

© 2022 Justin Szaday

BUILDING BLOCKS FOR HIGH-PERFORMANCE LANGUAGES

BY

JUSTIN SZADAY

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant “Sanjay” Kale, Chair

Professor Nancy Amato

Professor David Padua

Affiliate Professor Brad Chamberlain, University of Washington

ABSTRACT

As machine complexity and diversity rise, the importance of high-level abstractions to the high-performance computing ecosystem rises. These tools, like adaptive runtimes and domain-specific or general-purpose languages, promote developer productivity and performance portability, lowering the barrier to building large-scale, high-performance parallel applications. However, their development and maintenance costs can be discouragingly high, relegating us to the use of classical, low-level tools.

Observing this, we propose that a language framework with high-level constructs built on a rich runtime system can significantly improve the productivity of language authors. To explore this hypothesis, we designed and implemented such a framework, EIR, then targeted it with an extensible language, Ergoline. Within Ergoline, we embedded several DSLs derived from past works built on the underlying RTS, Charm++. We explored code reuse among them and offered shared optimizations within EIR. To close semantic gaps between Charm++ and EIR, we developed Hypercomm—a suite of optimized abstractions. Through our experiences developing these tools, we conclude that our approach can offer performance and productivity benefits to language authors and their users alike.

ACKNOWLEDGMENTS

This work continues a long lineage of language-related work in UIUC’s Parallel Programming Lab. It would not be possible without the infrastructure and inspiration these works established. Thank you to my collaborators and predecessors for the invaluable assistance throughout my doctoral journey. In particular, my adviser Laxmikant Kale has lent me his deep insights into high-performance computing on many an occasion, guiding me toward solutions for performance and syntactic problems alike.

More broadly, thank you to the members of my dissertation committee; your evaluation and feedback of this work helped shape its outlook in numerous ways. Thank you to my collaborators from Intel and the Polaris research group as well. Although my work with you does not feature in this dissertation, you helped shape my understanding of compilers and performance studies. I also owe a great deal to XSEDE, which provided me with access to machines like Bridges2 and Stampede2 to generate performance results.

Lastly, thank you to my friends and family for your support. This dissertation would not have been possible without you cheering me along.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
1.1	Preliminaries	3
1.2	Approach	10
CHAPTER 2	ERGOLINE	13
2.1	Design Considerations	13
2.2	Overview	14
2.3	Annotations	24
2.4	Standard Library	29
2.5	Example Programs	36
2.6	Summary	40
CHAPTER 3	EMBEDDED DSLS	43
3.1	SDAG	43
3.2	[Ergo]Stencil	52
3.3	Charisma	55
3.4	MSA	57
3.5	DivCon	65
3.6	Summary	66
CHAPTER 4	EIR	68
4.1	Compiler Architecture	68
4.2	Compiler Optimizations	75
4.3	Compiler Performance	79
4.4	Summary	80
CHAPTER 5	HYPERCOMM	82
5.1	Programming Model	82
5.2	De/serialization	89
5.3	Element-spanning Trees	92
5.4	Futures	98
5.5	Migratable Threads	99
5.6	Summary	100
CHAPTER 6	CONCLUSIONS	102
6.1	Retrospective	102
6.2	Prospective	104
REFERENCES		106

APPENDIX A	EIR CONSTRUCT LISTING	112
APPENDIX B	EIR INCORPORATION AND INTEGRATION	114
B.1	Using Hypercomm	114
B.2	Using EIR	114
B.3	Migrating EIR and Hypercomm	114
B.4	Enhancing Interoperability	115
APPENDIX C	ETYMOLOGY	116

CHAPTER 1: INTRODUCTION

Many computing communities have addressed challenges by adopting higher-level programming languages. For example, languages like Rust and Swift have made in-roads into their “traditional” counterparts’ spaces, bringing performance, productivity, and safety benefits. Rust is disrupting C/C++, even likely appearing in the Linux kernel soon [1][2]. Swift has ousted Objective-C from the Top 20 languages in the TIOBE index [3]; it has even begun to see use in data-intensive spaces (beyond its primarily mobile orientation) [4].

Python has seen rapid growth thanks, in large part, to its rich package ecosystem and highly-productive nature. According to the TIOBE index, since 2018, it has enjoyed a four-fold increase in reported use (12-13% in 2022 vs. 3-4% in 2018). This growth is also partly attributable to compounding innovations surrounding its performance and interoperability (Cython, PyBind, etc.). For example, Numba has begun to close the gap between Python and compiled languages like C and Fortran, enabling 100-200 $\times$ speedups over ‘conventional’ Python code [5]. These improvements have started to see high-performance computing spaces adopting Python.

Despite the potential for high-level languages to address the challenges of the exascale computing era, the high-performance computing community generally continues to favor lower-level languages. After all, lower-level tools can expose more hardware features to achieve better performance. However, developers utilizing explicit models or vendor-specific features must refactor their code to accommodate new machines (e.g., from MPI to MPI+CUDA). As supercomputers grow more complex and diverse, requiring more and more expertise, this continuous refactoring may become untenable. Furthermore, as data science converges with HPC¹ and cloud computing becomes more prevalent, non-traditional HPC developers have access to powerful distributed machines. Changing tides like these may lead to more adoption of high-level tools in HPC. Many of these tools decouple descriptions of computations from how they execute to facilitate optimizations and performance portability.

Conventional HPC development tools require programmers to rigidly specify all aspects of their applications’ execution and data decomposition. In addition to being burdensome for programmers, this specificity is not conducive to compile- or run-time optimization. Conversely, optimizable programming models typically delegate decisions regarding where and when computations occur to a compiler and runtime system (RTS). This added flexibility enables automated grain size,² load-balancing, and fault-tolerance decisions. The migratable

¹As in, supercomputers and the preferred platforms of data scientists are increasingly similar.

²Runtimes like STAPL can modulate how much data-parallelism is contained within tasks throughout a computation, relieving users of these decisions.

objects model, for example, decomposes problems into parallel entities arranged in collections. These entities communicate with one another using active messages that correspond to method invocations in their peers. Since they wholly encapsulate computations and their immediate dependencies, they are migratable, i.e., able to execute these messages at any physical location. They can enjoy seamless communication/computation overlap when they outnumber the physically available processing elements (PEs); the RTS executes objects whose remote dependencies are ready while others wait for theirs to arrive. Along with automation, features like these address shared needs of scalable HPC applications, reducing the burden of programmers utilizing adaptive runtimes.

Most automatic optimizations for HPC programs occur at execution time (i.e., minimal or no user intervention). Generally, compilers for languages like C/C++ and Fortran are unaware of HPC matters (parallel control flow, data distribution, etc.) and unable to optimize them. Here, specializing languages for HPC can enable automatic compile-time optimizations; Chapel, as well as numerous HPC DSLs, have demonstrated these advantages. Furthermore, compilers can close ‘knowledge gaps’ of runtimes, compounding their benefits.

Past works demonstrated that adaptive runtime systems are productive platforms for language construction [6][7]. These works represent the fourth approach in Figure 1.1, which compares strategies for building HPC applications and libraries. Under the first approach, developers build their software on low-level infrastructure, implementing nearly everything from scratch. The second and third approaches delegate components to adaptive runtime or high-level languages, respectively, with the fourth approach combining their benefits. We extend these productivity benefits by incorporating a language framework built on an adaptive RTS, proposing the fifth approach. Our strategy relieves language authors of repetitive and tedious tasks like implementing type systems and shared optimizations. Furthermore, it offers performant implementations of high-level abstractions to close semantic gaps between languages and adaptive runtimes. Before this work, a suitable language framework to realize these possibilities did not exist.

We introduce Ergoline, an extensible language that raises migratable objects to first-class citizens. Behind Ergoline is EIR, a high-level intermediate representation that provides its analysis and transformation facilities. EIR generates code that targets Charm++, a proven adaptive runtime system with distributed load balancing and fault tolerance capabilities. Alongside it, we employ an optimized library (Hypercomm) to close semantic gaps, ensuring we deliver competitive performance in areas where EIR diverges from Charm++. We hypothesize that our language framework based on migratable objects forms a capable platform for proposing, exploring, and evaluating various topics regarding HPC programming abstractions and languages. Our “full spectrum” approach has multiple entry points into

its ecosystem and minimizes semantic gaps by providing a generic and object-oriented type system. Aside from using Ergoline to demonstrate EIR’s generality, we embed several DSLs within it to demonstrate EIR’s extensibility. Here, we note that future languages could directly target EIR as well.

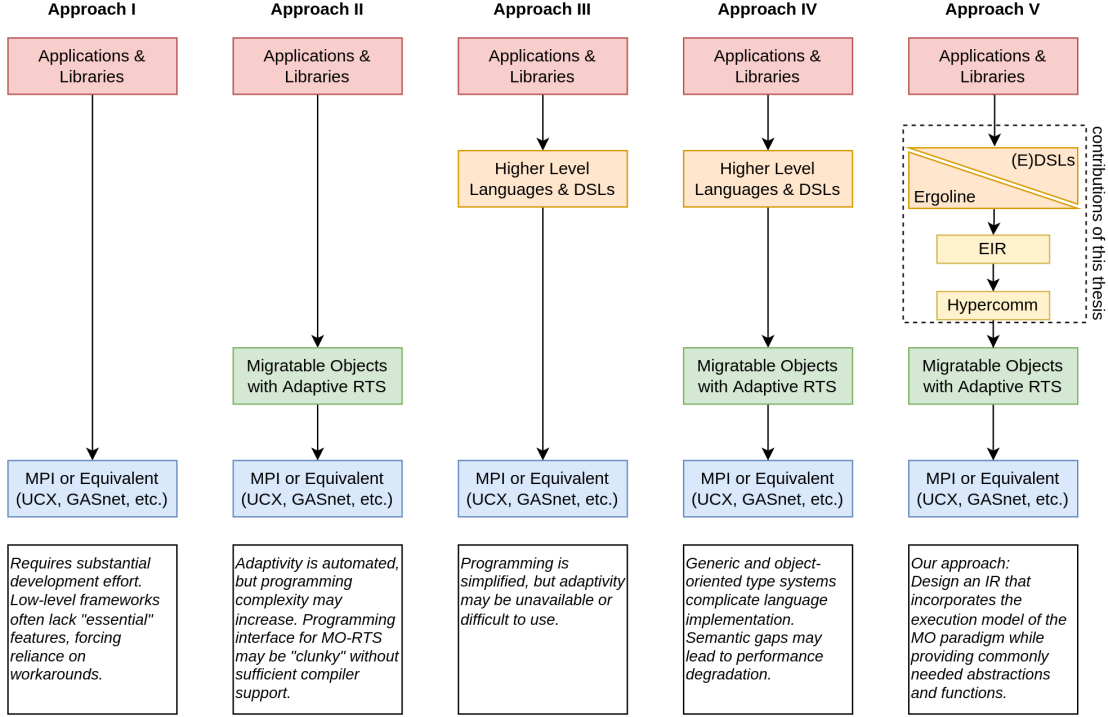


Figure 1.1: A visual representation of the approaches for application and library development in high-performance computing.

1.1 PRELIMINARIES

This section establishes the vocabulary we use to describe our work in relation to others. It also defines the mechanics of Charm++ relevant to our work.

1.1.1 Programming Models in HPC

In this work, we generally classify HPC programming models by their views of control and data, either global or local. Programmers cannot define the behaviors of specific parallel entities under global control views. For example, pools represent global views of threads. Programmers “post” tasks for idle threads to execute; however, they cannot select which

thread will run them. POSIX threads represent a local control view; programmers must instruct (e.g., suspend and resume) individual threads.

All parallel entities have shared access to all data under global views of data, typically with a uniform addressing mechanism. Under local views, each entity only has access to a particular data partition. In another OS-level analogy, threads have shared access to their host process’ data.³ On the other hand, each process has a unique address space and cannot directly share data with its peers. Instead, they must communicate through Inter-Process Communication (IPC) techniques.

Although these examples are still relevant in the context of HPC, we are more concerned with data and control views that scale beyond a single node. We compare several parallel programming models in Figure 1.2. MPI has local views of control and data; ranks communicate data through messages and are individually programmable. At their extreme, PGAS models have global control and data views—all messaging is implicit in distributed contexts.⁴ We discuss the other models shown in the contexts of related work or integrating them in EIR. Our approach of supporting diverse programming models within the same framework ensures programmers always have the “right tool for the job.” All views have their expressive strengths, i.e., operations they concisely represent.

1.1.2 Adaptive Runtimes

Alongside Charm++, many asynchronous many-task (AMT) systems with distributed capabilities exist, like HPX, Legion, PaRSEC, STAPL, Uintah, and VT. These systems promote computation-communication overlap through overdecomposition and message-driven execution schemes. Modules enjoy composability, seamlessly sharing processor time through their RTS’s scheduler. These are, typically, work-queue-based schedulers with varying degrees of adaptivity. For instance, they may utilize work-stealing to address load imbalance, either in terms of “push” (proactively send tasks to under-loaded PEs) or “pull” (steal work from over-loaded PEs) schemes. VT and HPX have Charm-like notions of migratability, moving work to be “closer” to communicating peers or data (e.g., to minimize non-local data-dependencies [8]).

³Hence, they implement a form of Shared Memory Parallelism (SMP).

⁴Counterexamples include PGAS models that distinguish between local and remote data accesses (e.g., Co-array Fortran); they stray toward local control and data views since users are aware of data distribution. Also, Chapel encourages using a global programming style, but programmers *can* use MPI routines or single-sided set/put calls.

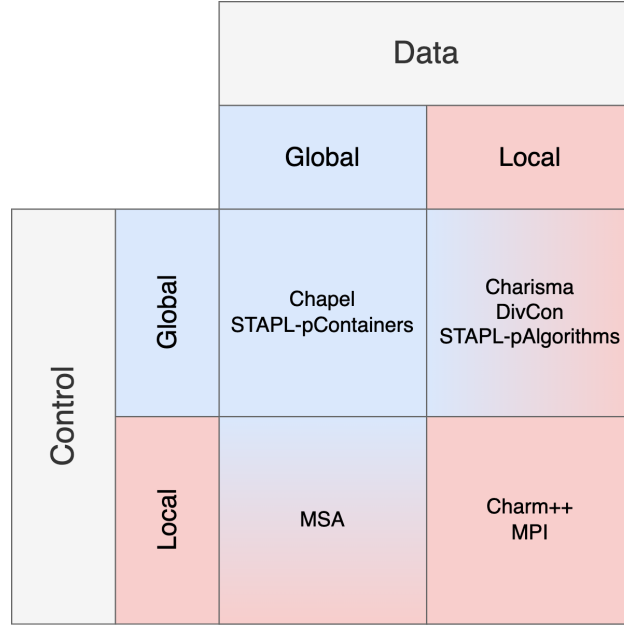


Figure 1.2: Example classifications of data and control views of parallel programming models.

Charm++

The cornerstone of this work is Charm++—the foundational implementation of the migratable objects model. As established in the introduction, over-decomposition and processor virtualization enables Charm++ to provide load-balancing and fault-tolerance facilities (e.g., automatic checkpoint/restart). Its asynchronous, message-driven execution model promotes modularity and parallel composition; applications can mix data- and task-parallelism.

Migratable Objects A migratable object is called a *chare* in Charm++. Users communicate with chares using *proxies*, which expose a chare’s remotely invocable methods, called *entry methods*. Charm++’s interface translator (Charmxi) generates code for marshaling entry method parameters, copying them to and from message buffers.

Proxies can target a single chare, an entire collection of chares, or a subset of a chare collection (called a *section*). Along with broadcasts and multicasts, programmers can also conduct asynchronous reductions at these granularities. Users compose these collective primitives to form the full complement of collective operations found in other programming models, like MPI.

Most applications in this model will feature over-decomposition, with multiple chares residing on each processing element. Each PE has a queue that its scheduler polls for messages; it invokes the corresponding methods of the target objects non-preemptively, with control

returning to the scheduler when they finish. The RTS routes messages to PEs using location managers that track where chares reside through a combination of caching and procedural updates. This caching can cause messages to be routed to a chare’s old location when they migrate. In such cases, the RTS re-routes messages onward.

Workflow There are two entry points into the Charm++ ecosystem: the conventional C++-based approach and a Python-based approach called Charm4Py. In the first approach, users write a combination of “Charm interface” (.ci) and C++ source files. Charmxi translates these (.ci) files into registration code for chares and their entry methods, along with implementing other domain-specific functionality (e.g., SDAG, discussed later in Section 3.1).⁵ In the Pythonic approach, users do not need to write separate files or explicitly annotate their chare’s entry methods; instead, the RTS dynamically registers all these constructs. Subsequently, Charm4Py represents an excellent entry point into the Charm++ ecosystem, sharply decreasing how much code users need to write. In part, our work with Ergoline aims to blend the simplicity of the Charm4Py approach with the performance of the (compiled) C++ approach.

1.1.3 General-purpose HPC Languages

Of the three languages developed with funding from the High Productivity Computing Systems (HPCS) project, i.e., Fortress, Chapel, and X10, only Chapel is still being actively developed. Chapel has a rich feature set and can comfortably compete with UPC and MPI, having seen radical performance improvements since its debut. Its compiler supports retargeting code for GPUs and optimizations at various abstraction levels; communication-related optimizations are feasible even at the LLVM IR level [9]. It has productive library support for prefetching [10] and message aggregation [11].

Regent is a high-level language primarily targeting the Legion RTS [12]. Its programming model blends the task and logical region (i.e., a hierarchical collection of objects) abstractions. Its compiler can auto-parallelize and optimize code for SIMD/T or distributed execution.

Noteworthy in this context, Regent is a Terra language extension. Terra is an extensible language embedded in Lua that employs Lightweight Modular Staging (LMS) to generate and optimize LLVM IR [12]. Subsequently, Regent could implement its parsing, analysis, and optimization facilities using Terra. The relationship between Terra and Regent is similar to the projected relationship between EIR and Ergoline; however, we do not employ LMS. We

⁵Charj cited Charmxi’s limitations and replicated information as one of its primary motivations [6].

discuss this further in the context of Delite. We developed EIR instead of using Terra since it does not have built-in support for object-oriented features like inheritance and sub-typing. Although it has proof-of-concept implementations of Go- and Java-like class systems, Terra’s abilities to optimize their constructs are limited compared to C++’s (particularly regarding virtual methods).

Charm-based Languages

Numerous works have explored using Charm++ as a language substrate, including Charisma [13][14], Charj [6], DivCon [7], Jade [15][16], and Structured Dagger (SDAG) [17]. Three of these languages were domain-specific, Charisma (for data-independent data flows), DivCon (for divide-and-conquer parallelism), and SDAG (for the dependencies between message arrival and local control flow); we leave detailed discussions of them to other parts of this dissertation. Charj and Jade were general-purpose, Java-inspired languages designed with first-class support for migratable objects. Both translated to C++ and featured liveness analysis-based data un/packing optimizations. Charj included a prototypical EDSL for the Multi-phase Shared Arrays (MSAs) library with control-flow analysis-based optimization passes—ideas first proposed in the context of Jade [18]. Scala-based revival efforts of Charisma and Charj began after their Java-based counterparts became obsolete;⁶ later, we compare our experiences on these projects with EIR and Ergoline.

1.1.4 Domain-Specific Languages

The term “domain-specific languages” (DSLs) refers to a broad swath of approaches with a common goal: incorporate domain knowledge to enhance productivity. They offer a restricted set of familiar abstractions and notations to focus their expressive power, amplifying a domain expert’s intent. Compared to general-purpose programming languages, these restrictions translate to better testability, reliability, maintainability, and portability. DSLs with translators can also enjoy domain-level optimizations that produce high-quality performance results. This distinction brings us to some of the terms used to categorize DSLs:

- **Internal/External.** We consider DSLs embedded within other programming languages as ‘internal,’ whereas those with bespoke compilers are ‘external.’ We classify internal DSLs by their embedding depth, either *deep* or *shallow*. Highly-extensible

⁶Their code can be found here and here.

languages enable deeply embedded DSLs to have correctness checking, code transformations, custom syntax, etc. Conversely, host languages constrain shallow DSLs, e.g., providing a limited set of keywords and overloadable operators. The exact definitions of these terms vary between sources [19][20].

- **Horizontal/Vertical.** Horizontal domains are activities or patterns that multiple applications share (e.g., data-independent data flows, stencil operations). Vertical domains refer to a particular type of application (e.g., weather and climate applications). DSLs can exist anywhere in the space these “axes” define. For example, being either horizontal, vertical, or somewhere in between.

Other factors to consider when comparing DSLs to general-purpose languages include their:

- **Target Audience.** DSLs tend to target different audiences than GPLs. They usually assume some expertise in their domain. They may appeal to non-programmers by adopting familiar or graphical notations (e.g., spreadsheets). Alternatively, they may target programmers by providing concise representations of particular tasks (e.g., array processing or parsing).
- **Implementation.** Many DSLs are effectively high-level APIs for libraries—as OpenPME is to OpenFPM [21]. Others require purpose-built abstraction-support infrastructure, i.e., custom implementations of domain abstractions, schedulers, etc.

1.1.5 DSLs in HPC

To exercise the vocabulary introduced in 1.1.4, let us apply it to several exemplary HPC DSLs. There is a great deal of diversity among them regarding their approaches and capabilities. For example, they may exhibit performance portability or focus on within-node, heterogeneous, or distributed parallelism.

Most DSLs in HPC favor an internal approach within C++, Python, or Scala. GridTools is a DSL based on C++ that supports stencil-like computations [22]. PyOP2 is a DSL based on Python that parallelizes kernels on unstructured meshes and graphs [23]. Liszt is based on Scala and targets mesh-based PDE solvers [24]. OptiGraph, OptiML, and OptiQL (Query Language) are based on Scala using Delite [25]. GraphIt is a DSL for Graph Analytics with bleeding-edge optimizations for ordered algorithms [26].

These DSLs are broadly horizontal since they target a class of computations rather than a particular scientific domain. However, GridTools exhibits some degree of verticality since its authors developed it primarily for weather and climate applications. Scala-based languages

can straddle the line between internal and external. While some may consider them to be “embedded,” Scala is not truly extensible since it still imposes syntactic constraints [19].

There are a few “standalone” DSLs in HPC (those not based on a GPL), like OpenABL which focuses on agent-based simulations [27]. OpenPME, which is supported by the OpenFPM library and implemented with JetBrains MPS, focuses on particle and mesh methods [21]. Marlon, a DSL for Multi-Agent Reinforcement Learning on Networks, notably supports adaptive load-balancing [28].

Finally, DSLs may help with the design, development, or deployment phases of HPC applications. Hi-PaL helps convert sequential programs to message-passing ones [29]. ParDSL aids in mapping parallel algorithms to parallel computing platforms [30]. Although these DSLs apply to applications from any domain, they appeal to a narrow audience (i.e., HPC system users) and a specific stage of the HPC development process. Subsequently, we consider them to be more vertical than horizontal.

1.1.6 Language Development Tools

Language frameworks and intermediate representations (IRs) simplify the language development process by providing “building blocks” for common syntactic constructs and reusable operations over them. For example, type-checking and AST-to-IR compilation may be delegated to a language framework, whereas IRs focus on low-level optimizations and machine code generation. However, these concepts are loosely defined, and their provided operations may take on many forms at varying abstraction levels. HPC language studies have involved language frameworks and IRs since their inception; LLVM IR is of particular note as a component of many studies [31].

Language Frameworks

Systems like AnyDSL and Delite help construct DSL compilers. AnyDSL employs partial evaluation to parallelize code for CPUs and GPUs [32]. Its front-end language (Impala) maps closely to its IR (Thorin), similarly to Ergoline and EIR. Its authors describe Impala as syntactic sugar for Thorin, albeit with support for direct-style functions in addition to continuation-passing style ones. Subsequently, the abstraction gap between them is quite narrow.

Delite is a Scala-based DSL framework that generates parallel code for CPUs and GPUs [25]; it does not support distributed parallelism. DSL authors can rely on Delite’s built-in optimizers or transform their code at runtime using LMS. Scala’s rich macro system

facilitates compile-time transformations; Rust also adopted this idea, and we plan to pursue it in Ergoline. Ergoline only supports compile-time transformations through EIR at this time. EIR also requires dedicated type-checking facilities, where Delite relies on Scala’s.

DSL frameworks may be specialized for particular domains as well. For example, QUARC is a framework for embedding DSLs with lattice and grid domains into C++ [33]. Instead of modifying C++, it extends its arrays with implicit data parallelism through template meta-programming (TMP). This strategy enables its LLVM-based code generator, QOPT, to “recover” high-level abstractions without requiring modifications to Clang (LLVM’s C/C++ frontend). From there, QOPT can perform low-level optimizations and automatic parallelization (targeting either SIMD or MPI).

Dawn is an optimizing compiler toolchain for weather and climate DSLs [34]. It has frontends for Python, Fortran, and a customized C++ variant (GTClang); these produce a high-level, sequential IR that can express entire models. Like QUARC, Dawn generates parallel implementations from its sequential specifications, optimizing them along the way. It can generate CUDA and MPI code, paving the way for scalability and performance portability.

Intermediate Representations

Many proposals have attempted to extend LLVM IR with dedicated facilities for parallelism. Tapir supports the fork-join model [35], ARES supports higher-level abstractions (e.g., channels, parallel-for, and parallel-reduce) [36], and HPVM caters to heterogeneous workloads [37]. Parallelware uses LLVM IR to trace HPC programs, seeking “patterns” that indicate parallelization opportunities [38].

The INSPIRE, LIFT, and XcodeML IRs are notable among those that do not involve LLVM. INSPIRE sought to provide a uniform representation for hybrid applications, e.g., OpenCL/MP and MPI; subsequently, their parallel control flow would be wholly accessible to compilers [39]. LIFT is an intermediate language (and closely related IR) that translates to OpenCL [40]; its predefined parallel patterns are the building blocks for expressing programs. XcodeML is an XML-based intermediate language for C and Fortran that is a component of the (source-to-source) Omni Compiler. Omni employs it to lower and optimize XcalableMP, a directive-based PGAS programming language [41].

1.2 APPROACH

The chapters of this work contextualize and detail the components of our approach. The second chapter presents examples of Ergoline, its syntactic constructs, and the parallel ab-

stractions in its standard library. The third chapter explores the DSLs we embedded within Ergoline: Charisma, DivCon, [Ergo]Stencil, SDAG, and MSA. The fourth chapter presents more details regarding EIR’s architecture. In the fifth chapter, we explore Hypercomm’s model and the abstractions we have built to support it. We conclude by discussing future directions for this work and how its ideas may propagate to other software packages.

1.2.1 Research Questions

In this work, we aim to tackle questions like:

- When does a shared compiler infrastructure benefit multiple DSLs targeting the same adaptive RTS? Analyses? Optimizations?
- Can we convert Charm++’s standalone DSLs to embedded ones?
- Can we productively blend diverse views of data and control?
- Can we enhance the expressiveness of migratable objects and their DSLs?
- When can RTS-level abstractions assist in building HPC languages?
 - Which semantic gaps arise when we raise the abstraction level?
 - Can we extend the underlying RTS to close these semantic gaps?

1.2.2 Contributions

Alongside demonstrating and discussing our approach in this dissertation, we introduce novel:

- RTS-level optimizations and abstractions, e.g.:
 - Eliminating copies during de/serialization
 - Building spanning trees over chare-arrays
 - Refining the ‘components’ model
- Compiler-level analyses and optimizations for:
 - The migratable objects model, e.g.:
 - * Identifying opportunities for message-buffer reuse

- Established DSLs, e.g.:
 - * Inferring synchronization points (MSA)
 - * Segmenting structured entry methods (SDAG)

CHAPTER 2: ERGOLINE

Ergoline blends aspects of functional and object-oriented programming models. It achieves parallelism by incorporating the migratable objects (MO) model. It aspires to be highly extensible to serve as a platform for researching embedded HPC DSLs. These features reflect the plurality of Ergoline’s role within this work; it advances both general-purpose and domain-specific MO-based languages.

We drew inspiration from Charj and Scala in our design of Ergoline. Charj established the intellectual underpinnings for an MO-based language and Scala lent its syntax. Broadly, Ergoline aims to be to C++ as Scala is to Java—blending the benefits of the underlying language with expressive syntax and functional programming ideas. To that end, some of its features mirror C++’s to promote interoperability. Lastly, we drew inspiration from contemporary languages like Elixir, Rust, and Swift.

We start this chapter with a discussion of Ergoline’s features, noting their inspiration and the basis for their inclusion (extensibility, performance, productivity, etc.). For example, many features enhance Ergoline’s capabilities or expressiveness relative to the combination of Charmxi and C++. Then, we present a few example programs written in Ergoline.

2.1 DESIGN CONSIDERATIONS

Great care was taken to uphold a handful of core tenets for Ergoline, including:

- Reduce copying, a huge waste of cycles in de/serialization.
- Focus users on impactful HPC development areas—reduce “tedium” by:
 - Automating memory management.
 - Providing cleaner syntax for Charm++ features.
- Enable optimizations by making stronger assumptions like:
 - Favor copy-only communication semantics to discourage data-sharing.
 - Restrict access to the underlying objects of local chares.¹

¹Providing local access to chare-objects allows free agents within the system to bypass entry-method calling conventions; this can lead to data-races. While *Ergoline* should retain support for valid use-cases of these abilities (e.g., yielding ownership of a large, heap-allocated object from one chare to another), it should do so with clarity and caution.

- When data must be shared between parallel actors, provide clear syntax for both the compiler and user.
- Mirror the underlying model when practical, extending it when possible.

2.2 OVERVIEW

Ergoline is a general-purpose, object-oriented programming language. It is feature-rich, and many of its features will be familiar to those familiar with its contemporaries (Scala, Swift, etc.). Subsequently, we present a broad overview of Ergoline’s features in this section, particularly highlighting those relevant to HPC developers.

2.2.1 Built-in Types

Ergoline has class-like, lambda, tuple, and reference types. It maps its primitives (e.g., `int`) to class-like types, enabling them to have member functions (e.g., `(42).toString()`). This mapping is ‘zero-cost’ since we achieve it through helper functions and compile-time specialization (unlike the JVM ‘boxing’ method). We plan to add class-like wrappers for lambda and tuple types in the future, providing utility methods as use-cases arise.

Static Expressions

Ergoline provides rich template meta-programming capabilities, and can statically evaluate expressions like C++. Values of primitive types, like booleans and integers, can be used as type parameters and manipulated with operators. Consider the following function:

```
def foo<N : int>() where (N % 2 != 0) { ... } (2.1)
```

Here, `foo` has a template argument `N` that accepts `int` values. Its `where`-clause is not specializable when `N` is divisible by two. Subsequently, `foo<4>()` will generate a substitution failure, whereas `foo<5>()` will not. Note that the operators introduced in *Variance Relationships* may be used in static expressions.

Tuple Multiplication The tuple multiplication operator, `.*`, repeats the contents of a tuple type `N` times. For example, the operation `(int) .* 3` is equivalent to the tuple-type `(int, int, int)`. Ergoline’s STL uses this operation for determining the index-types of containers with a variable number of dimensions.

2.2.2 Scoping Constructs

Ergoline’s scoping constructs aim to encourage modular, object-oriented programming. These constructs will be most familiar to Scala programmers, but many have C++ counterparts. Note that, like Scala, Ergoline allows users to declare scope members in any order;² this is a boon for simplicity relative to C++, where users must manually resolve circular and forward references. We outline the structure and use of Ergoline’s scoping constructs in this section.

Namespaces

As the highest-order scoping construct, namespaces may contain any other constructs, including other namespaces. They are equivalent to their C++ counterparts and closely related to Scala’s ‘package objects.’ Unlike objects, their members must be accessed via the scope resolution operator (`::`, like `ck::numPes`). Their EBNF is shown here:

$$\begin{aligned} \langle \text{namespace} \rangle \mid &= \text{namespace } \langle \text{fully-qualified-name} \rangle \\ &\{ (\langle \text{namespace} \rangle \mid \langle \text{namespace-member} \rangle) * \} \end{aligned} \quad (2.2)$$

Functions

We allow functions at any scoping level and, generally, require that their definition is present alongside their declaration. Only purely virtual and specific annotated methods may be undefined (e.g., `@system`). Ergoline supports ‘auxiliary’ lists of implicit parameters (see *Implicit Variables*), and we plan to support partially-applicable functions in the future (i.e., binding values to parameter lists to convert a function to a sequence of functions). We show their EBNF here:

$$\begin{aligned} \langle \text{function} \rangle \mid &= \text{def } \langle \text{identifier} \rangle \langle \text{template-declaration} \rangle ((\langle \text{argument-list} \rangle)) * \\ & (: \langle \text{type} \rangle) ? \langle \text{where-clause} \rangle ? (\langle \text{block} \rangle \mid ;) \end{aligned} \quad (2.3)$$

Classes

Ergoline’s classes and structures most closely resemble Swift’s. They are syntactically similar to Scala’s but permit “static” members like C++ and Java. The primary differ-

²Although, like Chapel and Scala, variable declarations are ordered relative to each other to allow dependencies between their initializers.

ence between structures and classes is their storage convention, i.e., in-place or always-heap allocated.³

In-place structures trade some flexibility for performance since they are faster to de/allocate and access (e.g., on the stack within functions). In the context of HPC, fixed-sized “structs” also benefit from pointer-to-offset optimizations during de/serialization (discussed in Section 4.2.1). Note that we do not allow them to have virtual methods, so they may not be abstract or derive from classes and traits.

We manage instances of heap-allocated classes using Automatic Reference Counting (ARC) by default. Like Swift, we offer alternative ownership schemes to offset the overheads of ARC, either “borrowed” or un/owned. Chapel has similar notions; however, its variables are “owned” by default while ours are “shared.” If an entry method only receives one value via pointer-to-offset (see Sections 4.2.1 and 4.2.2 for discussion of this optimization), it can use ‘owned’ semantics. However, when they receive multiple pointer-to-offset values, they must use ‘shared’ semantics. When we enable our compiler to emit errors and warnings for these situations, we could align ourselves with Chapel’s better-performing default option.

Traits

Traits are effectively purely abstract classes, albeit with the caveat that classes may implement any number of traits but only extend one class. Java’s interfaces are similar to Scala’s traits, which inspired ours. Looking forward, Swift’s protocols or C++’s concepts would be more favorable in the context of HPC. They rely on compile-time specialization instead of virtual methods, lowering their overheads and enabling their use with structures. We deferred implementing them due to their additional complexity.

Objects

We adopt Scala’s notion of objects rather than C-like global variables. These classes use the ‘object’ keyword rather than ‘class’ or ‘struct.’ They have exactly one instance, lazily created by the RTS upon the first access of their properties or methods. We discuss them more and present an example in *Global Values*.

³Their allocation strategy also determines their default argument passing technique, i.e., pass by value or reference.

Generic Functions and Types

Generic functions and types are critical for effective code reuse, featuring in most modern languages (C++, Java, Rust, Scala, Swift, etc.). Charj sought to deliver expressive and performant code for generics, adopting Java’s straightforward syntax and C++’s underlying approach [6, p. 28]. Compile-time specialization gives C++ a performance advantage, whereas Java relies on runtime typing decisions.

We follow in Charj’s footsteps, albeit by applying C++’s approach to another JVM-based language, Scala. However, where Charj could easily compile Java’s syntax to C++, Scala’s support for covariant and contravariant type relationships remains under-supported in Ergoline. Although we correctly type-check them, the underlying C++ compiler may fail to compile the generated code.⁴ Looking toward EIRv2, we aim to resolve this gap by compiling to a custom object model (vs. reusing C++’s). Additionally, in EIRv2, we plan to support C++-like template instantiation (e.g., specializing the type parameter `A` of `class foo<A>` to `int` via `class foo<int!>`). While useful to abstraction authors, neither feature is immediately necessary to establish Ergoline as a proof-of-concept. Toward that end, Ergoline has already gone beyond Charj by supporting generic entry methods and chare-types—discussed in 4.1.4.

Variance Relationships Ergoline provides two operators for querying the relationships between classes, the upper-bound (`<:`) and lower-bound (`>:`) operator. These operators evaluate whether two classes are related by a parent-child relationship, e.g., `Cat <: Animal` returns true since all cats are animals. Here, it is our intent to mirror the functionality of Scala’s equivalent operators.

2.2.3 Top-Level Statements

This section describes all the top-level statements that interact with Ergoline’s module system. Our inspiration for it was Python’s module system, which infers package structures from the arrangement of files and directories. For example, the fully-qualified name `foo::bar` refers to a file `bar.erg` in a folder named `foo/`. Like Scala, `package foo;` must be the first line of the file `bar.erg`, which also has to publicly expose a construct named `bar`. Ergoline discovers all files in its module path at startup; however, it does not load them until it

⁴C++ provides no accommodations for these features. Consider a class ‘Cat’ that derives ‘Animal.’ If `std::vector`, a popular generic collection, supported contravariance, one could cast an instance of `std::vector<Cat>` to `std::vector<Animal>`. However, we are currently unaware of a way to facilitate these semantics.

encounters them during name resolution.

Import Statements

In Ergoline, modules gain access to code in other modules via the `import` statement. Typically, imported code is file-private, i.e., only accessible within the file containing the corresponding `import`; however, it may be made `public` so other modules can share it. Ergoline does not support Python-like import-aliases at this time (e.g., `import numpy as np`); however, we will support them in the future.

$$\begin{aligned} \langle \text{import-statement} \rangle \quad & \models \text{public? import } \langle \text{fully-qualified-name} \rangle & (2.4) \\ & (:: \langle \text{identifier-list} \rangle)? ; \end{aligned}$$

Ergoline minimizes its reserved keywords and default behaviors to maximize extensibility. Subsequently, it does not import its own namespace by default! Most Ergoline programs start with `import ergoline::-`; to import all modules within the `ergoline` namespace. Some programs only import the modules they need, e.g.: `import ergoline::{int, bool}`;

Package Statements

Well-formed Ergoline programs start with a `package` statement to indicate their host package. Packages correspond to folders on a machine; Ergoline source files must be in a directory whose name matches their indicated package. This property is enforced during the file-loading process, with mismatches causing compiler errors (a product of pedanticness).

$$\langle \text{package-statement} \rangle \quad \models \text{package } \langle \text{fully-qualified-name} \rangle ; \quad (2.5)$$

Using Statements

Ergoline’s “using” statements blend the functionality of C++’s “constexpr” (i.e., compile-time evaluated expressions) and “using” statements. Here, Ergoline only needs one construct since it does not distinguish between static and type expressions (unlike C++). This construct aliases its right-hand side with the name on its left-hand side; the compiler cannot distinguish between aliases and their true identities. Generic “using” statements pass type parameters through to their right-hand side. We show their EBNF form here:

$$\langle \text{using-statement} \rangle \quad \models \text{using } \langle \text{identifier} \rangle \langle \text{template-declaration} \rangle? = \langle \text{expression} \rangle ; \quad (2.6)$$

2.2.4 Control-Flow Statements

Branching

Ergoline features C-style if-then-else branching, as follows:

$$\langle \text{if-statement} \rangle \models \text{if } (\langle \text{expression} \rangle) (\langle \text{statement} \rangle \mid ;) (\text{else } (\langle \text{statement} \rangle \mid ;))? \quad (2.7)$$

Return

Ergoline features C-style return statements, as follows:

$$\langle \text{if-statement} \rangle \models \text{return } \langle \text{expression} \rangle? ; \quad (2.8)$$

Scala automatically returns the last expressions of a function’s terminal nodes, as determined by a control-flow analysis pass. Thus, using the ‘return’ keyword in Scala is generally discouraged. Eventually, we plan to implement similar features in Ergoline to make code less verbose. To that end, we have already implemented them for match-case expressions.

Loops

Ergoline supports while, do-while, and for loops, supporting both range-based and C-style headers for the latter. We present their EBNF forms here:

$$\langle \text{for-header} \rangle \models \langle \text{decltypes} \rangle < - \langle \text{expression} \rangle \mid \langle \text{declaration} \rangle? ; \langle \text{expression} \rangle? ; \langle \text{expression} \rangle? \quad (2.9)$$

$$\langle \text{for-loop} \rangle \models \text{for}(\langle \text{for-header} \rangle)(\langle \text{statement} \rangle \mid ;) \quad (2.10)$$

$$\langle \text{while-loop} \rangle \models \text{while}(\langle \text{expression} \rangle)(\langle \text{statement} \rangle \mid ;) \quad (2.11)$$

$$\langle \text{do-while-loop} \rangle \models \text{do}(\langle \text{statement} \rangle \mid ;) \text{while}(\langle \text{expression} \rangle); \quad (2.12)$$

Ergoline presently does not support **break** and **continue** statements in its loops, unlike other imperative programming languages. We would like to include Java-like labeled **break** and **continue** statements, provided a clean integration path exists relative to the segmentation pass (see Section 3.1). It is unclear how these statements, along with **return**, should interact with structured entry methods. For now, we generate a compilation error when a structured entry method contains a **return** statement; while we could do the same for a

`break` or `continue`, we believe that it would be better to give them well-defined semantics. Note that Ergoline supports empty for-loop headers (i.e., `for(;;)`) like C/C++. Like infinite while-loops (i.e., `while(true)`), their utility is questionable without `break` since they can only be exited using `return`.

Match-Case

Ergoline has similar pattern matching facilities to Scala. The EBNF form of its match-case construct is shown here:

$$\langle \text{match-expression} \rangle \models \text{match } (\langle \text{expression} \rangle) \{ \langle \text{case} \rangle * \} \quad (2.13)$$

$$\begin{aligned} \langle \text{case} \rangle \models & \text{case } \langle \text{pattern-list} \rangle (\text{if } \langle \text{expression} \rangle)? \Rightarrow (\\ & \langle \text{statement} \rangle \mid ;) \end{aligned} \quad (2.14)$$

$$\langle \text{pattern-list} \rangle \models (\langle \text{pattern} \rangle ,) * \langle \text{pattern} \rangle \quad (2.15)$$

$$\begin{aligned} \langle \text{pattern} \rangle \models & \langle \text{extractor-pattern} \rangle \mid \langle \text{constant-pattern} \rangle \mid \\ & \langle \text{expression-pattern} \rangle \mid \langle \text{identifier-pattern} \rangle \end{aligned} \quad (2.16)$$

Rather than presenting their EBNF forms here, we show examples of the four pattern types in Figure 2.1. In Ergoline, the wildcard pattern (`_`) is a special type of identifier pattern which matches all values. Note, matching expressions step through their `case` statements in program order. If they reach the end without finding a match, the program aborts.⁵

Extractor Objects Extractor objects ‘extract’ values from objects via statically-accessible `unapply` methods. For example, the pattern `case some(x) => ...` corresponds to a call to `some::unapply(x)`. Well-formed `unapply` methods return an optional, whose value is present when the arguments match a condition. Pattern-matching expressions move onto the next `case` when the result is undefined.

2.2.5 Expressions

Ergoline has all the expression kinds that one would expect from a modern programming language. Since their users know these features well, we will highlight those that benefit HPC developers in this section.

⁵Aborting reflects Ergoline’s lack of exception-handling facilities.

```

def testTimekeeper(x: option<timekeeper>) {
  match (x) {
    case some(x: calendar) => {
      val leapYear =
        (x.numYears % 4 == 0) && (
          (x.numYears % 100 != 0) ||
          (x.numYears % 400 == 0)
        );
      x.numDays match {
        case 366 if leapYear =>
          println("valid leap year")
        case 365 if !leapYear =>
          println("valid year")
        case _ =>
          println("invalid number of days");
      }
    }
    case some(x: clock) =>
      (x.numHours, x.numMinutes, x.numSeconds) match {
        case (_ >= 24, _, _) => abort("too many hours");
        case (_, _ >= 60, _) => abort("too many minutes");
        case (_, _, _ >= 60) => abort("too many seconds");
        case (hh, mm, ss) if ((hh < 0) || (mm < 0) || (ss < 0)) =>
          abort("negative time");
        case _ =>
          println("valid time");
      }
    case some(_) => abort("unknown timekeeper");
    case none() => abort("missing timekeeper");
  }
}

```

Figure 2.1: An example demonstrating all kinds of non/compound patterns.

Lambdas

Modern languages with first-class functions feature lambda or arrow expressions to create anonymous functions. Ergoline supports them with Scala-like syntax, shown below. Lambdas *capture* in-scope variables when they reference them. Ergoline only supports by-reference captures—while other languages may also allow by-value captures (e.g., C++). Effectively, this means that users can always mutate the values of variables referenced in Ergoline’s lambdas (whereas by-value captures cannot be re-assigned).

$$\langle \text{lambda-expression} \rangle \models (\langle \text{identifier} \rangle \mid (\langle \text{argument-list} \rangle)) \Rightarrow \langle \text{expression} \rangle \quad (2.17)$$

```

val x = 42;
println('x equals ${x}'); // => x equals 42
println('${t-strings are nested like this}');
println('${
    "and they may be multi-line!"
}');

```

Figure 2.2: Example of interpolated string in Ergoline.

De/serialization HPC-oriented C++ libraries rely on macros or other ‘non-idiomatic’ solutions (e.g., ‘functors,’ classes that act like functions) to support lambda de/serialization. To our knowledge, none support lambdas with captures. Restrictions like these will be present until C++ includes mechanisms for identifying lambdas and which variables they capture.⁶ Although the workarounds satisfy users’ functional requirements, they demonstrate an area where language support can benefit HPC libraries.

Ergoline’s compiler enables it to support de/serializing lambdas with captures with idiomatic syntax. This process involves two steps. First, it statically registers all the lambdas, generating a table that maps IDs to function pointers. This table is essential for establishing consensus since we generally cannot expect processes to load executables at the same address. The second step is to generate helper routines that de/serialize any variables captured by lambda expressions; subsequently, lambdas have unique copies of such variables after deserialization.

Interpolated Strings

Modern languages provide string interpolation as a convenient way to generate formatted strings. Ergoline borrows JavaScript’s syntax for its “template literals.” Their grammar is non-trivial, so we defer to Figure 2.2 for an example.

Operator Overloading

A key strength of Scala for language authors is its rich support for operator overloading. These facilities enable the natural expression of software abstractions and cleaner APIs. When used responsibly, Scala supports multiple embedded DSLs, i.e., via non-interfering

⁶A proposal like P1240R2 (2022), adding reflection capabilities to future C++ revisions, would satisfy this requirement.

```

class foo {
  def plus(i: int): foo { ... }

  def +(i: int): foo {
    return self plus i;
  }

  def unary_!(): foo { ... }
}

def bar() {
  var f: foo = new foo();
  // these statements are equivalent
  f = !f + 42;
  f = f.unary_!().+(42);
}

```

Figure 2.3: Operator overloading example.

overloads. Conventional embedding approaches may require a parsing step for each DSL; these steps have a greater risk of interfering with one another.

Ergoline replicates Scala’s operator overloading facilities, viewing operators as methods of classes. Virtually any identifier can serve as an infix operator, from `plus` to `+$#!%`. The first character of an identifier sequence decides its precedence, loosely according to the conventional ordering (with alphabetical symbols being the lowest). We provide Figure 2.3 as an example of overloads, but Scala’s resources may be more elucidating here [42].

Slicing Python’s slicing syntax provides a concise way to access and update ranges of values within collections. For example, `A[0:2:n]` selects the even elements of the container `A`. Like `Charj`, Ergoline incorporates slicing to inherit these benefits, e.g., reducing the Jacobi example’s LOC. Unlike Python, `Charj` and Ergoline’s slices are views of the original collection and not copies (an occasional source of performance problems in Python). Also, unlike `Charj` and `Charm4Py`, Ergoline’s `chare`-arrays are sliceable to create sections—an advancement facilitated by `Hypercomm`, see 5.3.2.

2.2.6 Advanced Features

Implicit Variables

Implicit variables hide information conveyance to reduce verbosity; function calls do not need to include implicit parameters when matching implicit variables are within scope. In Ergoline, both the types and names must match, whereas only the types of implicit variables and parameters must match in Scala.

Every entry method has an implicit variable set to the current element proxy within its scope. Since creating a future requires its creator’s identity, we frequently use this feature to pass the current proxy to STL functions that make futures. Our example Channel and DHT implementations demonstrate this use case in Section 2.4.

Exception Handling

HPC applications frequently disable exception handling during compilation (e.g., by using `-fno-exceptions`). The most noticeable effect is a decrease in binary size, with minor second-order performance effects (e.g., fewer branch mispredictions). Since languages like C++ employ “zero-cost” exception handling, they are unlikely to incur slowdowns until exceptions arise. Then, accessing cold memory regions and “unwinding the stack” will be costly. Regardless, disabling exception handling in C++ comes with a cost; many of its STL constructs rely on exceptions to communicate with users. Subsequently, this common HPC practice may result in hard-to-fix problems when exceptions would otherwise arise. For now, we rely on aborts with clear messages instead of including exception handling in Ergoline. Looking forward, we hope to incorporate a solution that HPC developers will find favorable.

2.3 ANNOTATIONS

Ergoline primarily uses annotations to specify its Charm++-related behaviors. While any syntax construct can be annotated, Charm++ annotations are limited to classes, members, types, and loops. In this section, we will explore our motivation for using annotations and present all the annotations in Ergoline.

2.3.1 Philosophy

Annotations allow users to add metadata to syntax constructs. For example, in Java, the

`@Override` annotation informs the compiler that a method is intended to override a method in a superclass. Often, compilers or libraries use annotations to facilitate extensibility, since their presence can cause semantic changes. For example, Python features the `@cache` annotation, which wraps a method with an LRU cache mapping its parameters to results. Whether languages and libraries utilize annotations is largely a philosophical decision, made relative to the other extensibility facilities that languages offer.

Where Java libraries heavily feature annotations, Scala libraries usually favor its “`implicit`,” type classes, and macros. To Scala users, these mechanisms avoid “polluting” logic with tangentially relevant code. Compared with annotations, these mechanisms are statically checkable and applied on use-sites rather than definitions; these advantages promote enhanced correctness and code reuse.

Python’s annotation use is somewhere between Scala’s and Java’s. While Python relies on annotations for some syntax extensions (e.g., using methods as properties) and to help convey a library author’s intent for a function (e.g., it should always be JIT’d), many libraries do not feature annotations at all. For example, Charm4Py does not use annotations to indicate chore entry methods; instead, all class methods are available through proxies. Like Scala, this decision results in cleaner code and better code reuse; any existing Python class is usable as a chore-type. Furthermore, this decision is generally consistent with Python’s lack of encapsulation. Without encapsulation, however, a developer’s intent is unenforceable, leading to subtle and avoidable failures.

We favored encapsulation and preservation of intent while designing Ergoline. Subsequently, we use annotations for many Charm-derived features, like entry method registration and configuration. We pass almost all user annotations through to EIR; this works well since it requires annotated IR from frontends (like other compiler frameworks). We plan to move towards annotation-free solutions in the future, striking a balance between expressiveness and preservation of intent. We may realize this goal by refining Ergoline (e.g., adding Scala-like macros) or introducing new EIR-based languages.

2.3.2 Listing

Many of Ergoline’s annotations mirror keywords in Charmxi or its generated code. Subsequently, its annotations should feel familiar to Charm++ users; however, its designated syntactic elements may appear foreign. We will outline Ergoline’s Charm++-specific annotations in this section, highlighting differences as they arise.

Collection	Section	Element	Mapping
@group	{@}group	[@]group	one per worker
@nodegroup	{@}nodegroup	[@]nodegroup	one per logical node
@array[n]d	{@}array[n]d	[@]array[n]d	arbitrary, n -dimensional
@	n/a	n/a	singleton

Table 2.1: All Proxy-suffixes in Ergoline

Type Annotations

Proxy suffixes are postfix annotations that, when attached to a type, make a corresponding proxy type—this process is called ‘proxification.’ Each indicates a specific arrangement of chares in a distributed system. Figure 2.1 shows all the proxy suffixes in Ergoline. Note that chares may be singletons or members of collections. All the collection kinds have suffix variants to refer to their entirety, sections, or individual elements.

Although proxy types follow the same typecast rules as their ‘base’ types, generally, one cannot change their suffixes via typecasting. There is one exception—an element proxy may be downcasted to a singleton proxy (but not vice versa). For example, if ‘Derived’ derives from ‘Base,’ one may cast `Derived[@]array1d` to `Base@` or `Base[@]array1d`. Finally, proxy base types may be generic; we discuss this later in Section 4.1.4.

Self-Proxy Operators The `self` keyword follows the same conventions as proxy-suffixes. For example, `self[@]contribute` and `self[@].contribute` are equivalent and read as “call the contribute method of this chare’s element proxy.” Note how the typical member access operator (i.e., dot) can be omitted for convenience.

Method Annotations

Only annotated members are accessible through proxies. Subsequently, while any type can be proxified, types without suitably annotated methods may be unusable or non-constructible. Among these annotations, the most common is `@entry`, registering its target as an entry method. Entry methods may be constrained to certain chare-types, for example, to enable classes to offer different behaviors as singletons or collective members; Figure 2.4 shows an example of these constraints. Lastly, entry methods cannot have return values by default, but other annotations can lift this restriction.

Threaded Entry Methods Typical entry methods cannot suspend and, once started, run until they complete. Threaded entry methods, however, can suspend and resume at-will

```

class foo {
    // usable in array2d collections
    @entry(array2d) def bar() { ... }
    // usable in either array1d or group collections
    @entry(array1d, group) def baz() { ... }
}

```

Figure 2.4: An example that controls entry method accessibility. Note that `@entry(array2d)` is syntactic sugar for `@entry(array2d=true)`.

since they run within user-level threads (ULTs). In this manner, `@threaded` entry methods are similar to coroutines in other programming models.

Both coroutines and threaded EPs are non-preemptive, only yielding when explicitly suspended, returning control to a scheduler. Where coroutines may suspend on one (worker) thread and resume on another, threaded EPs are bound to a particular ULT. Although, *Migratable Threads* can suspend on one PE and resume on another (enabled via `@migratable`). Note that only one entry method of a chore can execute at a time, regardless of threading. Generally, Ergoline employs the techniques discussed in Section 3.1 to ‘decay’ threaded entry methods into non-threaded ones, avoiding the complexities of ULTs (discussed later).

Asynchronous Entry Methods All entry methods are intrinsically asynchronous; subsequently, they typically do not `return` values. As the caller would have to wait for the callee, these semantics require synchronization, idiomatically achievable via futures in Ergoline. Users may pass futures as arguments to entry methods, waiting for them to set the futures’ values. However, this process is a little tedious and does not enable using `return` within entry methods.

To enable return values and eliminate user-level future creation, sending, and setting, Ergoline provides `@async` entry methods. A call to an `@async` method returns a future corresponding to its return type (e.g., `int` corresponds to `ck::future<int>`). The compiler creates and sends the future on the caller side, and replaces the “return” with a call that sets the future’s value. Figure 2.5 shows an example `@async` method and its usage.

Local Entry Methods Entry methods usually execute asynchronously—invocations generate messages that are processed later, potentially on another PE than the caller. The annotation `@local` alters this behavior, forcing them to execute synchronously on the same PE as the caller. The RTS aborts when the target is not locally present. Since they are synchronous, `@local` entry methods may have non-void return values. We utilize this property

```

class foo {
  @async @entry def foo(): int { return 21; }

  @entry def bar(f: future<int>) { f.set(42); }

  @threaded @entry def consume() {
    val f1 = self[@]foo();
    val f2 = future<int>();
    self[@]bar(f2);
    println('${(await f1)} ${(await f2)}'); // => 21 42
  }
}

```

Figure 2.5: Comparison between an `@async` entry method and its semantic equivalent.

in the Charisma EDSL, discussed in Section 3.3.

Demand Creation Chare-arrays usually buffer messages received for nonexistent elements. They will retain these messages until their destinations materialize, after which they attempt to deliver them. Demand creation alters this behavior, forming a specialized dynamic insertion operation. If an array receives a `@createhere` or `@createhome` message, it inserts a new element when the message’s destination does not exist. The chare-type must have a default `@entry` constructor (i.e., taking no parameters) for this operation to succeed. The variants of demand creation differ regarding where they create the new element, whether it is on the sender PE (here) or its home PE (determined by the array’s mapping scheme).

Mailboxes Although mailboxes are conceptually entry methods, they do not use the `@entry` annotation. Instead, Ergoline realizes them as body-less methods annotated with `@mailbox`, where mailboxes with bodies generate compilation failures. Users can post requests to mailboxes using when-clauses, specifying optional predicates with pattern matching. We discuss the compilation of these structures and their use in Section 3.1.

Proxy Methods Proxy methods (those annotated with `@proxy`) execute inline (on the caller side), and users must invoke them on a chare’s proxy. Generally, they only have access to proxy members (i.e., entry methods, mailboxes, and other proxy methods); however, they can gain access to local chares (i.e., those on the same PE) through the `ckLocal` proxy method. They only apply to element-like proxies by default (i.e., singletons and elements), but users can extend them to collections and sections. These methods provide a convenient way to hide “wrapper” code, shown later in Figure 2.10.

Main-chares

Like Charm++, users write main chares instead of functions in Ergoline, indicating them by annotating compatible classes with `@main`. Control enters user code via the default constructors of main-chares; there may be multiple. Typically, these kickstart the computation by creating other chares and chare-collectives.

Overlapped Loops

Annotating a for-loop within an entry method with `@overlap` indicates that its iterations can be performed independently, in any order. Each iteration can individually suspend and resume, but all run on the host chare’s current PE. Subsequently, they execute concurrently but not in parallel. In a sense, overlapped for-loops have an implicit barrier after them; all iterations must complete for execution to proceed. When nested, innermost loops do not block. Overlapped `for` loops correspond to `forall` loops in SDAG, although any kind of loop is overlappable in Ergoline.

2.3.3 Foreign-function Interface (FFI)

The `@system` attribute can be used to indicate that *Ergoline* types and methods correspond to C++ code, forming the basis of *Ergoline*’s C++ *foreign function interface* (FFI). It has various options for expressing code generation nuances, like whether a method should be treated as an operator or a type-cast expression. Additionally, `@system` classes may have methods written in *Ergoline*, permitting users to extend C++ classes with *Ergoline*-specific functionality. Figure 2.6 features *Ergoline*’s wrapper for C++’s `std::int32_t` to demonstrate these ideas.

2.4 STANDARD LIBRARY

A programming language’s standard library is a contributing factor to its usability. Ergoline’s Standard Template Library (STL) aims to include a comprehensive selection of parallel abstractions and other essentials for HPC development. Here, its runtime adaptivity, generic types, and operator overloading foster expressiveness without compromising performance and scalability. In this section, we highlight components of Ergoline’s STL and the C++ Foreign Function Interface (FFI) that occasionally facilitates them.

```

package ergoline;

@system(alias="std::int32_t", fromHeader="cstdint")
struct int {
    ...
    @system(operator, alias="-")
    def <=>(other: int): int;
    // Ergoline(1 <=> 2) => C++(1 - 2)

    def to(end: int): range<int> {
        return new range<int>(self, 1, end);
    }
    // Ergoline(1 to 2) => C++(int_to(1, 2))

    @system(cast) def toBool(): bool;
    // Ergoline(1.toBool()) => C++((bool)1)

    @system(static, alias="std::to_string")
    def toString(): string;
    // Ergoline(1.toString()) => C++(std::to_string(1))
    ...
}

```

Figure 2.6: *Ergoline*’s `int` type demonstrates many of the features of its C++ FFI. Note, attribute options without values are assigned the default value of `true`.

2.4.1 Global Values

HPC applications often compute values based on user inputs at startup, e.g., determining data distribution from command-line arguments. They distribute these values to all parallel entities, rarely updating them later. To accommodate this common need of HPC applications, Charm++ provides `readonly` variables. After users assign their values in startup routines, the RTS broadcasts them to all PEs. PEs start the rest of the computation after their copies are updated; afterward, users cannot change their values again. Charm4Py provides similar functionality via its global variables, although users must explicitly broadcast their re-assignable values.

In *Ergoline*, we blended Charm4Py’s scheme with Scala’s notion of singleton objects. These are classes with one instance, created after the first reference. Like global variables in Charm4Py, a PE can broadcast a local object’s state to all other PEs using the `ck::updateGlobal` function. This function accepts a callback as an argument, invoking it

after all PEs have updated their object’s state. A call to `ck::updateGlobal` inside a mainchare constructor can replicate `readonly`-like semantics, shown in Figure 2.7. Note, multiple unsynchronized calls to `ck::updateGlobal` with the same target object will cause undefined behavior.

<pre> // in the .ci file: mainmodule example { readonly std::string inFile; mainchare Main { entry Main(CkArgMsg*); entry void loadFile(void); }; } // in the cpp file: Main::Main(CkArgMsg* msg) { inFile = msg->argv[1]; thisProxy.loadFile(); } </pre>	<pre> object parameters { var inFile: string; } @main class main { @entry def self(args: array<string>) { parameters.inFile = args[1]; /* broadcast this PE's value for * 'parameters', then call * 'loadFile' after all PEs * updated their copies */ ck::updateGlobal<parameters>(self@loadFile); } @entry def loadFile() { ... } } </pre>
(a) Read-only variables (Charm++)	(b) Singleton objects (Ergoline)

Figure 2.7: Comparison of read-only variables and singleton objects.

2.4.2 Extended Future Facilities

Many concurrent programming models feature future values as a cornerstone abstraction; for example, those of C#, C++, Rust, and Scala. Futures are, effectively, proxies for results whose values have not been computed yet. Concurrent entities can suspend until a future’s value becomes available, enabling synchronized program execution. Charm++ supports futures primarily to bridge gaps between it and other programming models; since they are synchronous, they are rarely the most “Charm-like” way to solve problems (vs. SDAG’s asynchrony, for example) and, subsequently, have seen little development. Nonetheless, we enhanced Ergoline’s support for futures to strengthen these bridges, enabling them to interact with more aspects of the Charm++ ecosystem (migrations, SDAG, etc.).

The primary difference between Charm++’s and Ergoline’s futures is their issuers. Both

implementations represent futures as issuer and (issued) tag pairs. When one sets a future value, it routes to its issuer, where they are re-routed to the agent in waiting (usually a ULT). In Charm++, only a chare co-located with the issuer can wait for a future value, mitigating the performance impact of this indirection step. This property is a product of Charm++’s choice of issuing agents—PEs; all chares on the same PE share a (bounded) pool of future tags. Comparatively, in Ergoline, each chare can issue future tags, enabling unhindered migration and greatly expanding the number of potential futures.

Hypercomm implements Ergoline’s futures in terms of its components model; this is discussed further in *Futures*. Critically, chares send value-forwarding requests when they need the value of remote futures, where values are always sent to the future’s issuer and routed from there.⁷ Enforcing that values can be retrieved only once enables the system to manage the memory associated with futures; it simply transfers memory ownership when values are retrieved. Comparatively, in Charm++, users may retrieve a future’s value multiple times since the RTS retains a copy until `CkReleaseFuture` is called. Supporting this use case in Ergoline would require adding a future replication mechanism, enabling n -many retrievals.

We plan to add ‘and’ and ‘or’ methods to Ergoline’s futures soon. The ‘and’ function joins the results of all its components into a single tuple. On the other hand, the ‘or’ method generates a variant containing the first available value. Subsequently, these features are effectively syntactic sugar for the wait ‘any’ and ‘all’ methods of the underlying model (i.e., Charm++). Figure 2.8 shows examples of their use.

```
// given:
val f: future<int> = ...
val g: future<double> = ...
// then:
f or g // => future<either<int, double>>
f and g // => future<(int, double)>
```

Figure 2.8: An example demonstrating the types generated by the ‘and’ and ‘or’ methods of Ergoline’s futures. If we alias the ‘pipe’ operator with variant types (e.g., Swift, Python), then `future<(int | double)>` would be the most concise way to express the result of the ‘or’ method.

2.4.3 Channels

Many programming models incorporate channels as a mechanism for point-to-point messaging between parallel entities. Since channels have numerous design choices associated

⁷We proposed a similar solution for Charm++, albeit in terms of PEs instead of chares.

with them, their implementations can vary widely. Even within a single programming model, channels may offer many different options.

For example, Go’s channels are multi-producer, multi-consumer; however, users can restrict their direction to alter their behavior. By default, a channel operation blocks until it handshakes with its corresponding receive or send. Buffering enables non-blocking sends; without which, Goroutines, the Go notion of coroutines, could not send values to themselves. Goroutines can use the ‘select’ construct to poll for messages from multiple channels, shown in Figure 2.9. We plan to enable Ergoline’s when-clauses to receive values from channels and futures, allowing its ‘await any’ construct to behave like Go’s ‘select.’ Outside of Go, other programming models may have similarly nuanced channel implementations.

The diversity of channel implementations implies that there is not a “definitive” way to implement channels. Towards this end, if Ergoline’s channels do not meet a programmer’s need; they are free to implement their own. Ergoline’s STL channels are buffered with non-blocking sends and asynchronous receives (via futures), and they are implemented as singleton chares in less than 40 lines of code. Figures 2.10 and 2.11 show their complete implementation and an example use-case, respectively. This succinctness and usability generally cannot be achieved in Charm++ due to the complexities of generic types discussed in Section 4.1.4.

```
c1 := make(chan string)
c2 := make(chan string)

...

for i := 0; i < 2; i++ {
    select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
    }
}
```

Figure 2.9: Excerpt of a Go program demonstrating its ‘select’ construct.

```

package ck;

import ergoline::queue;

class channel<A> {
  private val values: queue<A>;
  private val pending: queue<future<A>>>;

  @entry def self() {
    values = new queue<A>();
    pending = new queue<future<A>>>();
  }

  @entry def send(a: A) {
    if (pending.isEmpty()) {
      values.append(a);
    } else {
      pending.head().set(a);
      pending.removeHead();
    }
  }

  @entry def receive(f: future<A>) {
    if (values.isEmpty()) {
      pending.append(f);
    } else {
      f.set(values.head());
      values.removeHead();
    }
  }

  @proxy @sync def get()(implicit __proxy__: proxy): A {
    val f = future<A>();
    self@receive(f);
    return f.get();
  }
}

```

Figure 2.10: Reference channel implementation in Ergoline.

2.4.4 DHTs and Tuple Spaces

Distributed collections help HPC applications share data between parallel entities, often resembling sequential counterparts. Distributed hash tables (DHTs) use keys to map values to PEs. Requests route to values' locations, waiting until they become available then

```

package examples;

import ergoline::_;
import ck::channel;

@main class receiver {
    val ch: channel<int>@;

    @entry def self() {
        ch = new channel<int>@();
        new sender@(ch);
        self@receive();
    }

    @threaded @entry def receive() {
        for (var i = 0; i < 16; i += 1) {
            println('received value: ${await ch}');
        }
        exit();
    }
}

class sender {
    @entry def self(ch: channel<int>@) {
        for (var i = 0; i < 16; i += 1) {
            ch.send(i);
        }
    }
}

```

Figure 2.11: Example program using a channel to exchange values between two chares.

consuming them (i.e., they become unavailable to other requests). Distributed tuple spaces function similarly, storing arbitrary tuples instead of key-value pairs. Most tuple spaces allow querying for tuples by any of their constituents. However, works like DTuples implement tuple spaces as DHTs, restricting value retrieval to by-keys [43]. Buravlev et al. discuss this and other tuple space design factors [44].

DHTs have been a component of the Charm++ ecosystem since its inception. The predecessor of Charm++ (i.e., Charm) featured DHTs as an information-sharing mechanism [45]. The Charm++ tutorial guides users through implementing a straightforward key-value store. Subsequently, we wanted to include a DHT implementation in Ergoline’s STL. We split our implementation across two classes, using one as a chare-array and the other as a

lightweight wrapper. The wrapper provides convenient operator overloads for utilizing the DHT’s features. It hashes keys and decides where to route values and requests; we modulo the hashes with the count of array elements to obtain indices. Figure 2.12 shows these classes, with Figure 2.13 showing an example of their use. We implemented a tuple-space as a lightweight wrapper around our DHT in the style of DTuples (not shown).

2.5 EXAMPLE PROGRAMS

In this section, we present snippets of example programs written in Ergoline. Many resemble their counterparts in Charm++, albeit with tighter integration between the programming model (migratable objects) and its host language (C++ vs. Ergoline). In practice, this integration leads to less “boiler-plate” code and the elimination of repeated information across C++ and Charmxi source files.

2.5.1 Fibonacci

Figure 2.14 contains a program that recursively creates “fib” chares that compute the N^{th} Fibonacci number and send the result to their parent. Since the chares disperse throughout systems, this program exhibits divide-and-conquer parallelism. It calculates numbers below a threshold sequentially to limit the number of chares spawned. Here, we favor clarity instead of performance; e.g., we could use table lookups instead of sequential computations.

There are a few points of interest here. First, a type with an `@` suffix is a proxy that provides access to its underlying type’s `@entry` methods. Constructing an instance of a proxy-type creates a chare. Second, within `@entry` methods, chares can access their proxy using the `self@` syntax; note, this syntax always refers to a chare’s ‘most complete’ representation—the entire collection for its members and, for singletons, the chare itself.. Lastly, users must declare variables within functions or classes in Ergoline. The `configuration` class exemplifies this requirement and, as an `object` type, automatically has a singleton instance.

Our Ergoline example is 54 LOC, whereas Charj’s was 58 LOC [6, p. 35]. Here, Ergoline’s support for abstract chare-types (i.e., traits with entry methods) simplifies this program since “fib” chares no longer need to distinguish between sends to “main” or “fib” chares. Ergoline’s compiler generates abstract proxies to support this feature, as discussed in Section 4.1.4.

```

package ck;
import ergoline::_;

class dht<K, V> {
  private val proxy: dht_<V>@array1d;
  private val n: int;

  def self(=n: int) {
    proxy = new dht_<V>@array1d(n, n);
  }

  def get(k: K)(implicit __proxy__: ck::proxy): future<V> {
    val h = hash<K>(k);
    val f = future<V>();
    proxy[(h % n).toInt()].get(h, f);
    return f;
  }

  def set(k: K, v: V) {
    val h = hash<K>(k);
    return proxy[(h % n).toInt()].set(h, v);
  }
}

class dht_<A> {
  @hashExclude private val data: map<hashcode, A>;
  @hashExclude private val waiting: map<hashcode, queue<future<A>>>;
  private val n: int;

  @entry def self(=n: int) {
    data = new map<hashcode, A>();
    waiting = new map<hashcode, queue<future<A>>>();
  }

  private def enqueue(h: hashcode, f: future<A>) {
    if (!waiting.contains(h)) {
      waiting[h] = new queue<future<A>>();
    }

    waiting[h].append(f);
  }

  @entry def get(h: hashcode, f: future<A>) {
    if (data.contains(h)) {
      f.set(data[h]);
      data.remove(h);
    } else {
      enqueue(h, f);
    }
  }

  @entry def set(h: hashcode, a: A) {
    if (waiting.contains(h)) {
      val q = waiting[h];
      q.head().set(a);
      q.removeHead();
      if (q.isEmpty()) {
        waiting.remove(h);
      }
    } else {
      data[h] = a;
    }
  }
}

```

Figure 2.12: Reference DHT implementation in Ergoline.

```

@threaded @entry def example(table: ck::dht<int, int>) {
    val ours = self[@].index();
    val theirs = (ours + 1) % n;

    for (var i = (ours * k); i < ((ours * k) + k); i += 1) {
        table[i] = 2 * i + 1;
    }

    for (var i = (theirs * k); i < ((theirs * k) + k); i += 1) {
        val value = await table[i];
        println('${ours}> received table[${i}]=${value}');
    }

    self[@].contribute(mainProxy.done);
}

```

Figure 2.13: Example program using a DHT to exchange values between n chares, each sets a range of values then reads its rightmost neighbor’s values.

2.5.2 Matrix-matrix Multiplication

Figure 2.15 shows the core logic for Cannon’s matrix-matrix multiplication in Ergoline. In each iteration, elements wait to receive blocks of values from the input matrices. After the blocks arrive, the elements perform a local DGEMM operation and then forward the blocks onto the next element. We express this operation as a compound when-clause that requests messages from both input mailboxes. Blocks can arrive in any order since there is not a global reduction at the end of each iteration. Subsequently, we tag each block with its iteration then use pattern-matching to select the correct blocks. The compound clause’s constituents express this pattern matching using `_ == block`, which matches the messages’ tags with the current iteration.

The chares in this example are members of a collection; this means they have an element proxy, accessible via the `self[@]` syntax. These proxies provide access to an element’s position within a collection (i.e., via `self[@].index()`) and the ability to participate in reductions. The latter feature is demonstrated by `self[@].contribute(mainProxy.done)`, which calls the entry method `done` of the main-charge once all elements have contributed.

Since the `self@` syntax refers to a collection, we can subscript it to select elements, e.g., `self@[0]` chooses the element at index zero. Here, we use this feature to forward blocks once the DGEMM has finished. This example contains buffer reuse opportunities, since it forwards them to similar entry methods from whence they came (see Section 4.2.2).

```

package examples;

import ergoline::_;

@main class main with acceptor {
    val n: int;

    @entry def self(args: array<string>) {
        n = (args.size() > 1) ? args[1].toInt()
            : (configuration.threshold * 2);
        new fib@(self@, n);
    }

    @entry override def accept(x: int): unit {
        println('fib({n}) = {x}');
        exit();
    }
}

class fib with acceptor {
    var count: int = 0;
    var value: int = 0;
    val parent: acceptor@;

    @entry def self(=parent: acceptor@, n: int) {
        if (n <= configuration.threshold) {
            parent.accept(seq_fib(n));
        } else {
            new fib@(self@, n - 1);
            new fib@(self@, n - 2);
        }
    }

    @entry override def accept(x: int): unit {
        count += 1;
        value += x;

        if (count >= 2) {
            parent.accept(value);
        }
    }

    private def seq_fib(n: int): int {
        return (n > 1) ? (seq_fib(n - 1) + seq_fib(n - 2)) : n;
    }
}

object configuration {
    val threshold: int = 16;
}

trait acceptor {
    @entry def accept(x: int): unit;
}

```

Figure 2.14: A complete example that calculates the N^{th} Fibonacci number in Ergoline.

2.5.3 Jacobi Relaxation

This example program is a formulation of the Jacobi method as an Iterative Stencil Loop; it applies to problems like heat diffusion. We distribute the input matrix across a 2D chare-

```

@mailbox def inputA(tag: int, block: array<double, 2>);
@mailbox def inputB(tag: int, block: array<double, 2>);

@threaded @entry def run(alpha: double) {
  val idx = self[@]index();

  @overlap for (var block = 0; block < nBlocks; block += 1) {
    when inputA(_ == block, blockA: array<double, 2>),
      inputB(_ == block, blockB: array<double, 2>) => {
      blas::dgemm(alpha, blockA, blockB, beta, data);

      if ((block + 1) < nBlocks) {
        self@[idx[0] + 1] % nBlocks.inputA(block + 1, blockA);
        self@[idx[0], (idx[1] + 1) % nBlocks].inputB(block + 1, blockB);
      }
    }
  }

  self[@].contribute(mainProxy.done);
}

```

Figure 2.15: Run loop for Cannon’s matrix-matrix multiplication.

array. Each iteration begins with elements exchanging “halo” values with their neighbors. Then, they compute updated values and participate in a reduction to determine whether the maximum change is less than a threshold. Figures 2.16 and 2.17 show the iterative process and compute kernel, respectively. We omit the initialization logic for the sake of brevity. Note, when-clauses request a value from a mailbox and block until it is received, triggering their body afterward. We explain these concepts in depth later in Section 3.1.

Condensed versions of **Jacobi2D** in **Ergoline** and **Charm++** require 176 and 324 lines of code (LOC), respectively. **Ergoline**’s count is roughly consistent with **Charj**, which had a 170 LOC **Jacobi2D** example [6]. **Charj** has a slight advantage because it uses a 1D decomposition, whereas our 2D decomposition results in more complex halo exchange logic.

2.6 SUMMARY

In this chapter, we presented **Ergoline**’s syntax alongside example programs and features of its standard library. Its example programs showcased how it compares to **Charj** and **Charm++**, demonstrating areas where it enhances the migratable objects programming experience. We showed how its support for generic functions and types led to expressive

representations of common HPC abstractions. Lastly, we discussed the language’s philosophy and sources of inspiration, hinting at the extensibility that facilitates embedding DSLs within it.

```
@mailbox def receive_ghost(it: int, dir: int, data: array<double>);
@mailbox def receive_status(g_converged: bool);

@threaded @entry def run() {
  var it = 0;
  var converged = false;
  // for maxIters iterations or until all chares converge:
  for (; !converged && (it < globals.maxIters); it += 1) {
    // send our ghost values to our neighbors
    send_ghosts(it);
    // then process each of their messages
    for (var imsg = 0; imsg < numNeighbors; imsg += 1) {
      // ( filtering by the iteration is demonstrative, not necessary )
      when receive_ghost(_ == it, dir, data) => {
        process_ghost(dir, data);
      }
    }
    // compute the max difference for this iteration
    val maxDiff = check_and_compute();
    // compare it against the threshold to see whether we’ve converged
    converged = (maxDiff <= globals.threshold);
    // then check to see whether all chares have converged
    self[@].contribute(converged, bool::logical_and, self@receive_status);
    // and update our status accordingly
    when receive_status(x) => converged = x;
  }
  // the leader tells main how many iterations it took to converge
  if (self[@].index() == (0, 0)) {
    globals.mainProxy.done(it);
  }
}
```

Figure 2.16: Run loop for Jacobi2D

```

private def check_and_compute(): double {
    var maxDiff = 0.0;

    for (var i = iStart; i < iFinish; i += 1) {
        for (var j = jStart; j < jFinish; j += 1) {
            nextGrid[i, j] =
                0.2 * (grid[i, j] +
                    grid[i - 1, j] + grid[i + 1, j] +
                    grid[i, j - 1] + grid[i, j + 1]
                );

            val diff = math::abs(nextGrid[i, j] - grid[i, j]);
            maxDiff = math::max(diff, maxDiff);
        }
    }

    // swap grids for the next iteration
    (grid, nextGrid) = (nextGrid, grid);

    return maxDiff;
}

```

Figure 2.17: Compute function for Jacobi2D

CHAPTER 3: EMBEDDED DSLS

In this chapter, we discuss the EDSLs we explored within the context of Ergoline and EIR, ordering them in terms of the concreteness of their implementation. Our SDAG DSL comes first, delivering demonstrable advancements. Then, we discuss a Basilisk-inspired Stencil EDSL that reuses its components. Next, we reimagine Charisma as an embedded DSL, blending global and local control views within a single program. Our work on MSA comes next, introducing a novel approach for inferring synchronization points. Although we primarily implemented its code in the context of Scala-Charj, aspects of the work apply to EIR/Ergoline as well. Lastly, we sketch out a hypothetical implementation of DivCon as an EDSL in Ergoline, which would add Cilk-like task-parallelism to its repertoire.

Each DSL aims to address a common need of HPC programmers. Where most required unique infrastructure in previous approaches, they are unified under EIR and demonstrate the benefits of compositionality. Finally, they exercise Ergoline’s generality and leverage Hypercomm’s abstractions.

3.1 SDAG

Consider an asynchronous message-driven model where remote method invocations (RMIs) do not block. Parallel entities use callbacks to request data, directing where to send data when it is ready. Although models like this enable transparent communication-computation overlap, they can obfuscate control flow. The messages that code segments depend on may not be immediately evident since expressing conditional logic can be non-trivial.

Imagine that we want to wait for n messages of type A and B . After receiving an A message, we want to wait for a reply of C . In practice, this interaction pattern would translate to a series of if-then-else statements with multiple counters; comprehending them would require significant interpretation. In contrast, Figure 3.1 demonstrates how Structured DAGger (SDAG) can concisely express this slightly contrived example.

SDAG is a notation for coordinating dependencies between computations and messages. It is a deeply-embedded horizontal DSL that supports common idioms for applications depending on asynchronous method invocations.¹ It has a partitioned control view; each structured entry method directs an individual chore. Users write code that resembles sequential control flow within these methods, leveraging SDAG’s *when-clauses* to wait until message(s) arrive. Subsequently, structured entry methods translate to control-flow graphs whose execution is

¹Although it could be realized as a standalone DSL, it was also embedded within Charj and Charmxi.

message-driven. We will examine how we undertook this process in Ergoline, comparing it to Charm++ and Charj along the way.

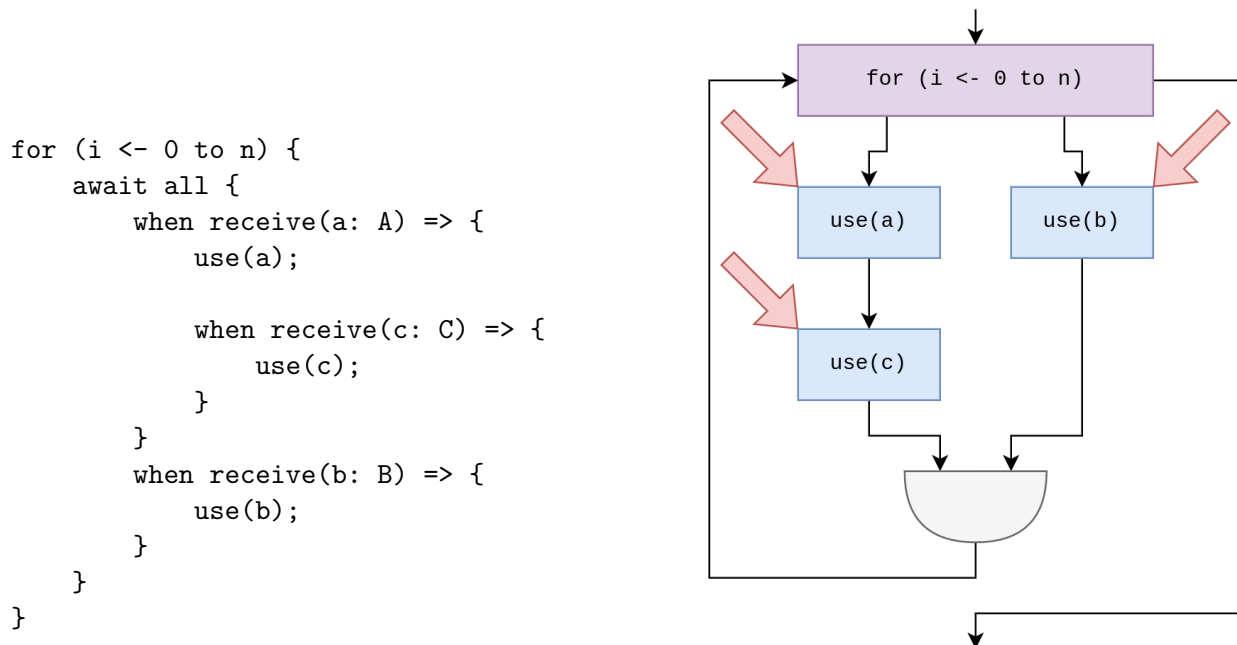


Figure 3.1: An example structured entry method and its corresponding DAG. Message arrival, shown as arrows, drives execution through the graph. We need to receive three messages in each iteration. Although they can arrive in any order, we must process ‘c’ after ‘a.’

3.1.1 Charm++ Implementation

The combination of Charmxi and a runtime library implements SDAG’s features in Charm++. The translation step consists of splitting a structured entry-method into numerous sub-methods, each corresponding to an SDAG construct (enumerated below). The translator statically encodes the sub-methods’ dependence relationships—each cannot execute until all their message(s) arrive and their predecessors finish (per program order). The library enforces these relationships at execution time, maintaining queues of messages and SDAG *continuations* (the runtime counterpart of sub-methods).

The SDAG library enqueues sub-methods whose predecessors have completed when their remote dependencies (messages) are unavailable (in a queue). Likewise, it enqueues messages when no dependent sub-method is available. Note that multiple sub-methods may depend on the same message arrival—most often when users enclose when-clauses within overlapped for loops. Since only one continuation may consume each message, in such cases, they execute

in a non-deterministic, first-come-first-served fashion. To counteract this, users can tag messages and when-clauses with reference numbers, shortened as *refnums*, that the library pairs up at execution time (e.g., when a particular iteration of a loop must process a particular message). We provided examples of such use-cases in Section 2.5.

Charmxi does not support local variables within structured entry methods; instead, users must store data within *chare* properties. This restriction is a side effect of the SDAG library, which does not interact with user data, effectively only tracking progress. Past works observed this limitation [6], along with Charmxi’s inability to emit meaningful SDAG-related errors and warnings (e.g., unable to type-check inline C++). Aiming to address these shortcomings, Charj implemented SDAG as an EDSL.

Ultimately, since Charj’s compiler emitted Charm++ and SDAG code, it could only generate constructs understood by the SDAG translator. This limited what its authors could achieve “without re-implementing the logic of the original SDAG translator within Charj” [6, pp. 63]. To overcome these limitations in Ergoline, we implemented a CFA-based SDAG segmentation pass and a code generation pass that uses its results. These passes effectively re-implement the logic of the SDAG translator within Ergoline, enabling it to support local variables and generate SDAG-related errors and warnings.

3.1.2 Syntax Constructs

The SDAG EDSL is a deeply-embedded DSL with bespoke syntax, interacting with the mailbox and overlap annotations introduced in 2.3.2.

When-clauses

When-clauses trigger computations upon receiving messages from one or more mailboxes. Users may attach predicates to screen messages, i.e., only accepting matching ones. From a user’s perspective, implicit barriers follow standalone when-clauses (i.e., those not contained within a bundling construct); everything after a clause does not execute until its ‘body’ (i.e., corresponding computation) has finished. We show various permutations of when-clauses in Figure 3.2. The matrix-matrix multiplication example from Chapter 2 (Figure 2.15) shows an example of a compound when-clause that requests values from multiple mailboxes.

Await All/Any

In Ergoline, users may bundle several when-clauses together with the await-all or await-any constructs. Their constituent when-clauses may execute in an overlapped fashion, with their implicit barrier being “pushed” outside of the bundling construct. An await-all construct unblocks once *all* its constituents are satisfied. Conversely, an await-any construct proceeds as soon as one of its components complete (i.e., *any* of them). Only one when-clause within an await-any construct may execute since they immediately invalidate all other paths. Figure 3.1 shows examples of an `await all` construct; changing it to an `await any` construct would convert the AND to OR semantics. These constructs are semantically equivalent to their counterparts in SDAG, `overlap` and `case`.

Related Work

In contemporary languages like Erlang and its sibling Elixir, similar mailbox-based syntax constructs that combine pattern-matching and message-processing are called *selective receives* [46]. That said, *Ergoline*’s incorporation of ideas from SDAG grant its selective receive constructs a greater degree of flexibility than those found in Erlang/Elixir. In such languages, only one branch/case of a selective receive may be taken—this means that they cannot be used as `await all`-like constructs, nor support the overlapped case execution that these constructs offer (when nested) in *Ergoline*. Additionally, these languages

```
@mailbox def receive_value<A>(tag: int, value: A);

@threaded def receive_values(n: int) {
  @overlap for (_ <- 0 to n) {
    await any {
      when receive_value<int>(tag, value) if tag % 2 == 0 =>
        println('received ${x} with the integer tag, ${tag}!');

      when receive_value<long>(_ % 2 != 0, value) => {
        println('received ${x} with an odd, long tag...');
      }
    }
  }
}
```

Figure 3.2: Ergoline introduces generic mailboxes, shown here along with different pattern matching styles.

do not have a direct counterpart to *Ergoline*’s compound **when**-clauses. Catering to latency-sensitive applications, Erlang and Elixir also include a timeout for their selective receives, triggering actions if messages do not arrive within a user-specified period. We consciously omitted this feature from *Ergoline*. Since we do not target real-time applications, we favor explicit cancellations. Lastly, note that *Ergoline*’s **await any** construct is also similar to Go’s **select** construct (although Go does not employ pattern matching).

Although Charm++ can match messages via refnums, they must fit within an RTS-reserved (message) header field, usually 16 bits long. While users can increase this length, it is generally discouraged due to its global performance impact (e.g., increasing latency in fine-grained applications). Additionally, if users want to match non-integer or multiple values, they must use a (potentially collision-prone) hashing scheme. In comparison, *Ergoline*’s pattern matching facilities do not suffer these limitations and can achieve results beyond hashing. For example, if one imagines that **A**, **B**, and **C** derive from a common class, **Letter**, and that **receive** was declared as `@mailbox def receive(_: Letter)`, then Figure 3.1 demonstrates selecting messages based on the type of object they contain.

3.1.3 Segmentation

Like the SDAG translator, *Ergoline*’s segmentation pass breaks a structured entry method into sub-methods. We arrange these sub-methods in a graph; each depends on their predecessors and, potentially, one or more message arrivals. Context switching overheads between sub-methods motivate their greed; they aim to maximize the work they accomplish with only their dependencies. This work corresponds to the statements in a structured entry method; only one sub-method may claim each piece of work.

Control-flow statements (e.g., loops) must be segmented when they include (when-)clauses within them. Otherwise, a sub-method can wholly encapsulate them. In the context of SDAG, we classify control-flow statements as either divergences or loops. Along with clauses, multi-clauses, and serial-blocks, these are node classifications in our SDAG implementation.

We implemented the segmentation pass as an EIR visitor, which visits each statement in a structured entry method and places them into a graph node. It creates new nodes when their predecessors are not reusable (e.g., when it encounters when-clauses for the first time). We present the pseudo-code for our visitor in Figure 3.1, along with an example program and graph in Figure 3.3.

We do not support **break**, **continue**, or **return** statements within SDAG constructs. Additionally, we do not support SDAG constructs within lambda expressions. These features do not have counterparts within Charm++, so we will need to define new semantics for them.

```

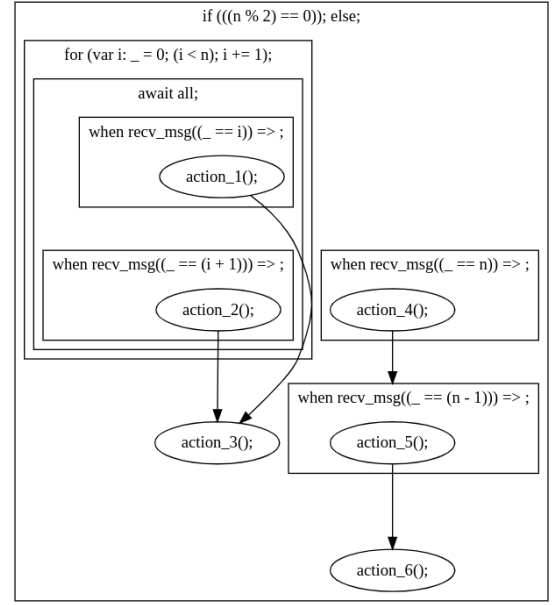
@threaded @entry def test() {
  if (n % 2 == 0) {
    for (var i = 0; i < n; i += 1) {
      await all {
        when recv_msg(_ == i) =>
          action_1();
        when recv_msg(_ == (i+1)) =>
          action_2();
      }
    }
    action_3();
  } else {
    when recv_msg(_ == n) =>
      action_4();

    when recv_msg(_ == (n - 1)) =>
      action_5();

    action_6();
  }
}

```

(a) Sample structured entry method.



(b) Graph corresponding to sample code.

Figure 3.3: Both paths are contained within the if-else node. Although `action_3()` only appears to depend on the clauses inside the await-all, it will not execute until the for-loop finishes all its iterations.

Until then, we generate compiler errors whenever we encounter these patterns. Note, `break` and `continue` are not supported by Ergoline at large, and entry methods typically do not have return values (so they do not use `return`).

3.1.4 Code Generation

Each segment produced by the previous pass corresponds to one or more generated methods. These methods implement their control flow behavior; for example, loops are composed of a preheader, header, and sentinel. They share local variables using ‘micro-stacks’ that form a linked list. Each method accepts the list’s last element as one of its parameters, looking up variables within it using offsets assigned by the compiler. When declarations exit lexical scope, we ‘unwind’ the stack by removing the last element of the list; all of its offsets become reusable. We will examine how we generate code for each kind of segment in this section.

Serial Blocks

When serial blocks contain local variable declarations, they add a new link to the micro-stack chain. They are simply methods containing the statements assigned to the node. The

Algorithm 3.1 Generate SDAG segments from an EIR AST node.

Precondition: *node* is an EIR AST node, *previous* is an optional segment.**Postcondition:** a segment that is generated based on *node*

```
1: function VISIT(node, previous)
2:   if x is a block then
3:     children  $\leftarrow$  partition the statements in x based on whether they contain when-
       clauses
4:     for child  $\leftarrow$  children do
5:       if child contains when-clauses then
6:         next  $\leftarrow$  VISIT(child, previous)
7:       else
8:         next  $\leftarrow$  segment with child that is either new or previous
9:       end if
10:      if next is not previous then
11:        make next the successor of previous
12:        previous  $\leftarrow$  next
13:      end if
14:    end for
15:  else if x is a loop or when-clause then
16:    previous  $\leftarrow$  segment with the result of calling VISIT on node's body
17:  else if x is an if-else, match, or await any/all then
18:    previous  $\leftarrow$  segment with the results of calling VISIT on all alternatives
19:  else
20:    previous  $\leftarrow$  segment with node that is either new or previous
21:  end if
22:  return previous
23: end function
```

last statement in the block is a call to the node's successor.

Divergences

Ergoline's branching constructs, e.g., if-else and match-case statements, correspond to SDAG divergences. We isolate each construct's branches within sub-nodes that share the successor of their parent divergence. The divergence node only contains the branching logic, calling the static methods corresponding to the sub-nodes.

Loops

We split loops into three parts, the pre-header, header, and sentinel. The pre-header allocates a micro-stack link and initializes any loop-specific variables. We omit the pre-header if there are no local variable declarations (e.g., while loops). The header contains a loop’s branching logic, executing its body based on the test expression’s result. Non-overlapped loops execute their successor when the test expression is false. For overlapped loops, we realize this branching as an ‘inline’ while loop instead of an if-then-else statement; this is a form of tail-call optimization that precludes stack overflow exceptions. Finally, sentinels are the successors of loop bodies and update any induction variables. Sentinels of non-overlapped loops return to the header, whereas those of overlapped loops move on to their loop’s successor once all iterations have finished.

Clauses

Clauses create components and generate mailbox requests for them. When these components receive values, they execute their corresponding clause’s body. We encapsulate these values within micro-stack links, passing them as arguments to the body’s method. As with divergences, clauses ‘push’ their successor to the last node of the body. Due to their implementation as recursive methods, the call-stacks for structured entry methods can get rather deep. The asynchronous execution of components can cull their depths, e.g., the call stack empties when remote dependencies are unavailable.

Multi-Clauses

Like overlapped for-loops, we split multi-clauses into headers and sentinels that track progress using stack links. Although, since multi-clauses cannot declare and initialize variables, they do not need pre-headers. Await-all multi-clauses wait for all their sub-clauses to complete, then move on to their successors. Await-any multi-clauses use *speculator* links which, before moving on, invalidate all pending sub-clauses once one completes. Recall that chares can only execute one action at a time, so there is no concern of invalidating active actions. Both variants create sub-clauses simultaneously without an iterative component (almost like fully-unrolled loops).

```

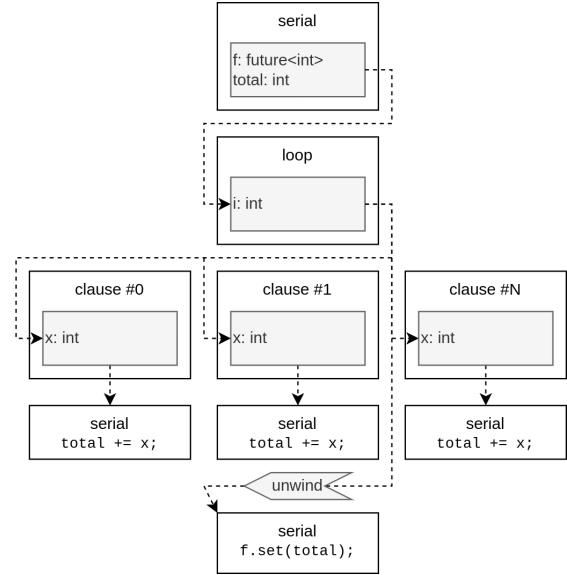
@entry
def add_values(f: future<int>) {
    var total = 0;

    @overlap
    for (var i = 0; i < n; i += 1) {
        when receive_value(x) => {
            total += x;
        }
    }

    f.set(total);
}

```

(a) Example SDAG code.



(b) Corresponding micro-stack links.

Figure 3.4: Segmentation demonstration with micro-stacks shown.

3.1.5 Example

Figure 3.4 loosely shows how an example loop translates to micro-stack links. SDAG nodes that declare variables create micro-stack links, while those that do not reuse their predecessor's. We insert a stack unwind operation after the loop since its declarations exit lexical scope. Subsequently, the last serial block connects to the chain's previous link, i.e., that of the first block. Note that we did not show all the clause nodes for brevity. Figure 3.5 demonstrates how to create linked lists using Hypercomm's micro-stack API.

```

auto hi = hypercomm::link(32);
auto mi = hypercomm::link(hi, 64);
auto lo = hypercomm::link(mi, 128);
// ancestors' values available in children
assert(lo->get<0>() == 32);
assert(lo->get<1>() == 64);
assert(lo->get<2>() == 128);
// offsets are consistent between links
assert(lo->get<1>() == mi->get<1>());
// unwinding returns each link's parent
assert(lo->unwind() == mi);
assert(mi->unwind() == hi);
assert(!hi->unwind());

```

Figure 3.5: Demonstration of the Hypercomm Micro-stack API. Note that offsets count up by the number of values in each link.

3.2 [ERGO]STENCIL

The Stencil EDSL looks beyond Charm-based languages, drawing inspiration from ‘external’ sources. Basilisk C is a set of C extensions based on discretization schemes on Cartesian grids [47]; it has a bespoke compiler (QCC). ParallelStencil.jl is a Julia EDSL capable of achieving near C/CUDA performance levels [48]. We aim to illustrate EIR’s benefits to the broader community by drawing from these sources. The Stencil EDSL reuses SDAG’s segmentation logic and does not require a custom parser. It leverages Hypercomm’s tasking infrastructure, which implements a worker pool that we launch tasks across. Here, tasks are Chant-like talking threads [49], which can suspend and resume based on communication from their peers. Although stencils are naturally balanced, they are a step on the pathway toward particle-in-cell (PIC) methods, which would benefit from Charm++’s automatic load-balancing.

3.2.1 Example

The Stencil EDSL achieves the same degree of scalability for Jacobi relaxation as Charj and Ergoline in roughly $4\times$ less code (40 vs. 170 LOC). Figure 3.6 shows its code. Since users have a global view of control and data, it resembles sequential code. The exception is the “boundary” function, which triggers halo exchanges, implying distributed execution. Both Basilisk C and ParallelStencil.jl have counterparts to this function; inferring when to exchange halo regions is beyond the scope of this work.

3.2.2 Analysis

For each annotated stencil function, the analysis pass visits all the statements, runs the SDAG segmentation pass, then generates a wrapper function. During the visitation phase, we seek information about the shape, halo region size, and boundary conditions of arrays. We seek out all local variable declarations since we promote them to task properties. Finally, we look for loops, identifying their array accesses, index spaces, and reductions. Along the way, whenever the program has to wait for remote data (i.e., from gather, halo exchange, or reduce operations), we insert a (dummy) when-clause for the segmentation pass. After these updates, we segment the function then generate its wrapper. The wrapper launches a task group and creates and returns futures. It passes all the user’s arguments, along with these futures, to the task group launch. Subsequently, if an array is among the stencil function’s arguments, it will be sent to all the tasks in its entirety. Such broadcasts are unnecessary when

each task only uses a subset of the array; in the future, we may detect these opportunities and generate scatter operations instead.

3.2.3 Code Generation

The code generation process of the Stencil EDSL is similar to that of SDAG—iterate through the segments, producing one or more methods that implement their control flow. Aside from its generation of domain-specific logic, e.g., for halo exchanges, a notable difference is that the Stencil EDSL targets Hypercomm’s tasking module instead of components. Subsequently, its methods exist inside tasks vs. chares, and it explicitly specifies continuations to post remote requests (see Figure 3.7 for an example). It also does not use microstacks, promoting local variables to task fields instead.

As mentioned previously, we generate when-clauses for three operations: gathers, halo exchanges, and reductions. During the code-generation phase, we implement their domain-specific logic. The Stencil DSL’s internal 1D decomposition simplifies the halo exchanges; we need only check whether we have north and south neighbors, sending the first and last rows of the grid(s), respectively. Tiles suspend when they expect halo updates, moving on to the next segment otherwise. When resumed by receiving an update, they continue when all their updates have arrived, suspending otherwise. Gathers and reductions follow typical MPI patterns, albeit complying with the suspend/resume pattern enforced by Hypercomm’s tasks.

The code generator partitions the arrays into tiles equal to how many workers are in the pool. Then, we generate headers for all the loops that iterate over each tile’s grid partition. We replace the accesses within these loops with specialized functions that consider the boundary conditions and halo regions. For example, out-of-bounds accesses return constant values when users have a padded grid. Lastly, we generate helper functions that de/serialize all the task’s fields, and return whether tiles have north and south neighbors.

3.2.4 Future Work

We plan to pursue higher-dimensional grids in the future. They are a stepping stone to particle-in-cell methods that benefit from the Charm++ RTS. Aside from that, porting more examples to the Stencil EDSL would help push its limits (e.g., acoustic waves, thermal-mechanical convection, etc.). Tapping into Charm++’s *LiveViz* infrastructure would allow users to visualize computations in real-time (as they progress). Finally, the Stencil EDSL would benefit from broader changes to Ergoline to make it more Scala-like. For example,

```

package examples;

import ck;
import ergoline::_;
import stencil::_;

// launches a distributed task, gathers tiles and returns a future
@stencil def relaxation(n: int, tol: double): array<double, 2> {
  var grid = array<double, 2>::fill((n, n), 0.0);
  var next = grid.clone();

  // set the grid's boundary conditions
  pad(grid, 1.0);

  var converged = false;
  while (!converged) {
    var max = 0.0;

    // update the grid's boundaries (halo exchange)
    boundary(grid);

    foreach({
      next[] = 0.2 * (grid[] +
        grid[-1, 0] + grid[1, 0] +
        grid[0, 1] + grid[0, -1]
      );

      val diff = math::abs(next[] - grid[]);
      max = math::max(max, diff);
    });

    (grid, next) = (next, grid);

    // automatically all-reduces 'max' from all participants
    converged = (max <= tol);
  }

  return grid;
}

@main class main {
  @entry def self(args: array<string>) {
    val n = (args.size() >= 2) ? args[1].toInt() : 16;
    val np = ck::numPes();
    // calls 'done' when the future's value is ready
    // ( round to nearest n divisible by np )
    relaxation(math::max(np, i + (i % np)), 0.004).then(self@done);
  }

  @entry def done(grid: array<double, 2>) {
    println(grid.toString());

    exit();
  }
}

```

Figure 3.6: Jacobi Relaxation in the Stencil EDSL.

dropping semicolons and parentheses when passing closures would strip `foreach({...})`; down to the more natural `foreach {...}`.

```

void block_7_begin(void) {
    this->all_reduce(&relaxation_task::block_7_continuation>(
        max, CkReduction::max_double);
    this->suspend(&relaxation_task::block_7_continuation>());
}
void block_7_continuation(hypercomm::tasking::task_payload &&__payload__) {
    auto __arguments__ = std::forward_as_tuple(max);
    hypercomm::flex::pup_unpack(__arguments__, __payload__.data,
                                std::move(__payload__.src));

    block_8_begin();
}
void block_8_begin(void) {
    std::forward_as_tuple(grid, next) = std::make_tuple(next, grid);
    converged = ((max - tol) <= 0);
    block_9_begin();
}
void block_9_begin(void) {
    if ((!converged)) {
        block_2_begin();
    } else {
        block_11_begin();
    }
}
}

```

Figure 3.7: An excerpt of the code generated for the example in Figure 3.6, corresponding the reduction and **while** loop’s header. Suspend and send-generating operations, like reductions, form pairs; the RTS calls the ‘successor’ of the last suspend operation upon receiving a corresponding message. This communication pattern is like that of conventional MPI, wherein receives are ‘expected.’

3.3 CHARISMA

Charisma is, broadly, a notation for expressing a subset of communication patterns found in HPC applications. More specifically, it captures data-independent data-flows, where senders determine message destinations without using data from previous messages. Subsequently, compilers can statically determine all possible communication patterns in Charisma programs.

Jetley et al. realized Charisma as a domain-specific language based on Charm++ [7]. Their compiler translated Charisma orchestration files, with global control views, into SDAG code (i.e., local control view). A combination of generated code with user-written Charm++ and C++ source files formed complete programs. Example applications for Charisma included dense LU decomposition and cutoff-based molecular dynamics, where performance matched hand-written Charm++ code while offering a 50-70% reduction in lines of code.

In this work, we reimaged Charisma as an Ergoline EDSL where users represent orchestrations as annotated entry methods within chares. This approach provides a single-file view of all class, method, and orchestration definitions where, previously, users needed to scan through multiple files to find them. EIR also brings strong type-checking to Charisma, something previous implementations lacked. Roughly twenty lines of Ergoline ‘skeleton’ code, along with an EIR analysis and transformation pass, implement our Charisma EDSL. Compared to a standalone DSL, with a custom parser, type-checker, analyzer, etc., this represents much less effort on the part of a language author.

3.3.1 Pairing Publishers and Consumers

At its heart, Charisma only needs a simple analysis pass to pair up consumers and producers, also called publishers. These pairs represent the dependence relationships of orchestrations. Consumers and producers correspond to read and write operations on placeholder arrays, respectively. We restrict the subscripts of consumers to be tuples of induction variables without offsets. This restriction ensures that senders know precisely where to send their values during the code generation step—receivers need only post requests (i.e., value expectations).

Figure 3.2 contains pseudocode for the pair pub-cons pass. Note that the map only considers the placeholder arrays accessed by publishers and consumers. For example, the expression `A[i]` matches `A[i+1]`.

3.3.2 Code Generation

Using ULTs simplified our Charisma implementation; we could reuse our sequential code generator with minor transformations. Placeholders become local variables. We reduce the orchestrated for-loop headers to induction variable declarations, setting their values to each element’s index in their respective collections.² We also replace all the publish-consume pairs and reductions with send/receive operations. Sends correspond to calling a proxy’s entry methods. Receives operate like SDAG when-clauses—they create components, post requests to them, then suspend the ULTs. Here, we use components as one-time tasks hosted within chares that, upon completion, fire a continuation. We ‘AND’ the completion events of an orchestration’s terminal nodes to track its progress (i.e., nodes without any dependencies). This tracking helps us enforce the implicit barrier at the end of a for-loop in an orchestration. For example, the chares of Figure 3.8 contribute to a reduction after completing their

²Orchestrated for-loops have the `@overlap` annotation or iterate through index spaces.

Algorithm 3.2 Pairs publishers and consumers within a Charisma orchestration.

Precondition: the list of all the *statements* in an orchestration**Postcondition:** the list of publisher-consumer pairs

```
1: function PAIRPUBCONS(statements)
2:   pairs  $\leftarrow$  empty list
3:   publishers  $\leftarrow$  empty map
4:   for statement  $\leftarrow$  statements do
5:     for publisher  $\leftarrow$  published values in statement do
6:       publishers(publisher)  $\leftarrow$  statement
7:     end for
8:     for consumer  $\leftarrow$  consumed values in statement do
9:       if consumer in publishers then
10:        pairs  $\leftarrow$  pairs :: (publishers(consumer), statement)
11:      end if
12:    end for
13:  end for
14:  return pairs
15: end function
```

portion of the orchestration; to enforce this behavior, we track the completion status of two components.

3.3.3 Example

We turn to Jacobi relaxation again for a Charisma example, shown in Figure 3.9. We use a 1D decomposition and omit the definitions of the boundary and compute methods for clarity. The boundary method returns the grid’s top and bottom rows; each element sends these rows to their north and south neighbors. The compute method uses the halo values from their neighbors to update the grid, returning the maximum change. This example demonstrates publishing tuples to multiple destinations and reductions over placeholder arrays.

3.4 MSA

The motivating observation of multiphase shared arrays is that programs typically access global address spaces in phases. For example, batched writes follow batched reads in a regular pattern, etc. The operations of histogramming and matrix-matrix multiplication exemplify this behavior.

MSA exploits this regularity to achieve determinism, albeit at the cost of flexibility. Un-

```

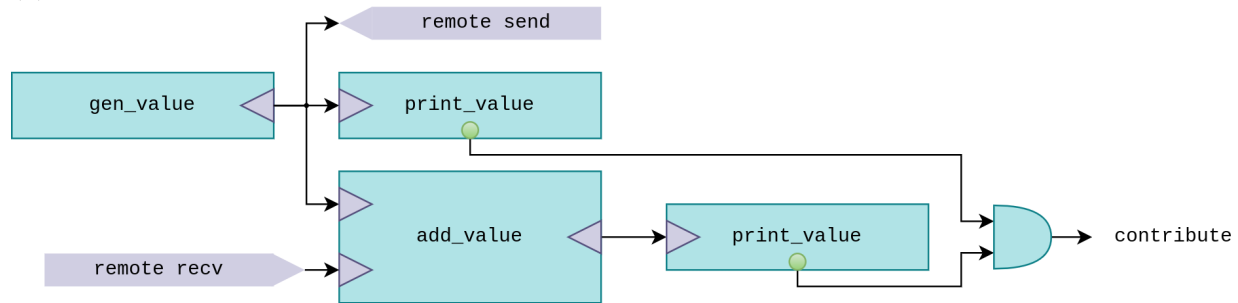
@charisma @entry def sample() {
  val values = or::placeholder<int, 1>(n);
  val sums = or::placeholder<int, 1>(n);

  for (i <- or::ispace(self@)) {
    val value = self@[i].gen_value();
    values[(i + 1) % n] = value;
    sums[i] = self@[i].add_values(value, values[i]);
    self@[i].print_value(sums[i]);
  }

  self[0].contribute(exit);
}

```

(a) Example Charisma orchestration.



(b) Component-based implementation.

Figure 3.8: In this example, each participant adds their generated values to their rightmost neighbor's.

der its model, users must explicitly synchronize between reading, writing, or accumulating phases. It disallows other operations during these phases and, during writing phases, multiple writes to the same element. MSAs employ caching and prefetching for efficiency, deferring change propagation until phase changes [50].

Past work has shown that MSAs benefit from compiler support. DeSouza et al. posited that compilers could optimize their remote accesses [18]. Becker demonstrated this by statically applying tiling and inserting prefetch operations within Charj [6]. Charj's compilers could also statically validate that operations only occur during their corresponding phase using control-flow analysis (CFA). Past works have not explored eliminating explicit, user-level synchronizations; doing so could lower the library's barrier to entry by closing the syntactic gap between it and sequential code.

We could trivially eliminate synchronizations by re-synchronizing whenever operations do not match their predecessors (at runtime); however, this may result in unnecessary synchronizations (e.g., during loops). Furthermore, if threads exhibit control-flow divergence,

```

@local @entry def boundaries(): (array<double>, array<double>) { ... }
@local @entry def compute(n: array<double>, s: array<double>): double { ... }

@charisma @entry def relaxation() {
    val north = or::placeholder<array<double>, 1>(n);
    val south = or::placeholder<array<double>, 1>(n);
    val maxima = or::placeholder<double, 1>(n);

    var converged = false;
    for (var it = 0; !converged && (it < maxIters); it += 1) {
        for (i <- or::ispace(self@)) {
            (north[(i + n - 1) % n], south[(i + 1) % n]) = self@[i].boundaries();
            maxima[i] = self@[i].compute(north[i], south[i]);
        }

        converged = (maxima.reduce(math::max) <= threshold);
    }

    self[@].contribute(exit);
}

```

Figure 3.9: Expressing Jacobi1D as a Charisma orchestration.

possessing control-flow paths with different operation counts, the behavior of this scheme would be undefined. With their insights into control flow and ability to reorder operations, compilers can statically generate correct and efficient synchronization schedules.

Consider the example program shown in Figure 3.10a. A naive schedule would have two synchronizations in each iteration, effectively $O(n)$ complexity. Here, compilers can split the loop, resulting in two synchronizations total, or $O(1)$ complexity (see Figure 3.10b). Since these synchronizations are expensive global barriers, the optimized version can be up to 1,000 $\times$ faster than its counterpart, shown in Figure 3.11.

3.4.1 Phase Analysis

Phase analysis has two stages: identification and synchronization. In the first stage, it labels the nodes of a control-flow graph with their MSA access kinds (i.e., read, write, or accumulate). It splits basic blocks into contiguous chunks of same-access operations. Then, during the second stage, it inserts psi-nodes—these correspond to phase change or synchronization events. The result is a naive synchronization schedule for the function under analysis, ensuring MSAs are in the correct phase for each access. Figure 3.3 shows the phase analysis algorithm, with Figure 3.12 showing the result of applying it to a CFG.

```

@async @threaded @entry
def foo(a: msa<int, 1>): int {
  val (lb, ub) = self.get_range();
  var sum = 0;

  // enroll this chare as one of n
  // MSA workers
  a.enroll(self.n);

  for ( i <- (lb to ub) ) {
    a[i] = f(i);
    sum += a[i];
  }

  return sum;
}

```

(a) Ergoline MSA eDSL example.

```

int foo(MSA<int> a) {
  auto [lb, ub] = this->getRange();
  auto sum = 0;

  a.enroll(this->n);

  auto w = a.getInitialWrite();

  for (auto i = lb; i < ub; i++) {
    w.set(i) = f(i);
  }

  auto r = w.syncToRead();

  for (auto i = lb; i < ub; i++) {
    sum += r.get(i);
  }

  return sum;
}

```

(b) C++ MSA example.

Figure 3.10: Comparison between Ergoline and C++ MSA programs. Phase-aware loop fission enables Ergoline’s program to be equivalent to its optimal C++ counterpart.

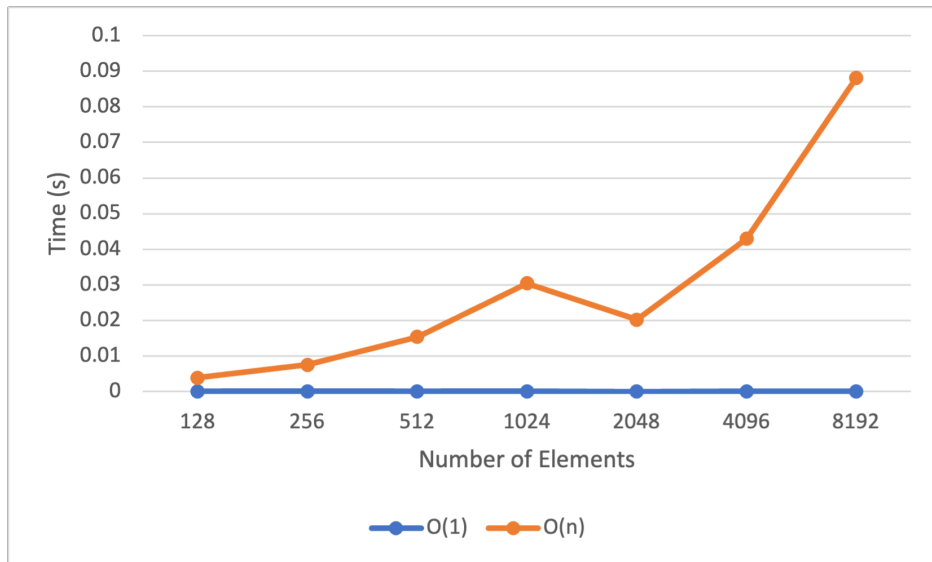


Figure 3.11: Comparison between $O(n)$ vs. $O(1)$ synchronizations, collected on a single node of Bridges2. The performance of the $O(1)$ version nearly remains constant, whereas the $O(n)$ version dramatically slows down.

Algorithm 3.3 Insert psi-nodes into a phase-aware CFG.

Precondition: S is the list of starting nodes, L is an optional list of natural loops.

```
1: function ANALYZE( $S, L$ )
2:    $queue \leftarrow$  a queue with the elements of  $S$ 
3:   while  $queue$  is non-empty do
4:      $node \leftarrow$  dequeue a node from  $queue$ 
5:     mark  $node$  as seen
6:     if  $node$  has a phase and is not a psi node then
7:       insert psi nodes on incoming edges or outgoing back-edges with different phases
8:     end if
9:     if  $node$  has back-edges then
10:       $loops \leftarrow$  the natural loop(s) they define
11:      ANALYZE(the  $loops$ ' bodies,  $loops$ )
12:    end if
13:    for  $out \leftarrow$  the unseen successors of  $node$  do
14:      unless an enqueued node reaches  $node$  without going thru  $L$ 's headers then
15:        enqueue  $out$  in  $queue$ 
16:      end unless
17:    end for
18:  end while
19: end function
```

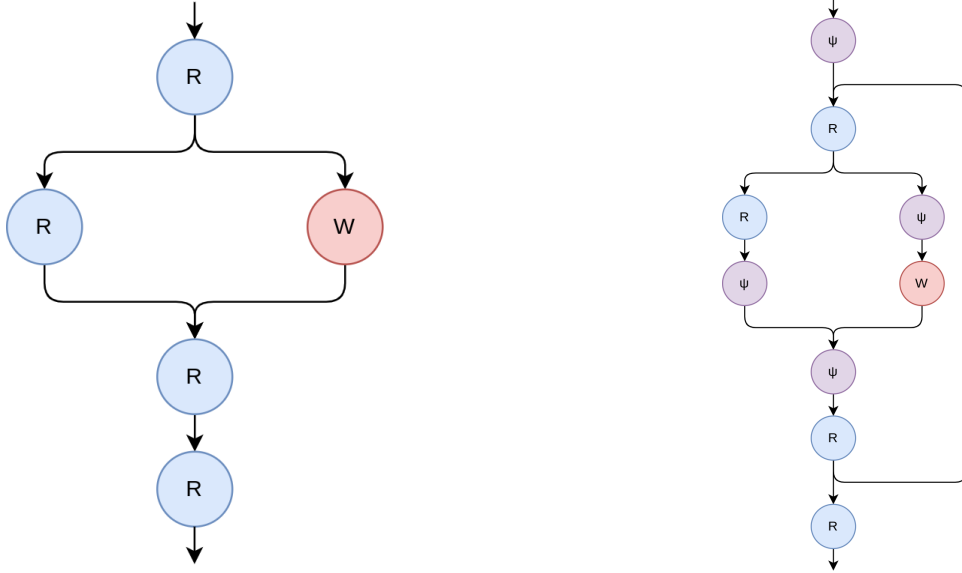
Phase analysis only applies to entry methods tagged with the `@msa` annotation. When users tag methods with `@msa`, they guarantee that:

- all participants will simultaneously execute the entry method
- no other entities will simultaneously access the entry method's MSAs

We aim to relax these constraints in future work since they limit when MSA synchronization can be automated. For example, only members of the same chore collection can communicate via an MSA under these constraints.

3.4.2 Optimizations

We use optimization passes to clean up the naive synchronization schedules generated by our phase analysis pass. When control-flow paths are isolated, i.e., followed by all participants, there may be opportunities to eliminate synchronizations. Generally, we also aim to move loop-carried phase changes due to their performance overheads.



(a) Abstract CFG of a program using MSAs. (b) Example CFG after inserting psi-nodes.

Figure 3.12: An example CFG and the result of applying phase analysis to it.

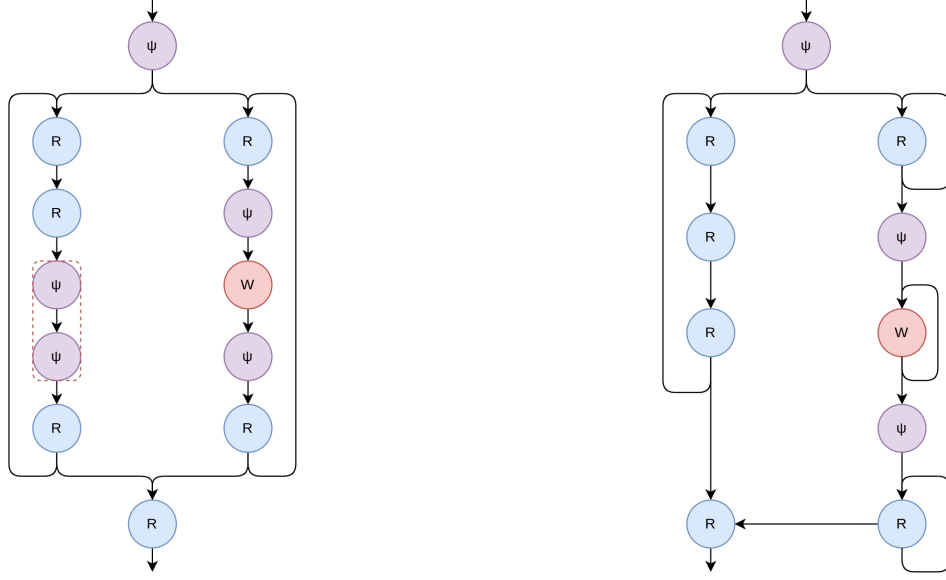
Unswitching

Fine-grained loops with internal branches may require multiple synchronizations per iteration, incurring performance overheads. Unswitching helps avoid these overheads when the control flow divergence is static, i.e., all participants follow the same path at runtime. This transformation applies to loops containing conditionals with loop-invariant test conditions [51]. It clones the loops inside each conditional branch, removing the original enclosing loops.

Figure 3.13a shows how this transformation applies to Figure 3.12b as a step toward the optimal number of phase changes. Afterward, redundant psi-node elimination can cleanup the phase changes on the isolated control-flow path (i.e., those in the red box). Additionally, we can apply loop fission to the other branch to eliminate all loop-carried phase changes.

Splitting

We apply fission to split loops along phase changes, enabling further optimizations to the isolated phase changes (e.g., code motion). Fission is legal when it does not break any cycles in the loop's dependence graph [51]. Figure 3.13b shows the result of applying fission to Figure 3.13a; note, the redundant psi-nodes were eliminated.



(a) CFG after applying unswitching to Figure 3.12b. Redundant phase changes are highlighted in red.

(b) CFG after eliminating the redundant phase changes from Figure 3.13a then applying fission.

Figure 3.13: Incremental optimizations of the CFG shown in Figure 3.12.

Redundant Psi-node Elimination

This pass cleans up semantically unnecessary phase changes along isolated control-flow paths. When multiple phase changes do not have MSA accesses interspersed, the redundant psi-node elimination pass can collapse them down to a single phase change (e.g., $R \rightarrow \psi \rightarrow \psi \rightarrow W$ to $R \rightarrow \psi \rightarrow W$). Phase changes between two same-phase accesses (e.g., $R \rightarrow \psi \rightarrow R$) are also redundant. Generally, we repeatedly run this pass after other transformations until the graph is unchanged.

3.4.3 Future Work

There are several directions for future work related to our MSA EDSL, with some requiring extensions to the underlying model. For instance, adding more transformations and support for accessing multiple MSAs are solely changes to the compiler. Our other proposals, like adding support for ‘iota’ nodes and switching from threading to callback semantics, require changes to the MSA library. Interest in MSA from external sources may drive such developments.

Transformations

We believe that we could adapt many loop transformations to have phase-aware variations. For example, we could use an adaptation of loop invariant code motion to move phase changes outside of loop nests. We could use fusion to merge loops and simplify phase changes. Beyond these, even straight-line code transformations may be applicable. Statement reordering could ‘bundle’ same-phase accesses, eliminating transitions (e.g., changing $R \rightarrow W \rightarrow R$ to $R \rightarrow R \rightarrow W$).

One potential downside of expanding the set of transformations is complicating the compiler’s search process. We currently iterate through all the legal optimizations until reaching a minimal set of phase changes; this naive scheme is only performant because our search space is small. In the future, we may need to use heuristics and stratified optimization levels to decide which transformation to choose next and when to stop the search. Newer transformations may require more sophisticated dependence analysis implementations as well.

Optimizing Multiple MSAs

Our current approach builds a phase-aware CFG for each MSA used within a function. Our nodes carry metadata for all the MSAs, such that optimizations do not violate correctness by rearranging nodes or restructuring loops. This solution may artificially constrain our optimizations since they cannot find compromises that satisfy all the MSAs. In the future, it would be better to build a more holistic graph that contains all the MSAs’ phase-changes.

Idle Barriers

Thus far, we have only considered loops where all participants execute the same number of iterations. However, there are many areas where this assumption may not hold. For example, if chares are iterating through non-uniform tiles of an MSA. In these cases, loop-carried phase changes will be problematic; chares that completed the loop will not synchronize with their still-iterating peers. We suggest adding a new function to MSA, the idle-barrier, represented in analyses as an iota node. This function marks a participant as idle, suspending the chare until all participants reach the barrier. Synchronization events proceed normally across the subset of active chares, enabling forward progress.

It would be straightforward to add support for idle barriers to the compiler—insert an iota node after all loops that we cannot prove to be uniform across participants. However, the MSA library does not account for workers de/activating throughout a computation. It is

unclear what has to change to facilitate the pseudocode implementation shown in Figure 3.14.

```
void idle_barrier(void) {  
    nActiveWorkers -= 1;  
    barrier(nWorkers);  
    nActiveWorkers += 1;  
}
```

Figure 3.14: Example implementation of the iota node.

Callbacks vs. Threads

MSA necessitates using `@threaded` entry methods since it can only resume threads when synchronization operations finish. If it could fire a callback instead, we could generate segmented entry methods for MSA a la SDAG. In doing so, we could improve the migratability of programs using MSAs. Our existing segmentation logic would not require significant changes to facilitate these benefits; however, modifying the MSA library could be non-trivial since it assumes a threaded model in several areas.

3.5 DIVCON

DivCon was a standalone domain-specific language designed for scaling recursive and divide-and-conquer computations using the Charm++ RTS. From an operational perspective, users wrote the parallel parts of their computation in *DivCon* and its compiler generated a “skeleton” for users to “fill in” with the sequential host language (i.e., C++). Individual tasks were created using a Cilk-like `spawn` command, which could also operate on collectives; this allowed mapping data-parallel operations over distributed arrays called *DivConArrays*. Tasks were *agglomerated*, grouped within chares, using a data dependency-aware scheme, minimizing off-node communication, fostering communication-computation overlap, and enabling migrations (i.e., tasks migrate with their host chare). Task agglomeration is a form of adaptive grain size control, grouping fine-grained entities into coarse-grained “chunks” at execution time.

We fold DivCon into Ergoline’s suite of languages via embedding. DivCon has similar embedding benefits to Charisma—users write code in a single language (Ergoline vs. a mix of C++ and DivCon) and benefit from Ergoline’s rich type system (e.g., support for generic functions). DivCon tasks also had limited syntactic range, e.g., no support for arbitrary expressions or control flow, like while loops. Lastly, Ergoline brings modernization benefits—

DivCon did not have automatic memory management or type inference. Generally, DivCon is a great embedding candidate since its syntax closely resembles Ergoline’s.

Figure 3.15 compares DivCon and Ergoline implementations of the quick sort algorithm. The primary difference is Ergoline’s use of ‘divcon::spawn’ instead of a bespoke ‘spawn’ keyword. We could resolve this discrepancy by enhancing Ergoline’s support for custom keywords—a goal for EIRv2 since true extensibility would benefit many other DSLs. In the meantime, Ergoline’s compromise is workable and resembles contemporary facilities, like ‘std::invoke’ in C++. Mechanically, spawns could leverage Hypercomm’s infrastructure.

We can easily map DivCon’s tasks to Hypercomm’s components. These components would be launched within a worker pool, spawning their children within their peers. We could implement a component-stealing mechanism, such that idle workers can steal work from their peers. Replicating DivCon’s more advanced features, like adaptive grain size control and task agglomeration, would require additional effort. However, establishing infrastructure within Hypercomm for these features would be worthwhile since it could benefit other DSLs targeting EIR.

3.6 SUMMARY

In this chapter, we explored embedding diverse programming models in Ergoline, highlighting how they benefited from EIR’s flexibility and common passes (type-checking, etc.). We demonstrated how Hypercomm’s abstractions can support these models, further reducing the burdens of language authors. Due to Ergoline’s immaturity, we had to implement the DSL analysis and transformation passes within EIR/Scala. In the future, we would like to implement these passes in Ergoline itself, potentially by incorporating macros like Rust [52, sec. 19.5] and Scala.³ Macros would lower the barrier of entry to EIR, as language authors would only need to learn Ergoline to gain access to its extensibility.

³Macros are emergent features in both languages that are starting to gain traction.

```

type Record;
extern int Tau;

serial seqSort(Array<Record>): Array<Record>;

qsort(Array<Record> A): Array<Record> {
  int len = A.length();
  if(len <= Tau) {
    return seqSort(A);
  } else {
    Array<Record> LT, EQ, GT;
    Record pivot;

    pivot = A.read(len/2);

    foreach(Record a in A) {
      if (a < pivot) {
        LT += a;
      } else if (pivot < a) {
        GT += a;
      } else {
        EQ += a;
      }
    }

    A.free();

    return (spawn qsort(LT))
      .concat(EQ)
      .concat(spawn qsort(GT));
  }
}

```

(a) DivCon version

```

object globals { val tau: int; }
using decay<A> = divcon::array<A>;

def seqSort<A>(arr: decay<A>): decay<A>;

def qsort<A>(arr: decay<A>): decay<A> {
  val len = arr.size();
  if (len <= globals.tau) {
    return seqSort(arr);
  } else {
    var lt: divcon::array<A>;
    var eq: divcon::array<A>;
    var gt: divcon::array<A>;

    val pivot = await arr[len / 2];

    for (elt <- arr) {
      if (elt < pivot) {
        lt += elt;
      } else if (pivot < elt) {
        gt += elt;
      } else {
        eq += elt;
      }
    }

    return divcon::spawn(qsort<A>, lt)
      ++ eq ++
      divcon::spawn(qsort<A>, gt);
  }
}

```

(b) Ergoline version

Figure 3.15: Comparison of quick sort written in DivCon and its EDSL counterpart. When users call ‘spawn’ on arrays, the RTS can launch multiple tasks and partition the elements among them.

CHAPTER 4: EIR

The Erg Intermediate Representation (EIR) provides a rich substrate for building high-level languages for high-performance computing. EIR includes support for high-level language concepts like generic types and class systems. Compared to conventional HPC-centric IRs, which omit support for these features, EIR’s higher abstraction level minimizes semantic gaps between it and languages built with it. Subsequently, EIR can offer common optimizations and reduce the burden of language authors.

4.1 COMPILER ARCHITECTURE

EIR operates on a mutable AST, represented as doubly-linked nodes that reference their parent and children. One can select all or class-specific children of a node (e.g., only the parameters of a function call). Node classes fit into a rich hierarchy, including types, scopes, expressions, etc. Please see Appendix A for a list of all EIR’s nodes and their classifications. We imagine this set of nodes as EIRv1, with future versions emphasizing (compiler) performance and canonicalization (i.e., only one way to represent operations).

From a list of user-specified files, EIR generates a combination of C++ and Charm interface files then compiles them with the host compiler (i.e., `charmcc`). Figure 4.1 gives an overview of the compiler’s work-flow, which can be summarized as:

1. Run the appropriate front-end on all user-specified files, generating AST nodes.
2. Run the “on-load” then “on-optimize” passes whenever an AST node is loaded.
3. After loading all files, run the code generation passes.
4. Run the back-end compiler(s) on the generated code:
 - (a) Compile the Charm interface files (`*.ci`) with `charmcc` to generate C++ headers (`*.de*h`).
 - (b) Then, compile all C++ source files with `charmcc` to produce a(n) executable or library.

4.1.1 Software Ecosystem

We wrote EIR in Scala to streamline its development process. JVM-based languages, like Scala, inherently support dynamically loading user-defined compiler passes and AST

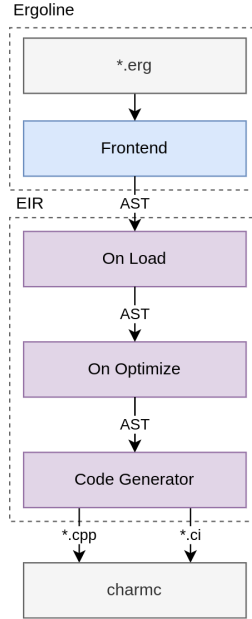


Figure 4.1: EIR compiler workflow.

nodes (i.e., via class-loading mechanisms). Furthermore, there is a rich ecosystem of JVM-based packages for compiler-related tasks. We utilized the ANTLR4 and FastParse2 parsing packages in this project. We plan to use the JVM’s built-in XML library for EIR’s textual representation.

4.1.2 Front-ends

EIR exposes a Scala API for building and manipulating ASTs. Multiple languages can utilize this API; for example, Java can directly target it, and C/C++ can target it through the Java Native Interface (JNI). We originally built an ANTLR4-based front-end for Ergoline written in Scala. Eventually, we switched to FastParse2 since ANTLR4 only supports context-free grammar (CFG). Regardless, both parsers construct an EIR AST that we hand off to the compiler infrastructure.¹ Ergoline does not require preprocessing since it is effectively syntactic sugar for EIR—other languages might. Looking toward EIRv2, we also plan to expose a textual interface for EIR. We imagine it being an XML-based representation to facilitate easy interchange between languages, like Omni’s XcodeML. Figure 4.2 shows what a “Hello, world!” example might look like in this representation.

¹Ergoline cannot be expressed as a CFG due to its use of the `<>` operators for templating, like C++.

```

<EirProgram>
  <!--
    import ergoline::_;

    def main() {
      println("Hello, world!");
    }
  -->
  <ImportStatement>
    <Symbol>ergoline::_</Symbol>
  </ImportStatement>
  <FunctionDefinition name="main">
    <Arguments />
    <Body>
      <Block>
        <FunctionCall>
          <Function>
            <Symbol>println</Symbol>
          </Function>
          <Arguments>
            <StringLiteral>Hello, world!</StringLiteral>
          </Arguments>
        </FunctionCall>
      </Block>
    </Body>
  </FunctionDefinition>
</EirProgram>

```

Figure 4.2: Hypothetical “Hello, world!” program in EIR’s XML representation.

4.1.3 Pass Infrastructure

EIR has on-load, on-optimize, and on-generate pass phases, which always execute in this order. EIR uses a topological sort to order passes according to their intra-phase dependencies, i.e., those that must run before them. We forbid cyclic relationships, generating errors when they occur.

EIR instantiates all the passes in its registry, which must derive from its `Pass` class (see Figure 4.3). It provides an interface for custom pass registration. We will give an overview of EIR’s built-in passes and code-generation infrastructure in this section. Note that we are still in the process of transitioning its on-generate phase to comply with this protocol. Before settling on its design, all EIR’s phases were ‘hard-coded,’ e.g., do *X*, then *Y*, then *Z*.

```

abstract class Pass {
  // which phase this pass executes within
  def phase: Phase
  // called to apply this pass to an EIR AST node
  def visit(n: EirNode): Unit
  // filters which nodes this pass applies to
  // (certain annotated nodes or, if empty, all nodes)
  def annotations: List[String]
  // the passes that must run before this pass
  def dependencies: List[Pass]
}

```

Figure 4.3: Structure of EIR’s `Pass` class (in Scala).

Type Checker

EIR runs a preliminary name-resolution sweep before type checking. Its primary objective is to load imported modules. While it can often generate candidate lists for symbols, it rarely selects a single resolution. There are numerous reasons why resolution can fail at this preliminary stage. For example, symbols within generic scopes are unresolvable until specializations are known. Additionally, resolving function calls depends on the types of their arguments since they inform overload selection. Subsequently, the name resolver defers many decisions to the type checker, which can handle these intricacies.

Along with selecting function overloads and specializing generic scopes, the type checker has numerous responsibilities. Aside from type validation (e.g., calling a function with correct arguments), it is responsible for validating accessibility (e.g., accessing a ‘private’ member), assignability (i.e., prohibiting re-assignment to value declarations), etc. EIR’s handling of this complexity is one of its key strengths since language authors need not worry about writing custom type checkers. For context—EIR’s type checker is over 2,000 LOC by itself, and it is only a prototype! Aside from this body of code, it also has several ‘helper’ passes, discussed below.

Class Checker The class checker’s responsibilities include validating use of the `override` keyword and enforcing properties like:

- Classes cannot have circular parent-child relationships.²
- Classes cannot implement base classes multiple times.

²For example, *Derived* cannot be *Base*’s parent if *Derived* is *Base*’s child.

- Classes can only extend classes and implement traits.
- Traits must extend or implement other traits.
- Traits cannot contain variable declarations.

Constructor Checker The constructor checker’s primary function is to validate that constructors assign all ‘mandatory’ class fields. These are fields that are not re-assignable and not default-initializable, mandatory since they would be permanently undefined if not set within a constructor. Otherwise, we check that constructors are only within classes (and not traits). Lastly, if a concrete class does not have any constructors, this pass generates a ‘dummy’ default constructor.

Function Checker The function checker validates that functions are correctly defined. For example, `@system` and trait methods cannot have bodies. Additionally, it checks that only constructors have field-assigning arguments (i.e., those prefixed with ‘=’). Finally, it ensures all overloads in the same scope are unambiguous. In other words, multiple functions cannot have identical names and arguments in a scoping construct (classes, namespaces, etc.).

DSL Passes

Our DSL passes are a mix of on-load and on-optimize passes. For example, the segmentation and control-flow analysis passes are on-load passes, while orchestration occurs on optimization. In practice, this distinction reflects whether a pass runs before or after type-checking. For example, Stencil EDSL programs would not type-check in EIR by default, so we must transform them first.

4.1.4 Code Generation

EIR has five stages of code generation: declarations, implementations, proxies, domain-specific logic, and Charm-interface files. This ordering is not absolute, e.g., generic types and EDSLs can cause variations. The declaration and implementation stages correspond to their C++ counterparts, outlining a class’s structure then defining its methods. The proxy stage wraps ‘proxified’ objects with code that un/marshals parameters and launches user-level threads. Lastly, EIR generates a ‘.ci’ file that registers an application’s chare-types and

entry methods with the underlying RTS (and outlines Hypercomm’s startup logic). Note that DSLs can customize code-generation, scheduling passes between their built-in peers.

Registration

EIR’s handling of generic and abstract proxy types during code generation is particularly relevant in this context.

Generic Types Charmxi compiles the parts of a user’s program that describe chares and their entry methods. It cannot determine how other parts use these constructs. Occasionally, users must inform Charmxi how they plan to use its constructs to account for its limitations. For example, users must register the specializations of generic methods and types through Charmxi. This requirement raises the barrier of entry to library usage; like, TRAM has multiple constructs that users must register for all data types they wish to aggregate. Figure 4.4 shows an example of user specialization registration.

Ergoline streamlines generic types and entry methods through its compiler. EIR maintains specialization lists then generates the corresponding registration code. Subsequently, users do not need to perform explicit registration. Figure 4.5 shows the Ergoline version of the program shown in Figure 4.4; it represents a roughly 60% reduction in LOC. In summary, this feature streamlines library development and usage—enabling the rapid prototyping of various parallel abstractions discussed in Section 2.4.

Abstract Proxy Types EIR iterates through all abstract classes and traits with entry methods during the declaration generation stage. For each one, it generates an abstract proxy with constructors and mock entry methods. These constructors accept proxies of derived types, recording the indices of their entry methods. These indices are analogous to virtual-method tables in object-oriented contexts, as the RTS uses them to select message handlers on the receiver side. The mock entry methods encode these indices into messages then route them accordingly (using the underlying chare information (collection identifier, index, etc.)). For example, if “foo” is the base of “bar” and “baz” and both override “qux,” the entry methods `bar::qux` and `baz::qux` will have different identifiers. The constructors from “bar” and “baz” to “foo” will set the `foo_proxy::qux_identifier` field to their corresponding value.

```

mainmodule example {
    mainchare printer {
        entry [nokeep] printer(CkArgMsg*);

        template <typename T>
        entry void print(T&&);
    };

    extern entry void printer print(double&&);
    extern entry void printer print(std::string&&);
}

```

(a) Charmxi source file.

```

#include "example.decl.h"

class printer : public CBase_printer {
public:
    printer(CkArgMsg*) {
        thisProxy.print(42.0);
        thisProxy.print<std::string>("hi");
        CkExitAfterQuiescence();
    }

    template <typename T> void print(T&&);
};

template <>
void printer::print<double>(double&& x) {
    CkPrintf("%lg!\n", x);
}

template <>
void printer::print<std::string>(std::string&& x) {
    CkPrintf("%s!\n", x.c_str());
}

#define CK_TEMPLATES_ONLY
#include "example.def.h"
#undef CK_TEMPLATES_ONLY

#include "example.def.h"

```

(b) C++ source file.

Figure 4.4: An example Charm++ program that demonstrates explicit specialization registration.

```

package examples;

import ergoline::_;
import ck;

@main class printer {
    @entry def self() {
        self@print(42);
        self@print("hi");
        ck::exitAfterQuiescence();
    }

    @entry def print<A>(a: A) {
        println('${a}!');
    }
}

```

Figure 4.5: The Ergoline version of the program shown in Figure 4.4.

Limitations of Generics

EIR’s code generator has limitations regarding multiple specializations of generic types and functions. Although it occasionally considers these factors, it generally takes one specialization and “runs with it.” This selectiveness can pose problems if, for example, a generic is specialized with both value and reference types. Figure 4.6 shows a program whose generated code would fail to compile under these circumstances. We plan to resolve this problem by generating unique code for all the specializations of generics. We need to address it before releasing EIR in a “production” capacity; however, it involves a healthy amount of refactoring.

4.2 COMPILER OPTIMIZATIONS

While raising the abstraction level, parallel programming systems may hide ‘unavoidable’ copies from users. In particular, we encountered unexpected copying during message un/packing—often from buffers encapsulated within classes or C++’s STL containers. We circumvented these copies through additional compiler and runtime support, delivering performance improvements. Specifically, co-designing our compiler and runtime facilitated the novel message un/packing optimizations discussed in this section.

```

class foo {
  ...

  def +(other: foo): foo {
    ...
  }
}

def bar<A>(lhs: A, rhs: A): A {
  return lhs + rhs;
}

def baz() {
  val f: foo = ...;
  // correctly generates 21+21
  ... bar(21, 21) ...;
  // should generate f->plus(f), but generates f+f... oops!
  ... bar(f, f) ...;
  ...
}

```

Figure 4.6: An example program that would fail to compile to due EIR's (present) limitations.

4.2.1 Pointer-to-offset

Data structures of plain ol’ data (POD) types are contiguously allocated in memory since they do not contain pointers or virtual method tables. Thanks to this contiguity, receivers can use pointers-to-offsets within message buffers to access the data. Avoiding a receiver-side copy like this is called a pointer-to-offset optimization. For example, type-casting an offset within a message to an integer array to avoid copying the data into a newly-allocated memory buffer. Note, this scheme is unlikely to provide benefits for small amounts of data (e.g., a single integer). Additionally, data is only accessible while users (or the RTS) retain messages. Subsequently, we recommend employing a size threshold for pointer-to-offset optimizations on machines with ‘pinned’ memory restrictions (i.e., regions directly accessible to the NIC) to avoid starving the RTS of messages.

Charmxi automatically applies pointer-to-offset optimizations to array arguments of entry methods. It retains an underlying message for the entry method’s duration, never exposing it to the user. Subsequently, users must copy an array’s data if they wish to retain access beyond the receiving entry method. Although switching to custom message types grants users message ownership, doing so incurs additional complexity (lifecycle management, un/packing, etc.). Lastly, note that anything outside Charmxi’s purview is not eligible for these optimizations, e.g., fields within classes.

Through full compilation and customized code generation, *Ergoline* can apply pointer-to-offset optimizations virtually system-wide. For example, all fields of classes are extended these optimizations during deserialization, eliminating the encapsulation overheads shown in Figure 4.7. Likewise, *Ergoline*’s standard containers have been designed in a pointer-to-offset compatible manner, reducing the impact of their use. Conversely, parallel programming systems based on C++ suffer “unavoidable” copies when using its STL containers, like `std::vector`; many of these data structures wholly encapsulate buffers that cannot be exchanged with pointers-to-offsets. Finally, through widespread use of ARC, *Ergoline* automatically retains ownership of message buffers until their last pointers-to-offsets exit scope.

4.2.2 Buffer Reuse

Re-sending values received via pointer-to-offset may present opportunities for buffer reuse, i.e., avoiding allocating and packing additional message buffers by reusing retained messages. *Ergoline* uses a control flow pass to identify buffer reuse opportunities, seeking areas where pointer-to-offset values are copied into message buffers and are unused afterward. EIR gen-

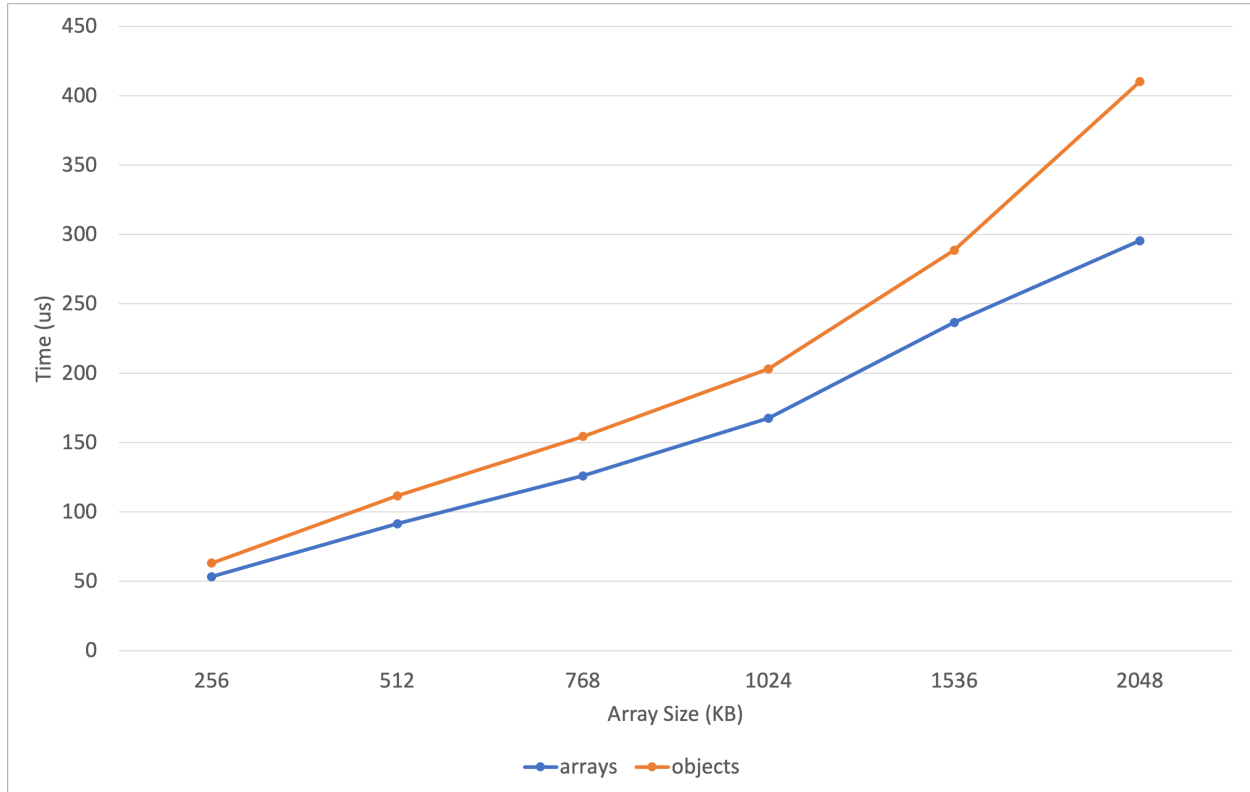


Figure 4.7: Here, we compare messaging arrays by themselves vs. within objects in Charm++. We collected this data within a node of OLCF Andes. The encapsulation overheads range from 20-40%, increasing with message size. Ergoline eliminates these overheads and makes both messaging schemes eligible for zero-copying through the combination of EIR and Hypercomm.

erates code to uphold copy semantics when it identifies these opportunities, i.e., the message is reusable when it is not shared. We verify unique ownership using the retained message’s runtime reference count. When its reference count is one, we can update it ‘in-place’ and resend it. Otherwise, since other parties could observe the updates, we must allocate a new message and copy the data.

An example algorithm eligible for buffer reuse is matrix-matrix multiplication, shown in Figure 2.15. In each iteration, chares forward the blocks they received to their peers. This algorithm has two opportunities for buffer reuse since both blocks arrive via pointer-to-offset, and it does not use them after the send operations. Subsequently, we only need to update the iteration counts within the messages to reuse them. The compiler recognizes this opportunity and, with this optimization, the code is $2.24\times$ faster at scale, shown in Figure 4.8. Note, since Charm++ is not eligible for this optimization, its performance is comparable to the baseline.

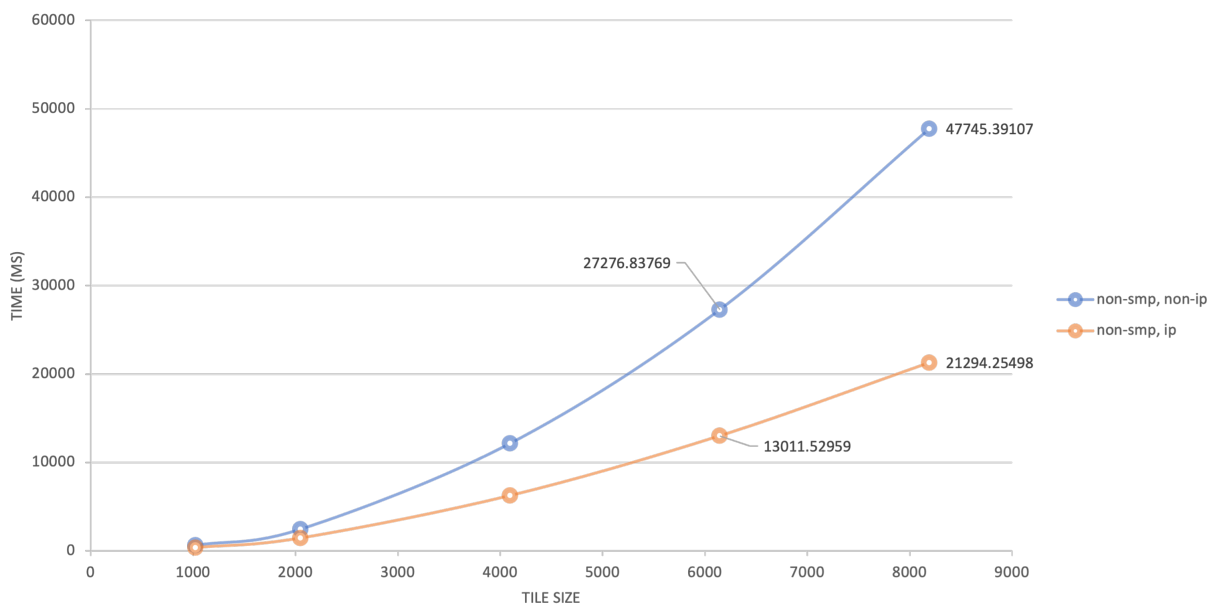


Figure 4.8: Effect of in-place optimizations on matrix-matrix multiplication on four nodes of Bridges2 (169 PEs). Note, this graph does not include computation time, only communication time.

4.3 COMPILER PERFORMANCE

The total compilation time for Charm++ programs is much less than that of EIR programs, shown in Figure 4.9. The performance difference is primarily due to the heavy use of

template meta-programming in Hypercomm, which drastically increases the C++ compilation time. If we update Hypercomm to C++17 or newer (from C++11), we could use the “if constexpr” pattern instead of SFINAE to mitigate this problem. Charmxi accounts for marginal amounts of time for EIR and Charm++ programs. Curiously, EIR spends $3.26\times$ less time in Charmxi than Charm++ (155 vs. 47 ms, on average), presumably because it does not require parameter marshaling. EIR accounts for less than 10% of compilation time on average; however, in the case of Jacobi2D, it requires 68% of Charm++’s overall time. Subsequently, if we refactor Hypercomm, we can reasonably expect EIR to account for a sizable portion of compilation time.

Future Work While the performance of our name resolver and type checker is adequate for our test programs, we do not expect it to scale. We frequently revisit symbols in both cases, leading to repeated work. We plan to employ caching to address this performance issue, mapping (scope, resolvable) pairs to (resolution, type) pairs. This scope captures specializations since, for example, the members of `array<int, 1>` differ from those of `array<int, 2>`. We expect that this change would dramatically reduce compilation times.

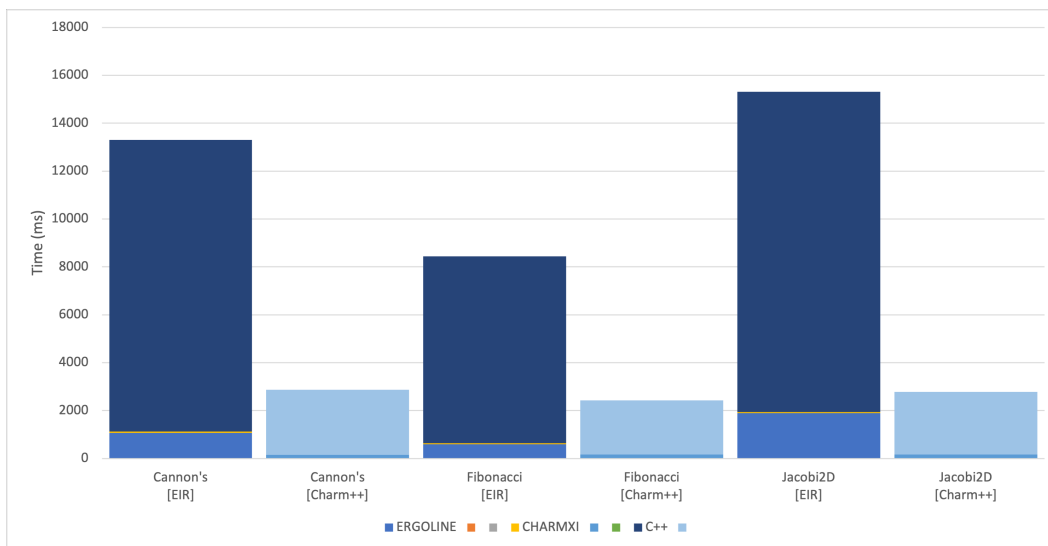


Figure 4.9: Compilation time breakdown, EIR vs. Charm++.

4.4 SUMMARY

We presented the features and optimizations of EIR in this chapter. These apply to all languages targetting it, relieving the burden of language authors and demonstrably improving performance. We discussed how EIR’s architecture facilitates Ergoline’s extensibility,

particularly its pass infrastructure. Although its structure is not novel to the compilers community, EIR’s position as a high-level language framework for distributed, high-performance computing is unique. Moreover, EIR’s focus on the migratable objects model gives it distinct runtime adaptivity and scalability.

CHAPTER 5: HYPERCOMM

In this chapter, we will present Hypercomm’s modules, optimizations, and programming model. Hypercomm provides EIR’s collective, de/serialization, section, and tasking infrastructure. Thus, where Converse is the underlying RTS of Charm++, Hypercomm underlies EIR. Generally, we built Hypercomm on top of Charm++, although it ‘drops down’ to Converse in some areas. Converse provides robust implementations of active messaging and scheduling, and Charm++ lends its load balancing and location management facilities. The resource-sharing relationship between Charm++ and Hypercomm demonstrates the composability of asynchronous many-task (AMT) models; they seamlessly share processor time since Converse schedules both entities. Thus, disabling Charm++ modules unused by Hypercomm is unlikely to affect performance. Since they will remain idle, i.e., unscheduled, the most likely effect of disabling them is a slight decrease in RTS startup time.

5.1 PROGRAMMING MODEL

Hypercomm’s programming model is centered around components, encapsulated computation fragments. Hypercomm established its “value-passing” model to ensure that data moves efficiently between components. Its spanning trees are built in service of component-based broadcasts and reductions. The entry ports of components form the basis for Hypercomm’s futures. This section focuses on these aspects of Hypercomm, providing a detailed overview of components and their use.

5.1.1 Components

In the early stages of this work, we needed a way to integrate its seemingly disjoint aspects. For example, we had to implement infrastructure for futures and mailboxes and found that chares were too coarse-grained to represent one-time tasks. These abstractions operate at ‘sub-charge’ granularities but may receive ‘external’ values (i.e., from other chares), requiring buffering facilities (e.g., in case of early value arrival). We developed Hypercomm’s notion of components to address these needs and, eventually, others (e.g., facilitating dependence relationships in SDAG).

Design Considerations

We designed components to be a flexible implementation mechanism for various parallel abstractions. We were concerned about SDAG- and task-based abstractions at the outset, then expanded our scope to include collective operations and futures. Toward this end, we broadly summarize our requirements for components as follows:

- Can express both one-time and persistent tasks.
- Can express dependencies between tasks.
- Less overhead than spawning a chore.

Functional Overview

Components encapsulate computational fragments, or *actions*. Actions express their dependencies and results in terms of input and output ports. When they receive values via input ports, they place them into value sets that may contain at most one value per port. If all existing value sets have values for an incoming value's port, they start new value sets. Complete value sets are staged, unrolled as parameters for a component's action.

Components have a three-stage lifecycle composed of standby, activated, and deactivated phases. They can only execute their action during the activated phase, outside of which they defer staging complete value sets. Subsequently, they buffer all received values during the standby phase. They can suspend instead of completing their action, leaving their phase unchanged until more value sets arrive. During their deactivated phase, components flush all held values and reject any received values, returning them to their senders. Figure 5.1 is a visualization of the lifecycle of components.

Components route the results of executing their action to their output ports. Users may connect output ports to any valid destination, e.g., another component's input port or a chore's entry method. Until connected, output ports buffer any values routed to them. Chores garbage collect deactivated components once their output port buffers are empty; this belies that the output ports of deactivated components are still connectable.

A completion or invalidation event will cause deactivation. Completion events occur when non-persistent components execute their actions. A mutable boolean flag determines persistence, enabling flexible lifespans. Receiving a null value or an explicit invalidation signal will cause an invalidation event. Unlike a completion event, components route invalidations to all their output ports when invalidated. We intend for this forwarding to help shut down chains of speculatively generated tasks.

Components are embedded within chares, migrating with their host and sharing its state. Chares store them as polymorphic base types, enabling extensive customization (discussed in the following sub-section). Input ports are not directly accessible outside of their component’s host chare. Instead, chares use entry ports to provide external access to their component’s input ports. Entry ports are tuples of hashable values that can be added or removed throughout a chare’s lifecycle. Note, chares buffer values received for non-existent entry ports.

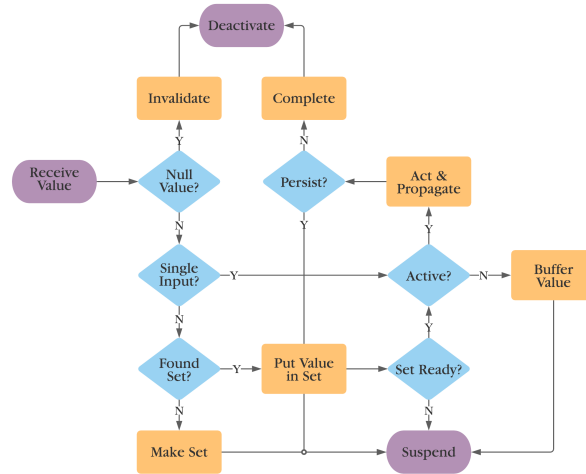


Figure 5.1: Component execution is driven by receiving values.

Programming Interface

Component behavior is highly customizable—users can control how they handle values and redefine their startup and shutdown routines. Figure 5.2 shows a minimal component implementation using the default behaviors. The syntax `hypercomm::component<std::tuple<T, T>, T>` specifies that the component has two input ports and an output port. Setting the persistent flag indicates that it should not deactivate immediately after performing its action. Figure 5.3 shows how to launch the component and connect its ports to other entities.

Mailboxes

Mailboxes are components that service requests, i.e., predicate-action pairs. When mailboxes receive a value that matches a predicate, they trigger the corresponding action, fulfilling the request. They buffer arrivals when they do not find a match. Incoming requests sweep the buffer, with the mailbox immediately fulfilling them if it finds a match. Otherwise, the mailbox adds them to the ‘outstanding’ request queue. We use this functionality to

```

template <typename T>
struct add_values : public hypercomm::component<std::tuple<T, T>, T> {
    using parent_t = hypercomm::component<std::tuple<T, T>, T>;
    using in_set = typename parent_t::in_set;
    using out_set = typename parent_t::out_set;

    add_values(const id_t &_1) : parent_t(_1) { this->persistent = true; }

    virtual out_set action(in_set &set) override {
        auto &lhs = std::get<0>(set)->value();
        auto &rhs = std::get<1>(set)->value();
        return lhs + rhs;
    }
};

```

Figure 5.2: An example component that perpetually adds values as they arrive.

```

auto com2 = this->emplace_component<add_values<value_type>>();
...
// output port com0:1 connects to input port com2:0
com0->template output_to<1, 0>(*com2);
...
// entry port (recv_value) connects to input port com2:1
this->connect(recv_value, com2, 1);
...
// output port com2:0 connects to input port com3:0
com2->template output_to<0, 0>(*com3);
...
// all connections ready---activate the component
this->activate_component(com2);

```

Figure 5.3: Launch the component shown in Figure 5.2, and connect its ports to un-shown entities.

implement SDAG when-clauses, effectively mailboxes paired with one or more components (see Figure 5.5a).

Mailboxes whose predicates implement reference number matching (like SDAG) can use a hash table for look-ups ($O(1)$ time). Otherwise, mailboxes with arbitrary predicates must use a linear scan, testing each outstanding request or buffered value ($O(n)$ time). If arrivals and requests are typically in order, the average performance of the linear scan and hash table may be similar. Regardless, mailboxes tend to meet or exceed the performance of Charm++’s SDAG—which always uses a linear scan.

Mailboxes are dramatically faster than SDAG when requests are fulfillable in any order. We demonstrate this using the synthetic ‘avalanche’ benchmark, which overwhelms match

queues to simulate simultaneously requesting numerous futures. It is, simply, a doubly-nested ‘forall’ loop containing a single ‘refnum’-matching when-clause. Figure 5.4 shows that mailboxes outperform SDAG by an increasing margin as more requests are issued. Outside of the suspected serialization within SDAG (i.e., that it processes requests in order), we require more investigation to determine the cause(s) of the performance differences. Note that we have not implemented $O(1)$ look-ups as of these results; they could further widen the margin.

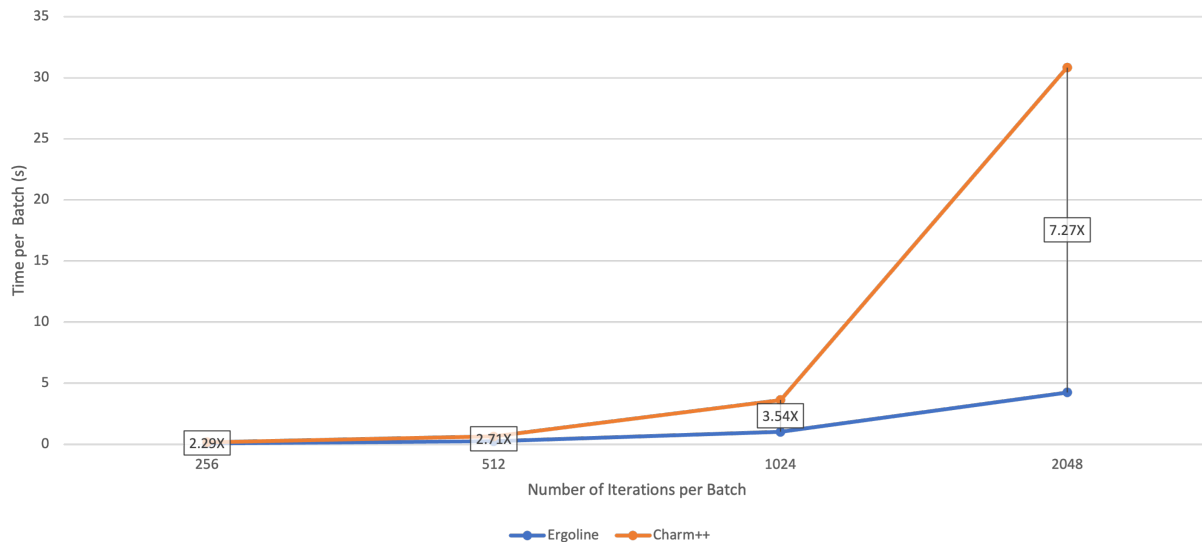


Figure 5.4: Comparison of Charm++ and Ergoline implementations of the ‘avalanche’ benchmark on a single Intel Skylake node with 18 PEs. Ergoline demonstrates linear scaling, whereas SDAG shows quadratic scaling.

Performance Concerns

Despite our best efforts to minimize component creation costs, they can still be burdensome in fine-grained scenarios. The memory allocations associated with this process (e.g., for components themselves, their entries within containers, etc.) account for most of its overheads. Subsequently, using a memory pool or faster allocator (‘jemalloc,’ ‘tcmalloc,’ etc.) may significantly reduce these overheads.

Since components are general-purpose abstractions with non-trivial lifecycles, not all use-cases require all their features. For example, one-time tasks do not need to buffer multiple values, nor the ability to suspend. We plan to explore ‘specialized’ components with only a subset of their features to reduce overheads further.

Optimizations

We realized component optimizations by targeting two overhead sources: creation and intra-process un/packing.

Multistate Components Chares may incur severe overheads when cloning components. For example, an overlapped SDAG loop will simultaneously generate numerous components (i.e., for all its when-clauses) that differ only by their parameters (i.e., the local variables they capture by value). We have designed ‘multistate’ components to optimize these scenarios.

Multistate components multiplex their ports to sets of internal states, simulating multiple conventional components. These states correspond to the input parameters and other individual properties of the multiplexed components. We encode the state’s unique identifiers and the ‘internal’ port number to form the ‘external’ port that other entities use as a destination. When values arrive, we decode the ‘external’ port to obtain this information, load the corresponding state, and route the values to the correct ‘internal’ port. Since the overheads associated with this process are minimal relative to creating multiple components, using a multistate scheme can engender savings. Figure 5.5 shows a high-level overview of both approaches.

We present the time per iteration of an overlapped loop that contains a when-clause in Table 5.1. Since the when-clause does not have a body, all the program’s execution time is spent within the RTS. Multistate components are faster than creating conventional ones every iteration in all cases. They are around 60% faster on AMD and ARM, whereas their gains are much smaller on Intel (around 3%). Although conventional components are slower than SDAG on AMD, multistate ones outperform it by roughly 50%. Thus, AMD’s characteristics demonstrate that multistate components are integral to achieving competitive performance with Charm++. They ensure that at least one Hypercomm configuration is faster than SDAG on all the platforms (likely due, in part, to overheads within the SDAG library).

	Intel [Kaby Lake]	AMD [Rome]	Apple [M1 Max]
SDAG/Charm++	3.97	3.35	1.80
Conventional	2.73	3.49	1.67
Multistate	2.65	2.18	1.07

Table 5.1: Time per iteration (in ms) of an overlapped `for` loop containing a when-clause.

Value-passing Message-passing between parallel entities in the Charm++ RTS incurs de/serialization overheads due to repeated value un/packing. Even within-node communi-

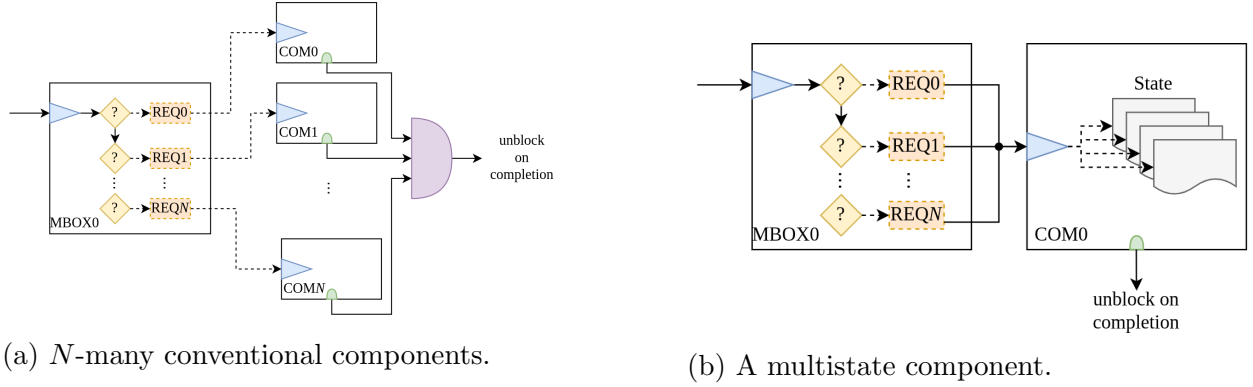


Figure 5.5: High-level overview of the approaches for implementing an overlapped `for` loop containing a `when`-clause.

cation incurs these overheads since entry methods use pass-by-copy calling conventions.¹ We directly pass values between components (within chares) to avoid these overheads. We wrap them in polymorphic containers to enable Converse to directly schedule them. Although value encapsulation incurs memory de/allocation overheads, these are a fraction of de/serialization overheads (i.e., copying and de/allocation). Subsequently, components can exchange information with their local peers very efficiently.

Under pass-by-copy calling conventions (message passing), value ownership and sharing are not concerns. However, under value-passing, multiple chares can share access to the same value. To sidestep the ensuing complexities (race conditions, etc.), we assume that sending values also forfeits ownership (i.e., the sender does not retain access afterward). We plan to incorporate other schemes in the future, like atomic or read-only shared-data semantics. Note that, although various parts of Hypercomm encourage and reflect our present assumptions (e.g., we use `std::unique_ptr` to exchange values), it does not strictly enforce them. Expert users could theoretically explore other sharing conventions within the current framework.

5.1.2 Related Work

HPX is a C++-based adaptive runtime system [53]. Whereas our components migrate with their host chares (i.e., ‘virtual’ localities), HPX’s components are individually migratable and exist within physical localities. SpECTRE is an astrophysics simulation built on Charm++ [54]. It has a template meta-programming-based interface for Charm++, featuring components and ‘inboxes’ (i.e., mailboxes) encapsulated in chares like ours. Although

¹The RTS has, historically, implemented these semantics using Charm++’s PUP module to generate a copy of a value, equivalent to full or partial de/serialization. As with all message-passing in Charm++, receiver-side pointer-to-offset optimizations avoid deserialization but not serialization.

neither HPX nor SpECTRE have connections and ports like ours, Bhandarkar previously viewed chares as components with these features [55]. We view our work as a refinement of Bhandarkar’s, moving components to smaller computation fragments *within* chares while maintaining a similar programming interface.

5.2 DE/SERIALIZATION

Infrastructure for migratable objects must provide efficient de/serialization facilities for migrations and parameter marshaling. Virtually all programs written in this model utilize these facilities, albeit in varying capacities. Parameter marshaling is avoidable by using custom message types, for example. The Charm++ module for de/serialization is called PUP. Our work builds on its ideas, incorporating new features that require a ground-up approach to explore. In particular, we extended its polymorphism and optimization capabilities to support Ergoline.

5.2.1 Polymorphic Traits

We discussed traits in 2.2.2—concrete types may inherit multiple of these purely abstract types. Although PUP supports de/serializing single-inheritance abstract types (i.e., classes), it does not support traits. Subsequently, Hypercomm must de/serialize them for Ergoline. We use C++’s RTTI facilities to identify which concrete base-type a trait instance corresponds to and de/serialize it accordingly. Charj also needed this feature but, to our knowledge, did not utilize RTTI and had a more complex solution (e.g., ScalaCharj implemented JVM-like type-erasure). Since then, C++ compilers have addressed many of the performance-related drawbacks of RTTI so we can confidently use it.

5.2.2 Optimizations

We implement two optimizations within Hypercomm’s de/serialization module to support Ergoline. One simultaneously relieves users of the burden of writing de/serialization routines for recurrent structures (e.g., doubly-linked lists or graphs). The other advances Ergoline’s support for RDMA network features, refining the ideas of Charm++’s ‘pup_buffer’ API.

Finer-graph Aware Packing

Under a finer-graph or identity-aware packing scheme, a message packer packs each unique object only once then “drops” back-references in subsequent encounters. When objects occur multiple times, this scheme decreases copying and the size of the packed message. Additionally, it trivializes packing recurrent data structures, which would otherwise cause infinite loops under identity-unaware schemes (e.g., Charm++’s PUP).

Zero-copy Aware Packing

Receivers can directly copy data from a sender’s memory-space through *Remote Direct Memory Access* (RDMA), a common feature of modern networks. This feature facilitates “zero-copy” (ZC) transfers, so-called since senders do not need to copy data into designated send buffers (compared to conventional techniques). These avoided copies translate to significant decreases in communication overheads; however, the control messages used to initiate and finalize ZC transfers can outweigh their benefits for small transfers. Note, these messages communicate which memory range the receiver should copy from and when it has finished copying data (so the sender can “free” it).

Charm++ offers various mechanisms for applying these ideas through its Zero-Copy API, each representing a different abstraction level. At the lowest level, users are responsible for “driving” the zero-copy process (i.e, exchanging control messages, etc.). Higher abstraction levels manage more of this process, but users lose the ability to fallback to conventional techniques if message sizes shrink. Regardless of their choice of abstraction level, users must “opt-into” the zero-copy API and explicitly rewrite their code to utilize it. Consequently, the impact of sending control messages is unavoidable even on machines without RDMA facilities where the ZC API must offer “fallback” routines to support them (for portability). An adaptive abstraction can offer the best of both worlds—dynamically selecting between zero-copy and conventional transfer techniques, potentially with runtime feedback.

Ergoline offers an adaptive solution through its zero-copy aware packing schemes. When a sender encounters a zero-copy eligible chunk of data during message packing, it “quick-sizes” it and decides whether to pack it in-line or defer it based on a threshold. The properties of the deferred data (i.e., its size and address) are packed in-line and, through ARC, the sender retains (shared) ownership of the data until the zero-copy transfer has completed. While unpacking a message, the receiver automatically detects deferred chunks of data, initiates the receive procedure(s), then postpones (final) delivery of the message until all deferred chunks of data have arrived. Appropriately-sized receive buffers may be implicitly allocated

(as necessary), or explicitly “posted” by the user elsewhere.

This strategy affords *Ergoline* several benefits, namely that zero-copy transfers become the *modus operandi* rather than an opt-in technique. Their management is delegated to the RTS, granting it more “knobs” for adaptive tuning. Furthermore, users do not incur the penalty of control messages for small data blocks or on non-RDMA enabled machines since the RTS can fall back to conventional, in-line packing. This strategy’s effects are most noticeable when comparing a naive bandwidth benchmark written in *Ergoline* to one written in Charm++, where the automatic switchover to zero-copying results in sizable gains for *Ergoline* (shown in Figure 5.6).

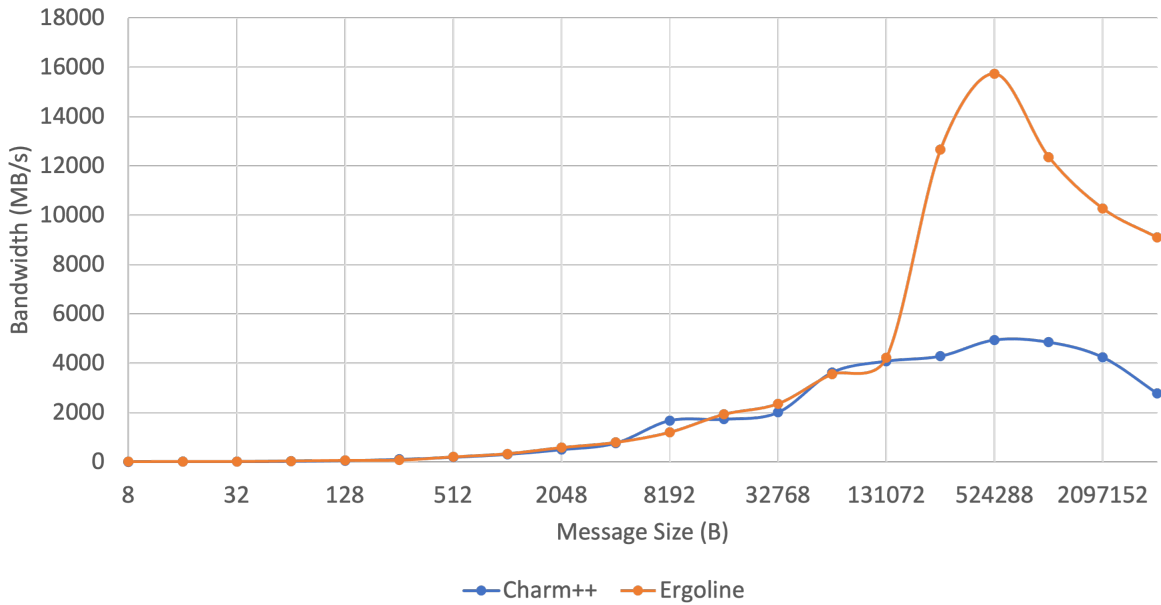


Figure 5.6: Intra-node bandwidth comparison between Charm++ and Ergoline on a single node of Stampede2. The switchover point for Ergoline is 256KB, after which it demonstrates the benefits of zero-copying. Ergoline is consistently slower than Charm++ with 8KB messages, potentially due to component creation overheads; switching to multi-state components will likely improve its performance across the board.

5.2.3 Writes After Sends

One cannot modify deferred data until all receivers have copied the data into their local buffers. One can designate a callback for the RTS to fire once the receiver(s) have finished copying the sent data, indicating it is ‘safe’ for the sender to modify its data. Interestingly, this is also an area where compiler support can benefit parallel programming—compilers

can automatically segment entry methods where users modify data after sending it via the ZC-API. This transformation would maintain the illusion of copy-only send semantics.

5.3 ELEMENT-SPANNING TREES

Sections enable efficient communication over subsets of collections in Charm++, e.g., a broadcast over the elements with even indices. Spanning trees over a section’s members are the basis for these operations. We refer to an element’s parent and children within the tree as its *associations*. This information determines where to send messages during collectives. Reductions use bottom-up motion (from children to parents), while broadcasts move from the tree’s root downward (i.e., top-down).

Tree construction must consider locality to ensure efficient messaging. In other words, it should preferentially connect elements that are physically close to minimize costly long-distance messaging. Existing solutions vary somewhat in their approach to achieve these results. Charm++ uses a centralized scheme that gathers all the section members’ locations at one location then partitions them. Then, it distributes the resulting tree among participants. Member migration triggers tree reconstruction. Members may hold an out-of-date version of the tree since reconstruction is asynchronous; typically, this only has performance implications (i.e., suboptimal connections). Charm4Py has a similar scheme, although it distributes the partitioning operation. It does not support migrations presently—a solution will likely involve indirection or rebuilding. Here, indirection refers to forwarding the messages from a chare’s old location to its new one. Neither approach supports deleting section members since users build them with explicit member lists (unlike more generic definitions, like “all the members with even indices”).

This section describes Hypercomm’s approach, a novel distributed scheme that utilizes dynamic reassociations. This scheme avoids completely rebuilding the tree when elements are created, destroyed, or migrated. Critically, for Ergoline, it enables ‘at-will’ section creation using membership functions; users can specify section members while issuing collective operations to avoid the typical “create then broadcast” step. This flexibility enables Ergoline’s sections to be expressive and support reduction-only use cases.

5.3.1 Construction

Our distributed tree construction scheme has two insertion modes: dynamic and static. Static insertions are unassociated and occur during designated phases, which users explicitly start and stop. Dynamic insertions, those outside of these phases, require reference points

within collections, i.e., associations. Elements use associations to decide when to wait for more reduction contributions or broadcast messages, ensuring newly-inserted ones may participate in collectives. Subsequently, without associations, one cannot reliably determine which collectives will include new elements. To account for this uncertainty, we buffer collectives started during static insertion phases, launching them once the phase concludes.

During static insertion phases, elements attempt to connect to local peers (i.e., on the same physical node) whenever they ‘come online.’ The search moves upstream when there are no local section members. Upstream refers to the node’s parent in the arithmetically or topologically defined node-spanning tree. The first element that arrives at the root of the node-spanning tree, via forwarding or creation, becomes the root of the element-spanning tree. When the node-spanning tree’s root does not have any elements, it routes subsequent arrivals to the element-spanning tree’s root. While upward-only routing avoids creating cycles, suboptimal connections can arise depending on the element arrival order (discussed in *Optimizations*).

Tree-builders use completion detection to unblock collectives after users finish inserting new members. Completion detection events include intermediate messages and creating and associating elements; these correspond to incrementing or decrementing local event counts. The detector routinely accumulates the local counts of all nodes. Completion occurs when the global total is zero; this condition indicates a tree-builder consumed all the events it produced. This process accounts for a tree-builder’s off-node communication, as a node may consume events created by one of its peers.

Figure 5.7 shows an example tree constructed by this scheme. Assume CH’s 0, 1, and 2 simultaneously appear on NODE0; they connect locally, and CH0 is elected the endpoint of the spanning tree. When CH5 is the first element to arrive on NODE1, its connection request goes to NODE1. Since nothing is present on NODE1, it directs the request upstream to NODE0, where CH1 accepts the connection request. The process continues until all the chares are connected.

Migrations

The connections between chares persist after they migrate. Since the spanning tree remains intact, collectives are unaffected by migrations. However, this may induce sub-optimal amounts of off-node messaging unless all locally connected elements migrate together. For example, an element’s parent may remain at its old location after it has moved to a new one. We mitigate this problem with re-associations, attempting to connect elements to those present at their post-migration location. We discuss this path further in *Optimizations*.

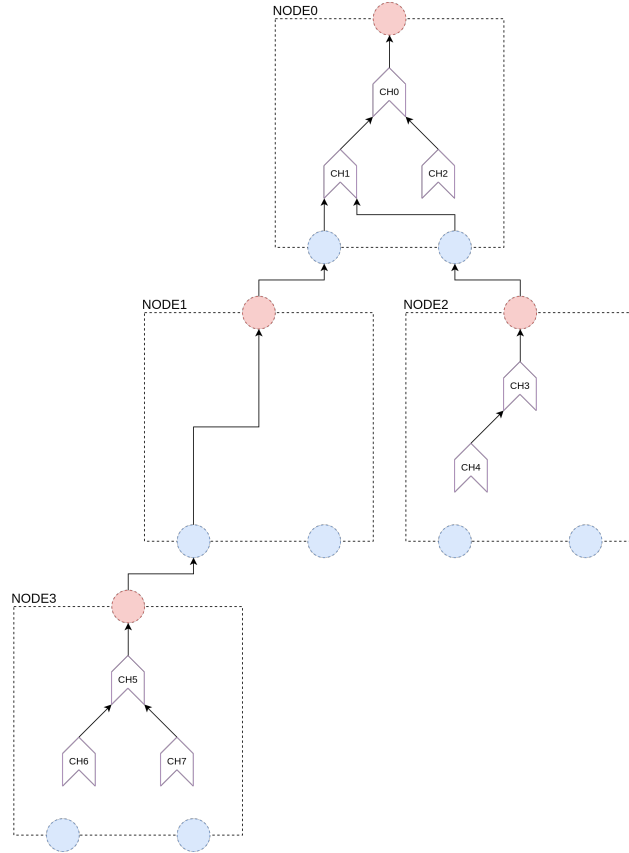


Figure 5.7: Tree-builder visualization.

Dynamic Insertions

Users explicitly associate all insertions outside of static phases with existing section members. To ensure the new member has a well-defined reference point for collectives, the established member becomes its parent in the spanning tree and forwards its generation counts (i.e., for broadcasts and reductions). All collectives after a new member’s insertion will include it. In Charm++, one cannot predict which collectives will consider new members since insertions are unassociated. For example, users may insert chares then be surprised when subsequent reductions do not include them. Note, we inherit this uncertainty for demand-created elements (i.e., via the create here/home annotations) since users cannot associate them.

Deletions

Destroying chares causes them to ‘collapse’ the tree, moving their children and records to their parents and requesting the RTS to forward their messages upward as well. Message

forwarding enables collective operations to continue unfettered by chare destruction; for example, we do not have to edit the destinations of in-flight contributions since they remain valid. The endpoint cannot use this scheme since it does not have a parent. Instead, it must promote one of its children to become the new endpoint and the parent of its other children. Promotion can result in cyclical messaging; ‘in-flight’ operations bound for the old endpoint will return to the new endpoint. Receiving the old endpoint’s records will help the next one resolve these ‘oddities;’ however, if it receives the records before being promoted, it will temporarily buffer them.

Optimizations

Migrations and the insertion order of elements can introduce suboptimal connections into the spanning tree. Here, suboptimality refers to the amount of inter-node messaging. Ideally, each node should have, at most, one outgoing and k incoming off-node connection(s), where k is the degree of the node-spanning tree. We explore how to restore balance to unbalanced trees in this section, thus optimizing communication overheads.

Dynamic Reassociations The RTS can reassociate elements in a locality with identical generation counts. This operation enables pruning the number of remote connections; however, it must not create cycles. In practice, this requirement restricts reassociation to elements without remote children. Figure 5.8 shows an example where reassigning an element’s parent would yield a cyclic relationship. We could relax this restriction by maintaining a cache of remote connections or sending speculative requests; however, it would complicate the implementation.

We try reassociating chares when they arrive from other PEs; this is an immediate opportunity to avoid remote messaging. If unsuccessful, we try again after they accept broadcasts or contribute to reductions since their generation counts will change. The worst-case performance for a linear scan for reassociation opportunities is $O(n)$, where n is the number of elements in a locality. Subsequently, the “try to reassociate” operation will not be overly time-consuming in general; however, if it proves to be a bottleneck, we could use a hash-map instead (for $O(1)$ look ups).

Insertion-time All connections may route through a single PE under some circumstances. For example, an element on PE2 becomes the endpoint in a 16 PE run. Then, if nothing appears on PE3, any elements inserted on PEs 5, 6, 7, 8, 9, and 10 would go to PE2. We have a pair of ideas to avoid building unbalanced spanning trees in such scenarios:

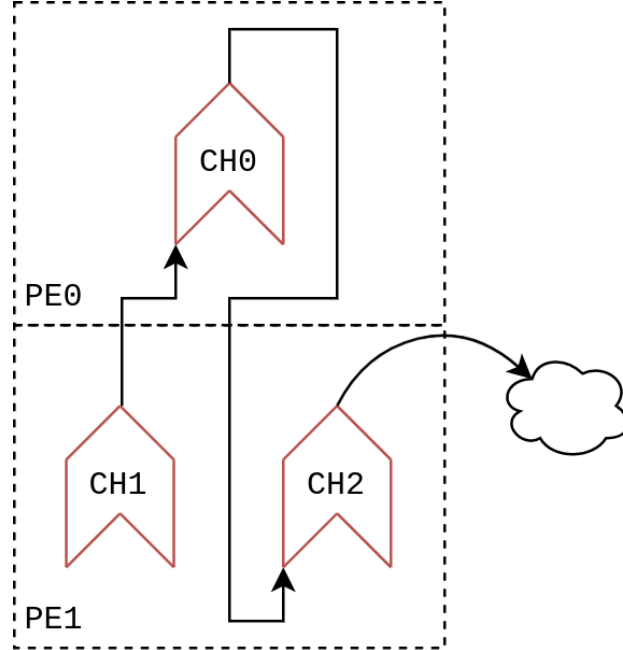


Figure 5.8: CH1 can reassociate as CH2’s child; however, the reverse would create a cycle.

- **Junction Caches.** Cache failed connection attempts at ‘junction’ PEs, i.e., those without elements. Route future requests the cached (element, PE) pairs instead of ‘going up,’ assuming they receive more up-to-date information.
 - In the above scenario, PE3 can route requests to PEs 6 and 7 instead of PE2 since they failed to connect locally.
- **k -child Enforcement.** Redirect requests when elements already have k children.

Since junction caching reroutes “upstream” requests to known-good peers, we believe its cycle-creation risk is low. However, we have not rigorously examined whether these schemes can create cycles. Subsequently, we have deferred implementing them for now. A potential solution may be actively tracking where requests and elements originate from to make “smarter” decisions.

5.3.2 Collectives

Instead of maintaining separate implementations of collectives for groups, node-groups, and chare-arrays (like Charm++), Hypercomm’s chare-collections share a common, component-based approach. It works with any tree that spans all elements. We use arithmetically defined trees for groups, node-groups, and their sections. For chare-arrays, we use the tree-builder

scheme introduced earlier. In theory, we could also use the centralized section-creation mechanisms from mainline Charm++. This flexibility is one of the core strengths of the component-based scheme; it enables research into new tree-construction schemes without substantial code refactoring efforts.

Broadcasts

Hypercomm sequentializes broadcasts at the root of the spanning tree, forming a leader-based scheme like Charm++’s. Each chare must track the broadcasts it has accepted to avoid duplicates. Since there is, effectively, only one origin point of broadcasts (i.e., the leader), this scheme requires $O(n)$ broadcast counts (where n is the number of chares). A naive leaderless approach would need, at worst, $O(n^2)$ storage since any chare could start a broadcast. These storage costs are prohibitive in a massively distributed system. However, these schemes are insensitive to the loss of the leader (since one does not exist) and may have better communication properties. Under a leaderless strategy, simultaneous broadcasts may utilize different network paths. Comparatively, all leader-based broadcasts follow the same route through the spanning tree (i.e., top-to-bottom). We may explore whether these potential benefits of a leaderless scheme outweigh their storage costs in the future.

We implement broadcasts as remote actions that chares execute upon arrival. Broadcast actions propagate self-replicas to the children of a chare, then locally deliver their value. Each broadcast is stamped with a monotonically increasing ID by the root. Chares process them in order, using a single counter to avoid accepting duplicates. Chares reject past broadcasts and buffer future ones; they flush their buffers whenever they handle one. We require that all participating chares accept each broadcast once; however, they may receive multiple actions corresponding to the same broadcast due to chare deletion.

Reductions

When chares receive contributions to reductions, they create reducer components. Reducers determine how many values to expect based on their host chare’s child count (via the spanning tree). When all the contributions have arrived, reducers combine them and propagate them to their host chare’s parent. If their host chare does not have a parent, reducers send their results to the reduction’s destination. Since Hypercomm’s sections are anonymous—created at the call site—Ergoline’s syntax for reductions can be much simpler than Charm++’s, shown in Figure 5.9.

```

// called on a single array element
void foo::make_section(const std::vector<int>& indices) {
    CProxySection_foo x =
        CProxySection_foo::ckNew(this->ckGetArrayID(), indices);
    x.make_contribution(...);
}

void foo::make_contribution(CkMcastBaseMsg *msg) {
    CkSectionInfo cookie;
    CkGetSectionInfo(cookie, msg);
    CkCallback cb(CkReductionTarget(Main, done), mainProxy);
    CProxySection_Check::contribute(
        sizeof(int), &thisIndex, CkReduction::sum_int, cookie, cb);
    delete msg;
}

```

(a) Example section reduction in Charm++.

```

def make_contribution(indices: array<int>) {
    self@[indices][@]contribute(self[@]index(), int::+, mainProxy.done);
}

```

(b) Example section reduction in Ergoline.

Figure 5.9: Comparison of section reductions between Charm++ and Ergoline. Where Charm++ enforces a create-then-multicast-reduce pattern for its sections, Ergoline allows anonymous, reduction-only sections. Hypercomm’s sections facilitate this simplicity.

5.4 FUTURES

Recall that all chares in Hypercomm are elements of collections. Futures, thus, correspond to (element, id) pairs, indicating the chare that issued them and their unique identifier. To support this, chares maintain counters, incrementing them whenever they create futures.

Users provide continuations whenever they request the values of futures. These continuations can, for example, resume threads or route values to other destinations. Whichever chare ‘contains’ the request opens an entry port corresponding to it. Upon receiving values, these ports fire the requester’s designated continuation (with the values as arguments) then expire. Futures not issued by the requesting chare require sending a forwarding request to the issuer.

Chares that issued futures can invalidate requests for them by deleting the corresponding entry ports; thus, when values arrive, they will not trigger continuations. Non-issuing chares are encouraged to “bounce” values back to their issuer in the event of an invalidation, giving another entity a chance to consume them. In both cases, unused future values will remain in buffers within their issuing chare; one can install “null” continuations to flush these buffers, albeit at the cost of multiple memory de/allocation (for the continuation itself, its spot

within a container, etc.). Worse yet, multiple invalidations by non-issuing chares results in needless value redirections. Thus, we encourage restraint while utilizing these features; if one issues a future, they should consider it a promise to be fulfilled.

In closing, our implementation of future variables is greatly simplified by reusing the buffering and lifecycle management provided by entry ports. Since futures are bound to chares, requests are resilient to migrations. We also have comprehensive features for requesting ‘remote’ values. Both capabilities are advancements in flexibility relative to the PE-bound scheme currently employed by Charm++; we hope to ‘back-port’ these improvements to the core RTS in the future.

5.5 MIGRATABLE THREADS

Charm++’s user-level threads (ULTs) are not migratable by default. Thread stacks contain local variables that may be pointers to heap-allocated memory blocks. Subsequently, capturing the entire memory space encompassed by a stack is non-trivial. The isomalloc module of Charm++ simplifies ULT migration by providing a mechanism for allocating memory blocks with globally unique virtual addresses. It divides the virtual address space into *contexts*, assigning each to a ULT. If a ULT’s isomalloc context contains all its heap and stack allocations, migration is as easy as copying the memory range from one machine to another. Pointers are unaffected since the context is assigned the same virtual addresses on the destination. However, local variables that point to memory outside their ULT’s isomalloc context will result in undefined behavior. ULTs maintain exclusive ownership of their context until they terminate; they are reusable afterward. Leveraging isomalloc is not the default behavior because there are a limited number of contexts available and performance implications for thread creation and context-switching time [56].

Charm++ does not have facilities for associating migratable ULTs with chares and automating the management of isomalloc contexts (i.e., de/allocation). Subsequently, it does not support migratable threaded entry methods. All a chare’s ULTs must migrate with it and obtain or release isomalloc contexts when they start or finish. Hypercomm addresses these needs through its “thread manager” interface.

Every PE creates several isomalloc contexts at startup, placing them into its free pool. Migratable threads draw from this pool when they start; if it is empty, we forward the request elsewhere.² They return their contexts to the PE they reside on when they complete. This strategy can lead to starvation, but it involves less communication than periodic re-

²The allocation will fail if the request cycles back to the PE that issued it.

distribution or sending contexts home. Chares have thread managers that implement these behaviors and maintain a list of active threads. To migrate, they simply de/serialize each item in this list. Since C++ does not provide a way to reassign the ‘this’ pointer (representing the chare’s state) post-migration, the thread manager also maintains a special ‘back-reference’ that ULTs should use instead. Figure 5.10 demonstrates all these ideas, showing how Hypercomm implements migratable threaded entry methods. We plan to update Ergoline’s compiler to leverage these facilities in the future.

5.6 SUMMARY

We explored Hypercomm’s modules and primary programming model in this chapter. Unhindered by supporting existing applications (i.e., legacy), we could explore a mix of new directions and improvements, occasionally taking a radical departure from Converse and Charm++ to offer novel optimizations. Future software engineering efforts may seek to ‘back-port’ these improvements to the ‘legacy’ frameworks. Recall that, in many cases, these changes reflect desired outcomes in EIR (e.g., cleaner sections) and, thereby, Hypercomm and EIR’s co-design. Whenever we introduced new ideas in EIR, we realized robust implementations for them in Hypercomm. We hope they will be helpful to others building Charm++ abstractions and applications looking forward.

```

struct example : public CBase_example {
    hypercomm::thread_manager man;
    int tid;

    example(void) : man(this) {
        // enables at-sync load balancing
        this->usesAtSync = true;
    }

    // restores the 'back-reference' on resume
    example(CkMigrateMessage*) : man(this) {}

    void pup(PUP::er& p) {
        // this will de/serialize all the threads
        p | man;
        // and the active one's id as well
        p | tid;
    }

    /* entry */ void run(CkMessage* msg) {
        // launch a thread using the manager
        auto thp = man.emplace(&example::run_, msg);
        // save its id so we can resume it later
        this->tid = thp.second;
        // then immediately launch it
        CthResume(thp.first);
    }

    // note how 'self' is a reference so the manager
    // can update it after we resume-from-sleep
    static void run_(example& self, CkMessage* msg) {
        // free the message (before we migrate)
        CkFreeMsg(msg);
        // print a message (as one does)
        auto idx = self->thisIndex;
        CkPrintf("ch%d@pe%d> I'm going to sleep...\n", idx, CkMyPe());
        // tell the load-balancer we're ready to migrate
        self->AtSync();
        // then suspend (so we can migrate)
        CthSuspend();
        // validate that (self) is still accessible
        // and it's value matches the old value
        if (idx != self->thisIndex) {
            CkAbort("ch%d> unsuccessful resume-from-sync! incorrect idx(%d)!\n",
                    idx, self->thisIndex);
        } else {
            // otherwise... we succeeded!
            CkPrintf("ch%d@pe%d> I'm alive again~!\n", idx, CkMyPe());
        }
    }

    // called after we finish migrating
    // ( resumes the thread using the saved id )
    void ResumeFromSync(void) { this->man.resume(this->tid); }
};

```

Figure 5.10: Example of using Hypercomm's thread manager.

CHAPTER 6: CONCLUSIONS

Recall that this work’s motivation is the non-triviality of language construction. Standalone languages require one to implement lexers, parsers, type checkers, code generators, etc. Embedded languages and frameworks can relieve developers of many such burdens but may impose syntactic constraints. HPC languages require runtime support to schedule work and manage communication, regardless of their approach. Existing HPC frameworks and intermediate representations have been low-level (MPI-level, no object-oriented programming support, etc.) or lack support for distributed computing (e.g., Delite). These shortcomings do not mesh well with Charm++, whose migratable objects model requires rich support for remote method invocations and object-oriented programming to reach its full potential. Yet, past works have shown that Charm++ can benefit language authors, providing sophisticated fault-tolerance and load-balancing facilities. Hence our proposal that a language framework built on Charm++ would be a highly-productive platform for DSL authorship.

In this dissertation, we developed abstractions and a framework to help language authors utilize the migratable objects model. We targeted them with a high-level, general-purpose language and embedded several DSLs within it to demonstrate our approach. Generally, we drew inspiration for these DSLs from past works built on Charm++ [6][7]; however, we looked to an external source for our Stencil DSL. In addition to these DSLs, we explored representing common HPC abstractions within our high-level language to exercise its generality. Advancements relative to prior art facilitated these abstractions, i.e., language features like automatic specialization registration.

We divide the rest of this chapter into retrospective and prospective views of our work. In the retrospective view, we discuss the insights we gained into our research questions while elaborating on the relevant context for this work. We outline directions for this work in the prospective view.

6.1 RETROSPECTIVE

Toward answering our research questions, this section begins by establishing the context of this work. This context helps illustrate how our unified approach is more productive than its immediate predecessor. Then, we discuss the insights into our research questions that we gained throughout this work—primarily in qualitative terms.

6.1.1 Context

Our experiences during Charisma and Charj modernization projects influenced EIR’s design in several key ways. Both efforts sought to update Java-based infrastructure and syntax to Scala-based alternatives. They sought to realize proposals from the original Charj project, like integrating Charisma and Charj and embedding MSA within the latter. We carried these aspirations and preferences forward to EIR and Ergoline, refining their ideas while achieving them.

Starting from the Scala-based version of Charj (Scala Charj), we began Ergoline’s design by adding tuple types and refining the semantics of operator overloads and types. Where Scala Charj took a JVM-inspired, type erasure-based approach for implementing its generics, we returned to Charj’s C++-inspired templates for performance. Additionally, we adopted an annotations-based approach for C++ interoperability where Scala-Charj used constrained “string templates” (i.e., inline, user-written C++ code with placeholders). These design decisions, and infrastructural factors, hindered Scala Charj’s usability for developing applications.

We derived cautionary lessons for EIR from the infrastructure of the Scala-based Charisma and Charj projects. First, these projects began with a standalone Charisma compiler with bespoke infrastructure (e.g., parsers for Charisma and Charmxi) rather than embedding Charisma within Charj at the outset. Ultimately, these were fruitless efforts that distracted from the goal of reviving Charisma. Second, Charj’s compiler was not flexible enough for language embedding. Modifying the AST based on the MSA analysis results was challenging, e.g., deleting or inserting phase changes. Many parts of the AST were immutable, and other passes frequently stored state within AST nodes, complicating piecewise cloning and rebuilding. We concluded that refactoring all the AST nodes for mutability would have been tantamount to entirely rewriting them. Lastly, the loss of institutional memory regarding Charj’s compiler compounded the impact of these factors, leading us to develop EIR.

6.1.2 Insights

Starting with the “easiest” questions—“can we blend diverse control and data views?” and “can we convert standalone DSLs to embedded ones?”—the answer to both is a resounding *yes*. This conclusion is, perhaps, unsurprising since Becker embedded MSA within Charj and proposed embedding Charisma within it [6]; after all, broadly, what applies to Charj applies to Ergoline. Ergoline carried the goals of Charj and the Scala-based modernization efforts forward, inheriting the benefits of hindsight from both.

Compared to our prior efforts, EIR’s distinguishing characteristics are its unified compiler and enhanced runtime infrastructure. These characteristics allow us to explore the questions “when does a shared compiler infrastructure benefit multiple DSLs targeting the same adaptive RTS?” and “when can RTS-level abstractions assist in building HPC languages?” Our investments in EIR’s extensibility allowed us to rapidly prototype Charisma and Stencil EDSLs in Ergoline—both can share its SDAG segmentation logic and message un/packing optimizations. Hypercomm’s components and task groups further simplified our EDSL implementations; their operations require minimal lowering to be represented by these abstractions, mitigating semantic gaps. Overall, we believe that our multi-layer language framework approach benefited our endeavors.

6.1.3 Acknowledgements

With all this context established, we hope that it is clear that this work follows in the footsteps of numerous language projects in the Charm++ ecosystem. It is quite likely that, had these projects continued, they would have produced results like ours. In particular, [Scala]Charj lit the fire for Ergoline’s language embedding and we carried its torch onward. Toward keeping the metaphorical torch alight, we present a prospective view of this work.

6.2 PROSPECTIVE

We view this dissertation as a vehicle for bringing new abstractions and approaches to the broader Charm++ ecosystem; we aim, in part, for its ideas “trickle out” to production software. Charm++ may incorporate our model for futures, and Charm4Py may adopt aspects of our zero-copy-aware packing scheme. The Parallel Tree Toolkit, ParaTreeT, a framework for massive tree-based simulations built on Charm++, has incorporated static registration techniques conceived during this work’s course. We also established CharmLite, a ground-up reimagination of Charm++ built on Converse. It eliminates the need for Charmxi via C++ template meta-programming and aims to bring Hypercomm’s other ideas into a production capacity (e.g., its tree-builder scheme). Our goal is for CharmLite to help ‘modernize’ Charm++’s programming interface, bringing it closer to Charm4Py or Ergoline.

As for research directions for this work itself, we view accelerators and aggregation as under-discussed topics. Both are increasingly relevant on modern supercomputers, where accelerators contribute most of the FLOPs, and Network Interface Cards (NICs) have restricted amounts of pinned memory. Applications generating numerous small messages (e.g., astrophysics, graph analytics) can encounter these restrictions, resulting in out-of-memory

failures. Aggregation, or the process of combining smaller messages into larger ones, might be a potential solution to this problem. In the future, we could integrate the APIs that Charm++ established into EIR—Hybrid API (HAPI) for accelerators [57] and TRAM for aggregation [58]. Minimally, TRAM requires adding an ‘aggregated’ annotation; however, since its effectiveness is highly case-sensitive, some degree of auto-tuning would be helpful (i.e., runtime support). HAPI, on the other hand, would require generating CUDA from EIR. Although CUDA generation would be more involved, Chapel and Python/Numba have set a precedent [59][60], showing it is feasible.

Finally, there are four paths for incorporating or integrating other software stacks with ours. First, one can use Hypercomm’s abstractions to optimize and simplify their applications. For example, Hypercomm could supplant the runtime layer Spectre’s authors built on top of Charm++. Second, one can migrate programming models onto the EIR software stack, offering them compiler and runtime optimizations. Like, one could extend EIR’s advantages to other C++-based adaptive runtimes (HPX, STAPL, etc.). Third, one can migrate the EIR software stack onto other platforms, e.g., re-targeting Hypercomm’s abstractions to Legion. The fourth path is to improve the interoperability of mainstream languages with EIR-based ones. Many Python packages could benefit Ergoline authors, and trivializing bidirectional interoperability with Charm++ would help promote Ergoline’s adoption. We elaborate on these paths in Appendix B.

REFERENCES

- [1] S. Vaughan-Nichols, “Rust takes a major step forward as linux’s second official language,” Dec 2021. [Online]. Available: <https://www.zdnet.com/article/rust-takes-a-major-step-forward-as-linuxs-second-official-language/>
- [2] M. Larabel, “Rust for the linux kernel sent out for review a fourth time,” 2022. [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=Rust-For-Linux-Kernel-v4
- [3] “The tiobe index.” [Online]. Available: <https://www.tiobe.com/tiobe-index/>
- [4] V. Amaral, B. Norberto, M. Goulão, M. Aldinucci, S. Benkner, A. Bracciali, P. Carreira, E. Celms, L. Correia, C. Grelck, H. Karatza, C. Kessler, P. Kilpatrick, H. Martiniano, I. Mavridis, S. Pllana, A. Respício, J. Simão, L. Veiga, and A. Visa, “Programming languages for data-intensive hpc applications: A systematic mapping study,” *Parallel Computing*, vol. 91, p. 102584, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167819119301759>
- [5] S. K. Lam, A. Pitrou, and S. Seibert, “Numba: A llvm-based python jit compiler,” in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: <https://doi.org/10.1145/2833157.2833162>
- [6] A. Becker, “Compiler support for productive message-driven parallel programming,” Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2012. [Online]. Available: <http://charm.cs.uiuc.edu/media/12-44/>
- [7] P. Jetley, “Incompleteness + interoperability: A multi-paradigm approach to parallel programming for science and engineering applications,” Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2014. [Online]. Available: <http://charm.cs.illinois.edu/newPapers/14-22/dissertation.pdf>
- [8] P. Gadikar, P. Diehl, and P. K. Jha, “Load balancing for distributed nonlocal models within asynchronous many-task systems,” *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Jun 2021. [Online]. Available: <http://dx.doi.org/10.1109/IPDPSW52791.2021.00103>
- [9] A. Hayashi, J. Zhao, M. Ferguson, and V. Sarkar, “Llvm-based communication optimizations for pgas programs,” in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: <https://doi.org/10.1145/2833157.2833164>

- [10] E. Kayraklioglu, M. P. Ferguson, and T. El-Ghazawi, “Lapps: Locality-aware productive prefetching support for pgas,” *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, Aug. 2018. [Online]. Available: <https://doi.org/10.1145/3233299>
- [11] L. Jenkins, M. Zalewski, and M. Ferguson, “Chapel aggregation library (cal),” in *2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, 2018, pp. 34–43.
- [12] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, “Regent: a high-productivity programming language for hpc with logical regions,” in *SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.
- [13] L. V. Kalé, M. Hills, and C. Huang, “An orchestration language for parallel objects,” in *Proceedings of Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR 04)*, Houston, Texas, October 2004.
- [14] C. Huang and L. V. Kale, “Charisma: Orchestrating migratable parallel objects,” in *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.
- [15] J. DeSouza, “Jade: Compiler-supported multi-paradigm processor virtualization-based parallel programming,” Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004.
- [16] J. DeSouza and L. V. Kalé, “Jade: A parallel message-driven Java,” in *Proc. Workshop on Java in Computational Science, held in conjunction with the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia and Saint Petersburg, Russian Federation, June 2003.
- [17] L. V. Kale and M. Bhandarkar, “Structured Dagger: A Coordination Language for Message-Driven Programming,” in *Proceedings of Second International Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 1123-1124, September 1996, pp. 646–653.
- [18] J. DeSouza and L. V. Kalé, “Msa: Multiphase specifically shared arrays,” in *Languages and Compilers for High Performance Computing*, R. Eigenmann, Z. Li, and S. P. Midkiff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 268–282.
- [19] C. Artho, K. Havelund, R. Kumar, and Y. Yamagata, “Domain-specific languages with scala,” 11 2015, pp. 1–16.
- [20] F. Tomassetti and V. Zaytsev, “Reflections on the lack of adoption of domain specific languages,” in *5th International Workshop on Open and Original Problems in Software Language Engineering, OOPSLE 2020*. CEUR, 2020, pp. 85–94.
- [21] N. Khouzami, L. Schütze, P. Incardona, L. Kraatz, T. Subic, J. Castrillón, and I. Sbalzarini, *The OpenPME Problem Solving Environment for Numerical Simulations*, 06 2021, pp. 614–627.

- [22] A. Afanasyev, M. Bianco, L. Mosimann, C. Osuna, F. Thaler, H. Vogt, O. Fuhrer, J. VandeVondele, and T. C. Schulthess, “GridTools: A framework for portable weather and climate applications.” [Online]. Available: <https://github.com/GridTools/gridtools>
- [23] F. Rathgeber, G. R. Markall, L. Mitchell, N. Lorient, D. A. Ham, C. Bertolli, and P. H. Kelly, “Pyop2: A high-level framework for performance-portable simulations on unstructured meshes,” in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 1116–1123.
- [24] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, “Liszt: A domain specific language for building portable mesh-based pde solvers,” in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2011)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2011. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1145/2063384.2063396> pp. 1–12.
- [25] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, “Delite: A compiler architecture for performance-oriented embedded domain-specific languages,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, apr 2014. [Online]. Available: <https://doi.org/10.1145/2584665>
- [26] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, “Graphit: A high-performance graph dsl,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018. [Online]. Available: <https://doi.org/10.1145/3276491>
- [27] B. Cosenza, N. Popov, B. Juurlink, P. Richmond, M. Kabiri Chimeh, C. Spagnuolo, G. Cordasco, and V. Scarano, “Openabl: A domain-specific language for parallel and distributed agent-based simulations,” 08 2018.
- [28] T. Molderez, B. Oeyen, C. De Roover, and W. De Meuter, “Marlon: a domain-specific language for multi-agent reinforcement learning on networks,” 04 2019, pp. 1322–1329.
- [29] R. Arora, P. Bangalore, and M. Mernik, “Raising the level of abstraction for developing message passing applications,” *J. Supercomput.*, vol. 59, no. 2, p. 1079–1100, feb 2012. [Online]. Available: <https://doi.org/10.1007/s11227-010-0490-3>
- [30] B. Tekinerdogan and E. Arkin, “Pardsl: A domain-specific language framework for supporting deployment of parallel algorithms,” *Softw. Syst. Model.*, vol. 18, no. 5, p. 2907–2935, oct 2019. [Online]. Available: <https://doi.org/10.1007/s10270-018-00705-w>
- [31] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, Palo Alto, California, Mar 2004.
- [32] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt, “Anydsl: A partial evaluation framework for programming high-performance libraries,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018. [Online]. Available: <https://doi.org/10.1145/3276489>

- [33] D. Deb, R. Fowler, and A. Porterfield, “Quarc: An optimized dsl framework using llvm,” 11 2017, pp. 1–11.
- [34] C. Osuna, T. Wicky, F. Thuerling, T. Hoefler, and O. Fuhrer, “Dawn: a high-level domain-specific language compiler toolchain for weather and climate applications,” *Supercomputing Frontiers and Innovations*, vol. 7, no. 2, p. 79–97, Jul. 2020. [Online]. Available: <https://superfri.org/index.php/superfri/article/view/314>
- [35] T. B. Schardl, W. S. Moses, and C. E. Leiserson, “Tapir: Embedding fork-join parallelism into llvm’s intermediate representation,” *SIGPLAN Not.*, vol. 52, no. 8, p. 249–265, Jan. 2017. [Online]. Available: <https://doi.org/10.1145/3155284.3018758>
- [36] N. Moss, K. Davis, and P. McCormick, “The ares high-level intermediate representation,” in *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC ’16. IEEE Press, 2016, p. 32–36.
- [37] P. Srivastava, M. Kotsifakou, and V. S. Adve, “HPVM: A portable virtual instruction set for heterogeneous parallel systems,” *CoRR*, vol. abs/1611.00860, 2016. [Online]. Available: <http://arxiv.org/abs/1611.00860>
- [38] M. Arenaz and X. Martorell, “Parallelware tools: An experimental evaluation on power systems,” 2021.
- [39] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer, “Inspire: The insieme parallel intermediate representation,” in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 7–17.
- [40] M. Steuwer, T. Remmelg, and C. Dubach, “Lift: A functional data-parallel ir for high-performance gpu code generation,” in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 74–85.
- [41] H. Murai, M. Sato, M. Nakao, and J. Lee, “Metaprogramming framework for existing hpc languages based on the omni compiler infrastructure,” in *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, 2018, pp. 250–256.
- [42] “Operators.” [Online]. Available: <https://docs.scala-lang.org/tour/operators.html>
- [43] Y. Jiang, G. Xue, Z. Jia, and J. You, “Dtuples: A distributed hash table based tuple space service for distributed coordination,” in *2006 Fifth International Conference on Grid and Cooperative Computing (GCC’06)*, 2006, pp. 101–106.
- [44] V. Buravlev, R. De Nicola, and C. A. Mezzina, “Tuple spaces implementations and their efficiency,” in *Coordination Models and Languages*, A. Lluch Lafuente and J. Proença, Eds. Cham: Springer International Publishing, 2016, pp. 51–66.
- [45] L. V. Kalé and A. Sinha, “Information sharing mechanisms in parallel programs,” in *Proceedings of the 8th International Symposium on Parallel Processing*. USA: IEEE Computer Society, 1994, p. 461–468.

- [46] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013. [Online]. Available: <https://books.google.com/books?id=Pg9QDwAAQBAJ>
- [47] “Basilisk c.” [Online]. Available: <http://basilisk.fr/Basilisk%20C>
- [48] S. Omlin, L. Räss, G. Kwasniewski, B. Malvoisin, and Y. Y. Podladchikov, “ParallelStencil.jl,” 2021. [Online]. Available: <https://github.com/omlins/ParallelStencil.jl>
- [49] M. Haines, D. Cronk, and P. Mehrotra, “On the design of chant: a talking threads package,” in *Supercomputing '94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, 1994, pp. 350–359.
- [50] P. Miller, A. Becker, and L. Kalé, “Using shared arrays in message-driven parallel programs,” *Parallel Computing*, vol. 38, no. 1, pp. 66–74, 2012, extensions for Next-Generation Parallel Programming Models. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167819111001360>
- [51] D. F. Bacon, S. L. Graham, and O. J. Sharp, “Compiler transformations for high-performance computing,” *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, 1994.
- [52] S. Klabnik and C. Nichols, *The rust programming language*. No Starch Press, 2019.
- [53] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, and T. Zhang, “Hpx - the c++ standard library for parallelism and concurrency,” *Journal of Open Source Software*, vol. 5, no. 53, p. 2352, 2020. [Online]. Available: <https://doi.org/10.21105/joss.02352>
- [54] N. Deppe, W. Throwe, L. E. Kidder, N. L. Vu, F. Hébert, J. Moxon, C. Armaza, G. S. Bonilla, Y. Kim, P. Kumar, G. Lovelace, A. Macedo, K. C. Nelli, E. O’Shea, H. P. Pfeiffer, M. A. Scheel, S. A. Teukolsky, N. A. Wittek et al., “SpECTRE v2022.03.07,” 10.5281/zenodo.6335350, 3 2022. [Online]. Available: <https://spectre-code.org>
- [55] M. A. Bhandarkar, “Charisma: A component architecture for parallel programming,” Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2002.
- [56] G. Zheng, L. V. Kalé, and O. S. Lawlor, “Multiple flows of control in migratable parallel programs,” in *2006 International Conference on Parallel Processing Workshops (ICPP Workshops 2006), 14-18 August 2006, Columbus, Ohio, USA*. IEEE Computer Society, 2006. [Online]. Available: <https://doi.org/10.1109/ICPPW.2006.58> pp. 435–444.
- [57] J. Choi, D. F. Richards, and L. V. Kale, “Achieving computation-communication overlap with overdecomposition on gpu systems,” in *2020 IEEE/ACM 5th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2020, pp. 1–10.

- [58] L. Wesolowski, R. Venkataraman, A. Gupta, J.-S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale, “TRAM: Optimizing Fine-grained Communication with Topological Routing and Aggregation of Messages,” in *Proceedings of the International Conference on Parallel Processing*, ser. ICPP ’14, Minneapolis, MN, September 2014.
- [59] “Chapel release notes, 1.25.1/1.26.0: Ongoing efforts.” [Online]. Available: <https://chapel-lang.org/releaseNotes/1.26/06-ongoing.pdf>
- [60] L. Oden, “Lessons learned from comparing c-cuda and python-numba for gpu-computing,” in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2020, pp. 216–223.

APPENDIX A: EIR CONSTRUCT LISTING

This is a high-level list of EIR’s constructs (or nodes). It is incomplete (e.g., does not include helper and type constructs), merely aiming to paint a picture of which constructs are available “out of the box” to language authors hoping to use EIR. Note that we aspire to consolidate some of these constructs in future versions of EIR via canonicalization. For example, we might remove `EirArrayReference`, `EirBinaryExpression`, and `EirUnaryExpression` since they are syntactic sugar for function calls. Additionally, we might replace instances of `EirFunction` with `EirDeclaration` nodes whose values are `EirLambdaExpression` nodes. We hope that these changes would lead to more consistent compilation since it is difficult to guarantee uniform behavior across these different options.

- **Top-level constructs:**

- `EirNamespace`—A named grouping of top-level constructs.
- `EirDeclaration`—A variable or named value declaration.
- `EirFunction`—An block of code that runs when invoked.
- `EirClass`—An abstract or concrete “blueprint” for objects.
- `EirTrait`—A purely-abstract set of shared object “traits.”
- `EirImport`—Makes entities from other namespaces accessible.

- **Function-level constructs (i.e., statements):**

- `EirBlock`—A grouping of function-level constructs.
- `EirAssignment`—Updates the value of a variable.
- `EirReturn`—Returns a value from within a function.
- `EirForLoop`—Represents C-style or C++-style range-based loops.
- `EirWhileLoop`—Loops “while” a condition is met.
- `EirIfElse`—Executes a statement when a condition is un/met.

- **Value-level constructs (i.e., expressions):**

- `EirSymbol`—References a named construct.
- `EirScopedSymbol`—References a named construct within a particular scope.
- `EirSpecializedSymbol`—References a particular (type-)specialization of a named construct.

- `EirNew`—Allocates and initializes an instance of a concrete class.
- `EirMatch`—Selects a value based on pattern matching.
- `EirLiteral`—A value of a given type (e.g., `1:int`).
- `EirFunctionCall`—Invokes a function with the given arguments.
- `EirArrayReference`—Subscripts an array (syntactic sugar for get/set).
- `EirTupleExpression`—Groups expression(s) within a tuple.
- `EirTernaryOperator`—Selects a value based on a condition.
- `EirLambdaExpression`—An anonymous function.
- `EirUnaryExpression`—Invokes a “unary” operator on a target (syntactic sugar for prefix/suffix function call).
- `EirBinaryExpression`—Invokes a “binary” operator on left-hand value with right-hand value as its argument (syntactic sugar for a function call).
- `EirInterpolatedString`—A string with one or more “placeholder” expressions.

APPENDIX B: EIR INCORPORATION AND INTEGRATION

B.1 USING HYPERCOMM

Hypercomm is a research package that was organically co-developed with EIR. As previously mentioned, it is presently best suited for use by generated code; it may be awkward for applications to use it directly. Additionally, bear in mind its long, SFINAE-induced compilation times. If these caveats do not deter you, you will find a library with rich features at your disposal. Hypercomm has advanced de/serialization facilities (ser/des) with an interface like Charm++’s PUP, implementations of the components and taskgroups models, and more (futures, threading, etc.). These modules have minimal standalone documentation—their example(s) and header-files are the best reference materials.

B.2 USING EIR

Standalone languages targeting EIR and custom syntax for (Ergoline) EDSLs should consult the Ergoline parser. Both may require bespoke analysis and transformation passes and tweaks to code generation; please refer to other EDSLs for examples. Most EDSLs need only update the main C++ code generator to call their custom code generation routines upon encountering correctly-annotated functions. We recommend only customizing code generation after exhausting all other options; it would be better to transform your EDSL into preexisting ones or “vanilla” EIR to benefit from code reuse. Notably, generating SDAG nodes is the most straightforward way to produce Hypercomm components and segmented entry methods; however, all lowering to SDAG must occur before its segmentation analysis (so the results are available later).

B.3 MIGRATING EIR AND HYPERCOMM

Since EIR generates a combination of Charmxi and Hypercomm-utilizing C++ code, migrating to another (adaptive) RTS would be non-trivial at this time. Indeed, one would effectively have to reimplement all the code generation for entry methods and chares; however, this may be straightforward if the target RTS already has registration capabilities comparable to Charmxi’s (e.g., HPX’s actions and components). Hypercomm is the more significant concern since it utilizes a mix of Converse and Charm++ features in its implementation,

not known to be replicated in any other ecosystem. To that end—it would be best to focus on implementing chares and Hypercomm-like components; their API is not Charm-specific in and of itself.

Lastly, targeting a different host language (besides C++) would be challenging. There are many areas where EDSLs generate custom C++ code, and the C++ code generation backend is substantial. The best course of action would be refactoring EIR to generate a mid-level IR that may be lowered to other languages—generally consistent with foreseeable goals for the project.

B.4 ENHANCING INTEROPERABILITY

Toward making EIR a more comprehensive ecosystem, we recommend enabling EIR programs to import Charmxi, C++, and Python modules. Each path would require parsing the imported modules and generating EIR stubs for type-checking, etc. Charmxi would, ostensibly, be the most “bang for the buck” of these options, requiring little effort besides sourcing a Scala-compatible Charmxi parser (like the one written for Scala Charisma). Automatic interoperation with C++ and Python would be non-trivial, likely involving a mix of open-source packages and compile-time/run-time interoperation.

APPENDIX C: ETYMOLOGY

In this blurb, we briefly clarify the sources of our names; some folks were curious about them in earlier stages of our work. Charm++ derives its name from *chare*, a Middle English word for a task or chore. Likewise, the Erg Intermediate Representation, EIR, derives its name from *erg*, an Ancient Greek inspired term that refers to a unit of work. We considered the name Ergolang for our language before, vaguely inspired by Halide, settling on Ergoline. Here, we sought to convey an ergonomic programming experience – or one that is efficient and comfortable. Lastly, in *Star Wars*, a Hypercomm is a hyperspace-capable communicator; consider it our “vader” (an OpenMPI byte transport layer).