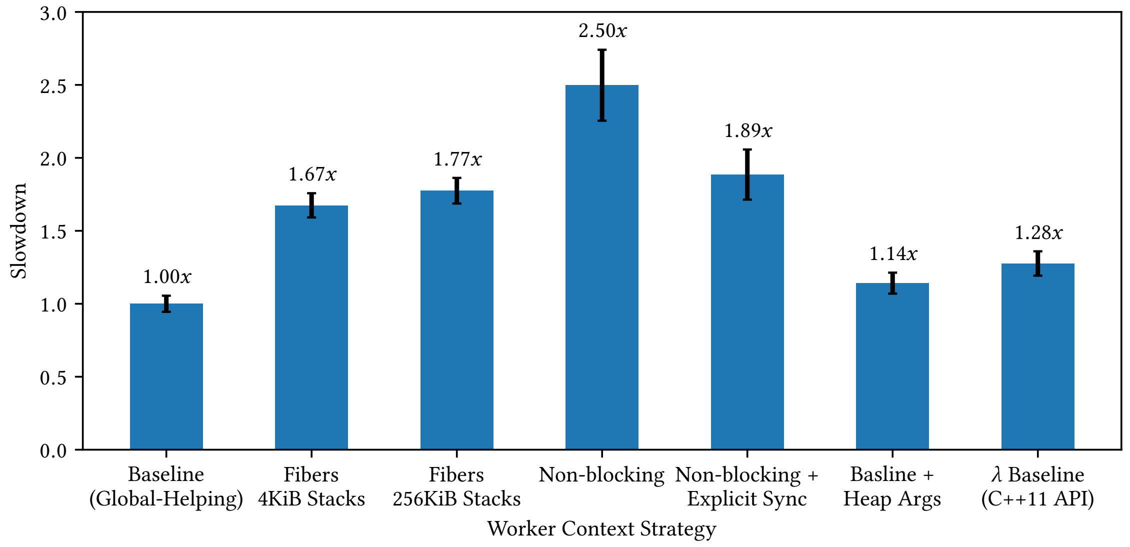


Figure 3.2: Comparing the global-helping baseline for HClib Fibonacci against fibers and non-blocking variants to highlight possible sources of overhead, specifically stack size for fibers. For non-blocking we show both `async`/`finish` bookkeeping overheads and overhead from the heap-allocation of `async` arguments. TurboBoost was enabled for these measurements.

very large impact even in this pathological scenario; therefore, we would not expect it to significantly influence performance in a more realistic application.

The Non-blocking variant has a relatively high overhead of $2.5x$. However, we can attribute a large chunk of that overhead to implicit synchronization of all of the `async` tasks with the top-level `finish` scope in the benchmark (all Habanero-C applications have an implicit top-level `finish` scope to ensure all tasks are finished before tearing down the runtime). However, this synchronization is completely redundant in this variant of the benchmark since all tasks are explicitly synchronized via promises and the `async_await` construct. Since all of our synchronization is explicit, we can specify that we do not need implicit synchronization on our `async` tasks[11] to eliminate that overhead. We see that using explicitly-synchronized `async`s drops the overhead from $2.5x$ to about $1.9x$.

[11] An `async` can be marked as explicitly-synchronized in HClib by using the `PROP_EXPLICIT_SYNC` flag.

Another source of overhead is the heap-allocation of arguments to the asyncs. Since the parent async does not block awaiting the children, it is not safe to allocate the child-tasks' arguments on the parent's stack; therefore, all task arguments must be allocated in the heap. To demonstrate the overhead for the heap-allocated arguments, we rewrote the baseline version running with global-helping to also heap-allocate all of its task arguments. The fact that this transformed version runs $1.14x$ slower than the baseline accounts for a fraction of the non-blocking variant's overhead, implying that without the heap-allocated arguments we could expect the explicitly-synchronized non-blocking version to have only a $1.66x$ slowdown compared to the global-helping baseline. We suspect that the extra allocations for creating promise objects also accounts for part of the remaining performance delta. However, the additional heap allocations (both for the task arguments and for the synchronization objects) are a necessary tradeoff for eliminating the long-lived tasks and their corresponding long-lived execution stacks.

The final bar in the graph shows the overhead introduced when we use the HClib C++11 API to implement the Fibonacci benchmark, using the same global-helping version of the runtime as the baseline. In this variant, we use the C++11 λ (lambda function) construct to automatically build closures for our async tasks, making the code simpler to write and to understand; however, we see that this causes a $1.28x$ slowdown compared to the pure C API baseline. While this overhead is less than that introduced by fibers, it is still significant, which is why we used the pure C API for most of the fine-grained overhead analysis on the Fibonacci benchmark.

Comparison with Other Runtimes

To demonstrate that HC and HClib provide a reasonable performance baseline, we compared the performance of our Fibonacci benchmark running in our Habanero-C variants to versions ported onto Cilk Plus, OpenMP and HPX-5. We use HPX-5 [66] version 4.1.0,[12]

[12] We chose to use the HPX-5 implementation of HPX since it uses a pure C API like we use in our baseline Fibonacci benchmark. This lets us avoid questions about language-construct induced overheads like those

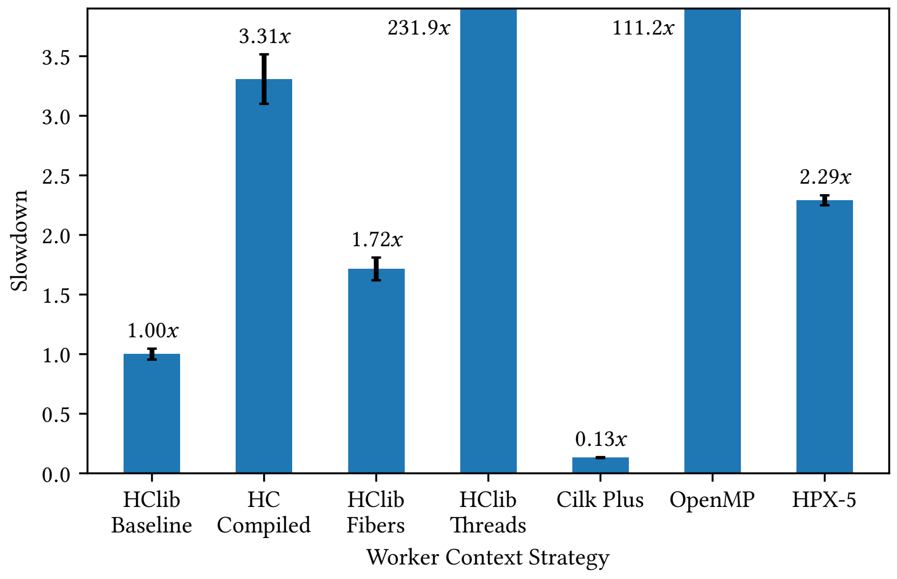
configured to link with libc `malloc`, which we override with TCmalloc by using `LD_PRELOAD`. We use the versions of OpenMP and Cilk Plus shipped with GCC 6.3.0, similarly using TCmalloc for allocation. We again use the HClib runtime with global-helping on a fixed worker thread pool as our performance baseline. Figures 3.3(a) to 3.3(b) summarize the results.

The first four bars show the results of our HClib version using global-helping, the compiled HC version, the HClib version compensating with fibers, and the HClib version compensating with fibers. These results are the same shown in figure 3.1(a). We include these as points of reference for comparison of the Habanero-C implementations with the other tasking runtimes that were tested.

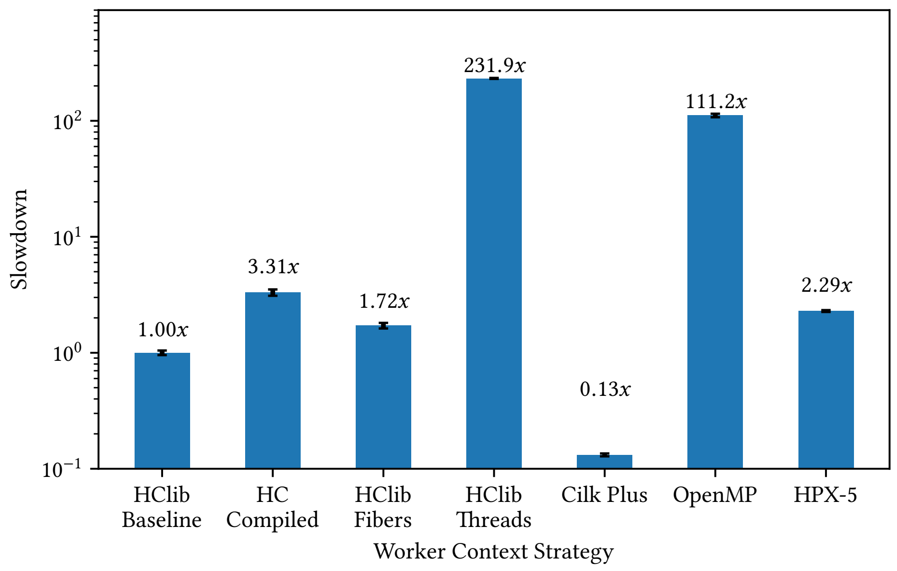
The Cilk Plus runtime is close to $8x$ faster than our HClib baseline. One of the key features of Cilk is its lightweight continuation support with its work-first scheduling policy. Basically, in the absence of work-stealing, a Cilk Plus application should invoking a subtask via `cilk_spawn` has close performance to the serial version doing normal function calls. Since the Cilk programming model only supports fork/join tasking (no support for futures or `async_await`-like constructs), it's not surprising that Cilk Plus significantly outperforms the other runtimes, all of which support a more general set of concurrency constructs. At the same time, since this benchmark represents the worst-case scenario for tasking overhead, the fact that the Cilk Plus version of Fibonacci only runs about $8x$ faster than our baseline HClib version tells us that the HClib performance is in the right ballpark.

OpenMP's task-based parallelism support is a relatively new feature, and as such the current OpenMP runtime implementations are not as fine-tuned as the more traditional features. The fact that the OpenMP version of our Fibonacci benchmark only outperformed the thread-compensating strategy in HClib by about $2x$ strongly supports the conclusion that current OpenMP tasking implementations need improvement. The fact that creating

shown in the HClib C++11 λ variant in figure 3.2. If we used the C++11 HClib API for our benchmark baseline, then the Stellar Group's implementation of HPX for C++ would have been a better choice.



(a) Tasking runtimes comparison with Fibonacci benchmark.



(b) Tasking runtimes comparison with Fibonacci benchmark (log scale).

Figure 3.3: Initial performance results for a subset of our selected tasking strategies using the recursive Fibonacci benchmark. Slowdown is measured against an HClib baseline that uses the global-helping optimization.

a new kernel thread is only a factor of two slower than using an OpenMP task directive suggests that there is still significant room for optimization.

Finally, the HPX-5 version of Fibonacci (which was adapted directly from the HPX-5 example code included with their source distribution) had a slowdown of $2.29x$ versus our baseline. This falls between our HClib fiber-compensating strategy ($1.72x$) and our HC compiler-transformed code ($3.31x$). We did not do a detailed performance analysis to find why the HClib version of Fibonacci using fibers outperformed the HPX-5 implementation of Fibonacci, but we suffice to say that this result confirms that our Habanero-C runtimes' task overheads are competitive.

3.5.4 Implications for Resilience

As a second facet of our evaluation, we now briefly analyze the implications of each strategy with regard to resilience.

All of the thread and fiber-based strategies involve long-lived task contexts. These contexts are not amenable to recovery since they may contain references to task-local temporary data (i.e., pointers into the stack), which is difficult and expensive to save and restore in case of a failure. Furthermore, the prospect that we will have sufficient resources in an extreme-scale system to allocate the additional thread and fiber contexts to support blocking tasks also does not seem reasonable.

In contrast, the CPS transformation (automated by the HC compiler) and (manual) non-blocking transformations of the application code actually split the long-lived blocking tasks into multiple short-lived non-blocking tasks. The memory footprint for saving the continuation state is no more than any other task's input data, which seems more reasonable for a memory-constrained extreme-scale system. Although the thread-based and fiber-based strategies are less restrictive from the application programmer's perspective, we hypothesize that there is a well-defined point with respect to the system's *mean time between failures* (MTBF) where the context-saving strategies cease to make computational progress (due to continual failure of blocked tasks), and a transformation of the application to a non-blocking style becomes a necessity.

3.5.5 Strategy Tradeoffs

We now discuss the strengths and weaknesses for each of our blocked-worker compensation strategies in terms of programmability, debugging, and deployment. For each property, we rank our four strategies from best (#1) to worst (#4).

Computation Time

1. **HC Compiled**: The compiled HC variants of our benchmarks performed very well in most cases, and actually outperformed the unsafe baseline in both the Cholesky and Quicksort benchmarks. Since the performance issue for UTS could be solved simply by removing the array-copy restriction in the compiler, we decided not to count that case against it. The only other case where the HC compiled variant performed poorly was on the Fibonacci benchmark, but since that is not a realistic workload, and it was still within a factor of 2 of the fibers performance even in that pathological case, we feel that the HC code supported by its custom compiler toolchain gave the best overall for computation time.
2. **HClib Fibers**: The fibers variants of our benchmarks gave good, stable performance on every benchmark; therefore, we rank the fibers-compensating strategy in second place overall for for computation time.
3. **HClib Non-blocking**: The non-blocking strategy suffered from extra overhead in the Fibonacci, Cilk-sort and Cholesky benchmarks. For Needleman-Wunsch, this strategy only outperformed the others because we purposely used less-efficient `future.get()` calls in order to test blocking future operations in that benchmark; however, we were forced to use the more efficient `async_await` construct for the non-blocking code, which is actually more idiomatic and equally applicable for all of our blocked-worker compensation strategies.

4. **HClib Threads**: We rank the thread-compensating strategy last in terms of for computation time due to its disproportionately large slowdown on both the Fibonacci and Cilk-sort benchmarks,

Memory Usage

1. **HC Compiled**: The compiled variants of our benchmarks have relatively low memory overhead since they run on a fixed-sized worker pool and the compiler is fairly efficient at saving the state for suspended `async` tasks.
2. **HClib Non-blocking**: Although our non-blocking HClib applications can also run on a fixed-sized pool of worker threads, we saw that the C++11 API used to capture state for `async` tasks is less efficient than the HC compiler (see figure 3.2).
3. **HClib Fibers**: While our fiber-compensating strategy *technically* uses a fixed-sized worker thread pool, the fact that it allocates a new fiber stack to be used by the worker each time a task blocks means that the actual memory overhead is close to the same as creating a new thread each time a task blocks, especially if the thread stack size is set to match our fiber stack size.
4. **HClib Threads**: The thread-compensating strategy exhibits a similar memory footprint to our fiber-compensating strategy when the stacks are set to the same size; however, kernel threads have additional data structures that add to the memory overhead of creating new threads. In addition, since most users do not manually limit the thread size before running an application, we would generally expect a much larger memory footprint overhead when using threads than when using fibers.

Programmability

1. **HClib Threads**: Using the HClib C++11 API affords all of the programmability of an internal DSL (naturally expressing our domain-specific `async`, `finish` and other constructs using macros, lambdas, and other language features), while the thread-compensating strategy provides the same intuitive behavior as one would

expect if the runtime created a new thread for each `async`. However, if the finish-helping optimization is enabled, then the programmer may need to take care with how thread-local variables are used across blocking calls.

2. **HClib Fibers**: Using the fiber-compensating strategy to manage blocked workers affords almost all of the same benefits as using thread-compensating, and is almost indistinguishable from threads in terms of programmability and deadlock-avoidance. One downside of using fibers and not threads is that application code using thread-local state will not have a consistent view of the thread-local state across blocking calls (since fibers may be migrated among worker threads), which may cause some programmability issues. However, all but the thread-compensating strategy have this same issue with thread-local state (including the thread-compensating strategy when the finish-helping optimization is enabled); therefore, proper support (or lack thereof) for thread-local state is really a feature of the thread-compensating strategy rather than a shortcoming of the fiber-compensating strategy.
3. **HC Compiled**: Although HC adds the Habanero-C concurrency constructs (e.g., `async` and `finish`) as first-class constructs to the C language, we find that the internal DSL support affords almost all of the same benefits without the need for a specialized compiler toolchain. The HC compiler has some quirks that hurt programmability, such as the restriction on closing over arrays (previously discussed in the context of the UTS benchmark), reserving common variable names like `in` and `out`, and poor diagnostic messages in the case of a syntax error. While many of these compiler-related issues could be overcome with an overhaul of the compiler's code base, the HC compiler was under active development for several years, but these issues were never corrected. For HC or a similar custom compiler toolchain, one would expect these types of quirks until the custom language gained a larger user base and thus merited full time support and more focus on user experience.
4. **HClib Non-blocking**: Writing a non-blocking version of an application in HClib basically amounts to the programmer doing by hand what the HC compiler would

do automatically. Eliminating a blocking call from serial task code usually requires both splitting the task in two—the computation before the blocking call and the continuation after the blocking call completes—as well as adding some new synchronization mechanism to trigger the continuation task. All of these additional tasks and additional synchronization logic increase the complexity of the code, making it harder for the programmer to initially write the code, to reason about its correctness, and to maintain in the long-term. The programmer also must take extra care to only use blocking constructs at the top level (to synchronize the sequential `main` function with the concurrent HClib code), otherwise potential deadlocks may be inadvertently introduced in the code. While there may be some applications or algorithms where a non-blocking solution in simple and intuitive, we believe that in the general case the fully non-blocking approach ranks lower than our other strategies for programmability and productivity.

Debugging

1. **HClib Threads**: Using the thread-compensating strategy—although it has higher overhead—works very well when debugging for several reasons. Firstly, mainstream debuggers (e.g., *gdb* and *lldb*) have built-in support for inspecting the state of all kernel threads in the application currently attached to the debugger; e.g., if at the time the debugger his a breakpoint there are currently three threads blocked in `finish` scopes and five threads executing `async` tasks, then running the `info threads` command in *gdb* would reveal all of this state information. If the finish-helping optimization is enabled, the state is a bit more opaque (since multiple blocked tasks may be "buried" in one thread's stack), but inspecting the stack trace of each thread would quickly reveal the number of blocked tasks and where they are blocked. Secondly, debugging problems with stack overflows is easier since kernel threads allocate a guard page at the bottom of the stack. Diagnosing dependence on thread-local state is likewise easier if you can confirm an application works when using

thread-compensating but breaks when switching to one of the other strategies. Finally, third-party debugging tools such as *valgrind* work better when using standard kernel threads rather than custom fiber support or some other transformation. All of these factors together make the thread-compensating strategy the strategy of choice for debugging.

2. **HClib Fibers**: While fibers and threads are very similar conceptually, debuggers are not aware of our fiber contexts, which can potentially confuse tools like *gdb* and *valgrind*. When inspecting the program state in the debugger, the currently-running `async` tasks' fiber states are readily available via the worker threads' contexts; however, any blocked fiber states would only be visible by traversing runtime data structures, which would either require intimate knowledge of the runtime internals or HClib-specific extensions to the debugger. Debugging stack overflows when using fibers is also more difficult since the fiber stacks are allocated in the heap, meaning that stack overflows degenerate into heap buffer underruns. It would not be difficult to also allocate guard pages when allocating fiber stacks—but this requires additional system calls, and we find it simpler to just use the thread-compensating strategy, which gives us guard pages and the other advantages described above.
3. **HClib Non-blocking**: Application code that has been manually transformed into continuation-passing style can be difficult to debug for a few reasons. First, the code can simply be hard to read due to its complexity when compared with equivalent straight-line code. If the code is difficult for the programmer to understand, then debugging that code will also become much more difficult. Likewise, tracing the flow of control through a continuation-passing style (CPS) program is much more complex than tracing the execution of equivalent straight-line code. If an error occurs in some continuation `async` task, but the error was caused by a logic bug in an earlier task, then tracing the error back to the source can be very difficult. This is a problem in all programs that have asynchronous computations, but the problem is compounded in CPS programs by the potential explosion in number of tasks.

4. **HC Compiled**: The HC compiler-transformed code has all of the same problems as the manually-transformed non-blocking HClib code, but with the added complication that the debug symbols stored in the binary don't even map back to the HC source code that was written by the application programmer. Since the HC compiler toolchain performs a source-to-source transformation, generating pure C89 code from the HC source files, debug symbols generated by the C compiler correspond to the mangled C89 code rather than to the code that the programmer writes and sees. In fact, the C89 code is usually stored in a temporary file that is deleted by the HC compiler toolchain after it is compiled, meaning that the application programmer typically doesn't even know that the intermediate C89 source code exists. Again, these shortcomings should be fixable through more work on the compiler and/or debugger toolchains—but again, expecting this level of support for a toolchain with a small user base (even the entire HPC community is small compared to the general user base of GCC or Clang) does not seem reasonable. Therefore, even if HC development and maintenance were resumed immediately with a full-time staff of several researchers, we wouldn't expect the same user experience as what's available for the pure C++11 library solution used by HClib.

Deployment

1. **HClib Non-blocking**: Since the non-blocking HClib code uses no special toolchain, relies on no system-specific code, and can run correctly on a fixed-sized thread pool, applications using this strategy be easily deployed on most systems.
2. **HClib Threads**: Although POSIX threads are very portable—meaning that applications using the thread-compensating strategy should compile and run on a wide variety of systems—applications that create a large number of blocking tasks may run into problems with system resource limits set by the OS. For example, the hard limit of around 2k threads per process on Mac OS X could cause problems when deploying on that platform or on other platforms with similar resource limits.

3. **HClib Fibers**: Our fibers support depends on platform-specific assembly code to save and swap fiber contexts, which can cause problems when deploying on a less-common platform. We derived our fibers support from the Boost.Context library [64], which allows us to use the platform-specific machine code for each of that project's supported platforms directly with HClib, which in theory should make it simple to port to almost any platform where a user might want to deploy an HClib application; however, we have run into problems with bugs in the machine code for some platforms that are relatively common in the HPC community,[13] suggesting that we may encounter similar problems in the future when deploying to other non-x86-based platforms such as ARM.
4. **HC Compiled**: While there are some small obstacles to deploying applications that use the other strategies as described above, the custom toolchain supporting HC is much more difficult to deploy on a new platform or even a new machine. Due to complex dependencies in the HC compiler toolchain, it is very difficult to build the compiler on a modern Linux or OS X system. To support the benchmarks described in this section, we created scripts to perform the HC source-to-source translation either within a Docker appliance (a lightweight virtual machine) or on a remote machine, and then copy the resulting C89 code back to the local machine for compilation with the local C compiler toolchain. While this solution is workable, it is much less user-friendly than using HClib.

Resilience

1. **HC Compiled**: Based on the assumption that long-lived tasks are bad for resilience, the HC compiler is in the best position to transform application code that uses blocking API calls into something more resilience-friendly. The compiler could even automatically transform known blocking calls that are not necessarily part of the Habanero-C API (e.g., blocking I/O functions) into non-blocking variants, and split

[13] See this bug report about 64-bit PowerPC support: https://github.com/boostorg/context/issues/50

straight-line task code across those calls (i.e., doing a CPS transformation). The compiler could also help to facilitate restarting an application after a failure, by providing an alternate restart entry point that bypasses the bootstrapping code in the `main` routine and directly launches the tasks that need to be resumed. Although the compiler does not currently do these transformations, they seem like an obvious extension for failure-prone systems (failures have not been considered as a key issue in any past work in the Habanero-C programming model).

2. **HClib Non-blocking**: While HClib code written using only the non-blocking subset of the API should closely resemble code generated by the HC compiler, the fact that the programmer must write it by hand makes it error-prone; e.g., the programmer may inadvertently make a blocking call by invoking a function written by someone else. Furthermore, we always make use of blocking calls at the top level in HClib applications in order to synchronize all Habanero-C `async` tasks before returning control back to the sequential C program that drives our parallel Habanero-C program (i.e., we need to block the main program thread at some point so that the C `main` function doesn't return before our HClib code completes). Again, if we assume that long-lived blocking tasks are bad for resilience, then these top-level blocking calls would be problematic.
3. **HClib Fibers**: Using the fiber-compensating strategy to support blocking calls does not eliminate long-lived tasks; instead, it creates additional live-task contexts so that worker threads can continue doing work while awaiting completion of the task-blocking calls. The limited size of the fiber stacks may make it easier to save and restore the state of blocked fibers to support checkpointing and restarting of blocked tasks; however, under our assumption that long-lived tasks are bad for resilience, the fiber-compensating strategy does nothing to avoid long-lived blocking tasks.
4. **HClib Threads**: Using the thread-compensating strategy has all of the same problems associated with the fiber-compensating strategy, plus the extra complication that the blocked tasks each occupy a dedicated kernel thread, which would most

	Threads	Fibers	Non-blocking	HCC
Performance (3x)	4	2	3	1
Memory Footprint	4	3	2	1
Programmability (2x)	1	2	4	3
Debugging	1	2	3	4
Deployment	2	3	1	4
Resilience (2x)	4	3	2	1
Weighted Total	29	24	27	20

Table 3.1: Summary of the rankings of our blocked-worker compensation strategies, as discussed in section 3.5.5.

likely make saving and restoring the state of blocked tasks for automatic recovery more complicated than when using fibers.

3.5.6 Recommendations for Strategy Selection

Based on the analysis in this section—and specifically the strategy rankings given in section 3.5.5—we now give some general recommendations on how to select a good blocked-worker compensation strategy for a given situation.

Table 3.1 summarizes the results of the discussion in section 3.5.5. We assigned weights to three of our properties, and calculated the weighted totals of the scores for each strategy to get a rough idea of the general applicability of each strategy. We assigned the heaviest weight, 3x, to performance since that is usually the primary concern of HPC application developers (developers are usually willing to deal with other issues if it gets their apps to run significantly faster, which is also why the majority of this evaluation is dedicated to our performance analysis). We also assigned a 2x weight to both programmability and resilience. If a developer can throw together an app in half the time that gets 90% of the performance they could have achieved using a toolchain with poor programmability, the 10% performance hit is probably worth the productivity gain. Conversely, if an app is easy to write and gets good performance, but can't run to completion on a failure-prone extreme scale system because it has poor resilience properties, that is most likely not an acceptable tradeoff.

Based on the weighted total scores, the HC compiler (HCC) toolchain was the best choice overall. The HC compiled code had good performance on the majority of our benchmarks, it has a lower memory footprint than the other strategies (which will be very important in memory-constrained extreme-scale systems), and it had the best resilience properties. In addition, many of the issues with programmability and debugging could be addressed with more work on the compiler toolchain, which would result in an even better overall experience. However, due to the difficulty of actually deploying the HC compiler toolchain—specifically its lack of public availability[14]—our general recommendation for running Habanero-C applications is to use HClib with the fiber-compensating strategy. In fact, using the fiber-compensating strategy with the finish-helping optimization is currently the default configuration for the HClib runtime.

Although the thread-compensating strategy has poor performance for some workloads and a much higher memory footprint in general, the many advantages for debugging when using this strategy makes it our favorite configuration when actively debugging an application. Since the HClib blocked-worker compensation strategy can be selected via environment variables when launching any HClib application not built in production-mode, application developers can easily choose to use thread-compensating (with or without finish-helping) when they encounter a difficult bugs during the development process, instantly improving the user experience with *gdb*, *valgrind*, and other debugging tools.

[14] Unlike HClib—which is open source and publicly distributed under the Apache 2 license on GitHub—the HC compiler and runtime are not currently available to the public. The Habanero group has shared the toolchain with collaborators in the past, but due to license issues around portions of both the compiler and the runtime code, it was never approved for a public release. Since the toolchain is no longer under active development (being mostly superseded by the HClib project), it is unlikely to be released in the near future.

3.6 Related Work

Although many other lightweight task-based runtimes exist, the primary focus of this chapter is on the comparison of multiple task-scheduling strategies and their tradeoffs, rather than on the performance of Habanero-C versus other runtimes. Other C/C++-based runtimes, such as Sandia Qthreads [46] and Intel Thread Building Blocks (TBB) [58], implement similar strategies to those discussed here; however, evaluations of these models focus on performance compared to different programming models on different runtimes. While the TBB documentation encourages users to write applications in continuation-passing style rather than relying on the depth-bound worker "helper" strategy, we are not aware of any direct comparison of TBB benchmarks written for both strategies.

Yang et al. [48] performed a comparison of the compiler-based Cilk Plus framework included in GCC5—which uses a delimited CPS transformation on Cilk functions—with *Fibril*, their library implementation of Cilk. Fibril uses some clever optimizations to support the work-first execution policy without assuming special compiler support. Their runtime allocates a new thread context (not a new thread—just a new stack) at each Cilk function call, but remaps unused physical pages in the stack to maintain a reasonable bound on the total physical memory footprint. These memory optimizations could be applied to our fiber-compensating strategy to maintain a smaller resource footprint.

Imam et al. [67] presented a solution for cooperatively scheduling blocking tasks in HJ-lib, a work stealing runtime on the JVM. While their solution is sufficient for correctness, one-shot delimited continuations are not the minimal necessary programming element needed to support such scheduling. We precisely define the necessary characteristics for this type of scheduling, and compare several solutions of varying generality. The study includes a comparison of a cooperative work-stealing runtime and a thread-blocking work-sharing runtime, both part of HJ-lib. We believe the HJ-lib runtime would also be amenable to the Finish/Helper optimization described in this work.

The cooperative runtime in HJ-lib is very similar in design to that in Quasar [68]; in fact, both rely heavily on the same underlying continuations library [69], written by

Matthias Mann. It is interesting to note that Mann's continuations library—which is the currently the state of the art for continuations on the JVM—is basically a stack copying scheme triggered by a special exception. Since Java does not allow the user to capture references into the stack, it is safe to copy the stack-allocated values from one thread and restore them in another, whereas in C/C++ runtimes the possibility of pointers into the stack makes this much more difficult. In fact, many of the restrictions placed by the HC compiler center around the issue of capturing stack addresses in continuation frames, which makes frames non-migratable among worker threads.

The goal of HPX [47] is to provide a reference implementation of the runtime for a future version of the C++ standard library's parallel programming features. Since their work adheres strictly to the current C++ standard, it is possible to do a good comparison between current C++ standard library implementations of `std::async` and `std::future` versus their implementations. However, their studies instead compare with other non-standard C++ runtimes (Qthreads and TBB)—ostensibly because the performance of current standard-library implementations of these parallelism constructs is not at all competitive.

3.7 Summary

We have examined the difficulties tied to supporting blocking constructs in a lightweight tasking runtime, specifically in the context of the Habanero-C programming model. We have discussed the possible strategies used for scheduling tasks in the runtime, paying particular attention to the property of deadlock-freedom. We presented six strategies for handling blocked worker contexts in Habanero-C programs, including four main strategies, and two variants using the finish-help optimization. We evaluated our six selected strategies implemented in the Habanero-C programming model—including HC and HClib—with regard to usability, performance, and resilience. Finally, we presented our general recommendations based on our analysis for which strategy to use in each situation. While we found that the compiler-supported HC toolchain appears to be the best option, unfortunately we cannot recommend it to potential Habanero-C application programmers due to its lack of public availability. Instead, we suggest using HClib for application development, recommending the fibers-compensating strategy as the default runtime configuration choice, and the thread-compensating strategy specifically for debugging. We hope that compiler support can be leveraged in the future to take advantage of the productivity benefits afforded by domain-specific languages, as well as the performance and resilience benefits provided by the compiler-transformed source code.

Chapter 4

CnC-OCR: A Productivity Environment for OCR

In this chapter, we present CnC-OCR as a higher-level programming model and productivity environment for OCR, providing key abstractions for hierarchy, tuning, and dependence coordination. To the best of our knowledge, this is currently the only high-level programming model that maps to OCR while being faithful to OCR's data model.

4.1 Background

4.1.1 The Open Community Runtime (OCR)

The Open Community Runtime (OCR) is a set of low-level APIs aimed to provide a portable, resilient, and high-performance abstraction layer between extreme-scale hardware and applications. The API is intended as a target for compilers, a back-end for high-performance libraries, and as a tool for expert programmers writing performance-critical code. Being a low-level API, the design is much more focused on unambiguity and flexibility, rather than on programmability. Seeing as the average application programmer is not meant to program directly to the OCR API, it is necessary to provide one or more productivity layers on top of OCR for use by these developers.

4.1.2 The CnC Programming Model

CnC [70] is a graph-based *dependence programming model*. Note that CnC is a programming *model* rather than a programming *language*. There are many implementations of the CnC model in many programming languages, including C [71], C++ [72], Python [73], Scala [74], and Haskell [75]. The focus of this chapter is a C-based implementation of CnC, which is built on top of the C-language OCR API.

In CnC, programs are expressed in graph form. The CnC programmer decomposes an application into *item collections*, which represent data, and *step collections*, which represent computation. The item and step instances comprise the nodes of the CnC graph, while the edges represent data and control dependencies among the nodes. This specification of the inherent dependencies between the data and computation within a CnC application is called the *domain spec*.

While CnC is not an explicit parallel programming model, the dependence edges in the CnC graph implicitly restrict parallelism among task nodes, meaning that a CnC runtime implementation is free to schedule two tasks (or *computation steps*) in parallel so long as there is no dependence relationship specified between them.

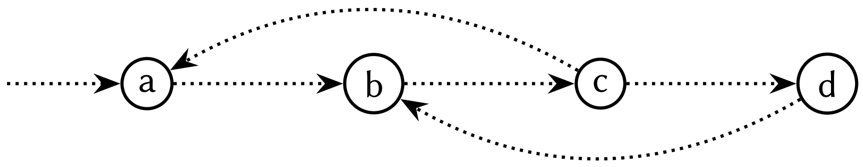


Figure 4.1: Example of a simple, abstract CnC graph that does not correspond to a traditional loop nest, since the back-edges $c \rightarrow a$ and $d \rightarrow b$ would create a pair of partially-overlapping loop regions.

Since many CnC programs are ports of various parallel research kernels, the dependence graph structures for these ported codes resemble the structure of the loop nest in the source application. However, the CnC model is in fact more general than traditional loop-based parallelism. One can easily construct a simple parallel CnC program that is not readily translatable to an equivalent parallel loop nest. Figure 4.1 shows an example of such a graph.

In CnC, the *domain specification* comprises all of the constraints that are necessary for correctness in the application. Any additional constraints that may be provided are classified as *tunings*.

4.1.3 Separation of Concerns in CnC vs. OCR

The mantra of CnC is *separation of concerns*. When designing a CnC application, the details of step function implementations are separate from the domain specification (i.e., the dependence graph), and application tunings are separate from both the domain specification and the step codes. This separation makes it easy for someone unfamiliar with a CnC application's codebase to get a high-level sketch of the program semantics (by studying the domain specification), analyze the implementation details of a particular computation step, or quickly understand the current performance-tuning strategy (by examining any tuning specifications currently in use).

Similarly, OCR aims for a separation of concerns between the application code, control (scheduling), and resource management. OCR has been designed with a modular, extensible scheduler and resource managers that can be customized to take advantage of performance tuning hints supplied in the application code. However, the separation of concerns in OCR is at more of a conceptual goal rather than a philosophy realized the level of the source code, since tuning hint code is embedded within the application code.

The fact that the tuning hint code is sprinkled throughout an OCR application's source code makes it difficult to get a big-picture view of the tuning strategy being employed in an OCR program. Likewise, having tuning and dependence information scattered throughout an application's source code can obfuscate the core computation of that application. Additionally, the verbosity of the OCR API makes it toilsome to modify existing source code in order to experiment with different tuning strategies. Furthermore, the interleaving of domain code with tuning code complicates the job of maintaining multiple tuning options within a single application.

We claim that all of these programmability issues in OCR can be overcome by offering a higher-level programming model to developers, and providing a programming system that automatically maps to OCR. We demonstrate this via the implementation of CnC on OCR, and an analysis of the resulting framework for developing OCR application.

4.1.4 CnC Graph Notation

In this chapter, we will use the following notation for CnC items and steps:

[X: i, j, k] An instance from an item collection named *X*, with the tag $\langle i,\ j,\ k\rangle$. The square brackets correspond with the rectangular nodes typically used in graphical representations of CnC item collection.

(Y: i, j, k) An instance from a step collection named *Y*, with the tag $\langle i,\ j,\ k\rangle$. The round brackets correspond with the elliptical nodes typically used in graphical representations of CnC step collections.

This notation is fairly standard in the CnC literature. See appendix B for a more detailed explanation of CnC graphical and textual representations.

4.2 Overview of Our Approach

We propose CnC-OCR—an implementation of the CnC programming model on top of the Open Community Runtime API—as a higher-level programming model and productivity environment for OCR. CnC-OCR is included in the OCR open source release [5], and is also available as a standalone project on GitHub [71]. The CnC-OCR toolchain consists of four main components:

1. A domain-specific language (DSL) for declaring a CnC application graph, including its step and item collections, as well as the dependence relationships among those collections.
2. A tuning language, based on the graph DSL, for expressing additional graph constraints for the purpose of performance tuning.
3. A prototype system for automatic inference of hierarchical structures within a CnC dependence graph, used for applying hierarchy-based optimizations.
4. An automatic code-generation toolchain for translating the domain and tuning specification into a skeleton CnC-OCR application, including function stubs for the computation step implementations.

We describe the implementation of the CnC-OCR toolchain, and demonstrate the benefits to programmer productivity and the improved separation of concerns attained by the use of the CnC domain specification for dependence coordination, as well as the external specifications for performance tuning. We define a set of rules for automatic generation of CnC application hierarchies, and demonstrate the benefits of hierarchy through improved distributed memory locality. We show that the performance of CnC-OCR applications is comparable to much more verbose hand-coded OCR solutions, and is competitive with an existing production-grade CnC implementation.

4.3 Design and Implementation of CnC on OCR

4.3.1 Mapping CnC onto the OCR Programming Model

While there exist other higher-level programming models built on top of OCR [59,76,77], it is our opinion that these fail to provide a true and faithful abstraction layer over OCR. This is because these solutions eschew key OCR concepts (e.g., datablocks), and thus produce applications that can only work correctly on a limited set OCR's target platforms. For example, programs that do not by store all shared data within OCR datablocks—and thus do not conform to OCR's data model—will not function correctly in distributed-memory implementations of OCR. Although we considered these programming models specifically in the context of OCR, we see this as a general problem when building higher-level productivity layers for exascale programming models.

The Open Community Runtime was designed based on a fundamental set of axioms, e.g., that all persistent data must reside in runtime-controlled datablocks. Similarly, existing candidates for higher-level layers were also designed under some set of fundamental assumptions. It is unlikely that a programming model designed for shared-memory—or even HPC clusters of the previous decade—will have axioms that match these assumptions of extreme-scale computing. This suggests that one of two things is necessary: (1) existing programming models must be heavily modified to fit the exascale paradigm, or (2) we need to design new higher-level programming models from scratch, based on the new axioms for extreme-scale computing.

However, the CnC programming model happens to share a very similar set of assumptions with that of the Open Community Runtime model. As shown in table 4.1, there is a direct correspondence between most of the core concepts in CnC and OCR.

Concept	**CnC construct**	**OCR construct**
Task classes (code)	Step collection	Task template
Task instance	Step instance	Task
Data classes	Item collection	—
Data instance	Item instance	Datablock
Unique instance identifier	Step/item tag	GUID
Dependence registration	Item get	Event add dependence
Dependence satisfaction	Item put	Event satisfy

Table 4.1: Mapping of software concepts between CnC and OCR.

While there is some divergence between the assumptions made in traditional CnC implementations and those for CnC-OCR, we were able to address all of those differences without radical changes to the CnC paradigm. In fact, in some cases the choices we made for CnC-OCR actually lead to a model that we found truer to the themes of CnC than what had been done in previous CnC implementations. These differences are discussed in more detail in section 4.3.4.

This clean mapping between the CnC and OCR programming models makes it fairly straightforward to generate scaffolding code for a CnC application to run on top of OCR. Similarly, declarations in the CnC tuning specification are used to automatically generate hint-annotated OCR code.

The CnC-OCR toolchain automatically generates OCR scaffolding code based on the dependences declared in the CnC graph specification, and the additional declarations in the tuning specification. For example, an OCR task is generated to wrap each CnC computation task, and the generated code automatically sets up the OCR task's dependences based on the declarations in the CnC graph specification. The CnC application programmer does not deal with any OCR-specific code, while the CnC program is entirely OCR compliant. This makes CnC-OCR an effective programming abstraction for OCR.

4.3.2 Software Architecture

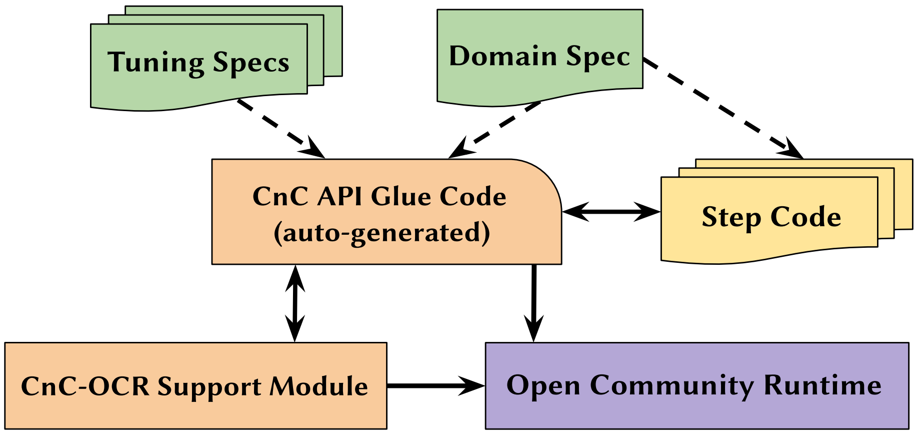


Figure 4.2: The software architecture of a CnC-OCR application. Dotted-line edges represent code generation relationships between programmer-specified inputs and the generated outputs, whereas solid-line edges represent coupling between software components. Green nodes represent specification files written by the application programmer. The yellow node represents source code that is partially generated from the domain specification, but must be completed by the application programmer. Orange nodes represent code that is viewed as part of the CnC-OCR runtime and API support, and thus are considered opaque by the application programmer. The purple node represents the underlying tasking runtime.

The software architecture of a CnC-OCR application is illustrated in figure 4.2. The application data and computation dependencies are described using the graph DSL, and provided as a domain specification file. This domain specification is use to automatically generate skeleton code for the application, including initialization, the compute steps, and finalization code. The application programmer then fleshes out the skeleton code to perform the desired computations.

In addition to the domain specification, one or more tuning specifications may also be provided. These tuning specifications are written in a DSL similar to that of the graph specification, and they are used to generate better-optimized code in the CnC API glue code and support module components. Since the tuning specifications are decoupled from the rest of the input, and they do not affect any code that is altered by the application programmer, tunings can be easily analyzed and adjusted. CnC tuning will be discussed in more detail in section 4.6.

The C language provides very limited facilities for type-specialized APIs. Basically all type-generic code involves pervasive use of `void*` types, which afford very little type safety, and thus can hurt productivity. Although CnC-OCR is implemented in C, we are still able to attain much of the type safety features that would be available through a C++-based API by leveraging our custom code generation toolchain. The type information from the domain specification is used to generate a set of customized API functions for the given CnC application, based on the declarations of its item and step collections. This auto-generated layer of glue code allows static checking of the parameters for CnC operations—such as verifying the correct number of tag components for a *prescribe* operation, or the correct payload data type for a *put* operation—before passing the arguments through to the CnC runtime support layer, which is implemented in terms of void pointers and byte arrays.

Note that none of the user-modified code (i.e., the domain and tuning specs, and the step code implementations) is directly coupled with the underlying tasking runtime; or in other words, the underlying tasking runtime API details do not leak through into the CnC application code. This design allows the framework to support other back-end runtimes in addition to OCR, while leaving all CnC application code unchanged. This feature is discussed further in section 4.3.4.

4.3.3 Development Workflow

We now give a brief description[1] of the typical workflow for a CnC-OCR application developer. The process of creating a new CnC application usually roughly involves the following steps:

1. **Write a CnC domain specification.** This often simply means translating a whiteboard sketch of the application's dependence graph into the CnC graph DSL syntax, even if the design wasn't originally made specifically with CnC in mind for the implementation paradigm. See figures 4.3 and 4.4 for an example of a whiteboard sketch of an application and the corresponding CnC graph.
2. **Generate a skeleton project.** The project is generated via the CnC framework graph translator tool, using the domain specification as input. The translator tool creates all of the auto-generated files shown in figure 4.2. Note that the generated skeleton project can be compiled and run immediately after generation, although the skeleton code will not do any useful computation. We feel that this feature enhances the incremental development process.
3. **Flesh out the skeleton step code.** Unfortunately, our toolchain is currently unable to infer the computation that the application is meant to perform. The programmer is still expected to do some coding. Automatic synthesis of the step code implementations is left as an exercise for the reader.
4. **Debug the application.** It is our experience that application programmers typically do not get the application right on their first try, and the code requires some debugging. The CnC toolchain supports various debugging techniques, such as execution traces and visualizations. If a bug is found in the domain specification rather than the step code, then it may be necessary to re-generate some of the project code using the graph translator tool. By default, the tool will not overwrite any of the files that are edited by the application programmer. For example, if the number of

[1] For a more detailed description of the CnC-OCR development workflow, see the Habanero CnC Framework wiki: https://github.com/habanero-rice/cnc-framework/wiki/Workflow

inputs to a step changes, it will be necessary to update the signature of the function implementing that step. In this case, we recommend that programmers save the existing code, generate a fresh copy of the skeleton step, and then use their favorite diff tool to manually merge the changes.

5. **Tune the application.** Once the application is functioning correctly, the focus shifts to optimization. The tuning process is discussed in section 4.6.

4.3.4 Unified CnC

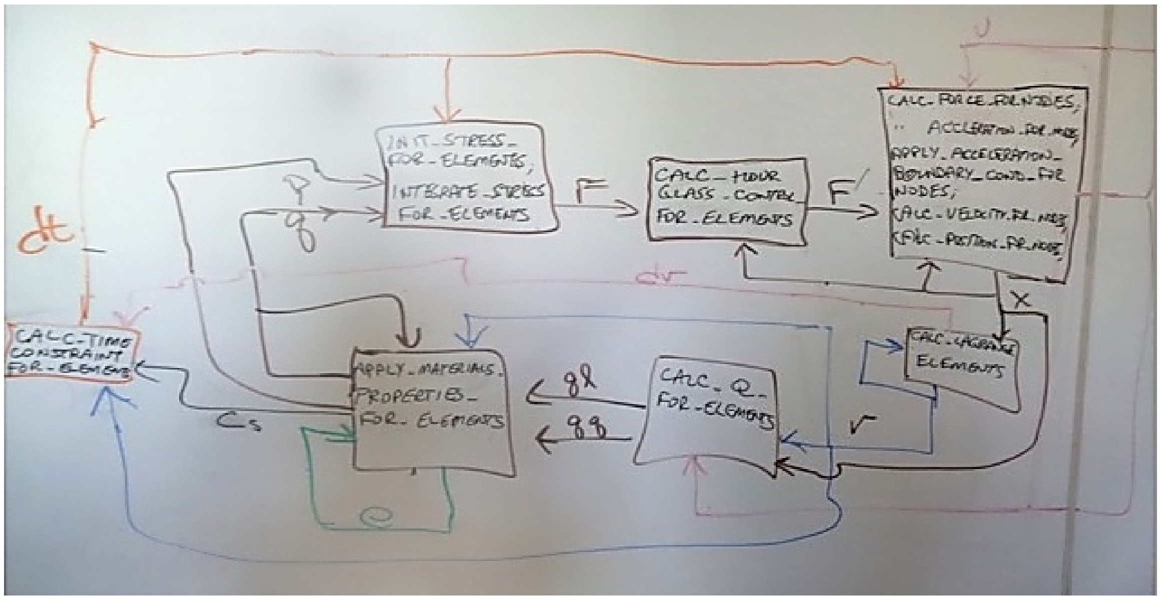
As discussed in section 4.3.2, the software architecture of the CnC toolchain used to support CnC-OCR is made up of several decoupled layers. This design allows us to support multiple backend runtimes for CnC applications developed using our framework. The common set of abstractions used to support CnC applications across multiple runtime is known as *unified CnC* [78]. The Open CnC community identified three pillars of CnC unification in a series of multi-institution discussions, all three of which are realized in our CnC framework:

1. **Unified graph language**

   Several variants of the text-based CnC graph representation have been introduced over the years; however, these past notations are always closely coupled with a particular implementation. We provide a graph language that is sufficiently expressive for defining detailed dependence relationships among the step and item collections in a graph, while leaving implementation-dependent features for external tuning specifications. See appendix D for more details on the graph language defined in our framework.

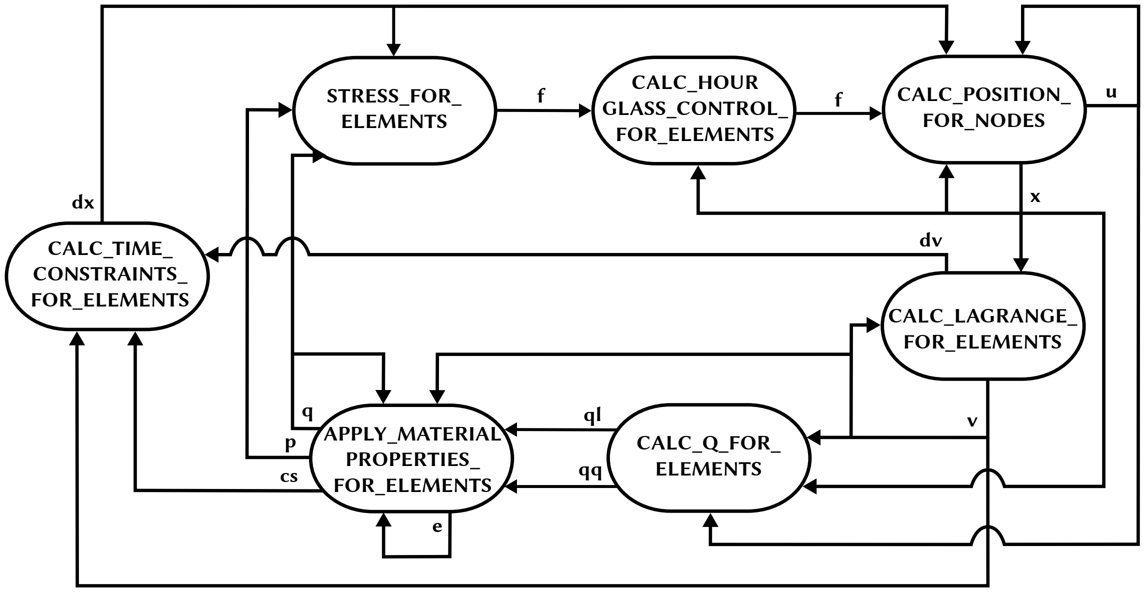
2. **Unified graph code generation toolchain**

   Just as there are several variants of the graph specification language, there are many tools that can generate a CnC project codebase from a graph specification. These tools are (understandably) each tightly coupled with the syntax of its respective


INIT_STRESS_FOR_ELEMENTS;
INTEGRATE_STRESS_FOR_ELEMENTS
CALC_HOUR GLASS_CONTROL_FOR_ELEMENTS
CALC_FORCE_FOR_NODES;
.. ACCELERATION_FOR_NODES
APPLY_ACCELERATION_BOUNDARY_COND_FOR NODES;
CALC_VELOCITY_FOR_NODES;
CALC_POSITION_FOR_NODES;
CALC_LAGRANGE_ELEMENTS
CALC_Q_FOR_ELEMENTS
APPLY_MATERIALS_PROPERTIES_FOR_ELEMENTS
CALC_TIME CONSTRAINT FOR_ELEMENTS
dt
p
q
F
F
dv
x
v
ql
qq
Cs
e
u

Drawing by John Feo, Pacific Northwest National Laboratory.

Figure 4.3: Whiteboard sketch of the LULESH proxy application. This graph was designed by a group of scientists with no specific knowledge of CnC, and the diagram was simply meant to describe the control and data flow of the application.


STRESS_FOR_ELEMENTS
CALC_HOUR GLASS_CONTROL_FOR_ELEMENTS
CALC_POSITION_FOR_NODES
CALC_TIME_CONSTRAINTS_FOR_ELEMENTS
CALC_LAGRANGE_FOR_ELEMENTS
APPLY_MATERIAL PROPERTIES_FOR_ELEMENTS
CALC_Q_FOR_ELEMENTS
f
f
u
x
dx
dv
v
q
p
cs
ql
qq
e

Adapted from a figure by Ellen Porter, Pacific Northwest National Laboratory.

Figure 4.4: CnC graph structure of the LULESH proxy application. Item collections are represented as labels on edges rather than independent nodes in order to emphasize the symmetry with the whiteboard sketch in figure 4.3.

graph language, which is in turn tightly bound to a specific CnC implementation. We introduce a modular code generation framework that is designed for supporting multiple front-end graph representations and multiple back-end runtime targets for translating a CnC graph specification into application code.

3. **Unified developer API**

   While porting application code among the several CnC implementations is usually a fairly straightforward process, it seems counterproductive to have such a level of fragmentation in the CnC ecosystem—especially for implementations with compatible host languages. We define a C language API for use in CnC step and environment code, allowing programmers to write application code that is compatible with any supported C or C++ back-end runtime of the translator tool. See appendix E for a full description of the API.

Our unified CnC framework currently provides support for two different runtime backends: OCR [5] and the Intel CnC runtime (iCnC) [72]. This allows for a more straightforward performance comparison of CnC-OCR applications against a production-quality CnC implementation (iCnC) on current x86-64 shared-memory and distributed-memory systems. Providing multiple target runtimes also improves debuggability: If an application works on iCnC but not CnC-OCR, the problem is likely somewhere in the CnC-OCR layer, rather than in the logic of the user's application.

As mentioned in section 4.3.1, the final API for our CnC framework diverged in some ways from the patterns in previous CnC implementations. Most of these divergences were motivated by restrictions in the OCR programming model. We give a summary of the four major differences below.

Explicit Environment Pseudo-steps: To the best of our knowledge, all previous CnC implementations used the following pattern at the top level in CnC applications: initialize the graph, make a blocking call to await completion of the graph computation, and then read the desired output data. However, OCR applications use a purely event-driven model, where such blocking semantics are best avoided if possible. We determined that in previous CnC implementations, the blocking semantics were simply an artifact of the CnC application being embedded in a serial C/C++ program. In our framework, we instead split the setup and teardown code into two pseudo-steps, thus avoiding the need for blocking calls in the API. These pseudo-steps are denoted by the special identifiers `$initialize` and `$finalize` in the CnC graph and tuning DSLs.

Runtime Hooks for Memory Allocation: As explained in section 2.2.2, all data that persists across more than one OCR task must be allocated within an OCR datablock; therefore, we must provide an API abstraction for managing persistent item data. The `cncItemAlloc` and `cncItemFree` functions are used for allocating memory for items, which typically have a scope larger than a single CnC step. We also provide the `cncLocalAlloc` and `cncLocalFree` functions to allow optimized allocation of step-local temporary data.

Opaque Data Items: As explained in section 2.2.2, OCR considers datablocks to be completely opaque, simply doing byte-for-byte copies to relocate datablocks; therefore, the items in our CnC programs are also assumed to be self-contained, trivially-copyable objects. In contrast, iCnC automatically serializes/deserializes objects when transferring items in distributed memory systems. The techniques introduced in chapter 2 also could be applied in CnC applications to help work around this restriction for CnC-OCR.

Optional Static Data Dependencies: While some CnC implementations require all step inputs to be declared statically [79], most previous CnC implementations provide a mechanism for *data-dependent gets*; i.e., a computation step can request access to additional CnC item instances based on data read from already-obtained items. While OCR requires that all datablocks be acquired before a task begins execution, we provide a simple mechanism in the graph DSL—the `$when` clause—to provide a static function (defined in terms of instance tag components) to determine whether or not a particular input is required for a given step instance. Note that while one previous CnC implementation explored a *flexible* *precondition* model for inputs [80], this was in fact a hybrid approach allowing both static and data-dependent input dependencies, not a way to support optional dependencies that are determined statically. We found that the flexibility provided by the `$when` clause was sufficient for the majority of our CnC applications; therefore, due to both implementation restrictions and lack of demand, data-dependent gets are not supported in our framework.

4.3.5 Code Generation Support

For source code generation, we use Jinja2 [81], a well known Python-based source code template framework. Although Jinja2 is typically used for generation of template-based web pages, this template framework also works well for generating our CnC skeleton projects from our source code templates written in C, GNU Make, and other languages.

We make heavy use of template inheritance, a key feature in Jinja2, in order to better support multiple runtime backends. Jinja2 templates allow the declaration of named blocks, which can later be overridden to specialize the template. For example, we use this feature to provide a common *makefile* template for CnC projects across all runtime targets, while allowing specific implementations to override blocks to add additional build targets, extra compiler flags, linker options, etc.

This flexible template framework allows us to reuse all of the templates for the templates for the skeleton source code files (which are runtime agnostic), leaving only the runtime-specific API implementation files to be defined when targeting a new backend runtime. Our CnC framework system was designed from the start to support two different runtime backends—OCR [5] and Intel CnC (iCnC) [72]—and we heavily reuse template code to generate CnC projects that target these two different backends.

Table 4.2 gives a summary of the number of lines of code required to implement each of our currently supported platform targets. These line counts can be used as a rough estimation of the effort needed to add support for a new backend runtime. Note that all implementations make heavy use of template inheritance. For example, the OCR x86-64 distributed memory platform extends and specializes the shared memory platform

	Common	**iCnC**	**OCR**			
			Base	TG	x86-64	
					Shared Memory	Distributed Memory
Source Code	458	734	1101	240	81	53
Makefiles	57	86	0	53	58	21
Total	515	820	1101	293	139	74

Table 4.2: Line counts of template files for the various CnC framework runtime backend targets. All counts are measured in terms of *logical software lines of code* using UCC [82].

templates, which in turn extend and specialize the OCR base templates. All OCR and iCnC targets also reuse the templates that are common across all CnC implementations; however, since the majority of the common template code is used for generating the runtime-agnostic application code skeleton, most of these templates are not extended by the OCR or iCnC implementations. (The only exception is the base makefile template, which is specialized for the platform-specific project builds.) The common template code also provides some macro functions for common tasks, such as printing the variables for a step tag, or generating a loop nest for processing a range of input items.

Note that the implementation of CnC-OCR for the Traleika Glacier (TG) hardware simulator [83] is implemented purely in terms of OCR constructs—as per the OCR v1.1.0 specification [84]—and thus will run correctly on any platform supporting OCR. (In other words, the x86-64 specializations make optimizations that may be unsafe on other platforms.) Similarly, the same iCnC code works for shared memory, distributed via MPI, and distributed via raw TCP sockets. The only difference among the three iCnC targets is their makefiles, all three of which are included in the count in table 4.2.

For both CnC-OCR and iCnC projects, the runtime-agnostic common templates constitute over 30% of a project's template code. For the various CnC-OCR specializations, the majority of the runtime-specific template code is contained in base templates, which are extended by each of the specialized targets. The specialized templates for x86-64 and

TG need only provide item collection implementations and platform-specific makefile configurations, since all of the remaining logic is contained in the base templates.

Past efforts have added support for HCMPI and HPX-5 as alternative backend runtimes [85, 86]. Since neither of these projects are actively maintained, accurate source code line counts were not available for inclusion in table 4.2; however, both of these projects required an amount of code in the specialized templates that is comparable to the iCnC backend (i.e., less than 1000 logical lines of source code). We believe this supports a general claim that our toolchain design allows for the addition of new runtime backends with only a modest effort.

4.4 CnC Programming Example: Fibonacci Numbers

In this section, we give a simple example of programming an application in CnC. We cover steps 1–3 from the workflow described in section 4.3.3. Step 4 (debugging) is not discussed in this thesis, but suggestions on debugging are available on the CnC framework wiki. Step 5 (tuning) is discussed in section 4.6.

Our example CnC application will compute the *n*th Fibonacci number, using the traditional Fibonacci recurrence relation shown in equation (4.1).

$$Fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ Fib(n-2) + Fib(n-1) & \text{otherwise} \end{cases} \tag{4.1}$$

4.4.1 Writing the CnC Graph Specification

Based on equation (4.1), we will want at least two collections in our CnC application graph: a step collection to compute each $Fib(n)$ value in the recurrence relation, and an item collection to store each of those values. We will call the step collection for computing the values `compute_fib`, and the item collection for storing the values `fib`. The CnC graph specification for our Fibonacci numbers application is shown in listing 4.1.

```
1  $context { int n; };
2
3  [ int *fib: i ];
4
5  ($initialize: () ) -> ( compute_fib:$rangeTo(0, #n) );
6
7  ( compute_fib: i )
8   <- [ x @ fib: i-2 ]$when(i > 1),
9      [ y @ fib: i-1 ]$when(i > 1)
10  -> [ z @ fib: i   ];
11
12 ($finalize: () ) <- [ fib: #n ];
```

Listing 4.1: CnC graph specification file for computing Fibonacci numbers.

We now give a brief description[2] of the syntax used to describe our CnC application. Line 3 of listing 4.1 shows the declaration of our `fib` item collection, which stores integer values identified by the tag i. The step collection `compute_fib` is declared on line 7, with its input relations on lines 8 to 9, and its output relation on line 10. The step collection also has a single tag component, named i, which directly corresponds with the i in `fib`. This correspondence is defined on line 10, which says that each (compute_fib: i) step *puts* (i.e., produces) the corresponding [fib: i] item value. Lines 8 to 9 specify that each (compute_fib: i) instance reads the item values [fib: i-2] and [fib: i-1] as input, but only when $i > 1$. This conditional input relationship enables the base cases for $n = 0$ and $n = 1$ shown in equation (4.1).

The *context* declaration on line 1 of listing 4.1 specifies the global parameters available to the entire CnC graph computation. In this case, we have a single integer value n, which represents the argument value for our $Fib(n)$ computation. The `$initializer` pseudo-step declaration on line 5 bootstraps the computation by *prescribing* the step instances for (compute_fib: 0) through (compute_fib: n). The `#n` notation refers to the parameter n in

[2] See appendix D for a complete description of the CnC graph DSL.

the graph context. The `$finalize` pseudo-step declaration on line 12 specifies that the desired output of the application is the value of [`fib`: `#n`], i.e., $Fib(n)$.

4.4.2 Generating the CnC Project Skeleton

Assume that we have created a new directory for our CnC project, and saved the code from listing 4.1 in a file named *Fibonacci.cnc* within that directory. If we run our `ucnc_t` translator script in the project directory,[3] then the following files are automatically generated (listed here in lexicographical order):

- *Fibonacci.c:* Contains the skeleton functions for the `$initializer` and `$finalizer`.
- *Fibonacci_compute_fib.c:* Contains the skeleton function for `compute_fib`.
- *Fibonacci_defs.h:* A global header for the project, useful for things like definitions of custom types used in your graph.
- *Main.c:* Contains the `cncMain` function, which is the entry point for all unified CnC applications. This has the same signature as, and generally corresponds with, the traditional C/C++ `main` function.
- *Makefile:* A symbolic link to *Makefile.x86*.
- *Makefile.x86:* The platform-specific Makefile for building this unified CnC application on the shared-memory (x86-64) CnC-OCR platform.
- *cnc_support:* A directory containing the CnC runtime support modules. The CnC application programmer needs neither read nor modify the source files that are generated in this directory.

This skeleton program can immediately be compiled an run (via `make run`); however, the application gives no meaningful output since the skeleton functions have not yet been modified to do any meaningful computation.

[3] There are no arguments required for the `ucnc_t` script. By default, the script looks for a *.cnc* file in the current directory (which is assumed to be the target CnC graph specification), and assumes that CnC-OCR shared memory (x86-64) is the desired target platform.

4.4.3 Fleshing Out the Project Skeleton

For this simple application, the CnC framework's graph translator tool actually generates all but a few lines of the needed code. The source code for all modified files is shown in full in listings 4.2 to 4.4.

The two lines added in listing 4.2 verify that a command-line argument was given, and then parses that argument, and sets the value as the graph parameter n. The line added in listing 4.3 implements the Fibonacci recurrence relation function. Note that the conditional expression only adds the optional inputs when $i \geq 2$, as specified in equation (4.1). Finally, the `printf` statement added to the finalization function in listing 4.4 simply prints the computed result value for $Fib(n)$. Note that each place in the skeleton code where we needed to add some custom code was denoted in the auto-generated code with a `TODO` comment.

Our example CnC application is now complete! The project can be compiled and run, and it will display the correct value for $Fib(n)$, barring out-of-range inputs (e.g., $n < 0$) or integer overflow. The following is sample output for computing $Fib(5)$ and $Fib(8)$:

```
$ make run 5
Extracting WORKLOAD_ARGS from the command-line
WORKLOAD_ARGS used: '5'
cd ./install/x86 && \
        OCR_CONFIG=generated.cfg \
         ./Fibonacci 5
Fibonacci(5) = 5
$ make run 8
Extracting WORKLOAD_ARGS from the command-line
WORKLOAD_ARGS used: '8'
cd ./install/x86 && \
        OCR_CONFIG=generated.cfg \
         ./Fibonacci 8
Fibonacci(8) = 21
```

```c
#include "Fibonacci.h"

int cncMain(int argc, char *argv[]) {
    // Create a new graph context
    FibonacciCtx *context = Fibonacci_create();

    // TODO: Set up arguments for new graph initialization
    // Note that you should define the members of
    // this struct by editing Fibonacci_defs.h.
    FibonacciArgs *args = NULL;

    // TODO: initialize graph context parameters
    // int n;
    CNC_REQUIRE(argc >= 2, "Required argument N for Fibonacci(N)\n");
    context->n = atoi(argv[1]);

    // Launch the graph for execution
    Fibonacci_launch(args, context);

    // Exit when the graph execution completes
    CNC_SHUTDOWN_ON_FINISH(context);

    return 0;
}
```

Listing 4.2: Source code for *Main.c*. The highlighted lines were added manually, whereas all other lines were auto-generated.

```c
#include "Fibonacci.h"

/**
 * Step function definition for "compute_fib"
 */
void Fibonacci_compute_fib(cncTag_t i, int *x, int *y, FibonacciCtx *ctx) {
    //
    // OUTPUTS
    //

    // Put "z" items
    int *z = cncItemAlloc(sizeof(*z));
    /* TODO: Initialize z */
    *z = (i>1) ? *x + *y : i;
    cncPut_fib(z, i, ctx);

}
```

Listing 4.3: Source code for *Fibonacci_compute_fib.c*. The highlighted line was added manually, whereas all other lines were auto-generated.

```c
#include "Fibonacci.h"

void Fibonacci_cncFinalize(int *fib, FibonacciCtx *ctx) {
    /* TODO: Do something with fib */
    printf("Fibonacci(%d) = %d\n", ctx->n, *fib);
}

void Fibonacci_cncInitialize(FibonacciArgs *args, FibonacciCtx *ctx) {
    { // Prescribe "compute_fib" steps
        s64 _i;
        for (_i = 0; _i <= ctx->n; _i++) {
            cncPrescribe_compute_fib(_i, ctx);
        }
    }

    // Set finalizer function's tag
    Fibonacci_await(ctx);
}
```

Listing 4.4: Source code for *Fibonacci.c*. The highlighted line was added manually, whereas all other lines were auto-generated.

4.5 CnC Program Hierarchy

In this section, we present a prototype system for automatic inference of hierarchical structures within a CnC dependence graph, used for applying hierarchy-based optimizations. We assert that a *hierarchical organization* is essential for a program to achieve good performance when running at scale. This assertion is supported by past work [29, 87]. While one can hand-instrument such a hierarchical structure in a program, automatic inference of legal and useful hierarchies eases the burden on the programmer.

For a given CnC program, we automatically infer the set of legal hierarchies by iteratively applying restricted transformations on the intermediate graphs. While the set of all legal hierarchies holds theoretical interest, it is only possible to exhaustively examine all possible hierarchy options for the most trivial CnC application graphs; therefore, our search can be guided by a heuristic metric, which is used to identify potentially beneficial hierarchies. Possible metrics include the following:

1. Data locality (communication avoidance)
2. Improving parallel task prescription fanout
3. Item-lifetime scoping (ease of memory management)

In this work, we use data locality as the basis for our heuristic. To evaluate the effectiveness of our metrics in selecting useful hierarchies, we manually transform several copies of the input program to correspond with the granularity of the selected hierarchies.This transformation would be done automatically by a more mature version of the toolchain. We then compare the relative performance of the different versions, which should exhibit a similar trend to the heuristic metric scores.

4.5.1 CnC Collection Granularities

A CnC application is made up of a set of step collections, and a set of item collections. Although the collections have a definite granularity in their definitions in the graph, a sophisticated CnC compiler or runtime might choose to use different granularities than those given in the program source code for the actual execution of the program. Optimizing compilers that perform inlining, loop optimizations and data layout optimizations serve a similar function for traditional (non-graph-based) code.

Assume we have a simple CnC graph consisting of only two step collections, *X* and *Y*, each with a single tag component *i*. Using the notation from section 4.1.4, we would refer to instances from these collections as (X: i) and (Y: i), respectively. An optimizing CnC compiler could perform any of the following granularity-related transformations on these two step collections:

- **(X: i′) (Y: i)**: Chunk instances of step collection *X*, but do nothing to *Y*.
- **(X: i) (Y: i′)**: Chunk instances of step collection *Y*, but do nothing to *X*.
- **(X: i′) (Y: i′)**: Chunk corresponding instances of both *X* and *Y* identically.
- **(X: i′) (Y: i″)**: Chunk instances of step collection *X* with one chunking factor, and chunk instances of step collection *Y* with a different chunking factor.
- **(XY: i)**: Merge pairs of corresponding instances from step collections *X* and *Y*.

The compiler could also combine these transformations to perform more complex optimizations. Instances from two collections could be merged and then chunked, such as transforming (X: i) and (Y: i) from the example above into (XY: i′). Instances from a single collection could be chunked at multiple levels or across multiple tag components, such as chunking instances from a collection (Z: i, j) into (Z: i′, j″). Of course, these transformations are only legal if all dependences in the original program are preserved, as per the fundamental theorem of dependence [88].

4.5.2 CnC Hierarchy Defintions

We will now introduce a set of terms and definitions that we will use throughout the remainder of this thesis.

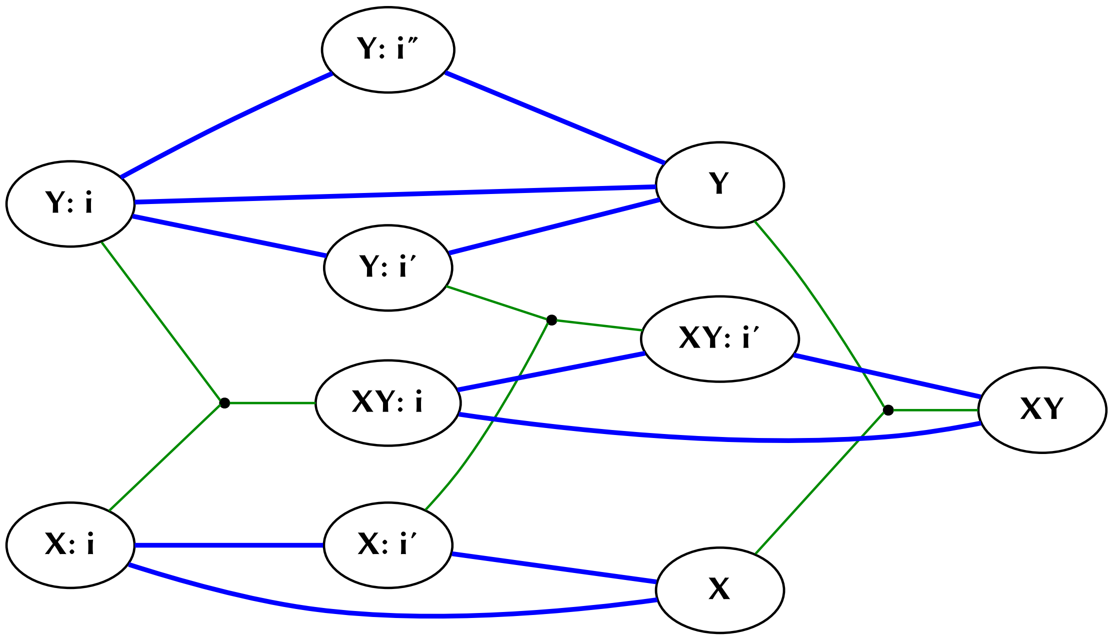
Homogeneous composition: The coarsening of a single collection in the graph by composing instances across the value of a single tag component. The transformation from

(X: i) to (X: i′) is an example of a homogeneous composition. If a single tag component is totally composed—i.e., all instances differentiated by that tag component are merged—then that tag component is eliminated. For example, the transformation from (X: i) to (X) totally composes the tag component *i*.

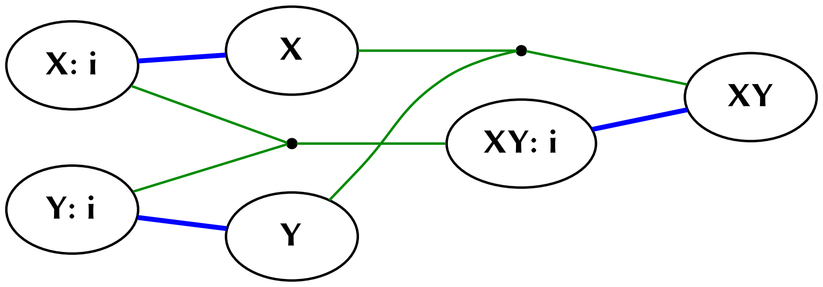
Heterogeneous composition: The coarsening of a pair of collections in the graph by composing pairs of corresponding instances from the two collections, resulting in a new composite step collection. The transformation of (X: i) and (Y: i) into (XY: i) is an example of a heterogeneous composition.

Hierarchy space: The union of all possible granularity choices for a CnC program. It constitutes a join-semilattice, with the granularity choices as the elements, and the homogeneous and heterogeneous composition options forming the partial ordering among the elements. The greatest element, or $\top$, is the singleton graph, where the entire CnC graph is composed into a single step instance. Since there exists a serial schedule for every dynamic CnC graph (although the serialization might not be known statically), it follows that such a singleton step exists for all graphs. Although the exact encoding of the step might differ, all encodings of the singleton step must be functionally equivalent due to the determinacy guarantees of CnC [89].

Figure 4.5 illustrates the hierarchy space of the simple CnC program discussed in section 4.5.1. Even for such a simple program, explicitly representing homogeneous composition options such as (Y: i′) and (Y: i″) significantly complicate the graph, even if multi-level chunking is not considered. In order to maintain simplicity and readability of such graphical representations of CnC hierarchy concepts, we assume that a homogeneous composition edge always represents the full range of corresponding legal transformations, up to and including totally composing a given tag component. Figure 4.5(a) demonstrates this simplified graphical representation.



(a) Hierarchy space representation including many homogeneous composition options.



(b) Simplified hierarchy space representation.

Figure 4.5: Two samples of graphical representation for the CnC hierarchy space of a simple CnC graph, corresponding to the example in section 4.5.1. The step granularity grows coarser from left to right. The **heavy blue edges** represent *homogeneous compositions* across a single tag component, whereas the green edges (with • join points) represent binary *heterogeneous compositions* of step collections. The simplified representation used in (b), which includes only *total* homogeneous compositions, is used throughout the remainder of this chapter.

Hierarchy: A set of valid granularity choices for the item and step instances in a CnC program. Note that each collection may have multiple levels of granularity choices in a given hierarchy, analogous to multi-level data tiling. In other words, each minimal element in the semilattice is dominated by exactly one maximal element in the hierarchy. This property follows from the intuition that any given collection that is coarsened through some set of *compositions* will still be a single collection after the coarsening transformations—although multiple collections from the original graph may comprise the resulting coarser-grained collection. More precisely, for any CnC hierarchy H in a hierarchy space (semilattice) L:[4]

$$D \equiv \{x \in H \mid \forall y \in H \; (y \preceq x \lor y \parallel x)\} \tag{4.2}$$

$$M \equiv \{x \in L \mid \forall y \in L \; (x \preceq y \lor x \parallel y)\} \tag{4.3}$$

$$\forall x \in M \; (\exists! y \in D \; (x \preceq y)) \tag{4.4}$$

Full hierarchy: A hierarchy in which no additional elements from the hierarchy space can be legally included (hence being *full*). In other words, for a full CnC hierarchy H in a heirarchy space (semilattice) L, equations (4.2) to (4.4) all apply, but with an additional constraint to ensure that any elements from the hierarchy space that are not included in the hierarchy must conflict with an element in the hierarchy:

$$x \,\mathrm{conflict}\, y \equiv x \parallel y \land \exists m \in L \; (m \prec x \land m \prec y) \tag{4.5}$$

$$\forall x \in L \; (x \notin H \iff \exists y \in H \; (x \,\mathrm{conflict}\, y)) \tag{4.6}$$

Hierarchy slice: A single granularity choice for each step and item collection in a CnC program. Note that a *hierarchy slice* is also a trivial case of a *hierarchy*, where each collection has a single granularity rather than simultaneously having multiple levels of granularity choices. In other words, for a CnC hierarchy slice H in a hierarchy space

[4] Note that $x \parallel y$ means that elements x and y are incomparable, i.e., $x \npreceq y \land y \npreceq x$ [60]. Additionally, $\exists! z$ denotes unique existential quantification of z, i.e., "there exists exactly one z" [90].

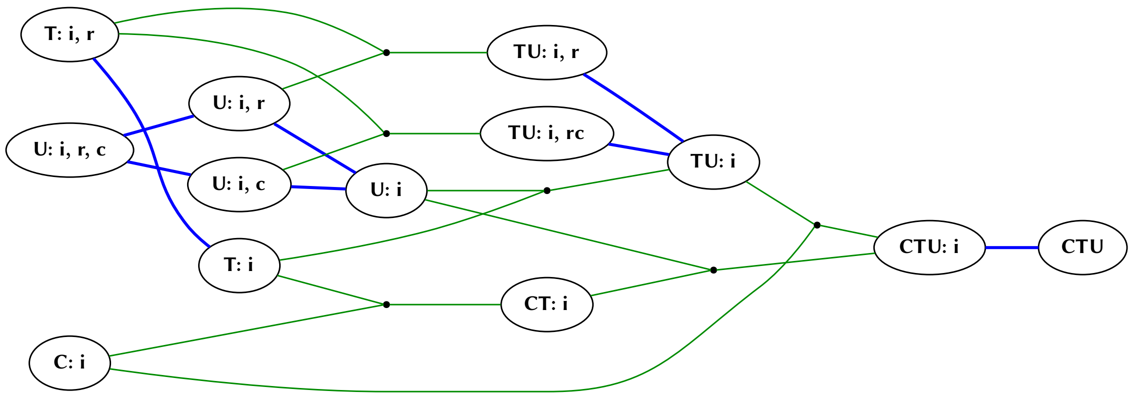


Figure 4.6: The hierarchy space for all computation steps in the Cholesky decomposition CnC kernel.

(semilattice) L, equations (4.2) to (4.4) all still apply, but with the addition of one more constraint that ensures each collection has a single granularity:

$$\forall x \in L \; (x \in H \implies x \in D) \tag{4.7}$$

4.5.3 Algorithmically Building Hierarchies

The equations in section 4.5.2 are useful for clearly defining the CnC hierarchy concepts; however, since the equations are framed as declarative logic expressions, it is not immediately obvious how one would efficiently derive the hierarchy space or the different types of hierarchy from an input CnC application. We now introduce a set of precise algorithms for CnC hierarchy derivations. These algorithms form the basis for the hierarchy-related tools implemented in our CnC toolchain.

To better illustrate the concepts from section 4.5.2, we will use a hierarchical version of the Cholesky decomposition kernel. Listing 4.5 shows the hierarchy space of the CnC graph specification for this application, and figure 4.6 shows the corresponding hierarchy space semilattice.

```
$context { int numTiles, tileSize; };

// Matrix tiles
[ double MC[]: i ];
[ double MT[]: i, r ];
[ double MU[]: i, r, c ];

// Sequential Cholesky step
( C: i )
 <- [ data1D @ MU: i, i, i ]
 -> [ MC: i+1 ];

// Trisolve step
( T: i, r )
 <- [ dataA1D @ MU: i, r, i ],
    [ dataB1D @ MC: i+1 ]
 -> [ MT: i+1, r ];

// Update step
( U: i, r, c )
 <- [ dataA1D @ MU: i, r, c ],
    [ dataB1D @ MT: i+1, r ],
    [ dataC1D @ MT: i+1, c ]
 -> [ MU: i+1, r, c ];

// Bootstrap computation
( $initialize: () )
 -> [ MU: 0, $range(#numTiles), $range(#numTiles) ],
    ( kComputeStep: () );

// Check results and print checksum
( $finalize: () ) <- [ MC: #numTiles ];
```

Listing 4.5: Graph specification for a Cholesky decomposition kernel in CnC. Note that *i* represents an iteration number, *r* represents a matrix tile row, and *c* represents a matrix tile column.

There are many possible transformations that we could apply at each step of the algorithm; however, for simplicity we restrict our model with the following choices and assumptions:

- We include only two types of transformations: homogeneous compositions across a single tag component of a collection, and heterogeneous compositions of disjoint collections. Future work may extend the foundational framework presented here with additional transformations and heuristics.
- All step-collection compositions in our hierarchy form new step-like collections. A broader definition of the CnC hierarchy space could also include graph-like collections, which may exhibit coroutine-like behavior. While such an abstraction might have useful applications for programmers reasoning about CnC graphs, it does not readily translate into a coarser-grained representation of the input graph. For this reason, we choose to only allow compositions that can be represented as coarser-grained steps in an optimized output graph.
- The programmer writes the application code at the finest granularity of interest for execution. This input code could already be tiled in some way, or it could be an element-wise computation. The CnC runtime does not distinguish between the two cases. Computation steps and data items are opaque to the runtime, meaning that any input program appears identical to an element-wise computation from the toolchain's perspective.
- All dependence functions are included in the input. These might be provided by the user, or automatically inferred using traditional dependence-analysis techniques [88]. These functions include the *produces*, *consumes*, *prescribes* and *prescribed-by* relationships for steps, whereas the functions for items include the *produced-by* and *consumed-by* relationships.
- The item collection structure is symmetric with the step collection structure, meaning that we can build a hierarchy for the step collections, and apply the same hierarchy to the item collections.

- If two tag components in different collections have the same name, then they must have a direct correspondence; i.e., when composing two collections, tag components with the same name may always be merged. This guarantee can be achieved by renaming distinct tag components before applying the hierarchy derivation algorithm.

Deriving the Hierarchy Space

Algorithm 4.1 describes the process used to derive the hierarchy space for a CnC program. Figure 4.6 was derived from the Cholesky domain specification using this algorithm. The intuition of the algorithm is simple: starting from the *base collections* (i.e., the input graph's elements), try adding a homogeneous composition for each tag components of each element, and try adding a heterogeneous composition for each element with each other element.

A collection can be homogeneously composed across a tag component if and only if removing that tag component from all dependence functions does not result in an indirect cyclic dependence. For example, (T: i, r) cannot be composed across i to form (T: r) because there is a conflicting input/output cycle: (T: 1, r) $\longrightarrow$ (U: 1, r, 2) $\longrightarrow$ (T: 2, r), which reduces to (T: r) $\longrightarrow$ (U: r, 2) $\longrightarrow$ (T: r) when the i components of the tags are removed. This is the basis of the `can_compose_across` method on line 10 of algorithm 4.1.

Similarly, two collections can be heterogeneously composed if and only if composing the two collections does not result in an indirect cyclic dependence. For example, (C: i) and (U: i) cannot be merged into a new collection (CU: i) because there is a conflicting input/output cycle: (C: i) $\longrightarrow$ (T: i) $\longrightarrow$ (U: i), which becomes (CU: i) $\longrightarrow$ (T: i) $\longrightarrow$ (CU: i) after the heterogeneous composition. This is the basis of the `can_compose_with` method on line 15 of algorithm 4.1.

The computational complexity of algorithm 4.1 is $O(|L|^2)$; i.e., it is quadratic in the number of elements in the hierarchy space. The space complexity of the algorithm is also $O(|L|^2)$, as a hierarchy space may have $O(|L|^2)$ edges in the case when virtually all compositions among elements are unrestricted. Note that the hierarchy space's element

```
    Input: CnC domain graph with complete dependence functions.
    Result: Join-semilattice comprising the hierarchy space for the input graph.
 1  let $BaseCollections$ be the set of all step collections in the input graph.
 2  let $HSpace$ be an empty data structure representing a CnC hierarchy space.
 3  let $Worklist$ be an empty queue of step elements to be processed.
 4  let $Processed$ be an empty list of step elements.
 5  $HSpace$ .add_all($BaseCollections$)
 6  $Worklist$ .enqueue_all($BaseCollections$)
 7  while not $Worklist$ .empty() do
 8      $Step \longleftarrow Worklist$ .dequeue()
        // Homogeneous compositions
 9      foreach $i$ in $Step$ .tag_components() do
10          if $Step$ .can_compose_across($i$) then
11              $Step' \longleftarrow Step$ .compose_across($i$)
12              $HSpace$ .add_derivation($Step, Step'$)
13              $Worklist$ .enqueue($Step'$)
        // Heterogeneous compositions
14      foreach $X \in Processed$ do
15          if $X$ .can_compose_with($Step$) then
16              $XStep \longleftarrow X$ .compose_with($Step$)
17              $HSpace$ .add_derivation($X, Step, XStep$)
18              $Worklist$ .enqueue($XStep$)
        // Record current element as processed
19      $Processed$ .add($Step$)
20  return $HSpace$
```

Algorithm 4.1: Hierarchy space derivation from a CnC graph.

count always dominates the tag component count of the individual elements. This is because each of those tag components are composed to create new elements in the hierarchy space; i.e., any element with n tag components will have at least n elements above it in the hierarchy space. Although the hierarchy space element count is exponential in relation to the number of base collections, the collection count is typically a small number in practice.

Deriving Hierarchy Slices

Algorithm 4.2 describes the process used to derive all hierarchy slices for a CnC program from that program's hierarchy space. The algorithm is easily described in terms of equations (4.2) to (4.4). We recursively search for slices by choosing to include or not include each element from the hierarchy space, adding elements to the partial slice that do not cause a violation of equation (4.2) (i.e., the base collection sets of the current slice and the new element do not intersect), with a base case yielding a complete slice whenever equation (4.4) is satisfied (i.e., all base collections in the hierarchy space are included in the current slice's base collection set).

The space complexity bound is proportional to the total number of hierarchy slices derived, or $O(|M| \cdot {}_{|L|}C_{|M|})$, where $|L|$ denotes the number of elements in the hierarchy space, and $|M|$ denotes the number of elements in the set defined in equation (4.3), which is simply the number of base collections. This is bound deduced from the fact that no slice can consist of more than $|M|$ elements from L. If implemented naïvely, the algorithm 4.2 has two recursive branches for each element in the hierarchy space, resulting an exponential time complexity; i.e., $O(2^{|L|})$. However, the algorithm can be optimized with dynamic programming, using the current index in the *Elements* list and the *base collections* set of the current partial slice as the memoization parameters. With this optimization, we potentially have one call per $\langle Index,\ Set \rangle$ pair. Since the number of possible base collection sets is $O(2^{|M|})$, the computational complexity in the optimized case is $O(|L| \cdot 2^{|M|})$. Since M is typically a very small subset of the elements of L, this optimization is important in practice.

```
   Input: The hierarchy space for a CnC application.
   Result: The list of all hierarchy slices in the hierarchy space.
 1 let $HSpace$ be the input data structure representing a CnC hierarchy space.
 2 let $Slices$ be an empty list.
   // Recursive function for finding all hierarchy slices
 3 function $find\_slices(Slice, Elements)$:
 4     if not $Elements$ .empty() then
 5         $X \longleftarrow Elements$ .first()
 6         $Elements' \longleftarrow Elements$ .rest()
           // Don't include element X, and recur
 7         $find\_slices(Slice, Elements')$
           // Try to include element X and recur
 8         if $X$ .base_collections() $\cap$ $Slice$ .base_collections() $= \varnothing$ then
 9             $Slice' \longleftarrow Slice \cup \{X\}$
10             if $Slice'$ .base_collections() $=$ $HSpace$ .base_collections() then
                   // Found a complete slice
11                 $Slices$ .add($Slice'$)
12             else // The current slice is incomplete, keep searching
13                 $find\_slices(Slice', Elements')$
   // Start the recursive search
14 $find\_slices(\varnothing, HSpace$ .elements()$)$
15 return $Slices$
```

Algorithm 4.2: Hierarchy slice derivation from a CnC hierarchy space. The result of algorithm 4.1 can be used as input for this algorithm.

Deriving Full Hierarchies

Algorithm 4.3 describes the process used to derive all full hierarchies for a CnC program from that program's hierarchy space. The basic intuition behind the algorithm is that each full hierarchy represents a single path from the top element down to each base element. The algorithm is recursive: each element aggregates the paths of the elements directly below it in the semilattice and prepends itself to each path, but heterogeneous compositions also require aggregating all possible combinations of the two sets of paths comprising the composition. Figure 4.7 shows all six of the full hierarchies that can be derived from the Cholesky graph's hierarchy space.

While the complexity of generating all the full hierarchies is very high, we can compute the corresponding count much more efficiently. Algorithm 4.4—which is structurally very similar to algorithm 4.3—computes the total number of full hierarchies in a hierarchy space. Note the symmetry between the Cartesian product computed on lines 7 to 10 of algorithm 4.3, and the product calculated in lines 6 to 8 of algorithm 4.4.

Algorithm 4.4 can be optimized by memoizing the sub-result of counting the hierarchy options under each element. This reduces the total number of recursive calls to $O(|L|)$, with one table lookup for each edge in the semilattice. The result is a $O(|L| + |E|)$ computational complexity (where $|E|$ is the number of edges in the semilattice), and $O(|L|)$ space complexity.

We can apply a similar optimization strategy to algorithm 4.3 by memoizing the sub-hierarchy results for each element, and reusing immutable sub-sequences of hierarchy elements to compose each of our full hierarchy results. This gives us a time complexity that is proportional to the total number of elements in each full hierarchy, which is a tight lower bound if we expect to process each element in each full hierarchy. The space complexity is similarly dominated by the memory used to store the generated results.

```
Input: The hierarchy space for a CnC application.
Result: The list of all full hierarchies in the hierarchy space.
// Recursive function for finding all full hierarchies
1  function find_full_hierarchies(Element):
2      let FullHierarchies be an empty list.
3      foreach X in Element .homogeneous_decompositions() do
4          foreach HX in find_full_hierarchies(X) do
5              FullHierarchies .add(HX .prepend(Element))
6      foreach ⟨X, Y⟩ in Element .heterogeneous_decompositions() do
           // Compute Cartesian product of X and Y hierarchies
7          foreach HX in find_full_hierarchies(X) do
8              foreach HY in find_full_hierarchies(Y) do
9                  HXY ⟵ HX .concat(HY)
10                 FullHierarchies .add(HXY .prepend(Element))
11     if FullHierarchies .empty() then
           // No decompositions (base case)
12         FullHierarchies .add(empty_seq() .prepend(Element))
13     return FullHierarchies
// Start the recursive search
14 let Top be the top element from the hierarchy space semilattice.
15 return find_full_hierarchies(Top)
```

Algorithm 4.3: Derivation of full hierarchies from a CnC hierarchy space. The result of algorithm 4.1 can be used as input for this algorithm. Note that the `concat` and `prepend` operations are functional (i.e., they return a new sequence).

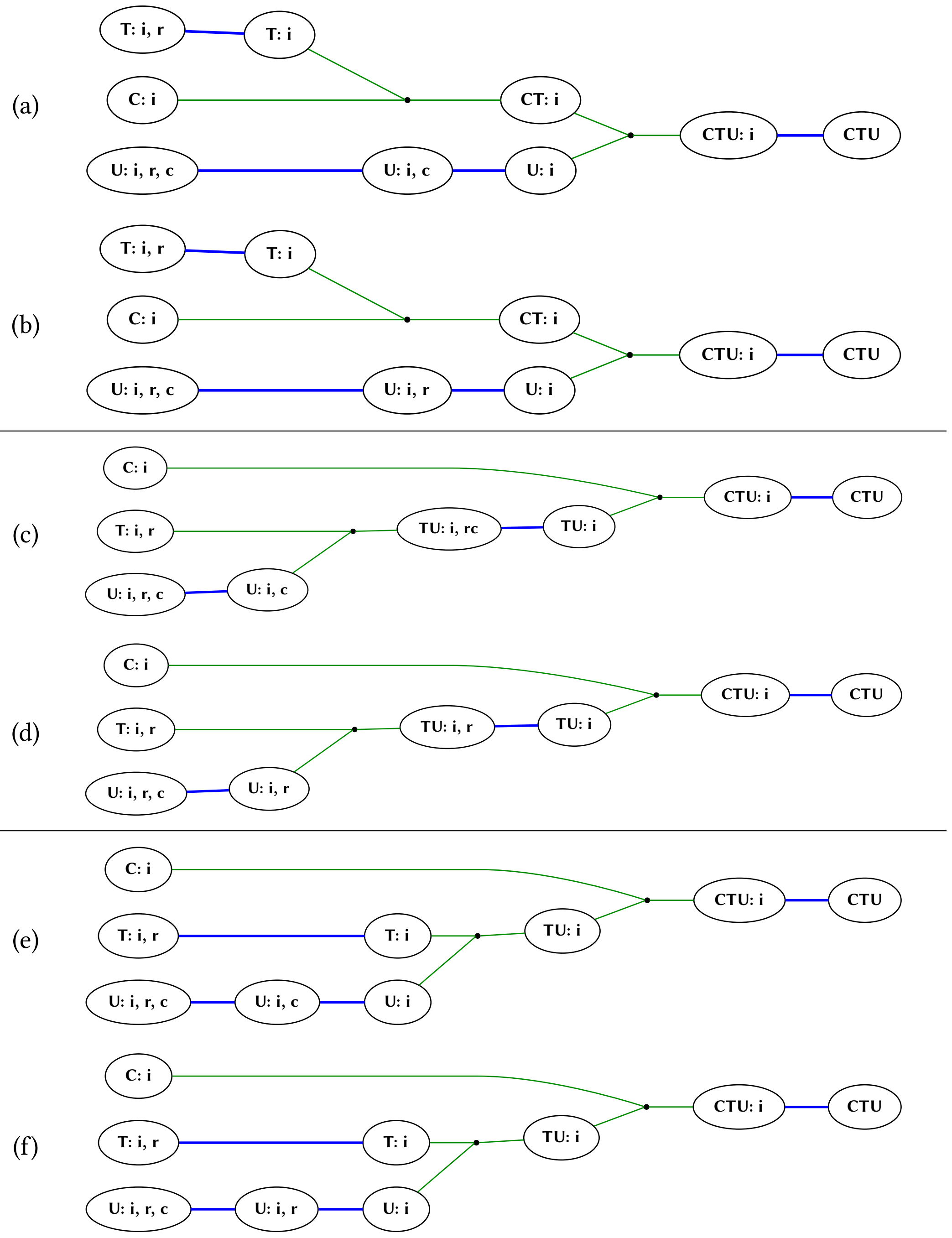


Figure 4.7: The six full-hierarchy variants of the Cholesky CnC graph specified in listing 4.5, corresponding to the hierarchy space in figure 4.6. Note that (a)–(b) differ from (c)–(f) in the choice of decomposition from (CTU: i), and (c)–(d) differ from (e)–(f) in the choice of decomposition from (TU: i). For each of the hierarchy pairs (separated by horizontal rules), the two hierarchies differ based on their choice of including either (U: i, r) or (U: i, c).

```
Input: The hierarchy space for a CnC application.
Result: The total number full hierarchies in the hierarchy space.
// Recursive function for counting all full hierarchies
1  function count_full_hierarchies(Element):
2      $Count \longleftarrow 0$
3      foreach $X$ in Element .homogeneous_decompositions() do
4          $Count \longleftarrow Count$ + count_full_hierarchies(X)
5      foreach $\langle X, Y \rangle$ in Element .heterogeneous_decompositions() do
6          $CX \longleftarrow$ count_full_hierarchies(X)
7          $CY \longleftarrow$ count_full_hierarchies(Y)
8          $Count \longleftarrow Count + CX \times CY$
9      if $Count = 0$ then
           // No decompositions (base case)
10         $Count \longleftarrow 1$
11     return Count
// Start the recursive count
12 let Top be the top element from the hierarchy space semilattice.
13 return count_full_hierarchies(Top)
```

Algorithm 4.4: Counting the full hierarchies in a CnC hierarchy space. The result of algorithm 4.1 can be used as input for this algorithm.

```
Input: The list of all full hierarchies for a CnC application.
Result: The set of all hierarchies in the hierarchy space.
1  let $FullHierarchies$ be the input list of all full hierarchies.
2  $Hierarchies \longleftarrow \emptyset$
   // Recursive function for finding all hierarchies
3  function $find\_hierarchies(Hierarchy, Elements)$:
4      if not $Elements$ .empty() then
5          $X \longleftarrow Elements$ .first()
6          $Elements' \longleftarrow Elements$ .rest()
           // Include element X, and recur
7          $find\_hierarchies(Hierarchy, Elements')$
           // Try removing element X
8          if $Hierarchy$ .can_remove($X$) then
9              $Hierarchy' \longleftarrow Hierarchy \setminus \{X\}$
               // Check if this hierarchy has already been seen
10             if $Hierarchy' \notin Hierarchies$ then
                   // Record this hierarchy and recur
11                 $Hierarchies \longleftarrow Hierarchies \cup \{Hierarchy'\}$
12                 $find\_hierarchies(Hierarchy', Elements')$
   // Start the recursive search
13 foreach $FH$ in $FullHierarchies$ do
14     $find\_hierarchies(FH, FH$ .elements()$)$
15 return $Hierarchies$
```

Algorithm 4.5: Hierarchy derivation from full CnC hierarchies. The output of algorithm 4.3 can be used as input to this algorithm.

Deriving All Hierarchies

Algorithm 4.5 describes the process used to derive all hierarchies for a CnC program from that program's full hierarchies.

The `can_remove` operation used on line 8 of algorithm 4.5 means that removing the given element from the current hierarchy would not cause the resulting set to violate equation (4.4). This can be efficiently checked if the base collections associated with each element in the hierarchy are precomputed, and we track the current element count for each base collection. If removing an element would cause a base collection counter to hit zero, then that element cannot be removed. Unfortunately, this algorithm can yield duplicate results as *find_hierarchies* is applied to each of the full hierarchies, which forces us to deduplicate the output using a set. If the hierarchies are represented as bitsets (with each bit corresponding to an index in the *Elements* list), then membership in the results set is easy to check; however, this implies that the set update operation on line 11 has a complexity proportional to $|L|$. This leads to a super-exponential compute complexity. The space complexity bound is also exponential, since the worst-case set of result hierarchies is similar to the power set of the elements of L.

Table 4.3 lists the element counts in the results of the algorithms in this section when run on our Cholesky example graph. These results can be reproduced via the `Cholesky.py` script included with the hierarchical Cholesky example in the CnC Framework repository.[5] The complexity of computing all hierarchies compared with the other hierarchy-related algorithms is demonstrated by the total hierarchy count, which is two orders of magnitude larger than the result set size of any of the other algorithms described in this section.

Table 4.4 shows some results for how hierarchy choices can impact performance. For these results, we used each of the possible hierarchy slices for the Cholesky application, and used the hierarchy information to distribute the steps across an 8-node cluster. Items were co-located with their producer steps. The different hierarchy slice choices resulted in vastly different performance, with execution times ranging from 3.2 seconds to over a

[5] https://github.com/habanero-rice/cnc-framework/blob/hierarchy2016/examples/hierarchy/Cholesky/

	Count
Hierarchy Base Collections	3
Hierarchy Space Elements	13
Full Hierarchies	6
Hierarchy Slices	17
All Hierarchies	1257

Table 4.3: Summary of sizes of the several classes of hierarchies, derived by applying algorithms 4.1 to 4.5 to our Cholesky example program graph.

Hierarchy Slice	Run-time
(CT: i) + (U: i, c)	3.2 seconds
(C: i) + (T: i, r) + (U: i, c)	5.6 seconds
(CT: i) + (U: i, r)	9.0 seconds
(CTU:)	41.6 seconds
(C: i) + (T: i, r) + (U: i)	60.3 seconds

Table 4.4: Performance results for a selection of the generated hierarchy slices for our Cholesky application. The hierarchy information was used for determining distribution in an 8-node cluster. The input matrix was 8100x8100, with items tiled at 50x50. All items were co-located with their producer steps.

minute. We feel that there is a great deal of potential for more applications of hierarchy in CnC and other programming models.

4.6 CnC Application Tuning

Programming directly to OCR requires using a very verbose, low-level C API. The verbosity of the API makes it difficult to experiment with different tuning strategies. Additionally, the separation of concerns in OCR is at more of a conceptual level than at a source-code level, since tuning hint code is embedded within the application code. The fact that the tuning hint code is sprinkled throughout the application source code makes it difficult to get a big-picture view of the tuning strategy being employed in an OCR program.

Fortunately, we can still get the best of both worlds. By using CnC on top of OCR, we get the expressiveness and clean separation of concerns provided by CnC, while still being able to take advantage of the runtime features and design in OCR.

Whereas OCR hints are specified inline throughout the application code, the CnC-OCR tunings are specified in a completely separate file. We believe this provides a better separation of concerns during the development process, and also makes it easier to get a big-picture view of an application's tunings (since all the tunings are in one place). We believe this is an important advantage available through higher-level abstractions built on top of OCR.

In this work, we demonstrate the utility of CnC as a high-level language on OCR for productivity and performance tuning. We define a set of five tuning annotations for CnC-OCR, describe the implementation of runtime-support for these tuning hints within OCR, and analyze the performance impact of the tuning hints on several CnC kernels. Our five tuning hints are as follows:

1. **Distribution functions**: Declare the affinity of the instances in a step or item collection as a function of the instance's tag components. (The affinity value typically corresponds to an MPI rank.)
2. **Step affinity with input**: A more declarative option for step instance affinities. Rather than declaring an explicit function, the instance can be affinitized to the same location as one of its input items.
3. **Step priority**: Give step instances priority weights, declared as a function of the step instance's tag components.
4. **Scheduler throttling**: Partition step collections into *stoker* steps, which create more work, and *quencher* steps, which complete some work. The *stoker* steps are preferred for stealing, whereas the *quencher* tasks are preferred for local execution.

5. **Item collection dense mapping**: Declare a mapping from an item collection's instance tags onto a dense array, allowing for more efficient storage and lookup.[6]

We also demonstrate the productivity boost in performance tuning—ascribed to our separation of concerns between the tuning and domain spec—provided by the ability to *mix in* several different tuning specifications with a single domain specification, and quickly switch among combinations of tunings to find an ideal configuration on a given platform or for a specific workload.

Finally, since our code-generation framework for CnC-OCR tuning was designed to support multiple targets, we have implemented support for running our CnC-OCR applications with both OCR and Intel CnC as compatible back-end runtime systems. For a subset of tunings that are available in both CnC-OCR and Intel CnC, we compare and contrast the performance results for application tunings.

4.6.1 Tuning Evaluation

We now demonstrate the benefits of our external tuning language by showing the large performance deltas between different tunings of the same application. We also compare the performance of the CnC-OCR version with an existing tuned version using Intel CnC. Additional tuning results for CnC-OCR are available in past work [91].

We use the Smith-Waterman kernel for this demonstration. The domain specification for this kernel is shown in listing 4.6, and the tuning specification is shown in listing 4.7. This tuning specifies that the items should be distributed row-block-cyclic, with each block consisting of 16 rows of the item values (where each item contains a tile of scores). Figure 4.8 shows the distributed performance of the Smith-Waterman kernel for both Intel CnC and CnC-OCR using a custom distribution tuning. By default (if no tuning is given), the items are distributed round-robin across the ranks based on the final component in the item's tag. For Smith-Waterman, the last tag component `j` index, which would

[6] In the general case, item collections are backed by a hashtable because the item collection is not guaranteed to be dense with regard to the tuple space of its instances' tags.

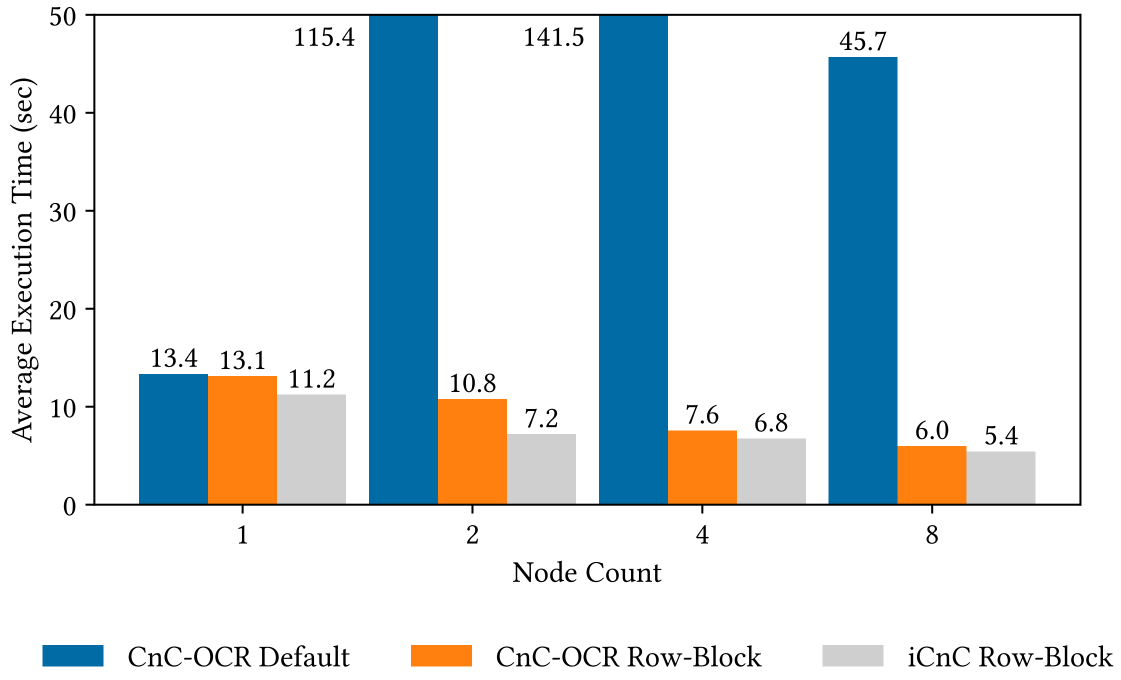


Figure 4.8: Smith-Waterman distributed tuning performance. Each input sequence has a length ≈200k, with tiles of size 177 × 153 and 1138 × 1322 total tiles. The distribution tuning is shown in listing 4.7.

```
[ int above[] : i, j ];
[ int left[]  : i, j ];
[ SeqData *data : () ];

( swStep: i, j )
 <- [ data: () ],
    [ above: i, j ] $when(i > 0),
    [ left: i, j ]  $when(j > 0)
 -> [ below @ above: i+1, j ],
    [ right @ left:  i, j+1 ],
    ( swStep: i+1, j ) $when(i+1 < #nth);
```

Listing 4.6: Domain specification for the Smith-Waterman kernel.

```
[ above ]: {
    distfn: (i / 16) % $RANKS
};

[ left ]: {
    distfn: (i / 16) % $RANKS
};

( swStep ): {
    placeWith: above
};
```

Listing 4.7: Distributed tuning specification for the Smith-Waterman kernel.

result in a column-cyclic distribution. We see that the row-block distribution results in much better performance than the naïve default distribution. This is due to the increased locality, and thus decreased network-communication overhead, afforded by the block-cyclic distribution. The fact that it was possible to get this large performance gain over the default simply by adding a few simple lines of code in our declarative tuning language obviously demonstrates the importance of performance-tuning functionality. However, the fact that we did not have to modify any of the existing application code, and that the tuning strategy is available as a stand-alone file (separate from the core computation logic of the application) is another powerful benefit.

4.7 Productivity in CnC-OCR

One way that CnC-OCR increases productivity is through code generation. By eliminating boilerplate code and generating statically-typed interfaces for the user's CnC domain specification, CnC-OCR eliminates a lot of the boilerplate and general verbosity that is typically present in an application written directly in OCR. As shown in table 4.5, the amount of user code in a CnC-OCR project—including the graph DSL code—is much less than an equivalent project written directly in OCR. Furthermore, since the CnC-OCR toolchain actually generates some of the user-code automatically as part of the skeleton project, the amount of user code can be argued to be less; e.g., we only had to add 4 lines to the generated code for the Fibonacci numbers example application in section 4.4, in addition to the code for the graph specification.

	Cholesky	**Smith-Waterman**
Base OCR	492	314
CnC-OCR	171	164
Reduction	65%	47%

Table 4.5: Comparison of Logical Lines of Code counts between the base OCR and CnC-OCR implementations of two kernels. These code counts were measured with UCC [82].

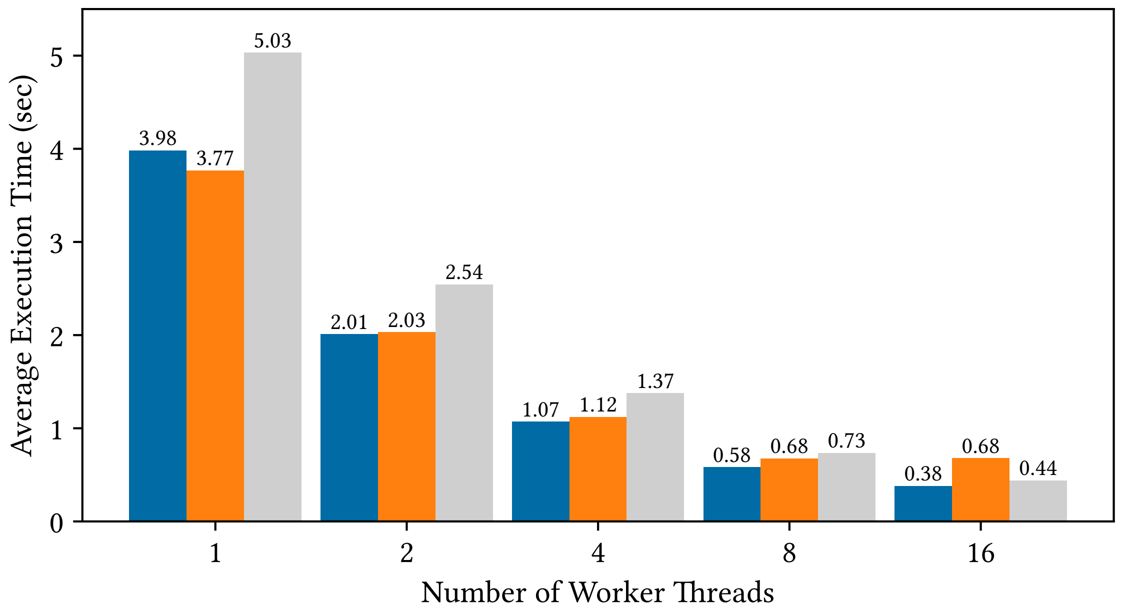


Figure 4.9: Single-node scaling with Cholesky, a 3000x3000 matrix with 50x50 tiles. Results are for Intel CnC, CnC-OCR, and OCR implementations of Cholesky.

In addition to the general reduction in lines of code, the static type information provided through CnC-OCR's code generation yields similar productivity gains to those from the `ocxxr` API; however, rather than using C++ templates, custom code is generated based on the type information derived from the user's domain specification.

These productivity benefits come with little overhead. As shown in figure 4.9, a carefully designed CnC-OCR application can meet or even beat the performance of a hand-coded OCR version. In this case, the increased performance can be attributed to the high-level abstractions provided by CnC-OCR, which allow for a more complex (but more efficient) coordination among tasks, whereas the hand-coded version opts for a simpler (but less performant) solution for coordination.

4.8 Related Work

There are only a few higher-level programming models that currently provide back-ends targeting OCR. Hierarchically Tiled Arrays for OCR (HTA) provides parallel, distributed array objects, and a set of highly-efficient linear algebra operations for the array objects [92]. HTA for OCR [76] removes the Intel Thread Building Blocks back-end from the previous C++ implementation, instead using OCR for dependence management and task scheduling. However, HTA does not currently integrate with the OCR data model since none of the HTA objects are allocated in OCR datablocks. HClib [59] similarly used OCR as a task and dependence management runtime, while eschewing datablocks and the OCR data model. Furthermore, as described in chapter 3, the blocking semantics of many of the HClib constructs can lead to deadlocks with the current OCR scheduler's limited legacy support for blocking tasks. Due to these limitations, current versions of HClib [37] no longer maintain an option to use OCR as the underlying tasking runtime.

Legion [4] and Regent [93] are two higher-level programming models that target the Realm runtime [15]. These two higher-level languages provide the productivity layer for the Realm runtime, much as is our goal for OCR with CnC-OCR. Since Legion and Regent have been co-designed with Realm, the assumptions in these higher-level models perfectly match those made in the lower-level Realm runtime. Work is currently underway for supporting the Legion programming model on OCR [77]; however, there are some mismatches between the concepts assumed by Legion and those provided by OCR, which will most likely require reworking the Legion programming model to adapt it to OCR. The effort to unify Legion's data model and blocking task semantics with the OCR programming model are still ongoing.

R-Stream [94] is a production-grade polyhedral optimizing compiler developed by Reservoir Labs. One feature of R-Stream is automatic translation of legacy C programs to OCR. The compiler has the usual limitations of polyhedral analysis and optimization (although their implementation is more scalable than academic alternatives), but that does not diminish the utility of R-Stream. It might not be possible to apply the R-Stream optimizations to a full application (due to limitations in analyzability and program size), however, we believe that there is good potential for using R-Stream in conjunction with CnC-OCR to optimize individual step function implementations.

There is a large amount of previous work on tuning for CnC. The flagship Intel CnC implementation provides several tuning options as part of its C++ API [28]. Our toolchain uses iCnC's tuning APIs when generating iCnC back-end code. While tuning code is not declared in a separate specification file, a separation of concerns can still be maintained since all tuning code is implemented in discrete tuning objects, which can be stored in separate source code files if desired. Knobe and Burke previously described a hypothetical declarative tuning language for CnC [95]. A variation of that proposed tuning language, specifically targeting tuning for distribution and hierarchy, was later implemented on a fork of the CnC-OCR codebase [29].

While our work is the first to automatically generate and select explicit hierarchies for a CnC program, there has been past work that performs related optimizations. Sharma et al. did auto-tuning for CnC programs, demonstrating a technique for automatic distribution function selection [96]. Automatic tiling, and other polyhedral optimizations, have also been implemented for the CnC programming model in DFGL [79] and PIPES [97]. Tuned-CnC allows the programmer to manually specify hierarchical affinity groups [29]. Liu et al. experimented with manual tiling and step fusion optimizations in CnC for LULESH [98].

4.9 Future Directions

The order of tag components as specified in a tag collection declaration do not restrict the underlying storage representation of individual items. For example, individual items in the collection $[\mathsf{Matrix}\colon \mathsf{row}, \mathsf{column}]$ could be stored in row major, column-major, or some other arbitrary order. Thus it is clear that an item collection specified at the finest granularity constitutes a layout-agnostic data representation. To improved locality for CnC applications, it follows that we could use an automatic layout selection framework (such as the framework presented by Sharma et al. [99]) to compute the metric value for selecting an optimal data hierarchy.

The long-term vision is to see this work integrated into an advanced toolchain, where the entire process would be automated and transparent. New hierarchies could be selected at runtime based on varying trends in the data, or a change of performance goals communicated to the runtime.

4.10 Summary

In this chapter we presented CnC-OCR, a higher-level programming model and productivity environment for OCR. We described the implementation of the CnC-OCR toolchain, and demonstrated the benefits to programmer productivity and the improved separation of concerns attained by the use of the CnC domain specification for dependence coordination, as well as the external specifications for performance tuning. We defined a set of rules for automatic generation of CnC application hierarchies, and demonstrate the benefits of hierarchy for distributed memory locality. We showed that the performance of CnC-OCR applications is comparable to much more verbose hand-coded OCR solutions, and is competitive with an existing production-grade CnC implementation.

Chapter 5

Conclusions and Future Work

In this thesis, we identified three problems associated with emerging programming models for extreme-scale runtimes—specifically the Open Community Runtime (OCR)—and presented solutions for each problem. The solutions are given in terms of new abstractions, either at the level of the runtime, user APIs, or the application programming model. We demonstrated the benefits and practicality of our proposed solutions by measuring the overhead of the new abstractions against an appropriate performance baseline for the given feature. Where appropriate, we also demonstrated the potential gains in productivity by measuring the lines of code necessary to write equivalent programs with and without our enhancements to the programming model and APIs. Table 5.1 summarizes the contributions in this thesis to the OCR programming model, presented in the context of other HPC programming models that are currently under development for future extreme-scale hardware.

	Pointer Safety	**Blocking**	**High-level languages**
Charm++	Programmer's responsibility	ucontext	Not yet supported
HPX	Programmer's responsibility	Fibers	Not yet supported
OCR	Static and dynamic checks	Several options	CnC
Realm	Programmer's responsibility	ucontext	Legion, Regent
UPC++	No data migration	SPMD blocking	Not yet supported

Table 5.1: Runtime Feature Comparison

5.1 General Conclusions

Sections 2.12, 3.7 and 4.10 summarized our conclusions on the effectiveness and practicality of our solutions to the problems presented in those respective chapters. In addition to those specific discussions, we now make three general conclusions about encouraging and supporting application development on extreme-scale runtime systems.

No one wants to be a "hero"

Higher-level languages and libraries are a critical component of a runtime ecosystem. Forcing application writers to become "hero programmers"—coding directly to a verbose, low-level API—does not encourage growth in the developer community around the runtime. If the application-facing API is just as complex as MPI+OpenMP, then it is hard to justify adopting a new programming model. While some runtimes may gain some traction by showcasing impressive performance with a carefully-tuned flagship application— unless new developers are provided with the tools to develop basic functionality quickly, and then incrementally fine-tune for performance and scalability, then the users won't be able to reproduce the advertised success. While a higher-level language or library might not allow users to obtain 100% of the optimal performance, a clear path to get a functioning application quickly, and then achieve incremental performance gains via a reasonable amount of additional work is a much more friendly path for new developers.

One size doesn't fit all

Providing alternative implementations help avoid forced choice between productivity and performance. For example, providing both the thread-based worker and fiber-based worker alternatives for the Habanero-C runtime scheduler allows the application programmer to develop and debug using the slower, but more easily inspectable threaded version, and then switch to the fiber-based version for production runs. Providing multiple, compatible implementations of a runtime that are tailored for development, debugging or production eases the burden on the application programmer; however, this convenience for the users comes at the additional cost of maintaining multiple implementations for the runtime team. Depending on the size and scope of the project, multiple implementations may not be possible (at least initially), but a runtime project that is serious about gaining users and supporting application development would be wise to invest time in productivity features as well as performance enhancements.

Concerns should be separate

Separation of concerns, as demonstrated with CnC-OCR, is simply an extension of a well-accepted principle of engineering: divide your problem up into smaller pieces, and attack them one at a time. Being able to reason about a complex application at different levels without being weighed-down with higher- or lower-level details makes an application easier to understand, which in turn should lead to better code and fewer bugs. Additionally, having separation between high-level dependencies among application components, low-level implementation details of individual modules, and platform-specific performance tuning strategies means that new developers and code reviewers also have less of a barrier to understand the software system. While the method of literally splitting the high-level graph specification, the step implementations and the tuning annotations into separate files (as done in CnC-OCR) may not be practical in other projects, providing tools to identify and extract the relevant information from an application would probably be just as useful. For example, having the ability to take an execution log and generate a graphical overview of the high-level flow of data and control in an application would give all of the same benefits as having a CnC-OCR graph specification. This also eases some of the burden on the developers for documenting the application behavior, and avoids the problem of the documentation being invalidated when the application is refactored.

5.2 Future Work

The earlier chapters include ideas for future work that directly builds on details of the solutions and techniques described in those chapters. We now discuss possible future research that relates to the overarching themes in this thesis.

The work in chapters chapters 2 and 3 addressed two big programmability problems for OCR applications: compatibility of many C++ prevalent features, and correctly supporting blocking synchronization constructs. While we have presented possible solutions to these two problems, there are still a plethora of other issues that have yet to be tackled. For example, our technique for sanitizing pointers stored in OCR datablocks assumes that the application data has already been partitioned into datablocks. Finding such a partitioning—especially one that works well in a distributed system—is not an easy task. Techniques for automatically finding an optimal partitioning, or even suggesting a set of partitions to a tuning expert, would be invaluable to the OCR application developer. Similarly, compiler or toolchain support for automatically transforming long-lived, blocking tasks into short-lived, non-blocking tasks would be very helpful for improving resilience guarantees for large-scale fault-prone systems.

Finally, while we presented CnC-OCR in chapter 4 as a higher-level productivity layer for OCR application development—as discussed in the previous section—there is seldom a one-size-fits-all solution to any problem. While CnC-OCR is a good fit for some HPC applications, there are undoubtedly many applications that would be much better expressed using some other programming model; therefore, it would be beneficial to develop additional higher-level productivity layers on top of OCR in order to give application developers multiple options. Two such options that are currently under investigation are implementations of Legion [4, 77] and Chapel [100] for OCR.

5.3 Possible Applications in Other Runtime Systems

While the techniques presented in this thesis have mainly focused on applications for OCR and Habanero-C, they also have value for potential future integration with other extreme-scale runtime systems.

The relative pointer object encodings and toolchain presented in chapter 2 were applied in the context of OCR; however, as explained in section 2.2.1, the pointer-object encodings presented in chapter 2 would be useful in any system using one-sided communication. Furthermore, other runtimes that use relocatable data blocks could also benefit by supporting our position-independent pointer variants. For example, although the Legion programming model (built on top of the Realm runtime) supports a position-independent pointer-object type for use within its relocatable data *regions*, it requires the target *region* handle to be provided when dereferencing one of these pointer objects.[1] Legion programmers could benefit from the increased productivity afforded by our `BasedPtr` types if the Legion system supported automatic *region* lookup for an embedded handle within the pointer object, or by using our `RelPtr` type for pointers with targets that are guaranteed to be in the same *region*.

The blocked-worker compensation strategies described in chapter 3 were presented in the context of the Habanero-C programming paradigm; however, we are currently working on implementing a subset of the strategies in OCR to support blocking APIs in the Legion on OCR project [77]. Additionally, the runtimes in table 5.1 listed as using the *ucontext* library (i.e., Charm++ and Realm) would gain the development and debugging benefits described in section 3.5.5 by implementing an alternative worker scheme using kernel threads. Swapping the *ucontext* library (deprecated from POSIX in 2004) for the Boost.Context-based fibers support used in our implementations would also[2] Finally, the *finish-helping* optimiza-

[1] See `logical_regions.cc` in the Legion/Realm tutorial for a description of the `ptr_t` type and an example of its usage with a local *region*:
https://github.com/StanfordLegion/legion/blob/legion-17.05.0/tutorial/04_logical_regions/

[2] The Boost.Context performance benchmarks report speedups of up to $20x$ compared to the deprecated *ucontext* library: http://www.boost.org/doc/libs/1_59_0/libs/context/doc/html/context/performance.html

tion introduced in section 3.4.2 can be applied in other runtimes using the `async`/`finish` model or similar constructs parallelism. The `async`/`finish` constructs in UPC++ could be implemented to take advantage of this optimization; however, current drafts of the UPC++ v1.0 specification [101] indicate that UPC++ is moving away from supporting blocking constructs and instead requiring users to manually CPS-transform their code (i.e., all of the UPC++ 1.0 APIs are non-blocking). There are other multitasking runtimes that support similar constructs that could benefit from the finish-helping optimization, such as X10 [31] (which also uses `async`/`finish`), and Chapel [100] (for the `cobegin` statement).

The high-level CnC programming model work presented in chapter 4 mainly focused on our CnC-OCR implementation; however, as a high-level programming model, CnC can be—and has previously been—ported to run on top of other runtimes (as long as they have sufficient support for event-driven constructs). As mentioned in section 4.3.5, there is a fork of the CnC-Framework that targets HPX-5 [86], but there also is a project called HPXnC that ports the Intel CnC API onto Stellar-HPX [102]. These ports of the CnC programming model bring all of the benefits of CnC (e.g., separation of concerns and explicit hierarchy) to the HPX-5 and Stellar-HPX runtimes. CnC should also map well onto the actor-like Charm++ model, as well as the event-driven UPC++ and Legion/Realm tasking models; however, to the best of our knowledge no implementation of CnC currently exists for any of those runtimes. Again, adding any of those runtimes as a new backend target for the CnC-Framework would allow existing CnC applications to work on those runtimes, and—assuming the tuning support is also properly implemented—allow users to automatically generate tuned application code for any (or all) of the supported backend runtimes.

Appendix A

Global Helping Deadlocks in OCR

Although our discussion of how the global-helping optimization can introduce new deadlocks (chapter 3) was focused on HClib, the same issue manifests in other runtimes. Specifically, global-helping induced deadlocks are a known problem in OCR. The Traleika Glacier implementation of OCR—which shares a common heritage with HClib—also uses the global helping[1] optimization within the runtime, but it is only enabled by default in distributed-memory (x86-mpi). Rather than supporting blocking operations in the application code, the goal was to hide the latency of synchronous network communication by running another OCR task while the runtime awaits a response from a remote OCR peer process. Although this application seems fairly innocuous, it is possible to construct a legal OCR program that can deadlock when using the global-helping optimization, but that would never deadlock without the optimization. Listing A.1 shows one such program.[2]

Although the global-helping optimization is not enabled by default for shared-memory OCR builds, it is often enabled in order to support blocking calls in legacy APIs. For example, the Legion-OCR project [77] uses this functionality to support blocking a task until a given event has been triggered. The naïve recursive Fibonacci example distributed with Legion will almost deadlock when run on Legion-OCR if the global-helping-based blocking support is used. To address this problem, a branch of OCR is currently under development to support legacy blocking constructs for Legion-OCR using fibers integration,[3] which is

[1] The global helping optimization is referred to using the terms *master-helper* or *work-shifting* in the context of OCR. We use the term *global helping* because we find it more intuitive, and it allows us to extend the **-helping* naming pattern to similar optimizations (e.g., *finish helping*).

[2] See https://xstack.exascale-tech.com/redmine/issues/964 for original bug report, along with links to the original version of the deadlock example code.

[3] https://xstack.exascale-tech.com/git/public?p=ocr.git;h=refs/heads/sandbox/nbvrvilo/blocking-fibers

based on our work integrating fibers with HClib. Fibers are also used by the flagship Legion implementation from Stanford [4], making the application of fibers in Legion-OCR an obvious solution. However, it's interesting to note that the Stanford Legion implementation uses the *ucontext* library to support fibers, which is not only deprecated from the POSIX standard (removed in 2004), but also up to $20x$ slower than using the Boost.Context library for fibers.[4] Therefore, the flagship implementation would likely also benefit by switching to the Boost.Context library to support blocking calls via fibers. However, since blocking calls are infrequent in well-written Legion code, the impact on most real-world applications would probably be minimal.

[4] The Boost.Context performance benchmarks report speedups of up to $20x$ compared to the deprecated *ucontext* library: http://www.boost.org/doc/libs/1_59_0/libs/context/doc/html/context/performance.html

```c
#include "ocr.h"
#define ENABLE_EXTENSION_AFFINITY
#include <extensions/ocr-affinity.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>

#define SZ_32MB ((size_t)(1L << 25))

#define SLEEP() sleep(2)

typedef struct {
    volatile u8 flag;
    ocrGuid_t signal;
    ocrGuid_t affinities[2];
} DbData;

ocrGuid_t aEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    DbData *data = depv[0].ptr;
    ocrGuid_t *affinities = data->affinities;
    char name[2] = { paramv[0], 0 };
    assert(name[0] == 'A');

    PRINTF("%s  started ...\n", name);

    ocrHint_t p1DbAffinityHint;
    ocrHintInit(&p1DbAffinityHint,OCR_HINT_DB_T);
    ocrSetHintValue(&p1DbAffinityHint, OCR_HINT_DB_AFFINITY,
    ↪  ocrAffinityToHintValue(affinities[1]));

    ocrEventSatisfy(data->signal, NULL_GUID);

    // Create a nice, big, remote datablock.
    // On x86-mpi, this operation will cause the current worker thread
    // to perform a "work-shift" to try to keep busy while communicating
    // with the remote policy domain.
    ocrGuid_t Y_DbGuid;
    void *Y_Ptr;
    ocrDbCreate(&Y_DbGuid, &Y_Ptr, SZ_32MB, DB_PROP_NONE, &p1DbAffinityHint,
    ↪  NO_ALLOC);

    PRINTF("%s (DB created) ...\n", name);

    data->flag = 1;

    SLEEP();
    PRINTF("%s  ended ...\n", name);
    return NULL_GUID;
}

ocrGuid_t bEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    DbData *data = depv[0].ptr;
    char name[2] = { paramv[0], 0 };
    assert(name[0] == 'B');

    PRINTF("%s  started ...\n", name);

    // NOTE: The OCR v1.1.0 spec specifically states in section 1.1.4
    // that it is legal for OCR programs to contain data races.
    // Furthermore, this statement is made specifically in the context
    // of defining Read-Write Mode access to datablocks, implying that
    // such accesses may race. I.e., as per the spec, this
    // busy-loop on the "flag" value is completely legal.
    while (data->flag == 0) {
        PRINTF("%s is spinning...\n", name);
        SLEEP();
    }

    SLEEP();

    PRINTF("%s  ended ...\n", name);
    return NULL_GUID;
}

ocrGuid_t shutdownEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    PRINTF("shutdownEdt  started ...\n");

    ocrShutdown();

    PRINTF("shutdownEdt  ended ...\n");
    return NULL_GUID;
}

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    //create templates
    ocrGuid_t aEdtTemplateGuid, bEdtTemplateGuid, shutdownEdtTemplateGuid;
    ocrEdtTemplateCreate(&aEdtTemplateGuid, aEdt, 1 /*paramc*/, 1 /*depc*/);
    ocrEdtTemplateCreate(&bEdtTemplateGuid, bEdt, 1 /*paramc*/, 2 /*depc*/);
    ocrEdtTemplateCreate(&shutdownEdtTemplateGuid, shutdownEdt, 0 /*paramc*/, 2
    ↪  /*depc*/);

    ocrGuid_t affinities[2];
    u64 affinityCount = 2;
    ocrAffinityGet(AFFINITY_PD, &affinityCount, affinities);
    assert(affinityCount == 2);

    // EDT Hint
    ocrHint_t p0EdtAffinityHint;
    ocrHintInit(&p0EdtAffinityHint,OCR_HINT_EDT_T);
    ocrSetHintValue(&p0EdtAffinityHint, OCR_HINT_EDT_AFFINITY,
    ↪  ocrAffinityToHintValue(affinities[0]));

    // DB hint
    ocrHint_t p0DbAffinityHint;
    ocrHintInit(&p0DbAffinityHint,OCR_HINT_DB_T);
    ocrSetHintValue(&p0DbAffinityHint, OCR_HINT_DB_AFFINITY,
    ↪  ocrAffinityToHintValue(affinities[0]));

    ocrGuid_t signalGuid;
    ocrEventCreate(&signalGuid, OCR_EVENT_STICKY_T, EVT_PROP_NONE);

    ocrGuid_t xDbGuid;
    DbData *xPtr;
    ocrDbCreate(&xDbGuid, (void**)&xPtr, sizeof(*xPtr), DB_PROP_NONE,
    ↪  &p0DbAffinityHint, NO_ALLOC);
    xPtr->flag = 0;
    xPtr->signal = signalGuid;
    xPtr->affinities[0] = affinities[0];
    xPtr->affinities[1] = affinities[1];
    ocrDbRelease(xDbGuid);

    // create EDTs A@P0 and B@P0
    u64 A = 'A', B = 'B';
    ocrGuid_t aEventGuid, bEventGuid;
    ocrGuid_t aEdtGuid, bEdtGuid;
    ocrEdtCreate(&aEdtGuid, aEdtTemplateGuid, 1, &A, EDT_PARAM_DEF, NULL,
                  /*prop=*/EDT_PROP_NONE, &p0EdtAffinityHint, &aEventGuid);

    ocrEdtCreate(&bEdtGuid, bEdtTemplateGuid, 1, &B, EDT_PARAM_DEF, NULL,
                      /*prop=*/EDT_PROP_NONE, &p0EdtAffinityHint, &bEventGuid);

    ocrGuid_t shutdownEdtGuid;
    ocrEdtCreate(&shutdownEdtGuid , shutdownEdtTemplateGuid , EDT_PARAM_DEF, NULL,
    ↪  EDT_PARAM_DEF, NULL,
            /*prop=*/EDT_PROP_NONE, &p0EdtAffinityHint, NULL);

    ocrAddDependence(aEventGuid, shutdownEdtGuid, 0, DB_MODE_RO);
    ocrAddDependence(bEventGuid, shutdownEdtGuid, 1, DB_MODE_RO);

    // A and B depend on X in RW mode (i.e., they may access it concurrently)
    ocrAddDependence(xDbGuid, aEdtGuid, 0, DB_MODE_RW);

    ocrAddDependence(xDbGuid,  bEdtGuid, 0, DB_MODE_RW);
    ocrAddDependence(signalGuid, bEdtGuid, 1, DB_MODE_RO);

    return NULL_GUID;
}
```

Listing A.1: Sample program that will deadlock with the default configuration for distributed memory OCR (x86-mpi), which uses the global-helping strategy. This example uses only core OCR API (no legacy blocking extensions), and the behavior is legal as per the OCR v1.1.0 specification.

Appendix B

Introduction to CnC

Concurrent Collections (CnC) is a system for describing the structure of parallel computation, or coordinating the data- and control-flow between the individual steps of a computation [70, 89]. A CnC application specifies a set of discrete step functions, and the data collections used as input to and output from those step functions.[1] The CnC coordination language describes the relationship between a specific invocation of a step function, its input and output data, as well as parent/child relationships between to other step function invocations.

B.1 Key Properties of CnC

In this section we outline several distinctive characteristics of the CnC programming model. These are the same characteristics that make the CnC programming model well suited for expressing large-scale parallel computations. The characteristics include graph representation, single-assignment data, monotonically growing state, discrete computation steps, and side-effect-free computation steps.

B.1.1 Graph Representation of the Application

A fundamental characteristic of a CnC application is that the entire computation flow is represented as a graph. An application is partitioned into collections of computation steps and data items, each of which describe a class of step (computation) or item (data) instances.

[1] This model varies slightly from the traditional CnC model in that it lacks *control collections*; however, this elision in our model reflects the absence of control collections in the Habanero variants of CnC developed at Rice University, on which this work is based. For a brief overview of control collections, and a discussion of the equivalence of this simplified CnC model with the traditional model, please see appendix C.

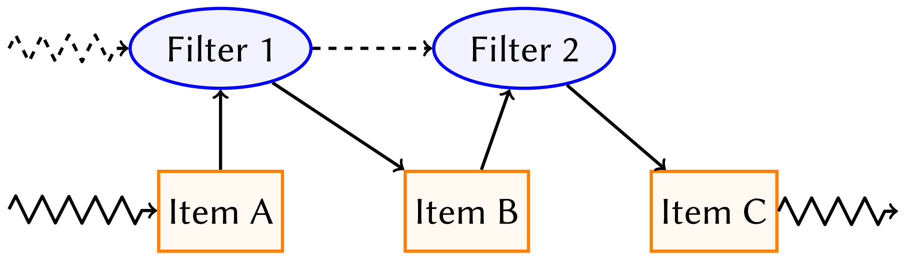


Figure B.1: The abstract graph representation of a simple data-filtering CnC application. The two ellipses represent step collections for two separate levels of filtering. The three rectangles represent item collections for holding all of the input data (*Item A*), the results of the first filter pass (*Item B*), and the final output from the second filter pass (*Item C*). Solid edges represent puts to and gets from item collections. Dashed edges represent prescriptions (creation) of new step instances. Jagged edges represent the interactions with the application environment that encloses the CnC graph.

These collections serve as the nodes of the graph. The *prescribe* (step creation), *put* (item creation) and *get* (item read) relationships among the collections are represented as edges in the graph. Figure B.1 shows a graph representation for a simple CnC application. By default, we mean a static program graph, when referring to a CnC graph. When necessary, we will differentiate between a static CnC graph, which defines a CnC program, and a dynamic CnC graph, which defines a CnC program execution.

By providing a high-level graphical representation of the application, the user provides the CnC runtime with all the necessary information to automatically track the incremental progress of the application. In the case of a failure, the runtime can restart the computation by simply restarting all of the computation steps that were running at the time of the failure, and providing the input data for those steps to run to completion.

B.1.2 Single-Assignment Data

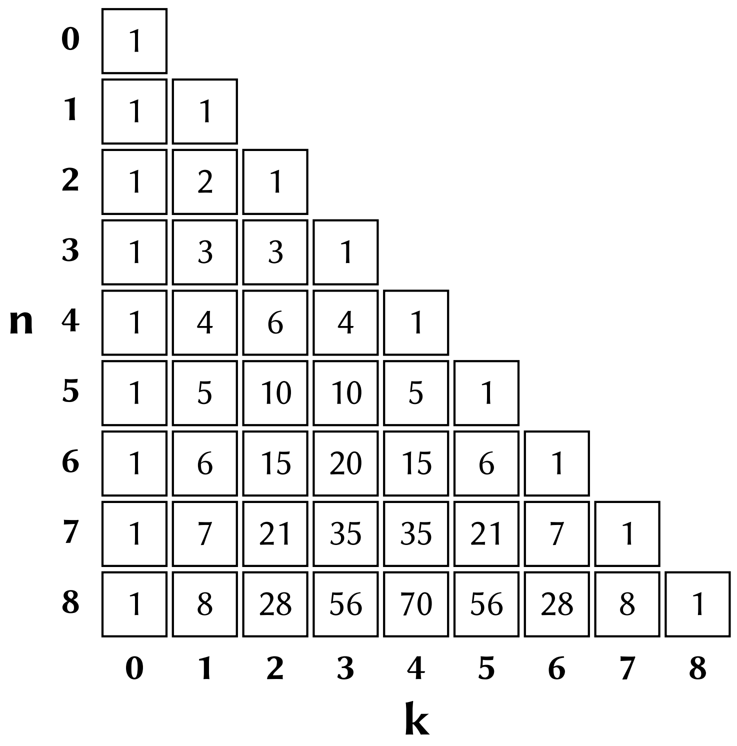
In a traditional imperative computation model, an application calculates incremental solutions to a problem by updating (or mutating) its in-memory data, eventually resulting in the final output. In such a system, it is possible to checkpoint an inconsistent state if some data is updated during the process of saving the checkpoint, such that part of the checkpoint reflects the update and part does not. In contrast, CnC takes a functional rather than imperative approach to modeling state. All data available at the level of the CnC computation graph is single-assignment, meaning that once a data item is created it is never updated. An individual computation step is free to mutate data local to that step, but all such mutations must be fully encapsulated within the step.

B.1.3 Monotonically Growing State

A property that follows from the single-assignment property is the monotonicity property. Since data cannot be updated after appearing in the graph, the overall state of a CnC application appears to only add new data, never removing or mutating previous data. A CnC implementation can optionally free data that is no longer required, though this process can, in general, be more complicated than garbage collection in functional languages [103].

B.1.4 Discrete and Side-Effect-Free Computation Steps

A traditional application may be implicitly divided into several logical computations, but the CnC programming model makes these divisions explicit. The CnC runtime takes advantage of discrete computation steps to run computation steps in parallel on multicore hardware. The fact that CnC applications have discrete computation steps with explicit inputs allows us to restart any given computation at the CnC step granularity. In addition, computation steps in CnC are side-effect-free because the only observable outputs of CnC steps are their items put and steps prescribed. This means that if a computation failed mid-step, there is no possibility that some incremental updates made by the step will corrupt the global CnC graphs state. These properties allow us to safely restart an application that failed at any point in the computation.



n \ k	0	1	2	3	4	5	6	7	8
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5	1	5	10	10	5	1			
6	1	6	15	20	15	6	1		
7	1	7	21	35	35	21	7	1	
8	1	8	28	56	70	56	28	8	1

Figure B.2: The first nine rows of Pascal's Triangle. The entries of Pascal's Triangle correspond to the binomial coefficients, such that the entry at row n, column k is ${}_nC_k$.

B.2 A Sample CnC Application

To better describe the CnC programming model, we now introduce a simple CnC application as an example. This sample application computes binomial coefficients—i.e., the values of ${}_nC_k$ (n choose k)—via Pascal's Triangle.

B.2.1 Review of Pascal's Triangle

Figure B.2 shows the first nine rows of Pascal's Triangle. If $P(n, k)$ is the value at row n, column k of Pascal's Triangle (where both the row and column numbers are zero-based), then for all $n \geq k \geq 0$:

$$P(n, 0) = P(n, n) = 1 \tag{B.1a}$$

$$P(n, k) = P(n - 1, k - 1) + P(n - 1, k) \tag{B.1b}$$

Equations (B.1a) and (B.1b) exactly match the recursive definition for the binomial coefficients [104]; hence, the entry of Pascal's Triangle at row n, column k corresponds to the binomial coefficient ${}_nC_k$ [105].

B.2.2 Structure of the CnC Graph

As explained earlier in this section, every CnC application must specify a set of step collections, corresponding to the functions used in computation, and a set of item collections, corresponding to the data on which the steps operate. To compute the value of ${}_nC_k$, we must compute n rows and k columns of Pascal's Triangle. Since the only type of data we use in this computation (both for building the triangle and in the result) is the set of values in the triangle, we only need a single item collection to hold that data, which we can call *pascal-entries*. Since we have two different equations for computing the entries of the triangle, we have one step collection for computing values based on equation (B.1a), and another based on equation (B.1b). We call the first step collection *edge-step* since it computes the values along the left and right edges of the triangle, and the second *inner-step* since it computes the remaining values inside the triangle. A high-level sketch of the CnC graph for this application is illustrated in figure B.3.

In CnC, instances of step and item collections are differentiated by a unique *tag*, often represented by an integer tuple; however, to differentiate step and item collections, we typically refer to the tag of an item instance as a *key*. We identify instances of both the item and step collections by the row and column of the corresponding entry in Pascal's Triangle; therefore, the tags and keys for instances in all three collections are integer pairs of the form $\langle \text{row}, \text{col} \rangle$. For simplicity, we use the notation (S: T) to denote an instance of step collection S with the tag T, where the round brackets correspond to the round nodes used for steps in the graphical representation (as shown in figure B.3). Similarly, we use the notation [I: K] to denote an instance of item collection I with the key K, or [I: K ⟶ V] to denote that the item instance has the value V, where the square brackets correspond to the rectangular nodes used for items in the graphical representation.

To give our application a more dynamic feel, each step instance with tag ⟨row, col⟩ prescribes the step instance with tag ⟨row+1, col⟩. Since each row of Pascal's Triangle has one more column than the previous row, steps with tags where $row = col$ also need to prescribe the step with tag ⟨row+1, col+1⟩. Each step instance also puts a single data item to the *pascal-entries* collection, with a key matching the step's tag, and the value ${}_{row}C_{col}$. Figure B.4 illustrates these relationships among the step and item collections, with the mapping between step tags and item keys shown explicitly.

It is often useful in a CnC application to parameterize some aspects of the graph structure. For example, one might want to parameterize the dimensions of the input matrices to a matrix kernel in order to make the code more generic. In our application, we want to parameterize the values n and k, which allows us to stop computation at row n of Pascal's triangle. We do this by setting values for n and k in the CnC graph's context, which is available to all CnC functions. These parameters are considered constant throughout the graph execution. The step functions in our application use these parameter values to compute whether or not to prescribe a new step instance corresponding to the next row of the triangle.

B.2.3 Executing CnC Steps

Before a CnC step instance is executed, that step must be *prescribed* (created) and all of its input data items must be available. The CnC runtime tracks the status of step and item instances via *attributes* attached to the instances. When some step instance (or the environment) *prescribed* a step in collection S with tag T, step (S: T) is created with the *control ready* attribute. If a step has a single input dependence on [I: K], the step gains

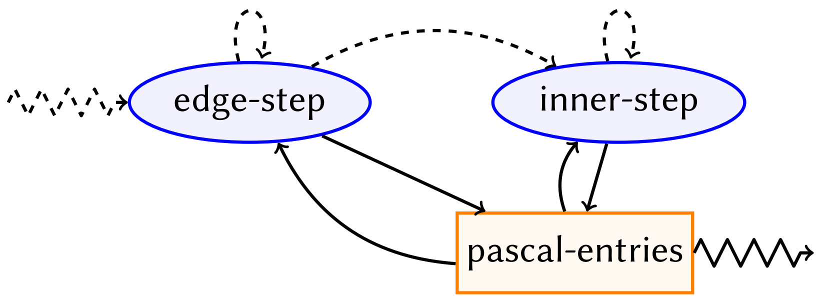
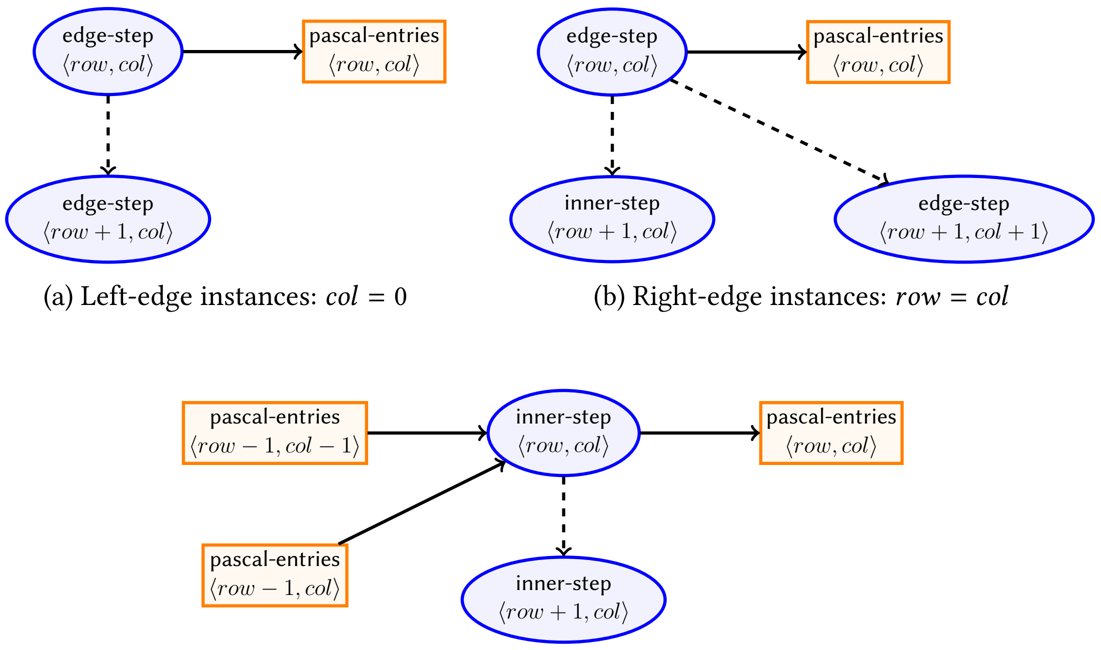


Figure B.3: The abstract graph representation of the Pascal's Triangle CnC application. Ellipses represent step collections (computation), and rectangles represent item collections (data). Dashed edges represent step prescriptions (creation), and solid edges represent puts to or gets from item collections. Jagged edges represent interactions with the application environment that encloses this CnC graph.


edge-step $\langle row, col \rangle$
pascal-entries $\langle row, col \rangle$
edge-step $\langle row+1, col \rangle$
edge-step $\langle row, col \rangle$
pascal-entries $\langle row, col \rangle$
inner-step $\langle row+1, col \rangle$
edge-step $\langle row+1, col+1 \rangle$
pascal-entries $\langle row-1, col-1 \rangle$
inner-step $\langle row, col \rangle$
pascal-entries $\langle row, col \rangle$
pascal-entries $\langle row-1, col \rangle$
inner-step $\langle row+1, col \rangle$

(a) Left-edge instances: $col = 0$

(b) Right-edge instances: $row = col$

(c) Inner instances: $0 < col < row$

Figure B.4: The *prescribe*, *put* and *get* relationships among the step and item collections in the Pascal's Triangle CnC application. Note that the topmost entry of the triangle, where $row = col = 0$, is actually a special case that does the *edge-step* prescription from the left edge and the *edge-step* prescription from the right edge without an *inner-step* prescription.

the *data ready* attribute when an item with key K has been put to item collection I. If a step has two or more such dependencies, the step is data ready only when all of the input items have been put. If a step has zero input dependencies then it is always considered data ready. Once a step is both control ready and data ready, it gains the *ready* attribute, and is only then eligible to execute.

In our Pascal's Triangle application, an instance of *edge-step* is ready as soon as it is prescribed because it has no input dependencies on the item collection. Since instances of *inner-step* depend on two item instances from *pascal-entries*, an *inner-step* instance is only ready to execute after it has been prescribed and both of the corresponding item instances have been put.

B.2.4 Interaction with the Environment

A CnC graph is typically embedded within a driver application, and we refer to the portions of the application that interact with the CnC graph as the *environment.* When CnC program execution is completed, the environment must put all data, prescribe all steps, and set any parameters necessary to properly initialize the CnC graph. The environment may also get values from item collections, which acts as an output mechanism for the graph.

In our Pascal's Triangle application, the environment initializes the graph's n and k parameters, then prescribes an (edge-step: $0, 0$), which corresponds to the topmost entry of the triangle. From that point, the CnC runtime has all the information it needs to compute the value for ${}_nC_k$. When the CnC graph has completed its execution, the environment gets [pascal-entries: n, k], which holds the computed value of ${}_nC_k$.

B.2.5 Example Execution

We will now outline an example of an execution trace for our Pascal's Triangle application. For simplicity in tracing the execution, we assume that the runtime has only a single worker thread, meaning that only one step can run at a time. This assumption eliminates any possible concurrency among steps in the computation and simplifies reasoning about program execution and execution trace creation.

We pick ${}_2C_1$ as the target value for this execution, therefore the environment initializes an instance of our CnC graph with the parameters $n = 2$ and $k = 1$. The environment also prescribes (edge-step: 0, 0) to start the graph's execution. Since (edge-step: 0, 0) has been prescribed and has no input dependencies, it is ready to execute. The graph execution is described textually below, and graphically in figure B.5.

(edge-step: 0, 0)
puts [pascal-entries: 0, 0 ⟶ 1];
prescribes (edge-step: 1, 0) and (edge-step: 1, 1).

All steps in row 0 have now run to completion.

(edge-step: 1, 0)
puts [pascal-entries: 1, 0 ⟶ 1];
prescribes (edge-step: 2, 0).
(edge-step: 1, 1)
puts [pascal-entries: 1, 1 ⟶ 1];
prescribes (inner-step: 2, 1) and (edge-step: 2, 2).

All steps in row 1 have now run to completion.

(edge-step: 2, 0)
puts [pascal-entries: 2, 0 ⟶ 1];
prescribes no steps since $row = n = 2$.

(inner-step: 2, 1) depends on [pascal-entries: 1, 0] and [pascal-entries: 1, 1], but since both items were already put, it is ready to execute.

(inner-step: 2, 1)
gets [pascal-entries: 1, 0 ⟶ 1] and [pascal-entries: 1, 1 ⟶ 1];
puts [pascal-entries: 2, 1 ⟶ 2];
prescribes no steps since $row = n = 2$.
(edge-step: 2, 2)
puts [pascal-entries: 2, 2 ⟶ 1];
prescribes no steps since $row = n = 2$.

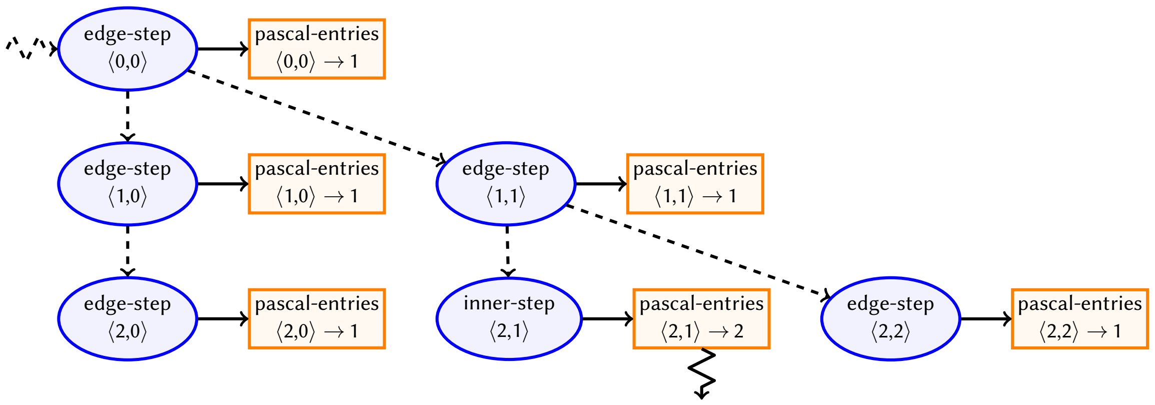


Figure B.5: Dynamic CnC graph for the computation of ${}_2C_1$.

All steps in row 2 have now run to completion. Since all prescribed steps have run to completion, the CnC graph execution is finished. The environment gets item instance [pascal-entries: 2, 1 $\longrightarrow$ 2] and correctly yields the answer ${}_2C_1 = 2$.

B.3 The CnC Continuum

CnC describes a programming paradigm rather than a specific runtime implementation. As a result, there is quite a bit of flexibility in how a particular CnC runtime may behave, and what requirements it might impose. One example of this is the static or dynamic nature of the CnC graph. CnC has no restrictions about how much of a application's graph structure must be computable statically versus computed dynamically at runtime. This results in a variety of requirements in the existing CnC implementations pertaining to the specification of inputs and outputs of CnC step functions. Some implementations require that some or all of step tags and item keys to be computed statically, whereas others allow all the inputs and outputs of a step instance to be computed dynamically.

Appendix C

CnC Sans Control Collections

In this appendix, we briefly explain how the CnC model used in CnC-OCR varies from the traditional CnC model in the literature in terms of explicit control. We also explain how the two models are functionally equivalent.

Control collections are traditionally used in CnC to help abstract away the creation of new step instances. In the traditional model, a CnC step can only create a new step instance indirectly by *putting* a control tag into a control collection, which in turn causes a new step instance to be *prescribed* in each step collection driven by that control collection [89]. Figure C.1 illustrates the graph structure of a simple application implementing a two-level filter (originally shown in figure C.1) in the CnC model with control collections.

Although this is a useful abstraction for reasoning about some programs, in practice we have found that the concept of control collections tends to confuse new users. Furthermore, we have found that in the majority of our CnC applications there is always a single step collection associated with each control collection. These observations are reflected in the absence of control collections in the Habanero variants of CnC, including CnC-OCR.[1]

We now demonstrate that CnC sans control collection has equivalent functional power to CnC with explicit control collections. In the case that the control–step relationship is a bijection, we can safely substitute each control *put* with the equivalent step *prescribe*, while maintaining identical program behavior. In the rare case that a control collection drives multiple step collections, each *put* must be replaced by one *prescribe* per associated step collection. These two transformations enable us to transform any traditional CnC

[1] See https://habanero.rice.edu/cnc for more information on other Habanero CnC implementations.

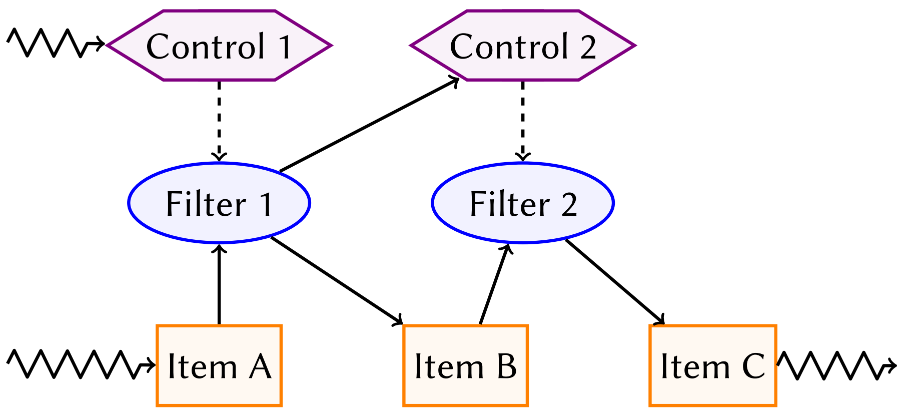


Figure C.1: The CnC graph representation of a simple data filtering application with explicit control collections. This corresponds with the graph shown in figure B.1, where control is represented by prescription edges between step collections. The two control collections, which drive the two step collections, are represented by the hexagonal nodes. Solid edges ending at a control collection represent a put of a tag to that control collection.

application to one without control collections; therefore, we know that modeling only the step and item collections of CnC is sufficient to describe CnC applications in general.

Recent work on hierarchical and modular graph organization in CnC has suggested that explicit control collections provide a crucial abstraction for connecting independent sibling graphs [106]. We maintain that explicit control collections are never necessary. The explicit control collections could be used in the domain specifications of the modular components, but transformed away in the actual program. It may be the case that control collections (or a similar abstraction) prove to be an essential software engineering abstract in future work; however, for current applications, it has been our experience that our users prefer using CnC sans control collections.

Appendix D

CnC Graph Domain Specific Languages

In this appendix, we provide detailed definitions of the domain specific languages used in our CnC framework. Appendix D.1 defines the DSL for CnC domain specifications, and appendix D.2 defines the DSL for tuning specifications. The most current version of this documentation is available on the CnC Framework wiki.[1]

D.1 The Graph Spec Language

CnC-OCR uses a graph specification file to generate a lot of the scaffolding code needed to interface CnC with the Open Community Runtime (OCR). It also uses the graph spec to generate some skeleton code for the user's CnC step functions. This page explains the ASCII representation used for this declarative CnC graph specification.

D.1.1 Structure

All CnC graph specifications should have the following structure:

- Begins with an optional *«context»* declaration
- Zero or more *«item-collection»* declarations
- Multiple *«function-io»* declarations

[1] https://github.com/habanero-rice/cnc-framework/wiki

D.1.2 Syntax

Comments and Whitespace

The graph spec supports C-style block comments (`/* ... */`) and C++-style line comments (`// ...`). Whitespace is not significant since all declarations are terminated by a sentinel (e.g. `;` or `,`).

Context

The *«context»* begins with the identifier `$context`, followed by a pair of braces `{ }` containing some field declarations in the host-language's syntax (C), and ends with a semicolon. Here's an example:

```
$context {
    int a, b, c;
    double x, y, z;
};
```

The *«context»* declaration is optional. You can safely leave it out if your application doesn't need to store any parameters in the "global" context for your graph.

Item Collection Declarations

Item collections are always denoted by square brackets, corresponding to the rectangle representation used for item collections in the graphical CnC representation. Each *«item-collection»* has the following general structure:

`[` *«data-type» «collection-name»* `:` *«key-id0»*`,` *«key-id1»*`,` … `] ;`

The *«data-type» «collection-name»* portion of the declaration should just look like a variable declaration in the host language. The current version should support most C types. However, there are a few cases that will not work, such as function pointers. If you have a complex data type you might consider using a `typedef` in your application's *AppName_defs.h* file, and used the type alias in the graph spec. To declare an array of some item type, simply use a pointer. The generated CnC-API includes parameters for creating items that contain some *count* of elements of the type referenced by the pointer.

The portion of the declaration after the colon corresponds to the declaration of the item collection's key shape. We assume that the keys are tuples of integer values (`s64`). The zero-ary tuple can be represented using a pair of empty parentheses `()`. Each tuple component should have a unique identifier. These identifiers are used in the generated API functions related to this collection.

Some examples of item collection declarations:

```
[ double *Lkji: i, j, k ];
[ double ts: x ];
[ struct timeval startTime: () ];
```

Function Declarations

Functions include all step functions, as well as the `$initialize` and `$finalize` pseudo-steps. Functions are declared with all of their input and output relationships relative to the other collections. Thus, a function declaration is actually a full specification of the I/O relationship for that function relative to the step and item collections declared in the graph. There should be a relationship declared for each item read, each item put, and each step prescribed by an instance of the function.

Function declarations have the following components:

- Declaration of the function, with its *name* and *tag*.
- An optional *inputs* clause, starting with the `<-` symbol, and followed by one or more item instance references.
- An optional *outputs* clause, starting with the `->` symbol, and followed by one or more step or item instance references.
- As with everything else, it's terminated by a trailing semicolon.

The *inputs* clause should always be omitted for the `$initialize` function, and same for the *outputs* clause for the `$finalize` function.

Step Declaration Step collections (along with the pseudo-steps) are always denoted by round brackets, corresponding to the ellipse representation used for step collections in the graphic CnC representation. A function declaration has the following form:

`(` *«collection-name»* `:` *«tag-id0»*`,` *«tag-id1»*`, ... )`

The *«collection-name»* is a single identifier used to identify the step collection. For the pseudo-steps, the special identifiers `$initialize` and `$finalize` are used. The portion after the colon is a list of identifiers used to specify the shape of the function's tag. These identifiers are used in the *tag functions* for the item and step instance references in the input and output clauses.

Collection Instance References Item and step instance references have the same form as their declaration, except that they *key* and *tag* components comma-separated lists of host-language *expressions* rather than identifiers. These expressions should be declared in terms of the tag component identifiers of the function for that input/output clause. This allows the CnC-OCR runtime to determine the inputs and outputs of a step (or psuedo-step) based on its unique tag. Here are some examples of function I/O declarations:

```
( addToInside:     row,   col   )
 <- [ cells:       row-1, col-1 ],
    [ cells:       row-1, col   ]
 -> [ cells:       row,   col   ],
    ( addToInside: row+1, col   );

( $initialize: () )
 -> [ cells:          0, 0 ],
    ( addToLeftEdge:  1, 0 ),
    ( addToRightEdge: 1, 1 );

( $finalize: n, k ) <- [ cells: n, k ];
```

Note that while the input relationships must be specified exactly, the output relationships are not as strict. The translator tool needs exact input relationships in order to create step instances with the correct inputs, but the output relationships are only used to generate skeleton code for the user. It's still good practice to include the output relationships in the graph specification, but if a particular output relationship is too complex to express in the specification, it can be simplified or omitted. This also means that outputs listed might actually be conditional outputs in the function's implementation.

Item Instance Binding Item instance references can optionally be bound to a local name for use in a function's implementation:

`[` *«instance-name»* `@` *«collection-name»* `:` *«key-expr0»*`,` *«key-expr1»*`,` … `]`

For example, in the step I/O declaration for `addToInside` above, the step has two different inputs from the `cells` item collection, and one output from the same collection. By default, the generated function will use the name `cells` for these values, appending integers to the end of the identifiers. In this case, you would end up with two parameters named `cells0` and `cells1`, and an output variable named `cells2`. If we wanted more succinct names instead—such as `a`, `b` and `c`—we could change the definition as follows:

```
( addToInside:     row,   col   )
 <- [ a @ cells:   row-1, col-1 ],
    [ b @ cells:   row-1, col   ]
 -> [ c @ cells:   row,   col   ],
    ( addToInside: row+1, col   );
```

Tag Ranges We provide two built-in functions for specifying *ranged* key/tag components: `$range` and `$rangeTo`. Both functions take a *start* and an *end* argument. The *end* is exclusive for `$range`, but inclusive for `$rangeTo`. Both functions also have a single-argument variant, where the *start* value is assumed to be zero. In other words, `$rangeTo(a, b)` is equivalent to `$range(a, b+1)`, and `$range(x)` is equivalent to `$range(0, x)`.

Ranges can be used on input or output instances. Let's say that we have an application that creates 5 item instances, updates them "iteratively" in some processing step, and then reads the results after 10 iterations. Here is an example of how we might specify this application using ranges:

```
[ SomeType data: index, iter ];

( $initialize: () )
 -> [ data:    $rangeTo(1, 5), 0 ],
    ( process: $rangeTo(1, 5), $rangeTo(1, 10) );

( process: index, iter )
 <- [ in  @ data: index, iter-1 ]
 -> [ out @ data: index, iter   ];

( $finalize: () ) <- [ data: $rangeTo(1, 5), 10 ];
```

Conditional I/O Not all applications have completely static input / output relationships. Sometimes it is helpful if we can make an input or output instance conditional. Let's say we have a compute step that depends on a data item at `i`, and at `i-1`. However, the step with tag `i=0` does not have a corresponding `i-1` value. We can make the `i-1` input conditional by using the `$when` built-in with a condition:

```
( compute: i )
  <- [ x @ data: i-1 ] $when(i > 0),
     [ y @ data: i   ];
```

If `data` has a pointer type, then `x` will be null when the condition is false. Conditions are currently not supported for inputs with raw (non-pointer) types.

D.2 The CnC Tuning Language

The Habanero CnC framework provides a domain-specific language (DSL) for tuning application performance. The DSL somewhat resembles the graph specification DSL.

D.2.1 Structure

All CnC graph specifications should have the following structure:

- Zero or more *«item-collection-tuning»* declarations
- Multiple *«step-function-tuning»* declarations

D.2.2 Syntax

Tuning declarations

Declaring an item collection tuning:

`[` *«collection-name»* `] : {` *«property-name»* `:` *«property-value»* `,` … `} ;`

Declaring a step collection tuning:

`(` *«collection-name»* `) : {` *«property-name»* `:` *«property-value»* `,` … `} ;`

Comments and Whitespace

The graph spec supports C-style block comments (`/* ... */`) and C++-style line comments (`// ...`). Whitespace is not significant since all declarations are terminated by a sentinel (e.g. `;` or `,`).

D.2.3 Currently supported tunings

The following tunings are supported on both the OCR and iCnC runtime backends.

Distribution functions

You can declare distribution functions using the `distfn` property. You can use the special variable `$RANK` to get the number of ranks (or policy domains in OCR).

Step priorities

You can set a weight for step exectution order using the `priority` property. A larger value means higher priority. The value is assumed to be a signed integer value, with a default priority of zero.

D.2.4 Using a tuning specification

You can specify one or more tuning files, via the `-t` flag, to use when generating the application/runtime scaffolding code.

```
ucnc_t -t tuningA.cnct -t tuningB.cnct graph.cnc
```

Since the translator tool can accept multiple tuning files as input, the programmer can keep orthogonal tuning specifications in separate files, and then mix-and-match the tunings to find the ideal combination for a given hardware platform or application input.

Appendix E

CnC Unified C API

This appendix describes the Unified CnC API for the C programming language. This API is utilized in all applications that use the Habanero CnC Framework, which allows the applications to work on top of all compliant runtime backends supported by the framework. The most current version of the API is available on the CnC Framework wiki.[1]

Below, italicized variable names indicate metavariables, which are replaced with concrete names during code generation. *`G`* stands for the graph name. *`S`* stands for a step collection name. *`t0`*, ..., *`tN`* stand for step tag component names. *`I`* stands for an item collection name. *`k0`*, ..., *`kN`* stand for item key component names.

Data structures

`GCtx`

The CnC graph context data structure. Contains runtime data structure information, as well as any members declared by the applications programmer as part of the `$context` in the CnC graph spec file. Most CnC API functions require a graph context pointer as the last argument.

`GArgs`

A struct used for passing arguments from external (environment) code into the CnC graph's initialization function. For many applications, this struct can be left empty. You can safely use a `NULL` pointer in place of a pointer to an empty argument struct.

[1] https://github.com/habanero-rice/cnc-framework/wiki

Generated skeleton functions

`int cncMain(int argc, char *argv[])`

Entry point to the CnC application. The `argc` and `argv` parameters contain the command line parameters passed to the application, just as in a vanilla C `main` function.

`void G_S(cncTag_t t0, ..., cncTag_t tN, TX x, ..., TZ z, GCtx *ctx)`

A CnC step function implementation. The parameters include the step instance's tag component values, along with all item instances specified as input to the step, which constitute the step inputs. The step can *prescribe* new step instances or *put* new item instances as output.

`void G_cncInitialize(GArgs *args, GCtx *ctx)`

This is the first function run after the CnC graph is launched. It corresponds to the `$initialize` pseudo-step in the CnC graph specification. This function considered a pseudo-step because it is step-like in its outputs, but the input does not come from the item collections. This function might use file I/O, or just read parameters from the `GCtx` and `GArgs` structs.

`void G_cncFinalize(cncTag_t t0, ..., cncTag_t tN, GCtx *ctx)`

This function corresponds to the `$finalize` pseudo-step in the CnC graph specification. Its tag is set via the `G_await` function. This function considered a pseudo-step because its input is step-like (item instances), but the output is not. This function might use file I/O or manipulate non-CnC data structures to produce output.

CnC graph operations

`GCtx *G_create()`

Creates a CnC graph context (`GCtx`) struct for use in the CnC computation. Since the runtime stores much of the information about an executing CnC graph within this structure, this is usually the first CnC function called within a CnC application.

`void G_destroy(GCtx *ctx)`

Frees memory used by the CnC graph's internal data structures (e.g. the item collections). After this call, the graph context is no longer valid.

`void G_launch(GArgs *args, GCtx *ctx)`

Launches the CnC graph. This is essentially the "prescribe" action for the initializer pseudo-step. The `args` parameter may be `NULL` if the initializer function does not require any arguments.

`void G_await(cncTag_t t0, ..., cncTag_t tN, GCtx *ctx)`

Sets the tag for the finalizer pseudo-step. This function is typically called within the `G_cncInitialize` function body. Typically we consider the computation complete when the inputs for the finalizer pseudo-step are available, which is why the CnC graph execution "awaits" the values computed via the provided tag.

CnC step operations

`void cncPrescribe_S(cncTag_t t0, ..., cncTag_t tN, GCtx *ctx)`

Prescribe an instance of a step from the collection `S` with the given tag. Prescriptions should only be made from the `G_cncInitialize` function, or from step functions within the same CnC graph instance.

CnC item operations

`void *cncItemAlloc(size_t bytes)`

Allocates a given number of `bytes` of memory that can be shared across steps as an item (or possibly just part of an item) in an item collection.

`void cncItemFree(void *item)`

Frees the memory associated with the given `item` value. Note that this function does not use the graph context. Freeing the item value will not free any memory for entries in item collections that are associated with this value.

`void cncPut_I(T *item, cncTag_t k0, ..., cncTag_t kN, GCtx *ctx)`

Associates the value given in `item`, with the given key, in the collection *`I`*. The `item` argument must be created by a call to one of the `cncItemAlloc*_I()` family of functions. Note that there's no corresponding "get" operation, since the gets are handled automatically in the generated code.

Miscellaneous functions and macros

`CNC_REQUIRE(condition, format_str, format_args...)`

This is just a convenience macro for checking required conditions in the code, and failing with an error message if the condition isn't met. The error message uses a `printf`-style format string with variable arguments. This macro provides no CnC-specific functionality, but is provided for convenience in writing CnC applications.

`CNC_SHUTDOWN_ON_FINISH(graph_context_ptr)`

Sets a trigger to automatically terminate the application when the graph execution is completely finished (i.e., all prescribed steps have completed, and the initializer and finalizer pseudo-steps have also completed).

`CNC_SHUTDOWN_ON_FINALIZE(graph_context_ptr)`

Sets a trigger to automatically terminate the application as soon as the finalizer pseudo-step completes (i.e., it doesn't wait for any straggling step function computations or even the initializer). It is usually better to use `CNC_SHUTDOWN_ON_FINISH` when possible.

`void *cncLocalAlloc(size_t bytes)`

Some implementations of the CnC runtime require the use of special-purpose memory allocators. This is a wrapper for the special-purpose runtime allocator. The pointer returned by this function should only be considered valid within of the current step (or pseudo-step). Using the pointer outside of that scope results in undefined behavior.

`void cncLocalFree(void *ptr)`

Frees memory allocated by `cncLocalAlloc`. Should be called from within the same step (or pseudo-step) as the matching `cncLocalAlloc` call, otherwise the behavior is undefined.

References

[1] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, *et al.*, "The Sunway TaihuLight supercomputer: system and applications," *Science China Information Sciences*, pp. 1–16, 2016.

[2] V. Sarkar, W. Harrod, and A. E. Snavely, "Software Challenges in Extreme Scale Systems," Jan. 2010. Special Issue on Advanced Computing: The Roadmap to Exascale.

[3] V. Sarkar, ed., *ExaScale Software Study: Software Challenges in Extreme Scale Systems*. DARPA IPTO, Air Force Research Labs, 2009. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.205.3944.

[4] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 66:1–66:11, IEEE Computer Society Press, 2012.

[5] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo, "The open community runtime: A runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, Sept. 2016.

[6] N. Vrvilo, L. Yu, and V. Sarkar, "A Marshalled Data Format for Pointers in Relocatable Data Blocks," in *Proceedings of the 9th International Symposium on Memory Management*, ISMM'17, (New York, NY, USA), ACM, 2017.

[7] R. D. Hornung and J. A. Keasler, “The RAJA Poratability Layer: Overview and Status,” Tech. Rep. LLNL-TR-661403, Lawrence Livermore National Laboratory, Sept. 2014.

[8] H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[9] A. Hendricks, T. Heller, H. Jordan, P. Thoman, T. Fahringer, and D. Fey, “The Allscale Runtime Interface: Theoretical Foundation and Concept,” in *Proceedings of the 9th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers*, MTAGS’16, (Piscataway, NJ, USA), pp. 13–19, IEEE Press, 2016.

[10] H. Shan, B. Austin, N. J. Wright, E. Strohmaier, J. Shalf, and K. Yelick, “Accelerating applications at scale using one-sided communication,” in *Proceedings of the Conference on Partitioned Global Address Space Programming Models (PGAS’12)*, 2012.

[11] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo, “The Open Community Runtime: A runtime system for extreme scale computing,” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, Sept. 2016.

[12] UPC Consortium, “UPC language specifications v1.2,” Tech. Rep. LBNL-59208, Lawrence Berkeley National Laboratory, 2005.

[13] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, “Introducing OpenSHMEM: SHMEM for the PGAS Community,” in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS’10, (New York, NY, USA), pp. 2:1–2:3, ACM, 2010.

[14] J. R. Hammond, S. Ghosh, and B. M. Chapman, “Implementing OpenSHMEM Using MPI-3 One-Sided Communication,” in *Proceedings of the First Workshop on*

OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - Volume 8356, OpenSHMEM 2014, (New York, NY, USA), pp. 44–58, Springer-Verlag New York, Inc., 2014.

[15] S. Treichler, M. Bauer, and A. Aiken, "Realm: An Event-based Low-level Runtime for Distributed Memory Architectures," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), pp. 263–276, ACM, 2014.

[16] R. Ramey, "Boost Serialization Library." Boost.org, 2002. http://www.boost.org/libs/serialization/.

[17] "cereal - a c++11 library for serialization." GitHub.com, 2013. http://uscilab.github.io/cereal/.

[18] "LibTooling." Clang 3.9 documentation, 2016. http://clang.llvm.org/docs/LibTooling.html.

[19] P. A. Ullrich, "A global finite-element shallow-water model supporting continuous and discontinuous elements," *Geoscientific Model Development*, vol. 7, no. 6, pp. 3017–3035, 2014. https://github.com/paullric/tempestmodel/.

[20] P. Ullrich, G. Jost, B. A. Lelbach, and H. Johansen, "Exascale-Ready Programming Models for Climate," in *Workshop on Advancing X-cutting Ideas for Computational Climate Science*, AXICCS '16, Jan. 2016.

[21] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," Tech. Rep. LLNL-TR-641973, Aug. 2013.

[22] E. Porter, K. Knobe, and J. Feo, "Experience Porting LULESH to CnC," in *CnC'14: The Sixth Annual Concurrent Collections Workshop*, Sept. 2014. Slides available online: http://cass-mt.pnnl.gov/cnc2014/. Source code:

https://xstack.exascale-tech.com/git/public?p=apps.git;a=tree;f=apps/lulesh-2.0.3/refactored/cnc-ocr/pnnl/per-element;.

[23] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "UTS: An Unbalanced Tree Search Benchmark," in *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, (Berlin, Heidelberg), pp. 235–250, Springer-Verlag, 2007.

[24] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 3.1," June 2015. http://mpi-forum.org/docs/.

[25] OpenSHMEM.org, "OpenSHMEM: Application Programming Interface, Version 1.3," Feb. 2016. http://www.openshmem.org/site/Specification/.

[26] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 647–658, Nov. 2014.

[27] T. Heller, "Removal of Boost.Serialization." Mailing list announcement (gmane.comp.lib.hpx.devel), Apr. 2015. http://thread.gmane.org/gmane.comp.lib.hpx.devel/196.

[28] F. Schlimbach, J. C. Brodman, and K. Knobe, "Concurrent Collections on Distributed Memory Theory Put into Practice," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 225–232, Feb. 2013.

[29] S. Chatterjee, N. Vrvilo, Z. Budimlić, K. Knobe, and V. Sarkar, "Declarative Tuning for Locality in Parallel Programs," in *Proceedings of the 45th International Conference on Parallel Processing*, ICPP '16, Aug. 2016.

[30] 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:*

Proceedings of the 2013 IEEE International Symposium on Parallel & Distributed Processing, IEEE Computer Society, 2013.

[31] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an Object-Oriented Approach to Non-Uniform Cluster Computing,” in *Proceedings of the Twentieth Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA ’05, pp. 519–538, Oct. 2005.

[32] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, “UPC++: a PGAS extension for C++,” in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1105–1114, IEEE, 2014.

[33] I. Gaztanaga, “Boost Interprocess Library.” Boost.org, 2005. http://www.boost.org/libs/interprocess/.

[34] “Based Pointers (C++).” MSDN Library, 2008. https://msdn.microsoft.com/en-us/library/57a97k4e.aspx.

[35] S. Chandra, V. Saraswat, V. Sarkar, and R. Bodik, “Type inference for locality analysis of distributed data structures,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP’08, (New York, NY, USA), pp. 11–22, ACM, 2008.

[36] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, “Using a “Codelet” Program Execution Model for Exascale Machines: Position Paper,” in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT’11, (New York, NY, USA), pp. 64–69, ACM, 2011.

[37] V. Kumar *et al.*, “HClib: A C/C++ task-based programming model for shared memory and distributed parallel computing.” GitHub.com. https://github.com/habanero-rice/hclib/.

[38] M. Flatt, R. B. Findler, and PLT, "Continuations." The Racket Guide (v.6.5), 2016. https://docs.racket-lang.org/guide/conts.html.

[39] B. Labs and L. Technologies, "First-class Continuations." SML/NJ Documentation, 1997. http://www.smlnj.org/doc/SMLofNJ/pages/cont.html.

[40] "Asynchronous Programming with async and await (C#)." MSDN Library, 2015. https://msdn.microsoft.com/en-us/library/mt674882.aspx.

[41] P. Haller and J. Zaugg, "SIP-22 - Async." Scala Improvement Process, 2013. http://docs.scala-lang.org/sips/pending/async.html.

[42] "Continuations API." Scala Standard Library Documentation, 2014. http://www.scala-lang.org/files/archive/api/2.11.8/scala-continuations-library/scala/util/continuations/package.html.

[43] E. Mittelette, "Coroutines in Visual Studio 2015 – Update 1." Visual C++ Team Blog, 2015. https://blogs.msdn.microsoft.com/vcblog/2015/11/30/coroutines-in-visual-studio-2015-update-1/.

[44] G. Nishanov, "Working Draft, Technical Specification for C++ Extensions for Coroutines." C++ Standards Committee Papers, 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4649.pdf.

[45] C. Kohlhoff, "Boost.Asio (Asynchronous I/O)." Boost.org, 2015. http://www.boost.org/libs/asio/.

[46] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, Apr. 2008.

[47] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *Proceedings of the 8th*

International Conference on Partitioned Global Address Space Programming Models, PGAS ’14, (New York, NY, USA), pp. 6:1–6:11, ACM, 2014.

[48] C. Yang and J. Mellor-Crummey, “A Practical Solution to the Cactus Stack Problem,” in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’16, (New York, NY, USA), ACM, 2016. https://github.com/chaoran/fibril/.

[49] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: An Efficient Multithreaded Runtime System,” in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’95, (New York, NY, USA), pp. 207–216, ACM, 1995.

[50] The Habanero Extreme Scale Software Research Project at Rice University, “Habanero-C.” https://habanero.rice.edu/hc/.

[51] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, “Habanero-Java: the New Adventures of Old X10,” in *PPPJ’11: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[52] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar, “Expressing DOACROSS Loop Dependencies in OpenMP,” in *9th International Workshop on OpenMP (IWOMP)*, Sept. 2013.

[53] E. G. Coffman, M. Elphick, and A. Shoshani, “System Deadlocks,” *ACM Computing Surveys (CSUR)*, vol. 3, no. 2, pp. 67–78, 1971.

[54] A. S. Tanenbaum, *Modern Operating Systems*, ch. 6, p. 438. New York, NY, USA: Pearson, 3 ed., 2007.

[55] C++ Standards Committee, “ISO/IEC 14882: 2011, Standard for Programming Language C++,” tech. rep., http://www.open-std.org/jtc1/sc22/wg21, 2011.

[56] D. Lea, "A Java Fork/Join Framework," in *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, (New York, NY, USA), pp. 36–43, ACM, 2000.

[57] "Interface ForkJoinPool.ManagedBlocker." Java Platform Standard Edition 8 Documentation, 2016. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.ManagedBlocker.html.

[58] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, ch. 9, p. 141. O'Reilly Media, 2007.

[59] V. Cavé, "HClib: A library implementation of the Habanero-C language." GitHub.com. http://habanero-rice.github.io/hclib-legacy/.

[60] Wikipedia, "Comparability — Wikipedia, The Free Encyclopedia," 2016. Accessed: 2016-12-05.

[61] Wikipedia, "Coroutine — Wikipedia, The Free Encyclopedia," 2016. Accessed: 2016-12-05.

[62] Flaise, "Semicoroutine," 2016. Accessed: 2017-05-05.

[63] Python 3 Documentation, "Yield expressions," 2001. Accessed: 2017-05-05.

[64] O. Kowalke, "Boost.Context." Boost.org, 2014. http://www.boost.org/libs/context/.

[65] S. Ghemawat and P. Menage, "TCMalloc: Thread-Caching Malloc." GitHub.com, 2007. https://gperftools.github.io/gperftools/tcmalloc.html.

[66] The Center for Research in Extreme Scale Technologies (CREST), "High Performance ParalleX: HPX-5." Indiana University, 2015. https://hpx.crest.iu.edu/.

[67] S. Imam and V. Sarkar, "Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns," in *28th European Conference on Object-Oriented Programming (ECOOP)*, July 2014.

[68] Parallel Universe, "Quasar: Fibers, Channels and Actors for the JVM," 2013. http://docs.paralleluniverse.co/quasar/.

[69] M. Mann, "Continuations Library." http://www.matthiasmann.de/content/view/24/26/.

[70] M. G. Burke, K. Knobe, R. Newton, and V. Sarkar, "Concurrent Collections Programming Model," in *Encyclopedia of Parallel Computing* (D. Padua, ed.), pp. 364–371, Springer US, 2011. A pre-print copy of this article is available at https://habanero.rice.edu/publications.

[71] "The Habanero CnC Framework." GitHub.com. https://github.com/habanero-rice/cnc-framework.

[72] "CnC: Intel Concurrent Collections for C++." GitHub.com. https://icnc.github.io/.

[73] "CnC-Python." Habanero Extreme Scale Software Research Project Wiki. https://habanero.rice.edu/CnC-Python.

[74] "CnC-Scala." Habanero Extreme Scale Software Research Project Wiki. https://habanero.rice.edu/CnC-Scala.

[75] "Intel Concurrent Collections for Haskell." GitHub.com. https://github.com/rrnewton/Haskell-CnC.

[76] C.-C. Yang and D. Padua, "Hierarchically Tiled Arrays for OCR." Bitbucket.org. https://bitbucket.org/cyang49/htaocr.git.

[77] "Legion-OCR." GitHub.com. https://github.com/srirajpaul/legion-ocr.

[78] N. Vrvilo, "The Habanero CnC Framework: A Demonstration of CnC Unification," in *Proceedings of the The Seventh Annual Concurrent Collections Workshop (CnC'15)*, 2015. Slides available online: https://engineering.purdue.edu/plcl/cnc2015/program.html.

[79] A. Sbîrlea, J. Shirako, L.-N. Pouchet, and V. Sarkar, "Polyhedral Optimizations for a Data-Flow Graph Language," in *The 28th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '15, Sept. 2015.

[80] K. B. W. Dragoş Sbîrlea, Alina Sbîrlea and V. Sarkar, "The Flexible Preconditions Model for Macro-Dataflow Execution," in *The 3rd Data-Flow Execution Models for Extreme Scale Computing Workshop (DFM)*, Sept. 2013.

[81] "Jinja2: The Python Template Engine." http://jinja.pocoo.org/.

[82] The Center for Systems and Software Engineering at the University of Southern California, "UCC: Unified CodeCount." http://csse.usc.edu/ucc_wp/.

[83] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu, "Runnemede: An Architecture for Ubiquitous High-Performance Computing," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, (Washington, DC, USA), pp. 198–209, IEEE Computer Society, 2013.

[84] T. Mattson and R. Cledat, "OCR: The Open Community Runtime Interface, Version 1.1.0." Modelado.org, 3 2016. https://xstack.exascale-tech.com/git/public?p=ocr.git;a=blob;h=c99da307;f=ocr/spec/ocr-1.1.0.pdf.

[85] "CnC on HCMPI with Tuning." GitHub.com. https://github.com/habanero-rice/cnc-ocr/tree/icpp2016-tuned-cnc.

[86] B. Chamith, "CnC Framework (forked)." GitHub.com. https://github.com/chamibuddhika/cnc-framework. This fork has experimental support for HPX-5 code generation.

[87] Z. Budimlić, M. Burke, K. Knobe, R. Newton, D. Peixotto, V. Sarkar, and E. Westbrook, "Deterministic Reductions in an Asynchronous Parallel Language," in

Proceedings of The 2nd Workshop on Determinism and Correctness in Parallel Programming, WoDet'11, Mar. 2011.

[88] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[89] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşırlar, "Concurrent Collections," *Scientific Programming*, vol. 18, pp. 203–217, Aug. 2010.

[90] Wikipedia, "Uniqueness quantification — Wikipedia, The Free Encyclopedia," 2016. Accessed: 2016-12-05.

[91] N. Vrvilo and R. Cledat, "Implementing a High-level Tuning Language on the Open Community Runtime: Experience Report," in *Runtime Systems for Extreme Scale Programming Models and Architectures (RESPA)*, Nov. 2015.

[92] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun, "Programming for Parallelism and Locality with Hierarchically Tiled Arrays," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, (New York, NY, USA), pp. 48–57, ACM, 2006.

[93] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A High-productivity Programming Language for HPC with Logical Regions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 81:1–81:12, ACM, 2015.

[94] B. Meister, M. Baskaran, B. Pradelle, T. Henretty, and R. Lethin, "Efficient Compilation to Event-Driven Task Programs," *ArXiv e-prints: arXiv:1601.05458v1 [cs.DC]*, Jan. 2016.

[95] K. Knobe and M. G. Burke, "The Tuning Language for Concurrent Collections," *16th Workshop on Compilers for Parallel Computing (CPC)*, 2012.

[96] K. Sharma, *Locality Transformations of Computation and Data for Portable Performance.* PhD thesis, Rice University, Aug. 2014.

[97] M. Kong *et al.*, "PIPES," 2016. To appear.

[98] C. Liu and M. Kulkarni, "Optimizing the LULESH Stencil Code Using Concurrent Collections," in *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '15, (New York, NY, USA), pp. 5:1–5:10, ACM, 2015.

[99] K. Sharma, I. Karlin, J. Keasler, J. R. McGraw, and V. Sarkar, "Data Layout Optimization for Portable Performance," in *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing*, pp. 250–262, Springer, 2015.

[100] C. Inc., "The Chapel language specification version 0.4," tech. rep., Cray Inc., Feb. 2005.

[101] UPC++ Specification Working Group, "UPC++ Specification v1.0 Draft 2," June 2017. https://bitbucket.org/upcxx/upcxx/downloads/upcxx-spec-V1.0-Draft2.pdf.

[102] H. Kaiser, "HPXnC - HPX Concurrent Collections." GitHub.com. https://github.com/STEllAR-GROUP/hpxnc.

[103] D. Sbîrlea, K. Knobe, and V. Sarkar, "Folding of tagged single assignment values for memory-efficient parallelism," in *Euro-Par 2012 Parallel Processing*, pp. 601–613, Springer, 2012.

[104] E. W. Weisstein, "Binomial Coefficient. From *MathWorld*–A Wolfram Web Resource." http://mathworld.wolfram.com/BinomialCoefficient.html. Accessed: 2016-12-05.

[105] E. W. Weisstein, "Pascal's Triangle. From *MathWorld*–A Wolfram Web Resource." http://mathworld.wolfram.com/PascalsTriangle.html. Accessed: 2016-12-05.

[106] Z. Budimlić and K. Knobe, "Declarative Communication for CnC," in *The Eighth Annual Concurrent Collections Workshop (CnC'16)*, 2016. Slides available online: https://cncworkshop2016.github.io/.