INCOMPLETENESS + INTEROPERABILITY: A MULTI-PARADIGM APPROACH TO PARALLEL PROGRAMMING FOR SCIENCE AND ENGINEERING APPLICATIONS

BY

PRITISH JETLEY

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 at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor Vikram S. Adve
Professor Samuel Kamin
Dr Vijay A. Saraswat, IBM Research.

ABSTRACT

We discuss an object-based, multi-paradigm approach to the development of large-scale, high performance parallel applications. Our approach is characterized by three essential ingredients: (i) *Plurality*, i.e. a programmer decomposes her application into a number of smaller, and relatively independent modules. Each one of these modules is written in the language/framework that allows for its most compact and elegant expression; (ii) *Specialization* of languages, i.e. each language is specialized for the expression of a particular and important subclass of parallel programs; and (iii) *Interoperability* between paradigms, which is to say that modules written in different languages and frameworks can actively interoperate. We believe that language specialization engenders productivity, since it allows programmers to employ higher-level abstractions that are closely attuned to the semantics of an intended domain. Specialization also affords performance benefits, since the designers of the runtime system can make assumptions about the dynamic behaviors of programs, and optimize for these behaviors. Finally, interoperability is a core requirement of the system, and allows it to achieve *completeness* of expression simultaneously with high-level specification in abstract notations.

As a proof of concept, we develop three specialized programming languages. Each one addresses an important subclass of computational patterns encountered in scientific and engineering applications. The first of these, Charisma, captures parallel programs with fixed communication pat-

terns that can be determined by static analysis. The second, *DivCon*, allows the succinct expression of *divide-and-conquer* applications, especially those that exhibit *generative* recursion on distributed collections of data elements. The third, *Distree*, is a flexible framework for the expression of iterative, tree-based algorithms.

We assume the presence of a common programming substrate on top of which translated specifications of these languages execute. In our work, we utilize the Charm++ [1] adaptive runtime system (ARTS) for this purpose. The Charm++ ARTS is based on a coarse-grained, message-driven *actor* [2] model. There are typically tens of such coarse-grained actors per processing element. In the terminology of Charm++, these actors are simply called *objects*. The co-location of multiple objects enables run-time optimizations such as automated overlap of computation on one object, with the communication latency of another, and migration-based dynamic load balancing. This leads to good performance of our translated codes.

The runtime system provides an object-based, message-driven substrate through which our specialized languages can actively interoperate. We shall see that the models of computation provided by *Charisma* and *Distree* are well-aligned with this object-based substrate.

Even though the *DivCon* language provides a mixture of imperative and functional semantics, it is ultimately translated into the interactions of coarse-grained, message-driven objects. This means that our three mini-languages can interoperate with each other, and also with Charm++. In addition, the message-driven nature of Charm++ allows the implicit transfer of control and data between modules. Multi-module programs are therefore automatically interleaved based on the availability of data. In fact, idle time in one module is automatically overlapped with useful work in another. Therefore, interoperability across different views of data and control is not only possible, but also efficient.

We believe that this combination of abstract specification, interoperation between modules, and an object-based view of parallelism backed by an adaptive runtime system affords our approach significant productivity and performance benefits. We substantiate our claims through discussions along the following themes. (i) We present the syntactic and semantic constructs of our specialized languages. We demonstrate their simplicity and the semantic consonance between the constructs provided by each language, and

the characteristics of programs that fall within its range of expression. (ii) We consider the expression of several common and important examples of HPC applications in these specialized languages. As we shall see, the specifications of these applications in our specialized languages are succinct and abstract away details such as the schedule of computation. (iii) We provide performance comparisons between hand-tuned codes and their counterparts written in the high-productivity programming systems. (iv) Finally, we identify and overcome the challenges in enabling interoperability between modules expressed in different paradigms. Support for interoperability allows the composition of large parallel applications from productively-expressed modules, without sacrificing performance. We will demonstrate this through a *Barnes-Hut* application that is composed from pieces of code written in *Charisma*, *DivCon*, *Distree* and Charm++.

ACKNOWLEDGMENTS

The members of my thesis committee have been instrumental in shaping the ideas of this work. I owe them many thanks for their patience, and their thorough review of the thesis. My adviser, Prof. Kale, has been especially supportive of me through my tenure as a graduate student. I thank him deeply for all his help over the years, and for allowing me the time and space to explore new ideas.

I would like to thank my colleagues at the Parallel Programming Laboratory. I have been inspired by their creativity and brilliance, and wish them the best of luck in their endeavors. My friends, few and loved, I thank you too for the pleasure of your company. You made life infinitely more interesting in the otherwise quiet hamlet of Champaign.

My family has been a source of comfort and motivation for me, especially in times of tumult. I thank my mother and father for all the sacrifices that they have made over the years in order that their sons could make something of themselves in this world. Had it not been for their tireless support, I would not be here. My brother has been a constant support for me from afar. I know that a sparkling future in anaesthesiology lies ahead of him.

One saves the best for last. I was fortunate to have met Mrinalini as a graduate student, and for that I shall always thank the forces that colluded to make it so. We have weathered many winters together, Mrinalini. From our self-inflicted bouts of vegetarianism, to our frequent raids on the Golden

Harbor, punctuated by heated debates on the nature of consciousness, I shall forever look upon our time in Champaign with warmth and fondness. It was only through your belief in me that I found myself. Without you, none of this would have been possible.

TABLE OF CONTENTS

CHAPTER

1

INTRODUCTION

The modern parallel programming language must strike the right balance between two opposing forces: performance and productivity. It is reasonable to take the stance that the whole point of parallel programming is to extract every last FLOP of performance from the machine on which one's code is executed. Indeed, currently dominant models of programming, such as message-passing, provide a means to optimize every aspect of a parallel programs execution. However, the process of performance optimization is intensive both in terms of effort expended and expertise required. The programmer must carefully consider a multitude of effects beyond algorithmic and data structure choices in optimizing her program. For instance, an appropriate grain size must be chosen, so as to balance concurrency and parallel overhead; work must be balanced across the set of processors on which the program is running; and, in multiphysics applications, parallel modules must be scheduled over the multi-processor machine, either by partitioning it among different modules, or by time-multiplexing the processors over the various modules. We think that these considerations place an unreasonable burden on the programmer.

1.1 Productivity through abstractions

We take the viewpoint that a programming language aimed at the expression of parallelism must strike a good balance between productivity (through the provision of an ample set of *high-level* constructs for parallelism and synchronization) and performance (by incorporating runtime strategies for dynamic optimization of programs).

As to the viability of an approach based on the abstract specification of parallelism, we appeal to the work of others who have argued for it far more eloquently: languages such as X10 [3] and Chapel [4] have reimagined the design of constructs for parallel programming. For instance, X10 combines primitives for the generation of task (`async`) and data parallel work (`ateach`) with constructs for locality (*places*) synchronization (*futures*, *clocks*, `when`, `finish`), and concurrency management (`atomic`). The language boasts a comprehensive repertoire of object oriented constructs, and appears in a familiar, Java/Scala-like syntax. The Chapel language provides a similar family of constructs for explicit task (`begin`, `cobegin`) parallelism, data parallelism (`forall`, `coforall`), and synchronization (`sync`). Importantly, both of these *Asynchronous, Partitioned Global Address Space* (APGAS) languages provide a departure from the SPMD style of programming, by espousing a global view of control flow, and data structures. The previous generation of *Partitioned Global Address Space* (PGAS) languages, namely Coarray Fortran [5] and UPC [6] formulated the notion of a *partitioned* global address space, thereby combining the productivity benefits of implicit, shared-memory communication with the performance benefits of programmer-specified data locality. UPC provides a host of efficient, one-sided bulk-communication operations (together with synchronization primitives such as *fences*) for performance. However, with the possible exception of UPC, few large-scale scientific and engineering HPC codes have been developed in any of these languages.

1.2 Performance and productivity through adaptive run-time strategies

The Charm++ [1] programming language addresses the forces of productivity and performance by combining a simple, asynchronous message-driven

exection model, with an adaptive, measurement-based runtime system. The Charm++ adaptive runtime system forms the bedrock for a number of scalable parallel applications, including molecular dynamics (NAMD [7]), computational cosmology (ChaNGa [8]), quantum chemistry (OpenAtom [9]), and stochastic optimization [10]. The runtime system continuously measures performance metrics such as task load and communication patterns, to perform dynamic optimizations such as load balancing, communication agglomeration and adaptive overlap of computation with communication. The availability of this infrastructure is a productivity benefit to the programmer as well, since it needn't be reinvented in every application.

The Charm++ message-driven model allows for the transparent composition of different modules, wherein tasks execute as the data on which they depend become available: There is no need for time- or space-division of the underlying parallel machine. However, a criticism of the message-driven model of Charm++ is that it fosters a *reactive* style of programming, in which it is difficult to discern the global flow of control from the local specifications of tasks.

1.3 This thesis

The present work aims to combine the two broad themes above, namely of improving programming productivity through abstract constructs, and of improving application performance using an ARTS that incorporates a host of dynamic strategies for optimization. We restrict our scope of applications to high performance, scientific and engineering codes. This allows us to exploit the regularity inherent in our domain of applications.

Asanovic *et al.* [11] have organized applications in computational science and engineering into categories based on their patterns of computation and communication. This categorization has an immediate consequence for designers of programming models for the HPC community. In this thesis, we aim to exploit this structural regularity to provide a modular, productive and performance-oriented means of constructing parallel HPC applications. We will identify three application classes that are of interest to the HPC community, and explore the formulation of a *specialized* programming paradigm for each one. We target the following classes of parallel program by developing

specialized languages, listed in parentheses:

1. Programs with data-independent communication patterns. (*Charisma.*)

2. Generatively recursive programs. (*DivCon.*)

3. Programs based on distributed trees. (*Distree.*)

Specifically, our aim is to design a *set* of higher-level programming languages that allow for the productive and performance-oriented expression of HPC applications. We achieve abstraction of language constructs through their *specialization* for particular domains of interest. Each language provides high-level constructs that are well-suited for the expression of a sub-class of all possible HPC applications. Given this specialization of languages, it is often cumbersome, and sometimes impossible, to express a certain kind of HPC application (or a module thereof) in a given language. In such an event, we expect that the programmer will either identify a different specialized language in which that application or module is expressed more elegantly, or that she will instead phrase that part of the application in a so-called *recourse* language. In the case of this thesis, the recourse language would be Charm++, which provides a general-purpose programming model. One can see that *interoperability* between modules expressed in different languages is then a key requirement of any (and in particular our own) multi-paradigm programming system. Our work allows for the tight coupling of modules through through compiler intervention. **In summary, we explore the following idea: that given the inherent structural regularity of HPC applications, the principle of language specialization fosters their productive and performance-oriented expression.**

We take an *empirical* approach to the design of languages. The key ingredients of our approach are: (i) Synthesis of useful language constructs from the requirements of *common* and *important* classes of HPC applications; (ii) An *explicitly parallel*, *object-based* expression of algorithms; (iii) *Plurality* of expression, by which a programmer decomposes her application into a number of smaller, and relatively independent modules. (iv) *Specialization* of languages, namely each language is specialized for the expression of a particular and important subclass of parallel programs; and (v) *Interoperability* between modules written in different languages and frameworks, which is to

say that modules written in different languages don't just coexist, but actively interoperate. Therefore the functionality of the program emerges from the interactions of parallel modules written in different languages.

We advocate *specialization* of notations for the same reason as the domain-specific language community, namely that it allows the language designer to provide programming constructs that bridge the semantic gap between the application domain and the abstract syntax of the program that represents the application [12]. *Plurality* of expression allows for different types of program, or different parts of a single program, to be captured in different programming languages. Each program (or module thereof) can then be written in the language that best captures its structure. The price of language specialization, however, is that of completeness of expression. That is, not all types of parallel computation can be expressed in any one of these languages. Indeed, we expect the programmer to identify the language best suited for the expression of each application (or module thereof). If none of the specialized paradigms suffices, the programmer must express the computation in a *recourse* programming paradigm, that is general purpose in nature. In this thesis, we employ Charm++ as the recourse language. Our work enables interoperability between such differently expressed parallel modules.

Whereas we have outlined a rather general multi-paradigm approach, this thesis focuses on the development of three specific specialized programming languages. Our work provides for interoperability between modules written in these languages, as well as modules written in Charm++. Each of the specialized paradigms, as well as the recourse language (Charm++), is based on the *object-based, message-driven* execution model. This allows module execution to be automatically interleaved, based on availability of data for each module.

We demonstrate the feasibility of this multi-paradigm and multilingual approach by developing an HPC application that exhibits highly irregular patterns of computation and communication. We develop a *Barnes-Hut* code that comprises modules written in *Charisma*, *DivCon*, *Distree*, and Charm++. As we argue later, such interoperability is made easier by the presence of a message-driven execution substrate in the form of Charm++.

This is not the first work to attempt the development of a multi-paradigm system for programming. Consider the Oz [13] multi-paradigm programming language, which combines elements of functional programming, imperative

programming and message passing, and constitutes an arguably more elegant approach to programming than the present work. However, Oz programs are slow, and the underlying runtime system, named Mozart [14], seems better suited for the development of loosely-coupled distributed systems, than for the development of tightly-coupled, high performance codes for scientific and engineering applications. This points to a fundamental difference between our approach and that taken by the developers of Oz/Mozart. We view our work as an exercise in pragmatic design: we intend to develop *specialized* constructs that are shaped by the tradeoffs between abstraction, ease of translation into lower-level notations and, most importantly, high performance.

Specialization of programming notations has been extensively investigated prior to this thesis. The most recent example of such an effort is the Stanford Delite project [15]. Indeed, that group has had notable successes in the design of specialized notations for machine learning [16], convex optimization and graph analysis. Unlike the present work, the Delite project provides an elegant means of embedding domain specific languages within the syntactic framework of the Scala language. And whereas the code generated by the Delite compiler targets heterogeneous systems, it is currently restricted to shared memory platforms augmented with accelerator hardware. Therefore the present work is distinct from the Delite project in terms of application domain and scope of execution. Further, we seek specialization on the basis of interaction patterns, rather than application domain. Thus, we hope that our abstractions are more broadly applicable.

CHAPTER

2

PRELIMINARIES

2.1 The role of Charm++ in our work

Although alternative implementations are no doubt possible, our decision to base our specialized notations on the object-based programming model of Charm++ was strategic. We believe that our design allows the programmer to leverage a number features built into the adaptive runtime system of Charm++. These include: migratability (over the network, or even to disk during checkpointing) of objects; a dynamic instrumentation infrastructure that enables the balancing of computational load across PEs at run time; a message-driven model of computation, which automatically engenders overlap of communication and computation; support for fault tolerance, etc. The Charm++ runtime system has been ported to a wide range of architectures, ranging from multicore processors [17], to clusters of SMPs [18], heterogeneous clusters [19–21], and some of the largest supercomputers operational today [22–24].

Our strategy is to translate higher-level notations developed in this thesis, into the *message-driven, coarse-grained objects* paradigm of Charm++. In

this chapter, we provide an overview of the Charm++ execution model. The chief features of this model are:

1. Explicit, programmer-identified parallelism.

2. Expression of computation in terms of coarse-grained, communicating objects whose behavior is best modeled by *actors*.

3. Cooperative concurrency, achieved through non-preemptible, message-driven function invocations.

To illustrate the features of this model concretely, we conduct the following discussion in the context of a simple example application written in Charm++. The application performs a Jacobi relaxation over a two-dimensional domain.

2.2 Explicit decomposition of work and data over coarse-grained *chares*

The programmer explicitly decomposes the data and parallel work over coarse-grained objects called *chares* [1] [25]. Pre-declared *entry* methods can be remotely invoked on such objects, through appropriate handles. The granularity of these invocations is an important consideration. Chares should be such that each entry method invocation leads, on the average, to a reasonable amount of sequential computation. In practical terms, this constrains the HPC programmer to assign a *chunk* of work or data to each chare, and, typically, on the order of tens of chares to each processing element (abbreviated *PE*: a processor core, a thread, etc.) A trade-off must be struck between concurrency and overhead: To increase concurrency, chares must be made as finely-grained as possible. However, they must encapsulate enough work to amortize the parallel overheads of scheduling, communication, and dynamic instrumentation.

If one were writing a parallel prefix sum computation over integers in Charm++, one would assign to each chare a reasonably sized subrange of the input array of integers, and not a single integer. If, on aggregate, too

[1] The word *chare* is Old English for *chore*, or *task*.

few integers are assigned to each chare, the overhead of parallel method invocation can become significant.

Chares can be organized as indexible collections called *chare arrays*. The chare array construct gives the ability to create a handle to an object from anywhere in the system, as long as its index within the array is known. This obviates the communication of handles for individual objects through some external protocol.

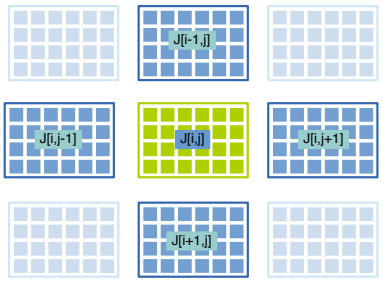
In the Jacobi relaxation, computation is explicitly decomposed over a chare array. Each chare encapsulates a tile of several points from the two-dimensional structured grid over which we perform the relaxation. A chare is mapped to a PE by the runtime system, although the programmer may intervene in the placement decisions. We will often refer to chares as *residing* on PEs. However, a PE is rarely part of the Charm++ programmer's ontology, so we shall avoid the use of the term to the extent possible.

Figure 2.1(a) shows the schematic of a two-dimensional, tiled decomposition of the relaxation domain over a two-dimensional chare array. Let us use a subscript notation to refer to chares based on the geometric layout of their corresponding tiles over the domain: given a collection of tiles J, expression $J[i, j]$ refers to the tile in the i-th row and j-th column. For convenience, we identify the tile at position (i, j), as well as the coarse-grained chare/object that encapsulates it, with the label $J[i, j]$. Figure 2.1(b) shows the assignment of chares to PEs.

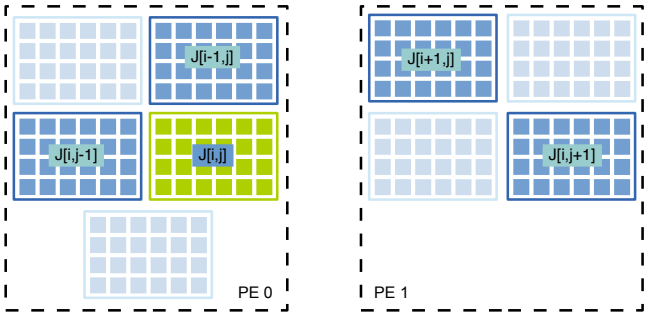
Expressing communication between chares

In Charm++, a message-send is realized by the remote invocation of a method on a target chare. The receipt of a message by a chare triggers some computation on it. Messages encapsulate data from possibly remote sources, and are passed to chare code via *entry method* invocations. For the purpose of this thesis, remote method invocation is assumed to be the only means of communication between objects in the Charm++ model. In particular, the model assumes that chares do not use shared memory for communication. Therefore, the programmer is forced to incorporate the distributed memory cost model into the design of algorithms.

For example, in the Jacobi relaxation algorithm, each chare $J[i, j]$ sends messages to its *neighboring* chares, $J[i + 1, j]$, $J[i - 1, j]$, $J[i, j + 1]$ and



(a) Decomposition.



(b) Mapping.

Figure 2.1: Decomposition of work and data onto an indexible collection of chares: In (a) the Jacobi relaxation problem is decomposed onto a two-dimensional chare array. Chare $J[i, j]$ and its neighbors are shown in bold shading. Each chare contains a *tile* of points from the original relaxation domain. A chare performs the relaxation computation over the tile of points that it contains. Hence, chares are *coarse* in the amount of computation they perform. In (b) is shown the mapping of these chares to two PEs. Note that several, coarse-grained chares reside on each PE.

$J[i, j-1]$. This is shown in Figure 2.2.

To clarify the vocabulary further, a chare (the message source) can invoke an entry method on (i.e. send a message to) another chare (the target of the message) through a handle. Specifically, the method is invoked on a *representative* (a *stub* in Java's RMI, and a *proxy* in CORBA) (non-chare) C++ object called the *proxy* of the target chare. Typically, the proxy of each chare is available (or can be constructed) on any PE. Therefore, in order to invoke entry method `leftBoundary` on its right neighbor chare, chare $J[i, j]$ invokes the the corresponding method on its right neighbor's proxy. Similarly for the other three neighbors of a chare identified by the indices '`i`' and '`j`':

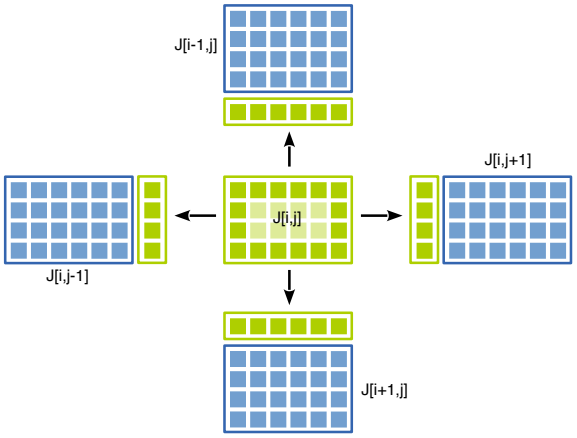


Figure 2.2: Communication of tile boundaries via messages in Charm++. Each chare sends four boundary messages, one to each of its four neighbors, and in turn expects a message from each one of them. A boundary message encapsulates several data elements copied from an appropriate location in the tile of its source chare.

```
JProxy[i, j+1].leftBoundary(makeRightBoundaryMessage());
JProxy[i, j-1].rightBoundary(makeLeftBoundaryMessage());
JProxy[i+1, j].topBoundary(makeBottomBoundaryMessage());
JProxy[i-1, j].bottomBoundary(makeTopBoundaryMessage());
```

Upon invocation, an entry method performs certain computations on the state encapsulated by the target chare. As shown in the code below, the receipt of each boundary message is followed by a check to see whether boundaries from all four neighbors have been received.

```
class Jacobi {
  public:
  void startComputation(){
    int i = thisIndex.x;
    int j = thisIndex.y;
    JProxy[i, j+1].leftBoundary(makeRightBoundaryMessage());
    JProxy[i, j-1].rightBoundary(makeLeftBoundaryMessage());
    JProxy[i+1, j].topBoundary(makeBottomBoundaryMessage());
    JProxy[i-1, j].bottomBoundary(makeTopBoundaryMessage());
  }

  void leftBoundary(BoundaryMsg *msg){
    copyToLeftBoundary(msg->getData());
    numBoundariesReceived_++;
    checkAllBoundariesReceived();
  }

  void rightBoundary(BoundaryMsg *msg){
    copyToRightBoundary(msg->getData());
    numBoundariesReceived_++;
    checkAllBoundariesReceived();
  }

  ...

  void checkAllBoundariesReceived(){
    if(numBoundariesReceived == 4){
      relax();
      numBoundariesReceived = 0;
    }
  }
};
```

Message-driven execution naturally leads to a *reactive* style of programming, wherein chares react to received messages by performing certain actions. This causes control flow in Charm++ programs to become *fragmented* over individual entry methods.

As shown in the `checkAllBoundariesReceived()` method above, and as

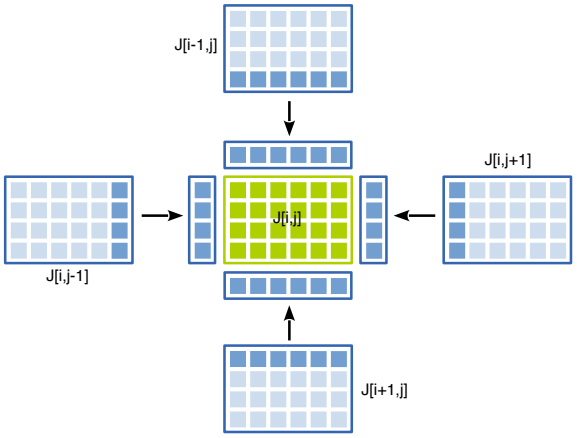


Figure 2.3: A chare receives one boundary message from each one of its neighbors, and performs the local relaxation computation using the elements contained therein.

depicted in Figure 2.3, upon receiving messages from all of its neighbors, a Jacobi chare performs a coarse-grained computation, namely the stencil-based relaxation operation on all the grid points within its tile. Such an invocation is *non-preemptible*, and executes to completion. When an entry method invocation finishes, control is transferred to a *scheduler*, which picks a message from a *queue* of remotely received messages. Therefore, chares are cooperatively scheduled. Each message specifies its target chare, as well as the entry method to be invoked upon its receipt. The message is passed as a parameter to the corresponding entry method invocation.

2.3 The role of the runtime system

The Charm++ system combines the idea of message-driven objects, with an adaptive runtime system that dynamically optimizes program performance. In particular, Charm++ incorporates a dynamic instrumentation module, which records the amount of computational load exerted by each chare in the program, as well as the communication pattern of each chare. This information is then funneled to other dynamic strategies to alter the mapping of chares to PEs at run-time. In this way, chares can be migrated to balance load. The Charm++ infrastructure also provides other amenities, such as a dynamic communication optimization library [26, 27], run-time tuning

of program parameters [28] quiescence detection algorithms [29] and fault tolerance [30, 31]. However, a substantive discussion of these aspects of the Charm++ system is beyond the scope of this thesis.

2.4 Modular composition

Typically, multi-module MPI codes employ either the temporal division of the parallel machine on which the job is executed, or its spatial division among the modules. In the first approach, one module occupies the entire machine for a certain duration (measured perhaps in time steps of execution), and then transfers control to the other module, which then occupies the entire machine for some time, and so on. The problem with this technique is that the idle time suffered by one of the modules (for instance, due to communication latency) cannot be overlapped by useful work in the other module. An alternative is to spatially partition the machine among the modules, whereby modules execute in parallel, but on disjoint sets of processors. This technique has two drawbacks, namely (i) The need to determine *a priori* the appropriate ratio in which to divide the machine, and (ii) Unnecessary communication between corresponding subdomains in different modules These subdomains simulate different physics on the same region, but are forced to communiate because of their artificial separation by a module boundary.

A third alternative involves the intrusive rewriting of communication code in each of the modules, so as to insert wildcard MPI receives (instead of module-specific ones), and thereafter funnel each intercepted message to the appropriate module, based on its identifying tag. This allows the interleaved execution of different modules in each MPI process.

By contrast, efficient composition of modules comes naturally in the message-driven model of Charm++. Chares of several modules reside on each PE; entry methods are invoked on these chares based on the local availability of data-carrying messages addressed to each one. Therefore, we don't have to artificially separate modules, and idle time incurred by chares of one module on a PE may be dynamically overlapped with useful computation performed by local chares of other modules. Moreover, the Charm++ scheduler automatically handles the addition of new modules without programmer intervention.

2.5 SDAG: Structured control flow for message-driven objects

Given the reactive nature of message-driven chares, Charm++ programs suffer from a fragmentation of control flow local to an object. Therefore, control flow local to a Charm++ object is *unstructured*, analogous to sequential programs that use *gotos* or equivalent control flow constructs. It is natural to want to give some structure to the sequence of events that occur during each chare's lifetime. Work by Kale and Bhandarkar [32] has made it possible to do just this.

Their notation, called *SDAG (Structured DAGger)*, consists of three main components: (i) `when` blocks, (ii) `serial` blocks, and (iii) conditional control flow constructs (e.g. `if`, `for` and `while`). SDAG statements are executed in program order. The `when` construct allows us to specify dependencies on received messages. A `when` block executes when all corresponding message receipt dependencies have been satisfied. A `serial` block encloses a number of C++ statements, and no SDAG code. Finally, control flow constructs allow the modulation of the otherwise sequential flow of control. Using SDAG constructs, one can express the message-driven behavior of Charm++ objects in an unfragmented manner. Details about the notation can be found elsewhere [32].

2.6 System architecture

The work done in this thesis is part of the Charm++ ecosystem, and leverages several of its components in order to achieve the twin goals of productivity of expression, and efficiency of parallel performance. We note the role of each one of these components below, not only to give the reader a complete picture of the context of this work, but also as an acknowledgement of the vast software infrastructure that enables the kind of research directions taken by this thesis. Therefore, in the following, we give a quick overview of the Charm++ software stack, and identify the specific location therein to which we contribute.

Let us start at the bottom of Figure 2.4. The bottom-most layer of Charm++ serves as the interface to the parallel machine on which the pro-

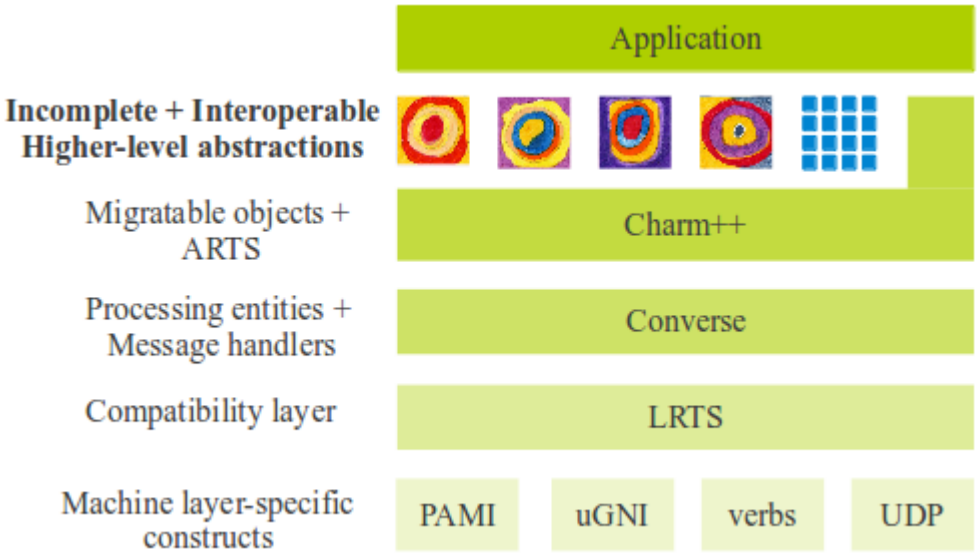


Figure 2.4: Pictorial depiction of the software architecture used in this thesis. The thesis describes work done to provide a multi-paradigm approach based on the idea of incomplete and interoperable abstractions (the *Kandinsky circles*). The programmer creates a multi-module application, each module of which is written in the higher-level, incomplete abstraction that suits it best. These modules are interoperable, allowing the composition of large applications from multiple modules, each of which is relatively self-contained.

gram is being executed. As such, it is called the *machine layer*. Charm++ has been ported to many different platforms, and the peformance of these ports depends to a large degree on the machine-specific optimizations that are incorporated by the corresponding machine layer. A basic machine layer based on UDP sockets has existed since the inception of Charm++, and support for new architectures, e.g. the IBM Blue Gene series [24, 33], the Cray Gemini network (*uGNI*) [23], Infiniband (*verbs*) [34], etc. has been developed over time.

The machine layer is the only one that has a different implementation for each parallel architecture to which Charm++ has been ported. As we traverse up the stack from the machine layer, we encounter layers that provide increasing levels of abstraction, and are (largely) hardware-agnostic. For example, the low-level runtime system, i.e. *LRTS* provides common memory management, user-level threads, and communication routines. The *Converse* [35] layer provides an abstract parallel execution model in the form of asynchronous function invocations on processors, and reactive *message han-*

dler code. The Charm++ layer, which provides migratable, message-driven objects, is present above the Converse layer. It manages the mapping of objects to PEs, as well as their migration across PEs for load balancing. Finally, the Charm++ layer provides various other amenities, such as completion/quiescence detection [36], efficient message aggregation and routing, fault tolerance [37], etc.

The work described in this thesis is based on top of the Charm++ layer. The language that we describe in Chapter 3 has an associated compiler that translates a global, data-independent, imperative specification of control and data flow into a Charm++ program (SDAG) with distributed, message-driven control. Chapter 4 details the translation of divide-and-conquer computations into Charm++ programs consisting of dynamically spawned chares, whose grain size is adaptively managed. We emphasize divide-and-conquer algorithms that operate on large, distributed arrays of data. In this context, each array of data is coarsely distributed over a chare array. In Chapter 5 we describe a framework that is meant to express parallel algorithms based on distributed trees. In that chapter, each tree is coarsely distributed over the elements of a chare array, several of which are present on each PE. The message-driven nature of Charm++ is exploited to foster overlap of computation and communcation.

CHAPTER

3

DATA-INDEPENDENT COMMUNICATION PATTERNS

3.1 Introduction

In their "View from Berkeley" Asanovic *et al.* [11] note that most parallel HPC applications fall into one of a handful of well-defined classes, which are typified by so called computational *dwarves*. From their perspective, such a characterization allows us to evaluate the efficacy of experimental hardware and software platforms against well-defined and commonly used patterns of computation and communication. From the point of view of programming language design, their characterization supports the idea that HPC programmers write programs subscribing to one of a handful of computation/communication patterns.

In this section, we examine the commonalities between an important subset of HPC applications. We intend to identify a collection of *data-independent* interaction patterns that can be supported by a new programming model. We take an empirical approach to the design of this model, and analyze several

common applications in the realm of HPC. Since we address only a subset of HPC applications, the model we develop is *specialized* and *incomplete*.

3.2 An examination of HPC applications for data-independent communication patterns

In this section, we use a simple notation to concisely capture the communication patterns of the parallel algorithms being considered. We will explore the class of communication patterns that can be expressed as a series of statements in this notation.

As in the rest of this thesis, we consider execution on distributed memory machines. We adhere to the explicit and coarse-grained decomposition of work and data over objects (*cf.* § 2). The notation that we use is meant to describe patterns of communication between sets of such objects. Objects communicate with each other by sending (medium-sized) messages.

These messages are identified through labels, each of which consists of a multidimensional subscript. The subscript comprises index expressions whose identifiers are bound by universal quantifiers, as we shall explain shortly. Identifiers within subscripts are *locally bound*. That is, if several messages are generated by a (set of) sender object(s), then we only use the message label to distinguish the set of target objects for each one. In particular, these labels are *not* available to the recipients themselves, and so cannot be used to discriminate between received messages. We place this restriction to simplify the notation and the accompanying discussion of communication patterns. In a real application this restriction would hamper the expression of computation. (For example, if a recipient of several messages operates upon them in a certain order, it must be able to determine the position of each received message in that order.) However, it does not affect our present purpose, which is the expression of *communication* patterns.

To summarize, our notation is used to *coordinate* or *orchestrate* the activities of sets of objects that send and receive messages. In this notation a parallel algorithm can be specified as a series of statements. A statement can be of one of three types, namely *sends-to*, *reduces-to* or *computes*:

1. *sends-to*: indicates that an individual coarse-grained object sends a

message to a set of similar receiver objects. The set of recipients may be a singleton, in which case a point-to-point message is implied. If the set of recipients has more than one member, the communication action is either a *multicast* or a *scatter*. A multicast is denoted by the sending of a single message to a set of recipients, whereas a scatter is implied when a differently labeled message is sent to each of a set of recipients. Finally, if a number of differently labeled messages is sent, and each such message targets a number of recipients, the communication operation is a *scatter-multicast*. In the case of scatters and scatter-multicasts, message labels are used to identify the recipient(s) of each message.

2. *reduces-to*: indicates that a set of objects collectively performs a reduction operation, the result of which is sent to a set of recipients. As in the *sends-to* case, the reduction could have a single recipient, or a set of recipients for the same result (*reduce-multicast*, similar to *all-reduce*). On the other hand, multiple subsets of a set of sources could each perform a reduction, leading to the generation of a corresponding number of reduced results. Each of these results could be a labeled message sent to a different set of recipients. If each labeled result is sent to a singleton object, the operation is a *reduce-scatter*. On the other hand, if the set of recipients for each labeled result is itself a set of objects, then we have a *reduce-scatter-multicast* operation.

3. *computes*: indicates that an object waits for the receipt of a certain number of messages, following which it performs some computational work using data extracted from those messages. As noted previously, the identity of received messages does not matter; only the number of messages awaited is important.

Below we use this notation to examine patterns of communication in several staple scientific and engineering applications.

3.2.1 Jacobi relaxation

A number of applications exhibit the *halo*-exchange pattern inherent in the stencil calculation of the Jacobi method. Examples include Cellular Gas Au-

tomata and the Lattice Boltzmann Method (for fluid mechanics), multiscale methods, and Particle-In-Cell (plasma physics) and Particle-Mesh methods (e.g. in computational cosmology).

Assume that a two-dimensional $N \times N$ toroidal domain has been decomposed onto a one-dimensional collection of N/g objects, J. The parameter g denotes granularity, and is typically chosen so that each of the N/g objects is *coarse* in the number of elements assigned to it. Therefore each $J[i]$, where $i \in S_J = \{0..N/g-1\}$, contains a number (g) of rows of the domain. In order to perform the relaxation operation on its subdomain, it needs the boundary rows of its two adjacent neighbors. Similarly, each of its neighbors gets one of *its* boundaries from the object in question. Therefore, each $J[i]$ sends a message containing its top and bottom boundaries to its adjacent objects, $J[i+1]$ and $J[i-1]$. When it receives boundaries sent by its neighbors, $J[i]$ can perform a relaxation operation on its subdomain.

$$\forall\ i \in S_J : J[i]\ \textbf{sends}\ [0]\ \textbf{to}\ J[(i+1) \bmod N/g]$$

$$\forall\ i \in S_J : J[i]\ \textbf{sends}\ [0]\ \textbf{to}\ J[(i-1+N/g) \bmod N/g]$$

$$\forall\ i \in S_J : J[i]\ \textbf{computes}(2)$$

Such a *halo-exchange* is typical of stencil codes, which are used extensively to solve partial differential equations. In our notation, each of the first two *sends-to* operations identifies a single recipient for a single message, implying a one-to-one communication pattern. The third statement says that each $J[i]$ waits for two messages before performing some computation.

3.2.2 Tiled matrix multiplication

The tiled algorithm for matrix multiplication relies on the existence of efficient section multicasts. Such a pattern is found in more sophisticated versions as well, including the state-of-the art algorithm of Agarwal *et al.* [38].

In our object-based decomposition, the input matrices A and B are decomposed onto one-dimensional object collections also named A and B. Each object $A[i]$ contains the i-th chunk of rows of A; similarly, $B[j]$ contains the j-th chunk of columns of B. The result C is computed by a third, two-dimensional, object collection, C, such that $C[i,j]$ computes the result tile

C_{ij}. In order to do this, it requires the entire i-th chunk of rows and j-th chunk of columns. Let $S_A = S_B = \{0..N/g - 1\}$, where N is the number of rows in the input (square) matrices, and g is the granularity, i.e. the number of rows assigned to each $A[i]$, or the number of columns assigned to each $B[j]$. Also, $S_C = S_A \times S_B$. Then, the matrix multiplication algorithm is:

$$\forall\ i \in S_A : A[i] \textbf{ sends } [0] \textbf{ to } \{\forall\ k \in S_B : C[i,k]\}$$

$$\forall\ j \in S_B : B[j] \textbf{ sends } [0] \textbf{ to } \{\forall\ k \in S_A : C[k,j]\}$$

$$\forall\ i,j \in S_C : C[i,j] \textbf{ computes}(2)$$

Again, we see the use of the singleton subscript [0] to denote a one-to-one communication pattern.

3.2.3 Transpose

The transpose operation involves a scatter followed by a gather. It is found in HPC applications that perform multi-dimensional FFTs and matrix inversion, and is otherwise ubiquitous in linear algebra.

Suppose that we have decomposed an $N \times N$ matrix over a one-dimensional collection of N/g objects, A. Therefore, each $A[i]$ contains a contiguous chunk of rows of the input matrix. In order to perform the transpose, $A[i]$ must send the j-th chunk of columns of its rows, to $A[j]$, where $i, j \in S_A = \{0..N/g-1\}$:

$$\forall\ i,j \in S_A \times S_A : A[i] \textbf{ sends } [j] \textbf{ to } A[j]$$

$$\forall\ i \in S_A : A[i] \textbf{ computes}(N/g)$$

The communication pattern here is a scatter: each $A[i]$ sends a number of messages, where each message is given the label $[j]$ ($j \in S_A$). Each message $[j]$ has a single recipient, $A[j]$. Following this, each $A[i]$ awaits the receipt of N/g messages and then performs some computation.

3.2.4 Decimation-in-time FFT

The FFT is an important part of many scientific and engineering codes: it has applications ranging from signal processing to field-potential calculations

used in cosmology, plasma physics and the like. Below we discuss the *butterfly* FFT algorithm [39], in which recursively-doubled message-exchange distances are used. Let A be the one-dimensional collection of objects over which the data are decomposed, and let $S_A = \{0..N/g - 1\}$, where N is the number of data elements (assumed to be a power of two for simplicity) and g is, once again, the granularity of decomposition, namely the number of data elements assigned to each object $A[i]$, with $i \in S_A$. In iteration k,

$$\forall\ i \in S_A : A[i] \textbf{ sends } 0 \textbf{ to } A[(i + 2^k) \bmod N]$$

$$\forall\ i \in S_A : A[i] \textbf{ computes}(1)$$

3.2.5 Dense LU decomposition

The dense LU decomposition is exemplary of a large class of dense linear algebra routines, each member of which can be expressed using section multicasts and reductions. For a two dimensional decomposition of the $N \times N$ input matrix onto an object collection A, we have the following structure for each iteration. In iteration k, $R_k = \{k + 1 : N - 1\}$, and

$$A[k,k] \textbf{ computes}(0)$$

$$A[k,k] \textbf{ sends } [0] \textbf{ to } \{\forall\ i \in R_k : A[k,i]\}$$

$$A[k,k] \textbf{ sends } [0] \textbf{ to } \{\forall\ i \in R_k : A[i,k]\}$$

$$\forall\ i \in R_k : A[k,i] \textbf{ computes}(1)$$

$$\forall\ i \in R_k : A[i,k] \textbf{ computes}(1)$$

$$\forall\ i \in R_k : A[k,i] \textbf{ sends } i \textbf{ to } \{\forall\ j \in R_k : A[j,i]\}$$

$$\forall\ i \in R_k : A[i,k] \textbf{ sends } i \textbf{ to } \{\forall\ j \in R_k : A[i,j]\}$$

$$\forall\ i,j \in R_k^2 : A[i,j] \textbf{ computes}(2)$$

Briefly, the object on the current element of the diagonal factorizes the diagonal tile (first statement) and has no data dependencies on any previous computation. It multicasts the triangular halves of the factorized tile to the objects in the k-th row to its right (second statement), and the objects in

the k-th column below it (third statement). When a targeted recipient receives a triangular submatrix, it performs a panel update (fourth and fifth statements) and in turn multicasts its result to the objects in the trailing row (sixth statement) or column (seventh statement) of the matrix. These updates are sent as sub-column and sub-row multicasts, such that each trailing tile receives two messages, which it uses to update its portion of the trailing matrix (eighth statement).

3.2.6 Geometric multigrid

Multigrid methods (and the associated family of more general *multiscale* methods) are used to solve certain classes of partial differential equations, and exhibit faster convergence rates than structured unigrid methods (e.g. the Jacobi method). A detailed description is beyond the scope of this document: we refer the interested reader to the book of Briggs *et al.* [40] for the mechanics of the algorithm, and also some of the terminology used below. Briefly, the geometric multigrid method consists of a localized gather operation during the *restriction* phase, a localized scatter operation during *interpolation*, and a halo-exchange during *smoothing*, which is similar to the Jacobi method's relaxation computation.

Assume that the input $N \times N$ domain has been decomposed over a two-dimensional collection of objects A, whose dimensions are $N/g \times N/g$, g being the granularity of decomposition. Therefore, every $A[i, j]$ holds a tile of the input domain. The interpolation phase has the effect of iteratively reducing the number of elements per A-object. To counter this reduction in grain size, a localized *coarsening* operation is performed. Let τ be the coarsening factor, and $k - 1$ be the number of coarsenings that have already occurred. Then $S_k = \{0..N/g\tau^k - 1\}^2$ is the set of indices of the active objects prior to the k-th coarsening. The k-th round of coarsening entails the following gather operation:

$$\forall\ i, j \in S_k : A[i, j] \textbf{ sends } [0] \textbf{ to } A[i/\tau, j/\tau]\}$$

$$\forall\ i, j \in S_{k+1} : A[i, j] \textbf{ computes}((\tau - 1)^2)$$

Similar observations apply to the refinement operations applied during interpolations.

3.2.7 Space- and Force-decomposed molecular dynamics

The work of Phillips, Zheng, Kumar and Kale [41] has resulted in a scalable method for computing the Newtonian interactions of large ensembles of atoms subject to molecular forces. Their algorithm is based on the fine-grained decomposition of data onto so-called *patches* of space, and of work onto *computes*. Each compute calculates the interactions between atoms in a *neighboring* pair of patches.

Expressed in terms of object collections, the two-dimensional domain is evenly distributed among members of the two-dimensional collection of patches, P, and the work of calculating the interactions between neighboring patches $P[i_1, j_1]$ and $P[i_2, j_2]$ is assigned to compute $C[i_1, j_1, i_2, j_2]$. The "neighbors" relation is denoted N, i.e. $N(i_1, j_1, i_2, j_2)$ if $P[i_2, j_2]$ is a neighbor of $P[i_1, j_1]$. Moreover, let $S_P = \{0..N-1\}^2$ be the set of all the indices of patches in P, and $S_C = S_P^2$.

$$\forall\ i_1, j_1 \in S_P : P[i_1, j_1] \textbf{ sends } [0] \textbf{ to } \{\forall\ i_2, j_2 \in S_P | N(i_1, j_1, i_2, j_2) : C[i_1, j_1, i_2, j_2]\}$$

$$\forall\ i_1, j_1 \in S_P : P[i_1, j_1] \textbf{ sends } [0] \textbf{ to } \{\forall\ i_2, j_2 \in S_P | N(i_2, j_2, i_1, j_1) : C[i_2, j_2, i_1, j_1]\}$$

$$\forall\ i_1, j_1, i_2, j_2 \in S_C : C[i_1, j_1, i_2, j_2] \textbf{ computes}(2)$$

$$\forall\, i_1, j_1 \in S_P : \{\forall\, i_2, j_2 \in S_P | N(i_1, j_1, i_2, j_2) : C[i_1, j_1, i_2, j_2]\} \textbf{ reduces } [0] \textbf{ to } P[i_1, j_1]$$

$$\forall\, i_1, j_1 \in S_P : \{\forall\, i_2, j_2 \in S_P | N(i_2, j_2, i_1, j_1) : C[i_1, j_1, i_2, j_2]\} \textbf{ reduces } [0] \textbf{ to } P[i_2, j_2]$$

$$\forall\ i, j \in S_P : P[i, j] \textbf{ computes}(2)$$

In the above, each patch $P[i_1, j_1]$, where $i_1, j_1 \in S_P$, multicasts its particles to those computes that calculate the interaction of its particles with those of its neighbors, namely $P[i_2, j_2]$, such that $i_2, j_2 \in S_P$ and $N(i_1, j_1, i_2, j_2)$ (first statement) or $N(i_2, j_2, i_1, j_1)$ (second statement). Conversely, each compute waits to receive particles from the two patches whose pairwise interactions it is to calculate (third statement). The results of the pairwise force interactions are reduced over corresponding sets of computes (fourth and fifth statements), and sent to patches for integration of particle trajectories (sixth statement).

Discussion

The above algorithms are characterized by the following features:

1. Data and work are decomposed over fixed sets of coarse-grained objects.

2. Communication is sender-driven, i.e. the sender of a message determines the identity of the receiver(s).

3. The computation of the receiver's identity *does not* depend on the contents of any messages previously received by the sender. However, it may depend on the identity of the sender itself, constants, and iteration index variables (as in LU and FFT). Therefore, the communication patterns between objects are independent of the state of the program. This allows a compiler to statically determine all possible communication patterns used in the program.

4. Finally, the receiver of a message does not make any reference to the identity of the sender.

For the purpose of our work, we take these to be the defining characteristics of *data-independent* communication patterns. Using even such a restricted notation, we are able to capture a variety of communication patterns that is prevalent in important domains of HPC applications. It is this observation that motivates the constructs of our higher-level, specialized programming language, called *Charisma*. We develop this language in sections 3.4 and beyond. In addition to constructs for specifying data-independent communication patterns, *Charisma* provides constructs for iterative computations and data-dependent *control* flow.

3.3 Some patterns that are not data-independent

Although the above list of applications makes the case that several important classes of algorithm can be expressed in a data-independent manner, there are several other algorithms that cannot be captured in this style. We enumerate some of these below:

3.3.1 Data-dependent data flows in Barnes-Hut traversals

The Barnes-Hut algorithm [42] provides an asymptotically efficient algorithm for computing the pairwise interactions betwen large particle ensembles. At the heart of this algorithm is the data-dependent *opening criterion*, which is used to determine whether two sets of particles are sufficiently distant from each other for the application of a hierarchical approximation. In this algorithm, whether or not two processors communicate depends strongly on the mass distributions on the two processors. Such data-dependent data flows cannot be captured by our notation.

3.3.2 Data scattering in the Quicksort algorithm

Consider the recursive quicksort algorithm operating over a distributed array. Suppose that the input array A to some invocation in the recursion tree, is evenly distributed over a set of processors P. Then, subsequent to the (out-of-place) partitioning of A about a pivot, we obtain three smaller arrays, A_1, A_2 and A_3. In the "team-parallel" scheme of Hardwick [43], each A_i must be distributed over a partition of P. In order to do this, a parallel prefix operation is performed over P to determine the number of elements of A_i held by each $p \in P$. The result of this parallel prefix operation is used by every $p \in P$ to determine the processors in P_i to which to send data elements of A_i held by p. Although this pattern of communication could be phrased in a data-independent manner, it would be extremely inefficient to do so.

3.3.3 Discrete event simulations

For many discrete event simulations, the communication graph between simulated entities varies as a function of time. In such simulations, the communication pattern depends on the state of the computation, and therefore cannot be effiiciently captured in a data-independent manner. Even for those simulations in which the communication graph remains static, whether or not data are exchanged between connected objects may be dictated by the local state of objects. A good example of such a computation is the simulation of a large neuronal network, in which the spiking of each individual neuron is determined by its internal state, which is in turn determined by the

data that it receives from neurons that are presynaptic to it. Of course, the communication pattern in such a simulation could be characterized as an all-to-all, but, especially for sparse graphs, such an implementation would be inefficient.

3.4 *Charisma*: Stance

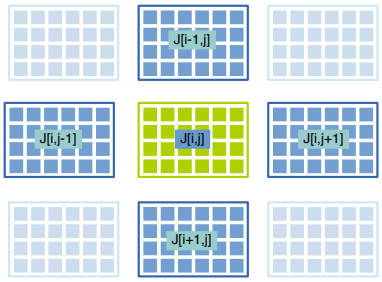
Our observations from § 3.2 bring us to the central focus of this chapter, namely the design of a specialized language for the productive expression of parallel programs with *data-independent* communication patterns. We have already seen that a number of HPC applications can be expressed with a data-independent communication pattern (*cf.* 3.2). We now synthesize from this notation the constructs of a high-level programming language called *Charisma*. Below, we briefly enumerate the features that we believe to be important to its design:

1. An explicitly decomposed description of the parallel program in terms of coarse-grained objects.
2. Separation of the program's serial computation from the specification of its parallel structure, thereby keeping the language for the latter simple, and simultaneously allowing the former to be expressed in one of many, well-established sequential programming languages.
3. An unfragmented representation of *global* control and data flows.
4. A simple and sequential semantics of parallel execution, that guarantees determinism.
5. An intuitive way to specify communication between parallel objects.
6. Incorporation of the automated resource management techniques of Charm++.

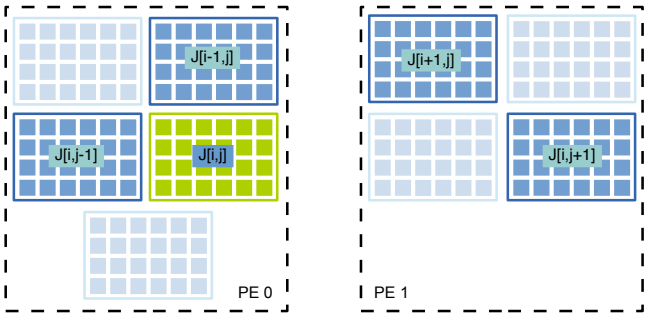
3.5 An example application written in *Charisma*

In order to give some concrete context to the constructs of *Charisma*, we present again the example Jacobi application from § 2. Fragments of *Charisma*

code are presented below, together with brief discussions of the language constructs that we encounter in this application. Having gained an intuitive understanding of Charisma from this example, we will return to a more comprehensive and formal description of the language in § 3.6.



(a) Decomposition.



(b) Mapping.

Figure 3.1: Decomposition of the problem domain over a collection of coarse-grained objects, and the ARTS-managed mapping of these objects to PEs, as in § 2.2.

3.5.1 Coarse grained objects

Similarly to the Charm++ version of Jacobi relaxation, the two dimensional domain is explicitly decomposed by the programmer over rectangular tiles, each of which is encapsulated by a coarse-grained object.

3.5.2 Object collections

The tiled objects are arranged into a named, two-dimensional collection. The objects are of a `class` specified by the programmer, and individuals in the

collection are identified through a two-dimensional index. Shown below is the *Charisma* declaration of the `Jacobi` class and that of a two-dimensional collection of objects of type `Jacobi`, namely `J`:

```
param NX, NY : int;

class Jacobi;
objects J : Jacobi[NX, NY];
```

Objects within collection `J` are depicted as encapsulating several points from the relaxation domain in Figure 3.1 (reproduced from § 2.2 for quick reference). As with Charm++, the mapping of objects onto processors is managed by the runtime system.

3.5.3 Program parameters

The object collection `J` can be thought of as a dense, two-dimensional array comprising `NX` elements along one dimension and `NY` along the other. Symbols `NX` and `NY` are *parameters* to the program, and are obtained extraneously, e.g. via the command line, or during module initialization. So, objects in collection `J` may be addressed as '`J[x,y]`', this expression being valid as long as index `x` evaluates to an integer between `0` and `NX-1`, and `y` is between `0` and `NY-1`, both ends inclusive.

3.5.4 Method invocations encapsulate serial work

The fundamental action prescribed by *Charisma* statements is the invocation of *serial* methods on objects. In describing them as serial, we mean that the methods invoked on objects do not themselves contain *Charisma* code. In fact, they are treated as blocks of serial, C++ code. As outlined below, method invocations may lead to communication between objects.

3.5.5 Communication via *values*

In the Jacobi application, each object performs a relaxation computation on the tile assigned to it. In order to do so, there must be a preceding

step in which communication is performed so as to exchange boundary elements between neighbors. In *Charisma*, communication occurs through the *publication* and *consumption* of *values*. The values published by an object may be thought of as named *messages*, intended for consumption by other objects. This mode of publication and consumption (resulting in the generation of messages) is the only way in which objects can communicate with each other. As an example, it is assumed that objects *do not* communicate implicitly via shared memory. Values have types, and, like objects, may be organized into named, indexed collections. A collection of values is called a *value space*. A value may encapsulate an array of elements of a user-defined type.

The following *Charisma* code declares value spaces `n`, `e`, `w` and `s` of `double` values. Like object declarations, value space declarations specify the range of valid indices along each dimension of the value subscript.

```
values n, e, w, s : double[NX, NY];
```

Below is the *Charisma* declaration of a singleton value '`err`', of user-defined type '`ReducibleDouble`'.

```
type ReducibleDouble;
value err : ReducibleDouble;
```

Publication

In our example, a object `J[x,y]` *publishes* four values (messages), each corresponding to one of its boundaries. These values are named after the four cardinal directions, and each value bears a subscript that is the same as the neighbor of `J[x,y]` to which it must be communicated. Therefore, the invocation of serial method `boundaries` on `J[x,y]` publishes the western boundary of its eastern neighbor (value `w[x+1,y]`), the eastern boundary of its western neighbor (`e[x-1,y]`), and similarly for values `n[x,y-1]` and `s[x,y+1]`:

```
(w[x+1,y], e[x-1,y],
 n[x,y-1], s[x,y+1]) <- J[x,y].boundaries();
```

Publishing values in serial methods

Computations performed by serial methods may generate data that are to be communicated via publication. For this purpose, *Charisma* provides a set of *conduit* classes, which serve as the interface between user-provided serial code, and compiler-generated code corresponding to the orchestration source. An example of such a conduit class is `Charisma::PublishedValue<>`, in the code below. Its purpose is to bind data in serial, C++ code to published values in the orchestration part of the program. The serial code for `boundaries` is as follows:

```
void Jacobi::boundaries(Charisma::PublishedValue<double> &w,
                        Charisma::PublishedValue<double> &e,
                        Charisma::PublishedValue<double> &n,
                        Charisma::PublishedValue<double> &s){
  int stride = nC;
  w.produce(&tile(1,nC), &tile(nR,nC), nR, stride);
  e.produce(&tile(1,1),  &tile(nR,1),  nR, stride);
  n.produce(&tile(nR,1), &tile(nR,nC), nC);
  s.produce(&tile(1,1),  &tile(1,nC),  nC);

}
```

The method is invoked with input arguments of type `PublishedValue`, which correspond to values published by this method in the *Charisma* orchestration code. In the above, `tile` is a member variable of the `Jacobi` class, and represents the subdomain assigned to the instance of the class on which the `boundaries` method is invoked. Similarly, `nC` and `nR` are the number of columns and rows in each tile. By invoking the `produce` method on the formal parameters to the C++ method, the code associates the leftmost column of `tile` with the published western boundary value, the rightmost column with the eastern boundary value, etc. The code demonstrates the use of strided- and non-strided variants of the `produce` method.

Consumption

An object is said to *consume* a published value when that value is named as an input argument to a serial method invoked on that object. The data

associated with a consumed value (which must previously have been published by another method invocation) are then passed into the invoked serial method.

```
J[x,y].relax(n[x,y], s[x,y],
             e[x,y], w[x,y]);
```

In the Jacobi example, each `J[x,y]` consumes the four values published by its neighbors for it, since they are arguments to the `relax` method invoked on it. This leads to the invocation of the `relax` method on `J[x,y]`. Arguments to this method encapsulate the boundaries of its neighbors.

Consuming values in serial methods

The data encapsulated within a consumed *Charisma* value are passed into serial C++ code by means of the `Charisma::ConsumedValue<>` class template. This type of object serves as a conduit for data from orchestration code to serial code. In our example the received boundary data, as encapsulated by `Charisma::ConsumedValue<>`'s, are accessed via the C++ square bracket operator defined for that class. If a consumed value encapsulates an array of data elements, its size can be obtained by calling the `size` method of the corresponding `ConsumedValue`, as shown below for values `e` and `n`:

```
void Jacobi::relax(Charisma::ConsumedValue<double> &n,
                   Charisma::ConsumedValue<double> &s,
                   Charisma::ConsumedValue<double> &e,
                   Charisma::ConsumedValue<double> &w){
  for(int i = 0; i < e.size(); ++i){
    tile(i,0) = w[i];
    tile(i,nC+1) = e[i];
  }
  for(int i = 0; i < n.size(); ++i){
    tile(0,j) = n[i];
    tile(nR+1,j) = s[i];
  }
  for(int i = 1; i <= nR; ++i)
    for(int j = 1; j <= nC; ++j)
      newTile(i,j) = 0.2 * (tile(i,j) + tile(i-1,j) +
                            tile(i,j-1) + tile(i,j+1]);
  swap(tile, newTile);
}
```

The code simply copies the boundaries into the appropriate positions within the `tile` of the object on which `relax` is invoked, and performs the stencil computation on it.

Reduction

In reality, when performing the `relax`ation operation, each object must determine a local error residual. We must find the maximum of this local error residual across all J-objects to check whether the computation has converged. In *Charisma*, published values may be *reduced* across all instances of a publish-consume statement in the following manner:

```
(+err) <- J[x,y].relax(n[x,y], s[x,y],
                       e[x,y], w[x,y]);
```

In the above, we denote `err` to be a value that is commutatively and associatively udpated by each invocation of the `relax` method. We will quantify the set of all such invocations in the discussion of the `foreach` statement below. For now, we only note that the invocation of `relax` on `J[x,y]` consumes values `n[x,y]`, `s[x,y]`, etc. and *reduces* the value `err`.

Contributing to reduced values in serial methods

Each invocation of the serial `relax` method above commutatively and associatively contributes to the reduction over all invocations. For this purpose, the `relax` method receives as input variable a C++ object representing the reduced value `err` in the orchestration code. The type of this variable is `Charisma::ReducedValue<>`, with the template parameter `ReducibleDouble`, which is a user-defined type.

```
void Jacobi::relax(Charisma::ConsumedValue<double> &n,
                   Charisma::ConsumedValue<double> &s,
                   Charisma::ConsumedValue<double> &e,
                   Charisma::ConsumedValue<double> &w,
                   Charisma::ReducedValue<ReducibleDouble> &err){
  for(int i = 0; i < e.size(); ++i){
    tile(i,0) = w[i];
    tile(i,nC+1) = e[i];
  }
  for(int i = 0; i < n.size(); ++i){
    tile(0,j) = n[i];
    tile(nR+1,j) = s[i];
  }
  double localMax = 0.0;
  for(int i = 1; i <= nR; ++i)
    for(int j = 1; j <= nC; ++j){
      newTile(i,j) = 0.2 * (tile(i,j) + tile(i-1,j) +
                            tile(i,j-1) + tile(i,j+1]);
      double error = newTile(i,j) - tile(i,j);
      if(fabs(error > localMax))
        localMax = error;
    }
  err.reduce(ReducibleDouble(localMax));
  swap(tile, newTile);
}
```

So, the `relax` method accrues the local maximum error in the variable `localMax`. This local error is contributed to the reduction of *Charisma* value `err` by passing it into the `reduce` call on the conduit `Charisma::ReducedValue err`. This causes a spanning-tree reduction over contributing invocations,

with the user-defined `ReducibleDouble::operator+=` being invoked at each node of the tree.

3.5.6 Parallel invocations through the `foreach` statements

We now address the question of how indices `x` and `y` are quantified in object references of the form `J[x,y]`. The *Charisma* `foreach` statement specifies a list of bound variables that take values from a given set of tuples. These tuples are used to create instances of the publish-consume statement enclosed by the `foreach` statement. Consider the example code below:

```
foreach(x,y in {0:NX-1} * {0:NY-1})
  (w[x+1,y], e[x-1,y],
   n[x,y-1], s[x,y+1]) <- J[x,y].boundaries();
```

The above code simply means that for each pair `[x,y]` in the Cartesian product `{0:NX-1} * {0:NY-1}`, one instance of the publish-consume statement

```
(w[x+1,y],...,s[x,y+1]) <- J[x,y].boundaries();
```

is executed. The order in which these executions occur is undefined. There is, however an implicit barrier at the end of each `foreach` statement. As a result, we invoke the method `boundaries` on each object in the collection `J`. Each invocation leads to the publication of four values, representing the boundaries of the corresponding object. A similar `foreach`-enclosed invocation can be made for the `relax` method. For brevity, multiple publish-consume statements can be enclosed within a single `foreach` statement; such a block is interpreted as a sequence of `foreach` statements, each enclosing one of the publish-consume statements, in order.

3.5.7 Iteration with the `while` construct

Finally, we require a mechanism to continually perform the boundary publication, followed by relaxation and error reduction, as long as the error is greater than a certain constant threshold (which is a `parameter` to the *Charisma* program). This is achieved through the `while` statement:

```
while(err > TOLERANCE)
  foreach(x,y in {0:NX-1} * {0:NY-1}){
    (w[x+1,y], e[x-1,y],
     n[x,y-1], s[x,y+1]) <- J[x,y].boundaries();

    (+err) <- J[x,y].relax(n[x,y], s[x,y],
                           e[x,y], w[x,y]);
  }
```

For completeness, we present the entire orchestration code for our *Charisma* Jacobi application below:

```
program jacobi;
parameter NX,NY : int;
parameter TOLERANCE : double;

class Jacobi;
objects J : Jacobi[NX,NY];

values n, e, w, s : double[NX,NY];
type ReducibleDouble;
value err : ReducibleDouble;

orchestrate {
  (err) <- J[0,0].initialize();

  while(err > TOLERANCE)
    foreach(x,y in {0:NX-1} * {0:NY-1}){
      (w[(x+1)%NX,y], e[(x-1+NX)%NX,y],
       n[x,(y-1+NY)%NY], s[x,(y+1)%NY]) <- J[x,y].boundaries();

      (+err) <- J[x,y].relax(n[x,y], s[x,y],
                             e[x,y], w[x,y]);
    }
}
```

To summarize, at a high level, *Charisma* code is divided into two parts:

1. The parallel structure of the application, specifying units of work, and the flow of control and data between them. This *orchestration code* is written in a specialized notation.

2. Serial computations, which are invoked from the orchestration code, and are specified in C++.

By invoking serial methods on objects in the orchestration code, one can publish and consume *values* (i.e. messages). The compiler infers (data-independent) communication patterns from the publications and consumptions, and generates the necessary sends and receives for messages. The flow of data between orchestration code and serial methods is mediated by so-called *conduit* C++ classes (`PublishedValue`, `ConsumedValue`, `ReducedValue`), which represent values within orchestration code. The results of computations in the serial methods can be associated with published values by invoking the `produce`/`reduce` methods on conduits.

3.6 The syntactic structure of *Charisma* programs

The following section is devoted to the consideration of *Charisma* syntax, whereas § 3.7 discusses the constraints placed on legal *Charisma* programs. In § 3.8 we present the operational semantics of the language. However, first we consider fragments of the Extended BNF specification of *Charisma*.

3.6.1 Top-level structure

⟨*program*⟩ ::= ("**program**" | "**module**")⟨*ident*⟩ ";"
⟨*includes*⟩⟨*decls*⟩
⟨*initialization*⟩
⟨*orchestration*⟩

The text of a *Charisma* program begins by identifying the following code as either a stand-alone, named program, or a named module. Modules can be incorporated into other *Charisma* programs, or external Charm++ code, as explained in § 3.13. This is followed by a *file inclusion* section, a *declarations* section and a sequence of *orchestration* statements:

3.6.2 Includes

$$\begin{aligned} \langle includes \rangle &::= (\langle include \rangle \text{ ";" }) * \\ \langle include \rangle &::= \text{"}\textbf{include}\text{" } \langle string \rangle \end{aligned}$$

The programmer may include C++ (header) files in order to use, for instance, user-defined data types (see below) in the *Charisma* program.

3.6.3 Declarations

$$\begin{aligned} \langle decls \rangle &::= (\langle decl \rangle \text{ ";" }) * \\ \langle decl \rangle &::= \langle parameter \rangle | \langle class \rangle | \langle objects \rangle | \langle vspace \rangle \end{aligned}$$

The declarations section consists of a number (possibly zero) of individual declarations. The programmer may declare:

Parameters

$$\langle parameter \rangle ::= \text{"}\textbf{parameter}\text{" } \langle ident \rangle \text{ ":" } \langle cppType \rangle$$

A parameter is an immutable identifier whose value is obtained from outside the *Charisma* code. For instance, the value of a parameter declared as

```
parameter myParam : int;
```

may be set to `198` in the command line for a stand-alone program, by including the string

```
--myParam 198
```

in the list of arguments passed to the binary of the program.

Classes

$$\langle class \rangle \quad ::= \quad \text{“}\textbf{class}\text{”} \ \langle ident \rangle$$

As in traditional object-oriented programming languages, a *Charisma* class serves to encapsulate logically related data structures and functions. Classes used in a *Charisma* program are not defined therein. Instead, the compiler only expects to encounter the *declaration* of every class that is used in the program. The *definition* of the class must be made in an external C++ (header) file, and included in the *Charisma* program through the `include` directive (above).

Objects and their collections

$$\begin{aligned} \langle objects \rangle \quad &::= \quad [\ \text{“}\textbf{sparse}\text{”}\ ]\ \ \text{“}\textbf{objects}\text{”}\ \langle ident \rangle\ \ \text{“:”}\ \ \langle objectsType \rangle \\ &\quad\ \ |\ \text{“}\textbf{object}\text{”}\ \langle ident \rangle\ \ \text{“:”}\ \ \langle ident \rangle \\ \langle objectsType \rangle \quad &::= \quad \langle ident \rangle\ \text{“[”}\ \langle idents \rangle\ \text{“]”} \end{aligned}$$

As in traditional object-oriented nomenclature, an object represents an instance of a class. In *Charisma* objects encapsulate both work and data. Such an object-based expression of a parallel algorithm allows generated *Charisma* code to leverage performance features provided by the Charm++ runtime system, e.g. automatic overlap of computation and communication; efficient, interleaved composability of parallel modules; object migration-based load balancing, etc. Method invocations on objects constitute the basic unit of computational work in *Charisma*.

Objects are arranged into named and indexible *collections*. Every collection of objects has a type (i.e. a declared class). Object collections may be multidimensional – the arity of the object can be one through six. A list of identifiers given in the collection declaration specifies the cardinality of the object collection along each of its dimensions. By default, object collections are *dense*. So, a dense two-dimensional collection, named `myObjects`,

of objects of (previously declared) type `MyClass` is declared as follows:

```
objects myObjects : MyClass[NX, NY];
```

where `NX` and `NY` are previously declared parameters. Since the collection is dense, every object reference `myObjects[x,y]` is a valid one, as long as `x` and `y` lie within their respective integer ranges of `0..NX-1` and `0..NY-1`. We will discuss the binding of identifiers `x` and `y` in the subsection dedicated to the `foreach` statement (§ 3.6.4).

On occasion, and especially for higher-dimensional collections, it is useful to have *sparse* collections of objects, in which the existence of object instances is dictated by the programmer. Consider the following four-dimensional, *sparse* collection of objects of type `MySecondClass`:

```
class MySecondClass;
sparse objects myOtherObjects : MySecondClass[N1,N2,N3,N4];
```

Given the declaration above, an object reference of the form

```
myOtherObjects[x1,x2,x3,x4]
```

(with `x1`, ..., `x4` appropriately bound), is a valid reference only if `x1` lies in the range `0..N1-1`, `x2` lies in the range `0..N2-1`, etc. The programmer specifies which indices correspond to actual objects through a *sparse collection insertion* statement (*cf.* § 3.6.6).

Finally, the programmer may also declare a singleton object, as follows:

```
object mySingletonObject :  MySecondClass;
```

Values

⟨*values*⟩ ::= "**values**" ⟨*ident*⟩ ":" ⟨*cppType*⟩[⟨*idents*⟩]
| "**value**" ⟨*ident*⟩ ":" ⟨*cppType*⟩

In the paradigm of objects communicating via messages, values correspond to messages, and provide a means of specifying communication patterns. An object may *publish* a value, or *consume* a value. If an object o_1 publishes a value and another object, o_2, consumes it, then the *Charisma* compiler infers

some form of communication between o_1 and o_2. We cover the syntactic expression of publication and consumption in § 3.6.4.

Values are typed, and a named collection of values of a particular type is called a *value space*. Values within a value space are identified by the name of the value space and a multidimensional subscript. By restricting the form of these subscripts, we are able to strike a balance between the expressive scope of *Charisma*, and the sophistication of static analysis required to generate communication code (*cf.* 3.12).

Consider the following *Charisma* declarations:

```
value error : double;
values forces : Force[NX, NY];
```

The first statement above declares a singleton value named `error`, of type `double`. The second statement declares a collection of values (i.e. a value space) named `forces`. This collection is two-dimensional, as indicated by the square-bracketed identifier list following the user-defined type `Force`. A reference to a particular value within the value space would have the form `forces[x,y]`, and would be valid as long as `x` and `y` evaluate to integers in the ranges `0..NX-1` and `0..NY-1`, respectively.

Unlike the notation that we developed in § 3.2, the use of value spaces allows us to separate the actions taken by the publishers from those taken by the consumers. This has the benefit that one can associate multiple consumption sites within the program, with a single publication site. However, it also means that communication is specified in an *indirect* manner, in that a compiler must examine the pattern of publications and consumptions to infer the message-sends occurring in the program.

3.6.4 Orchestration code

$$
\begin{aligned}
\langle \textit{orchestration} \rangle &::= \langle \textit{stmt} \rangle \text{ ";"} \\
\langle \textit{stmt} \rangle &::= \langle \textit{blockStmt} \rangle \mid \langle \textit{straightlineStmt} \rangle \mid \langle \textit{cflowStmt} \rangle \\
\langle \textit{blockStmt} \rangle &::= \text{ "\{" } (\langle \textit{stmt} \rangle \text{ ";" }) * \text{ "\}"} \\
\langle \textit{straightlineStmt} \rangle &::= \langle \textit{pubconStmt} \rangle | \langle \textit{foreachStmt} \rangle
\end{aligned}
$$

The orchestration part of a *Charisma* program consists of a single statement. This statement may be a curly-brace delimited *block* statement, which in turn encloses other statements. Other than block statements, *Charisma* has so-called *straightline* and *control-flow* statements. Whereas control flow statements mark points in the program where control diverges or converges, a straightline statement represents sequential flow of control.

A straightline statement may itself be of one of two subtypes. The first is the *publish-consume* statement, which we discuss next.

Publish-consume statements

A publish-consume statement has the form `LHS <- RHS`, where the symbol '`<-`' is called the *publication operator*. A publish-consume statement specifies the invocation of a *serial* method on a given object, possibly dependent on the prior receipt of a list of values by that invocation, and possibly resulting in the publication of a list of values.

$$\begin{aligned} \langle pubconStmt \rangle &::= \text{“(”}\ \langle pubVals \rangle\ \text{“)”}\quad \text{“}\leftarrow\text{”}\ \langle mthdInvocation \rangle \\ \langle mthdInvocation \rangle &::= \langle objRef \rangle\ \text{“.”}\ \langle ident \rangle\ \text{“(”}\ \langle conVals \rangle\ \text{“)”} \end{aligned}$$

Published values are either *produced* (i.e. each value is generated by the action of a single method invocation) or *reduced* (i.e. several method invocations contribute to a reduction). Reduced value expressions are prefixed with the '`+`' operator.

$$\begin{aligned} \langle pubVals \rangle &::= \langle pubVal \rangle(\ \text{“,”}\ \langle pubVal \rangle) * \\ \langle pubVal \rangle &::= [\ \text{“+”}\ ]\langle ident \rangle[\langle pubSubscript \rangle] \\ \langle pubSubscript \rangle &::= \text{“[”}\ \langle expr \rangle(\ \text{“,”}\ \langle expr \rangle) * \ \text{“]”} \\ \langle expr \rangle &::= \langle ident \rangle \mid \langle arithExpr \rangle \mid \langle boolExpr \rangle \end{aligned}$$

A published value expression may refer to a singleton value, or it may

contain a square-bracket-delimited list of expressions, identifying the particular value within a value space that is published by a given method invocation. These expressions may be identifiers or arithmetic expressions involving identifiers, parameters and loop index variables. However, in order to keep communication patterns independent of data, these index expressions may not make references to *Charisma* values.

A consumed value subscript is more limited in its form: it can either be an identifier or the *wildcard* expression, '`*(k)`'. The occurrence of '`*(k)`' in the l-th position in a consumed value's subscript signifies that the consuming method awaits the receipt of k values, with unspecified expressions in the l-th position. These constraints on the form of published and consumed value expressions allow the *Charisma* compiler to use very simple dependency analysis to perform the required communication pattern inference. We define precise constraints on these and other expressions in § 3.7.

$$
\begin{aligned}
\langle conVals \rangle &::= \langle conVal \rangle (\text{ ``,'' } \langle conVal \rangle) * \\
\langle conVal \rangle &::= \langle ident \rangle [conSubscripts] \\
\langle conSubscripts \rangle &::= \text{ ``['' } \langle conSubscript \rangle (\text{ ``,'' } \langle conSubscript \rangle) * \text{ ``]''} \\
\langle conSubscript \rangle &::= \langle ident \rangle | \langle wildcardExpr \rangle \\
\langle wildcardExpr \rangle &::= \text{ ``*'' } \text{ ``('' } \langle expr \rangle \text{ ``)''}
\end{aligned}
$$

Examples of publish-consume statements, preceded by the appropriate declarations, are given below:

```
class Aclass, Bclass, Cclass;
object A : Aclass;
object B : Bclass;
object C : Cclass;

value p : MyValueType;
value q : MyOtherValueType;

p <- A.f1();
q <- B.f2();
C.f3(p,q);
```

In the first of the three publish-consume statements above, method `f1` is

invoked on object `A`, leading to the publication of value `p`. Similarly the second statement invokes `f2` on `B`, leading to the publication of `q`. The invocation of method `f3` in the last publish-consume statement takes as input the values `p` and `q`, and occurs after the first two publish-consume statements in program order. That is, the invocation `C.f3` consumes the values `p` and `q` published by `A.f1` and `B.f2`, respectively; therefore, we say that `C.f3` is dependent on `A.f1` and `B.f2`.

Interfacing with serial code. The invocation of serial methods leads to the publication and consumption of values. Therefore, we must provide an interface between C++ code in serial method invocations and *Charisma* orchestration code. The purpose of this interface would be to bind values specified in orchestration code, to data generated and used by serial computation.

This is achieved by the C++ class templates `Charisma::PublishedValue` and `Charisma::ConsumedValue`. Specifically, by invoking the `produce` method on an (serial/C++) object of type `Charisma::PublishedValue` in serial code, the programmer's serial, C++ code can bind data to *Charisma* values, and thereby pass those data into compiler-generated, parallel code.

Likewise, the C++ class `Charisma::ConsumedValue` encapsulates consumed values in the *Charisma* orchestration code. A "`[ ]`" operator is provided to access data elements associated with each consumed *Charisma* value. These data can be used by serial C++ code to perform computational work.

For the contrived example code involving singleton objects `A`, `B` and `C` above, the definition of `f1` might look like this:

```
void Aclass::f1(Charisma::PublishedValue< MyType > &p)
{
  int numPublishedElements = ...;
  std::vector< MyType > data;
  for(int i = 0; i < numPublishedElements; ++i)
  {
    data[i] = MyType(i);
  }
  // bind generated 'data' to published value 'p'
  p.produce(&data[0], &data[numPublishedElements-1]);

}
```

The method performs some (trivial) computational work, and generates

elements of the `std::vector` '`data`' in the process. These data elements are mapped onto the singleton value `p` in the *Charisma* orchestration code, by the `produce` method. The definition of method `Cclass::f3` below shows how the data encapsulated by *Charisma* consumed values are passed into the programmer's serial C++ code:

```
void Cclass::f3(Charisma::ConsumedValue< MyType > &p,
                Charisma::ConsumedValue< MyOtherType > &q)
{
  int sum1 = 0;
  int sum2 = 0;
  for(int i = 0; i < p.size(); ++i) sum1[i] += p[i].getIntValue();
  for(int i = 0; i < q.size(); ++i) sum2[i] += q[i].getIntValue();
}
```

The code above shows how individual data elements within a consumed value may be accessed in serial, C++ computations via the square bracket operator.

Explicit parallelism and the `foreach` statement

$$\begin{aligned}
\langle foreachStmt\rangle &::= \langle foreachHead\rangle\langle pubconStmt\rangle \\
\langle foreachHead\rangle &::= \text{“}\textbf{foreach}\text{”}\ \text{“(”}\ \langle idents\rangle\ \text{“}\textbf{in}\text{”}\ \langle objIndexSpace\rangle \\
&\quad [\ \text{“:”}\ \langle expr\rangle]\ \text{“)”}
\end{aligned}$$

The `foreach` statement is used to create unordered instances of an enclosed publish-consume statement. One instance of the enclosed publish-consume statement is created for each element present in an associated *object index space*.

$$\begin{aligned}
\langle objIndexSpace\rangle &::= \text{“\{”}\ \langle expr\rangle\ \text{“:”}\ \langle expr\rangle[\ \text{“:”}\ \langle expr\rangle]\ \text{“\}”} \\
&\quad |\langle objIndexSpace\rangle\ \text{“}*\text{”}\ \langle objIndexSpace\rangle \\
&\quad |\ \text{“}\textbf{ispace}\text{”}\ (\langle ident\rangle)
\end{aligned}$$

An object index space can be explicit, i.e. either (curly-brace-delimited, strided) range of integer values, or a Cartesian product (signified by the '`*`' symbol) of two object index spaces. We can also have expressions that implicitly refer to the underlying object index space of a specified collection, using the `ispace` keyword. The elements of an object index space serve as valid indices in the subscripts of object references.

Notionally, the `foreach` construct commands (a subset of) objects of a collection to execute a particular method. There is no order implied between these method invocations. These invocations are constrained only by the availability of messages representing the values that they consume. Below is a simple example of a `foreach` statement, which causes the concurrent invocation of method `foo` on all objects of a collection named `A`.

```
parameter NX, NY : int;
class Aclass;
objects A : Aclass[NX,NY];

foreach(x,y in {0:NX-1} * {0:NY-1})
  A[x,y].foo();
```

Identifiers `x` and `y`, which serve as subscript indices in the object reference `A[x,y]`, are bound by the `foreach` statement to elements within the associated object index space (the Cartesian product `{0:NX-1} * {0:NY-1}`). One instance of the enclosed publish-consume statement is executed for each object `A[x,y]` in the collection `A`. However, we may choose to bind fewer object subscript indices than the arity of the collection, as below:

```
foreach(y in {0:NY-1})
  A[0,y].foo();
```

This would limit the set of objects on which `foo` is invoked, to the zeroth row of the object collection `A`.

The programmer may also associate an optional *filter* expression to constrain the set of object indices over which the `foreach` is applied. For instance, one could invoke the `foo` method on the set of all objects in collection `A` on its K-th anti-diagonal, as follows:

```
foreach(x,y in {0:NX-1}*{0:NY-1} : x+y == K)
  A[x,y].foo();
```

The `foreach` statement can be used to publish and consume values *en masse*. For instance, the following `foreach` statements publish and consume all the elements of value space '`p`', respectively:

```
parameter N : int;
class Aclass, Bclass;
objects A : Aclass[N];
objects B : Bclass[N];
values p : double[N];

foreach (x in {0:N-1})
  (p[x]) <- A[x].foo();

foreach (x in {0:N-1})
  B[x].bar(p[x]);
```

The orchestration statements above are interpreted as follows: (1) Method `foo` is invoked on each object `A[x]`, where `x` lies in `0..N-1`; each invocation leads to the publication of an element `p[x]` of the value space `p`; (2) Then, for each `x` in `0..N-1`, the invocation of `bar` on `B[x]` consumes `p[x]`. The *Charisma* compiler infers that each `A[x]` sends a single message to `B[x]`, implying a one-to-one communication pattern.

3.6.5 Control flow constructs

In a *Charisma* program control starts at the first orchestration statement, and is thereafter transferred from one statement within a block of statements to the one that follows it in program order. Control flow statements, in the form of loops and conditionals, can modulate this sequential flow of control.

$$\langle cflowStmt \rangle \quad ::= \quad \langle forStmt \rangle | \langle whileStmt \rangle | \langle ifElseStmt \rangle$$

`for`: computation-independent number of iterations

⟨*forStmt*⟩ ::= "**for**"
"(" ⟨*ident*⟩ "=" ⟨*expr*⟩ ":" ⟨*expr*⟩[":" ⟨*expr*⟩] ")"
⟨*blockStmt*⟩

Each `for` loop has an associated loop index variable, which is considered in-scope only for the block of statements enclosed within the `for` loop. The *Charisma* `for` loop resembles the Fortran `do` loop, in that it specifies an initialization expression for the loop index variable, a termination expression, and an increment expression. Each one of these expressions can involve numeric constants, `parameter`s passed to the program, and in-scope loop index variables. Therefore, a *Charisma* `for` loop executes for a statically determinable number of iterations. Moreover, these iterations execute sequentially.

`while`: computation-dependent number of iterations

⟨*whileStmt*⟩ ::= "**while**" "(" ⟨*boolExpr*⟩ ")"
⟨*blockStmt*⟩

The number of repetitions of a `while` loop's body may depend on constants, `parameter`s, in-scope loop index variables *and values*. The compiler generates code to collect all such values referenced within the `while` loop predicate, evaluates the predicate in a centralized manner, and begins the next iteration of the loop, contingent upon successful evaluation of the predicate.

`if-else`: conditional branching of global control

$$\begin{aligned}\langle ifElseStmt\rangle \quad ::= \quad & \text{“}\textbf{if}\text{”} \quad \text{“(”} \ \langle boolExpr\rangle \ \text{“)”} \\ & \langle blockStmt\rangle \\ & \text{“}\textbf{else}\text{”} \\ & \langle blockStmt\rangle\end{aligned}$$

The `if-else` statement has an associated predicate which is evaluated at run-time to determine whether the `then` branch or the `else` branch will be followed.

Whenever `while` and `if-else` expressions are determined by the compiler to be dependent only on numeric constants, `parameter`s and in-scope loop index variables, it generates code for their distributed (as opposed to centralized) evaluation, thereby increasing concurrency.

3.6.6 Initialization

Generally speaking, the initialization section is needed only when writing advanced *Charisma* programs. It allows the programmer to insert valid objects into *sparse* object collections.

$$\begin{aligned}\langle ifElseStmt\rangle \quad ::= \quad & \text{“}\textbf{initialize}\text{”} \ \langle initStmts\rangle \\ \langle initStmts\rangle \quad ::= \quad & \text{“\{”} \ \langle initStmt\rangle + \ \text{“\}”} \ |\langle initStmt\rangle \\ \langle initStmt\rangle \quad & \langle foreachHead\rangle\langle insertStmt\rangle \\ \langle insertStmt\rangle \quad ::= \quad & \text{“}\textbf{insert}\text{”} \quad \text{“(”} \ \langle objRefs\rangle \ \text{“)”} \\ \langle objRefs\rangle \quad ::= \quad & \langle objRef\rangle (\ \text{“,”} \ \langle objRef\rangle) *\end{aligned}$$

The initialization statement is structured similarly to the `foreach` statement, except that it can only enclose an *insertion* statement. In turn, an insertion statement commands the insertion of a single object, given by an object reference that possibly uses the square bracket notation seen in publish-

consume statements. Consider the following fragment from the molecular dynamics code in § 3.9.2:

```
parameter N, NIterations : int;

class PatchClass, ComputeClass;
objects Patch: PatchClass[N,N];
sparse objects Computes : ComputeClass[N,N,N,N];

initialize {
  foreach (x,y in {0:N-1}*{0:N-1})
    insert(Computes[x, y, ((x-1) + N)%N, ((y-1) + N)%N],
           Computes[x, y, ((x-1) + N)%N, y],
           Computes[x, y, ((x-1) + N)%N, (y+1)%N],
           Computes[x, y, x, y],
           Computes[x, y, x, (y+1)%N]);
}
```

In the code above, we insert a number of elements into the initially empty *sparse*, four dimensional object collection `Computes`. The idea is to insert objects in `Computes` for pairs of neighboring `Patches[x1,y1]` and `Patches[x2,y2]`. Two such `Patches` are considered neighbors if `[x2,y2] = [((x1-1) + N)%N, ((y1-1) + N)%N]`, or `[x2,y2] = [((x1-1) + N)%N, y1]`, etc. As was the case with the `foreach` statement in the orchestration section, a number of instances of the enclosed `insert` statement are executed, each with a different `[x,y]` pair drawn from the object index space `{0:N-1}*{0:N-1}`. Therefore, we insert one `Computes` object for the neighboring pair of indices `[x,y]` and `[((x-1)+N)%N, ((y-1) + N)%N]`; one for the neighboring pair `[x,y]` and `[((x-1) + N)%N, y]`, etc.

3.7 Well-formedness conditions

Not all syntactically correct *Charisma* programs are considered semantically acceptable. In the following, we present statically verifiable semantic constraints on the structure of *Charisma* programs. Dynamic errors are defined in § 3.8.

We use the following nomenclature: the orchestration part of a *Charisma* program is denoted π. The set of value spaces defined within π is symbolized by $\mathcal{V}(\pi)$, and the set of `parameter` declarations in π is $\mathcal{D}(\pi)$. The set of all object collections defined in π is denoted $\mathcal{O}(\pi)$. We write $E_x^{B,I}(e)$ to indicate that e is a well-formed *Expression* of type x, given the set of in-scope, `foreach`-bound identifiers B, and the set of in-scope, loop index variables, I. Elements within B are pairs of the form (e, i), where e is an atomic expression (defined shortly), and i is an integer. As discussed later, this allows us to express constraints on the ordering of `foreach`-bound identifiers in lists within certain kinds of expression. Similarly, $W^{B,I}(s)$ indicates the well-formedness of *statement* s, with B and I defined as previously.

Constraints are expressed either in plain English, or, after Plotkin [44] and Slonneger and Kurtz [45], as inference rules of the form:

$$\frac{premise_1 \quad \ldots \quad premise_n}{conclusion} \text{ conditions}$$

3.7.1 Expressions

Expressions form the structural building blocks of statements, and to a large extent, determine their meaning. As such, we use constrain the form of *Charisma* expressions to limit its expression scope.

Literal expressions

For any B and I, $E_{lit}^{B,I}(e)$ if e is a Boolean literal (`true` or `false`) or a numeric literal (e.g. `123`, `0.008`, `1.0e-9`, etc.), for any sets B and I.

Free identifiers

Given any B and I, An expression e is a well-formed *free* identifier expression under B and I, i.e. $E_{free}^{B,I}(e)$, if *none* of the following is true:

1. $\exists i.\ (e, i) \in B$, i.e. e is a `foreach`-bound identifier;
2. $e \in I$, i.e. e is a `for`-loop index variable;
3. $e \in \mathcal{D}(\pi)$, i.e. e is a parameter of the `program`/`module`

4. $e \in \mathcal{V}(\pi)$, i.e. e is the name of a value space

Bound identifiers

An expression e is a well-formed bound identifier expression for any B and I, i.e. $E^{B,I}_{bound}(e)$, if *exactly one* of the following holds:

1. $\exists i.\ (e, i) \in B$, i.e. e is a `foreach`-bound identifier;
2. $e \in I$, i.e. e is a `for`-loop index variable;
3. $e \in \mathcal{D}(\pi)$, i.e. e is a parameter of the `program`/`module`

We do not include value names as bound identifier expressions.

Atomic expressions

These are expressions that cannot be broken down into operations over operands. Numeric and Boolean literals, as well as bound identifiers (*cf.* above) are all well-formed atomic expressions.

$$\frac{E^{B,I}_{lit}(e)}{E^{B,I}_{atom}(e)} \qquad\qquad \frac{E^{B,I}_{bound}(e)}{E^{B,I}_{atom}(e)}$$

Non-value expressions

Well-formed expressions that do not contain any values are termed well-formed non-value expressions. Therefore, all well-formed atomic expressions (as defined above) are well-formed non-value expressions:

$$\frac{E^{B,I}_{atom}(x)}{E^{B,I}_{nv}(x)}$$

Expressions involving binary operators over well-formed non-value expressions are also considered well-formed non-value expressions:

$$\frac{E^{B,I}_{nv}(e_1) \quad E^{B,I}_{nv}(e_2)}{E^{B,I}_{nv}(e_1 \text{ op } e_2)}$$

where op $\in \{+, -, *, \%, /\}$.

Similarly, the unary negation, or parenthesization of a well-formed non-value expression is also a well-formed non-value expression.

$$\frac{E_{nv}^{B,I}(e)}{E_{nv}^{B,I}(!e)} \qquad \frac{E_{nv}^{B,I}(e)}{E_{nv}^{B,I}((e))}$$

Value expressions

Now we consider *Charisma* expressions that contain value references. As a base case, all value space names are considered well-formed value expressions. That is, $\forall p \in \mathcal{V}(\pi)$, given *any* B and I, it is the case that $E_v^{B,I}(p)$.

Given an m-dimensional value space p, and well-formed, *non-value expressions* $e_1, \ldots, e_m$, we use the square-bracket notation to denote a well-formed value expression that represents a particular element within p:

$$\frac{E_v^{B,I}(p) \quad \forall i.\ E_{nv}^{B,I}(e_i)}{E_v^{B,I}(p[e_1, \ldots, e_m])}$$

where $arity(p) = m$. Subscript index expressions for values must be non-value expressions. This means that the identity of a particular value published or consumed by a method invocation cannot depend on the evaluation of another value expression, so that the compiler analysis of published and consumed value indices can be kept simple. Although we sacrifice expressiveness through this constraint, in the domain of applications that we have considered, such data-dependent communication patterns do not arise. However, this constraint does limit the expressiveness of *Charisma*.

Returning to the definition of well-formed value expressions, when we combine a well-formed value-expression with a well-formed non-value expression using a binary operator, we get a well-formed value expressions:

$$\frac{E_v^{B,I}(e_1) \quad E_v^{B,I}(e_2)}{E_v^{B,I}(e_1 \text{ op } e_2)} \qquad \frac{E_{nv}^{B,I}(e_1) \quad E_v^{B,I}(e_2)}{E_v^{B,I}(e_1 \text{ op } e_2)} \qquad \frac{E_v^{B,I}(e_1) \quad E_{nv}^{B,I}(e_2)}{E_v^{B,I}(e_1 \text{ op } e_2)}$$

where op $\in \{+, -, *, \%, /\}$. Finally,

$$\frac{E_v^{B,I}(e)}{E_v^{B,I}((e))} \qquad \frac{E_v^{B,I}(e)}{E_v^{B,I}(!e)}$$

where '!' signifies the Boolean negation operator.

Range expressions

Range expressions occur in `foreach` statements, and consist of colon-separated start, end and (optional) stride expressions enclosed within curly braces:

$$\frac{E_{nv}^{B,I}(e_1) \quad E_{nv}^{B,I}(e_2) \quad E_{nv}^{B,I}(e_3)}{E_{range}^{B,I}(\{e_1 : e_2 : e_3\})}$$

Wildcard expressions

Wildcard expressions are used to specify the consumption of values whose subscript index along a particular dimension is unconstrained. Wildcard expressions are used only in the subscripts of consumed values, as stated shortly hereafter. A well-formed wildcard expression is simply the concatenation of the wildcard operator '$*$' and a parenthesized, well-formed *non-value* expression:

$$\frac{E_{nv}^{B,I}(e)}{E_{wild}^{B,I}(*(e))}$$

The parenthesized expression specifies the dynamically evaluated (but data-independent) number of values to consume.

Consumed value subscripts

We are now in a position to consider the requirements of well-formedness for expressions that occur as subscript indices in consumed and published value expressions. We consider the former first. Given B and I, e is a well-formed consumed-value subscript expression, or $E_{cexp}^{B,I}(e)$, if one of the following holds:

1. $\exists i.\ (e, i) \in B$, i.e. e is an identifier corresponding to a `foreach`-bound variable; or
2. $E_{wild}^{B,I}(e)$, i.e. e is a well-formed wildcard expression.

Published value subscripts

On the other hand, published value subscripts can be more complex. All well-formed non-value and range expressions are considered well-formed published value subscripts:

$$\frac{E^{B,I}_{nv}(e)}{E^{B,I}_{pexp}(e)} \quad \frac{E^{B,I}_{range}(e)}{E^{B,I}_{pexp}(e)}$$

Consumed values

A consumed value expression consists of the name of a declared value space, and a square bracket-enclosed list of consumed value subscripts. The length of this list must equal the arity of the value space. Additionally any foreach-bound variable in the subscript must occur in the *same position* as it does in the list of bound variables specified in the `foreach` statement. More rigorously,

$$\frac{E^{B,I}_{v}(p) \quad \forall i.\ E^{B,I}_{cexp}(e_i)}{E^{B,I}_{cons}(p[e_1, \ldots, e_m])}$$

such that, if $\exists j.\ (e_i, j) \in B$, then $i = j$.

For instance, the consumed values in the following publish-consume statements are valid. This is because whenever either one of the `foreach`-bound variables `x` and `y` occurs in a consumed value subscript, its position of occurrence is the same as its position in the list of bound variables for the enclosing `foreach` statement, namely `x,y`:

```
foreach (x,y in A)
{
  A[x,y].foo(p[x,y]);
  A[x,y].foo(p[x,*(N)]);
  A[x,y].foo(p[*(N),y]);
}
```

On the other hand, below are some instances of illegal consumed value expressions. These expressions are unacceptable because they transpose the positions of occurrence of `x` and `y` from their positions in the list of `foreach`-bound variables.

```
foreach (x,y in A)
{
  A[x,y].foo(p[y,x]);
  A[x,y].foo(p[y,*(N)]);
  A[x,y].foo(p[*(N),x]);
}
```

This ordering constraint helps to ensure that communication is sender-directed, and that communicatino patterns can be easily inferred by the accompanying compiler.

Published values

Unlike consumed value expressions, published value expressions are relatively unconstrained. Indeed, as long as the individual index expressions within the published value are well-formed published value subscripts, the square bracket notation gives us a well-formed published value expression:

$$\frac{E_v^{B,I}(p) \quad \forall i.\ E_{pexp}^{B,I}(e_i)}{E_{pub}^{B,I}(p[e_1, \ldots, e_m])}$$

Discussion. The reason behind the differential complexity of published- and consumed value subscripts is this. We would like to strike the right balance between (i) the expressive scope of *Charisma*, (ii) the sophistication of static analysis techniques required in order to infer communication patterns therein, and (iii) the amount of parallel overhead required to implement *Charisma* programs. Since we translate code to the message-driven mold of Charm++, the sender of a message is the one that must initiate its communication. Therefore, in the translated code corresponding to a *Charisma* program, it is the publisher object of a value that must determine the object(s) to which it must send a message containing the corresponding data. If both published and consumed value subscripts were similarly complex, we would have to apply sophisticated compiler analysis, for instance those prescribed by the rich and extensive literature on polyhedral analysis [46], to determine the intended recipients of each published value. However, even with such techniques it is not possible to resolve (or indeed express) all types of dependencies. Indeed, the most popular approach to polyhedral analysis expresses dependencies as formulas in the decidable Presburger arithmetic; although elegant, this approach cannot capture dependencies between expressions involving the multiplication, division or modulus operators. On the other hand, we could adopt techniques that resolve such dependencies dynamically, as done in Linda [47].

In constraining the forms of published and consumed value subscripts, we

sacrifice some generality, as in the approach based on polyhedral analysis. However, we are able to handle a much greater complexity of published value subscripts, *without using sophisticated static analysis*, simply because the consumed value subscripts expressions are of such a simple and restricted form. We detail the inference of communication pattern from published and consumed value subscripts, in § 3.12. Here we only mention that by coupling the asymmetric complexity of published and consumed value expressions, with simple compiler analysis, we are able to ensure the following property. That given a collection of objects that publish the elements within a value space, and a collection of objects that consume these values, each publisher can determine, via simple arithmetic calculations, the particular consumers to which it should send its data. Moreover, this can be done without any communication overhead, unlike in the tuple space matching scheme of Linda.

We now return to our discussion of well-formed *Charisma* expressions.

Object references

Object references occur in publish-consume statements, and are well-formed if the collection to which the referred object belongs has been previously declared, and each one of the subscript index expressions is a bound identifier (i.e. a parameter, loop-index variable or foreach-bound variable). Additionally, subscript index expressions must satisfy the same ordering constraint as consumed value subscripts. Formally,

$$\frac{\forall i.\ E^{B,I}_{bound}(e_i)}{E^{B,I}_{obj}(A[e_1, \ldots, e_m])}$$

such that $A \in \mathcal{O}(\pi)$. Further, for all $i \in \{1, \ldots, m\}$ if $\exists j.\ (e_i, j) \in B$, then $i = j$.

Inserted object references

Object references that appear within the sparse insertion statement (initialization section), are less restricted in form than the object references

embedded within publish-consume statements (above).

$$\frac{\forall i.\ E^{B,I}_{nv}(e_i)}{E^{B,I}_{insObj}(A[e_1, \ldots, e_m])}$$

such that $A \in \mathcal{O}(\pi)$. That is, for a well-formed inserted object reference, the programmer may choose as subscript index expressions *any* well-formed, non-value expressions, and not just identifiers, literals and parameters.

Object collection index spaces

We now consider the well-formedness of expressions that represent object index spaces. Recall that these are used in the `foreach` statement to invoke a given method on a particular subset of an object collection.

First, if $A \in \mathcal{O}(\pi)$, then ispace(A) is a well-formed object collection index space expression. We can construct an index space by taking the Cartesian product of several well-formed range expressions:

$$\frac{\forall i.\ E^{B,I}_{range}(e_i)}{E^{B,I}_{ispace}(e_1 * \ldots * e_m)}$$

3.7.2 Statements

We now have the building blocks to consider the well-formedness of *Charisma* statements at large. We begin with high-level statements, and proceed to refine the conditions for well-formedness: sequential blocks are covered first, followed by control flow statements, and finally `foreach` and publish-consume statements.

Statement blocks

Given two well-formed *Charisma* statements, their sequential composition is also well-formed.

$$\frac{W^{B,I}(s_1) \quad W^{B,I}(s_2)}{W^{B,I}(s_1; s_2)}$$

The `for` statement

A `for` statement is well-formed if its components satisfy the following requirements: its loop index variable are not bound; the strided range over which iteration occurs must consist entirely of *non-value* expressions; and the body of statements that it encloses must be well-formed. The enclosed body of statements is allowed to refer to the loop index variable. That is,

$$\frac{E^{B,I}_{free}(e) \quad E^{B,I}_{nv}(e_1) \quad E^{B,I}_{nv}(e_2) \quad E^{B,I}_{nv}(e_3) \quad W^{B,J}(s)}{W^{B,I}(\text{for}(e = e_1 : e_2 : e_3)\ s)}$$

where $J = I \cup \{e\}$.

The `while` statement

As mentioned previously, the loop continuation predicate of a `while` statement may be a well-formed arithmetic expression involving values, making it a value expression:

$$\frac{E^{B,I}_{v}(e) \quad W^{B,I}(s)}{W^{B,I}(\text{while}(e)\ s)}$$

The `if-else` statement

As with the `while` statement, the predicate evaluated in the `if-else` statement in choosing between divergent control flow paths, may include value references:

$$\frac{E^{B,I}_{v}(e) \quad W^{B,I}(s_1) \quad W^{B,I}(s_2)}{W^{B,I}(\text{if}(e)\ s_1\ \text{else}\ s_2)}$$

The `foreach` statement

A `foreach` statement has several components, each of which must be well-formed in order for the `foreach` statement to be well-formed: its list of bound variables must be free in its scope; its associated object index space (Ψ) must be a well-formed index space expression; its enclosed statement must be a well-formed, publish-consume statement; and finally, its associated *filter*

predicate expression (Q) must be a well-formed, non-value expression.

$$\frac{\forall i.\ E^{B,I}_{free}(e_i) \quad E^{B,I}_{ispace}(\Psi) \quad E^{C,I}_{nv}(Q) \quad W^{C,I}_{PC}(s)}{W^{B,I}(\text{foreach } (e_1, \ldots, e_m \text{ in } \Psi :\ Q)\ s)}$$

where $C = (\bigcup_{i=1}^{m} \{(e_i, i)\}) \cup B$. The enclosed statement (s) and the filter predicate (Q) may make references to the variables bound by the `foreach` statement.

The publish-consume statement

A publish-consume statement is well-formed if each of its components is well-formed: the object reference expression, the list of consumed values, and the list of published values.

$$\frac{E^{B,I}_{obj}(o) \quad \forall i.\ E^{B,I}_{cons}(c_i) \quad \forall j.\ E^{B,I}_{pub}(p_j)}{W^{B,I}_{PC}((p_1, \ldots, p_n) \leftarrow o.f(c_1, \ldots, c_m))}$$

The sparse collection object insertion statement

Recall that the insertion statement occurs within the initialization section of a *Charisma* program. Such a statement is well-formed if each one of the object references that it makes is a well-formed *inserted* object reference:

$$\frac{\forall i.\ E^{B,I}_{insObj}(o_i)}{W^{B,I}_{ins}(\text{insert}(o_1, \ldots, o_m))}$$

The initialization `foreach` statement

When a `foreach` statement appears in the initialization section of a *Charisma* program, it may only enclose a well-formed sparse object insertion statement:

$$\frac{\forall i.\ E^{B,I}_{free}(e_i) \quad E^{B,I}_{ispace}(\Psi) \quad E^{C,I}_{nv}(Q) \quad W^{C,I}_{ins}(s)}{W^{B,I}(\text{foreach } (e_1, \ldots, e_m \text{ in } \Psi :\ Q)\ s)}$$

3.8 Operational semantics

We now present a simple structural operational semantics that models the execution of *Charisma* programs. Given a sequence of *Charisma* statements, our intent is to explain its meaning in terms of what commands are executed, and their effect on the state of the program. We will also define precisely the meaning of publication and consumption of values.

Since a *Charisma* program has a clear separation between (parallel) orchestration code and serial, C++ code, we distinguish the semantics of the *Charisma* notation, from the semantics of the C++ code executed in serial methods. We abstract away the serial semantics by treating serial method invocations as (mathematical) functions that transform object state, with side-effects that are visible to the orchestration code.

Consider the state of an object to be the union of all the data structures that it maintains. We treat the state of one object as being disjoint from all others. The union of states of all objects in a *Charisma* program is referred to as the *serial* state of the program. Informally, the *parallel* state of the program comprises the evaluations of values and loop index variables, as well as immutable parameters. (We will shortly define the notion of parallel state more precisely.)

Serial method invocations on objects can then be modeled as having two types of side-effect:

1. The state of the object is modified as dictated by the semantics of the serial language in which the method is written (i.e. C++), thereby modifying the serial state of the *Charisma* program.

2. Moreover, the `produce` and `reduce` methods called on `PublishedValue`'s and `ReducedValue`'s modify the parallel state of the program (but not the serial state), by updating locations that correspond to their published values. `Charisma::ConsumedValue`s are immutable, so that no operation performed on a C++ object of this type can change serial or parallel program state.

This separation saves us the trouble of describing precisely the transformations in object state that occur when a serial method is invoked on it. We can then focus on the semantics of the orchestration part itself.

3.8.1 Notation

Before we present a small-step semantics of *Charisma* it is instructive to consider some conventions and notation.

Arity and cardinality of value spaces

The expression $arity(p)$ is the number of dimensions of value space p. This value is fixed when p is declared. We write $card(p, i)$ to mean the number of values in p along its i-th dimension, where $0 \leq i \leq arity(p) - 1$. By $\vec{y} \in \text{ispace}(p)$, we mean that $arity(\vec{y}) = arity(p)$, and $0 \leq y_j < card(p, j)$ for all $0 \leq j < arity(p)$.

Semantic store

The *store* of a program, σ, is a semantic construct consisting of a number of named locations. Given a *Charisma* orchestration program π, we have $\sigma = \langle \sigma_{p_1}, \ldots, \sigma_{p_n} \rangle$, where $\mathcal{V}(\pi) = \{p_1, \ldots, p_n\}$. Each σ_p is a function that maps a vector $\vec{e}$, with $arity(\vec{e}) = arity(p)$, to a value of type $Type(p)$.

Named values within the store are initially *undefined*. We write this as

$$\forall\, p \in \mathcal{V}(\pi).\ \forall\, \vec{i} \in \text{ispace}(p).\ \sigma_p(\vec{i}\,) = \bot$$

The serial state of objects

Each declared object in the *Charisma* program is assumed to encapsulate some *state*. The set $X(\pi)$ comprises the state of every declared object in the *Charisma* program. Usually, since the program under consideration is fixed, we refer to this set simply as X. Since objects are arranged into collections, we consider X to be composed of subsets X_C, such that $X = \langle X_C \rangle_{C \in \mathcal{O}(\pi)}$. The state of the object at index $\vec{e}$ in collection C is given by $X_C(\vec{e})$.

Transition system

We present the semantics of *Charisma* in the form of transition relations over *configurations* of the program. Configurations represent the global state of a *Charisma* program, and have the form $\langle S, \sigma, X \rangle$.

1. σ and X represent the mappings for named entities in the *Charisma* program.

2. S represents the statement in the program that is to be executed next. The semantic variable S ranges over individual statements, as well as program-ordered lists thereof. S may also have the special value of 'skip', whose meaning is discussed later.

3. The transition relation is defined over pairs of configurations. Intuitively, it describes the flow of control in the orchestration code, and the resulting evolution of program state.

Term substitution

Statements and expressions may contain identifier expressions that correspond to `foreach`-bound index variables and `for`-loop index variables. The binding of such variables to semantically evaluated values is decided by the particular type of statement (either a `foreach` or a `for` statement). However in both cases, the binding results in the substitution of occurrences of bound variables with certain evaluated values. To signifiy this, it is useful to have a notation for term substitution.

Let E be a *Charisma* expression. Then, $E\{i/y\}$ denotes a partially evaluated expression, which is the same as E, except with all occurrences of term y substituted by i. Multiple simultaneous substitutions are written as $E\{i_1/y_1, \ldots, i_m/y_m\}$, the shorthand for which is $E\{\vec{i}/\mathbf{y}\}$.

Substitutions of bound variables can be applied when evaluating statements as well.

Updates to store locations

For some $p \in \mathcal{V}(\pi)$, and $\vec{i} \in \text{ispace}(p)$ we write

$$\sigma'_p = \sigma_p\{v/\vec{i}\,\}$$

to mean that $\sigma'_p(\vec{i}\,) = v$, and $\sigma'_p(\vec{j}\,) = \sigma_p(\vec{j}\,)$, for all $\vec{j} \in \text{ispace}(p)$ such that $\vec{j} \neq \vec{i}$.

3.8.2 Expression evaluation

In order to describe the meaning of statements, we must first define the meaning of the expressions that are embedded within them. This theme is discussed next.

Boolean and arithmetic expressions

It is fairly straightforward to evaluate compound expressions involving unary or binary operators, in terms of their operands. As such, we take as given the operators $\rightarrow_A$ and $\rightarrow_B$, which mean the evaluation (in the standard sense) of arithmetic and Boolean expressions, respectively. For example, $\langle e, \sigma, X \rangle \rightarrow_A n$ means that arithmetic expression e evaluates to n with the value store σ and the object states X. Note that expression evaluation does not change σ or X.

Consumed value expressions

Now let us define the meaning of $\rightarrow_A$ for the evaluation of *Charisma* values. Here we consider *consumed values*. We will discuss the case of published values later, together with the semantics of the publish-consume statement.

Given value space p and an expression vector $(y_1, \ldots, y_m)$, where each y_j is an identifier expression, the consumed value expression

$$p[y_1, \ldots, y_m]\{i_1/y_1, \ldots, i_m/y_m\}$$

is evaluated thus:

$$\langle p[y_1, \ldots, y_m]\{i_1/y_1, \ldots, i_m/y_m\}, \sigma, X \rangle \rightarrow_A v$$

where $v = \emptyset$ if $\sigma_p(\vec{i}\,) = \perp$, and $v = \sigma_p(\vec{i}\,)$ otherwise.

The equivalent vectorized shorthand is given below:

$$\langle p[\mathbf{y}]\{\vec{i}/\mathbf{y}\}, \sigma, X \rangle \rightarrow_A v$$

A value expression with *wildcards* in its subscript is evaluated one wildcard operator at a time, from left to right. Let $i_1, \ldots, i_m$ be evaluated integers,

and $f_1, \ldots, f_n$ be unevaluated, non-value expressions. According to the well-formedness conditions, each f_i is either an identifier, or a *wildcard*-expression.

$$\frac{\begin{array}{ll} & \langle p[i_1, \ldots, i_m, l_1, f_1, \ldots, f_n], \sigma, X\rangle \rightarrow_A v_1 \neq \emptyset \\ \langle e, \sigma, X\rangle \rightarrow_A k & \ldots \\ & \langle p[i_1, \ldots, i_m, l_k, f_1, \ldots, f_n], \sigma, X\rangle \rightarrow_A v_k \neq \emptyset \end{array}}{\langle p[i_1, \ldots, i_m, *(e), f_1, \ldots, f_n], \sigma, X\rangle \rightarrow_A \bigcup_{j=1}^{k} \{v_j\}}$$

where $0 \leq l_j < \mathit{card}(p, m+1)$ and $l_{j_1} = l_{j_2} \Rightarrow j_1 = j_2$.

That is, if the i-th index of a consumed value subscript is a *wildcard*-expression with k possible matches, then the consuming method receives k values, with no restriction on the i-th index of each received value.

Object expressions

An object reference expression evaluates to the *state* of the specified member of a collection of objects:

$$\langle A[\mathbf{y}]\{\vec{i}/\mathbf{y}\}, \sigma, X\rangle \rightarrow_O X_A(\vec{i}\,)$$

This is done by substituting an evaluated integer for every term in the subscript of the reference. In the above, $A \in \mathcal{O}(\pi)$.

Object collection index space expressions

In the base case, an object collection index space expression evaluates to a strided, integer range:

$$\frac{\langle e_1, \sigma, X\rangle \rightarrow_A i_1 \quad \langle e_2, \sigma, X\rangle \rightarrow_A i_2 \quad \langle e_3, \sigma, X\rangle \rightarrow_A i_3}{\langle \{e_1 : e_2 : e_3\}, \sigma, X\rangle \rightarrow_I \{i \ \in \mathbb{N}_0.\ i_1 \leq i \leq i_2 \text{ with } (i \bmod i_3) = 0\}}$$

The '$*$' operator can be used to obtain the Cartesian product of index spaces for object collections:

$$\frac{\langle \Psi_1, \sigma, X\rangle \rightarrow_I \psi_1 \quad \ldots \quad \langle \Psi_m, \sigma, X\rangle \rightarrow_I \psi_m}{\langle \Psi_1 * \ldots * \Psi_m, \sigma, X\rangle \rightarrow_I \psi_1 \times \ldots \times \psi_m}$$

Finally, an expression that is an implicit reference to the index space underlying an object collection, namely ispace(A) for some $A \in \mathcal{O}(\pi)$, contains the index of every object in A. If A is dense, then the set of all objects in it is specified in the declaration of the collection. If A is sparse, the `initialization` section of the *Charisma* program is used to populate A. In this case, ispace(A) refers to the set of indices of all objects inserted into A.

The evaluation of any type of expression is free of side effects, including the update of the value store, or of the serial (object) state of the program.

3.8.3 Execution of statements

Charisma follows program order semantics: control is transferred from one statement in program to the one that succeeds it in program order, The meaning of each type of *Charisma* statement is given below, in terms of the semantic operator '$\rightarrow$' over pairs of program configurations.

Sequential composition

Here the operator ';' is part of the *abstract*, not concrete, syntax of a *Charisma* program. As such, it is a semantic object, and not a syntactic one. In the discussion that follows, the semicolon operator will be used to signify the sequence of operations:

1. Sequential execution:

$$\frac{\langle S_1, \sigma, X\rangle \rightarrow \langle S_1', \sigma', X'\rangle}{\langle S_1; S_2, \sigma\rangle \rightarrow \langle S_1'; S_2, \sigma', X'\rangle}$$

2. Null execution:

$$\langle \text{skip}; S, \sigma, X\rangle \rightarrow \langle S, \sigma, X\rangle$$

That is, statements compose together: the combined effect of executing two consecutive statements is the same as the effect of executing the first, and

then the second. A special type of statement, namely skip, has no effect on the state of the program.

Conditional execution

A conditional branching of control is executed by first evaluating the associated predicate expression; Depending on this result, either the `then` or the `else` branch is followed:

1. Following the `then` branch:

$$\frac{\langle e, \sigma, X\rangle \rightarrow_B T}{\langle \text{if}(e)\ S_1\ \text{else}\ S_2, \sigma, X\rangle \rightarrow \langle S_1, \sigma, X\rangle}$$

2. Following the `else` branch:

$$\frac{\langle e, \sigma, X\rangle \rightarrow_B F}{\langle \text{if}(e)\ S_1\ \text{else}\ S_2, \sigma\rangle \rightarrow \langle S_2, \sigma, X\rangle}$$

Iteration using the `for` construct

There are two cases that we must consider:

1. If the value of the 'begin' expression is less than or equal to the value of the `for` loop's 'end' expression, then one instance of the body S is executed. Every occurrence of the loop index variable in S is substitued by i, namely the value of the 'begin' expression. This statement is composed with a modified `for`-loop, using the ';' operator, as shown below:

$$\frac{\langle e_1, \sigma, X\rangle \rightarrow_A i_1 \quad \langle e_2, \sigma, X\rangle \rightarrow_A i_2 \quad \langle S\{i_1/I\}, \sigma, X\rangle \rightarrow \langle S', \sigma', X'\rangle}{\langle \text{for}(I = e_1 : e_2 : e_3)\ S, \sigma, X\rangle \rightarrow \langle S'; \text{for}(I = e_1 + e_3 : e_2 : e_3)\ S, \sigma', X'\rangle}$$

   if $i_1 \leq i_2$.

2. On the other hand, if the value of the 'begin' expression exceeds that of the 'end' expression, then we move to the next statement after the

`for` loop in program order.

$$\frac{\langle e_1, \sigma, X\rangle \rightarrow_A i_1 \quad \langle e_2, \sigma, X\rangle \rightarrow_A i_2}{\langle \text{for}(I = e_1 : e_2 : e_3)\ S, \sigma, X\rangle \rightarrow \langle \text{skip}, \sigma, X\rangle}$$

if $i_1 > i_2$.

Iteration using the `while` construct

The `while` statement involves the evaluation of an associated continuation predicate. The statements inside the loop's body are executed as long as this predicate evaluates to *true*

1. Continue execution of body:

$$\frac{\langle e, \sigma, X\rangle \rightarrow_B T \quad \langle S, \sigma, X\rangle \rightarrow \langle S', \sigma', X'\rangle}{\langle \text{while}(e)\ S, \sigma, X\rangle \rightarrow \langle S'; \text{while}(e)\ S, \sigma', X'\rangle}$$

2. Exit loop:

$$\frac{\langle e, \sigma, X\rangle \rightarrow_B F}{\langle \text{while}(e)\ S, \sigma, X\rangle \rightarrow \langle \text{skip}, \sigma, X\rangle}$$

The `foreach` construct

A `foreach` statement has an associated object collection index space Ψ, a list of `foreach`-bound variables, $\mathbf{y}$, which serve to identify members of Ψ, and a *filter* predicate, Q. The `foreach` encloses a single publish-consume statement, S_P, P being the set of value spaces published by it.

The `foreach`-statement specifies the sequential composition of all instances of S for which Q is satisfied:

$$\frac{\begin{array}{c}\langle Q\{\vec{i_1}/\mathbf{y}\}, \sigma, X\rangle \rightarrow_B T \\ \cdots \\ \langle Q\{\vec{i_n}/\mathbf{y}\}, \sigma, X\rangle \rightarrow_B T\end{array} \qquad \begin{array}{c}\langle \Psi, \sigma, X\rangle \rightarrow_I \psi \\ \forall \vec{j} \in \psi \setminus \{\vec{i_1}, \ldots, \vec{i_n}\}.\ \langle Q\{\vec{j}/\mathbf{y}\}, \sigma, X\rangle \rightarrow_B F\end{array}}{\langle \text{foreach } (\mathbf{y} \text{ in } \Psi : Q(\mathbf{y}))\ S_P, \sigma, X\rangle \rightarrow \langle \chi(P); S_P\{\vec{i_1}/\mathbf{y}\}; \ldots; S_P\{\vec{i_n}/\mathbf{y}\}, \sigma, X\rangle}$$

Let the object index space expression Ψ evaluate to the set of integer vectors ψ. Then, the `foreach` statement sequentially composes distinct, term-substituted statement instances $S_P\{\vec{i_1}/\mathbf{y}\}, \ldots, S_P\{\vec{i_n}/\mathbf{y}\}$, where $\{\vec{i_1}, \ldots, \vec{i_n}\}$

is a subset of ψ, and for each $\vec{i_k}$, the filter predicate Q evaluates to true. The set of such $\vec{i_k}$'s is maximal, in the sense that for no other $\vec{j} \in \psi$ does Q evaluate to true.

The symbol 'χ' denotes the clearing of value spaces, as defined below:

1. The statement $\chi(p)$, defined for $p \in \mathcal{V}(\pi)$, denotes the *clearing* of all locations in p, i.e.

$$\langle \chi(p), \sigma, X \rangle \to \langle \text{skip}, \sigma', X \rangle$$

   where $\forall q \in \mathcal{V}(\pi) : q \neq p.\ \sigma'_q = \sigma_q$, and $\forall \vec{i} \in \text{ispace}(p).\ \sigma'_p(\vec{i}) = \perp$.

2. We define the effect of $\chi(P)$ for sets $P = \{p_1, \ldots, p_k\} \subseteq \mathcal{V}(\pi)$ as follows:

$$\langle \chi(P), \sigma, X \rangle \to \langle \chi(p_1); \ldots ; \chi(p_k), \sigma, X \rangle$$

Therefore, before executing any instances of the publish-consume statement S_P, enclosed within a `foreach`-statement, we clear all value spaces that are published by it. This nullifies the effect of any previous publish-consume statements on the stores σ_p, for $p \in P$.

Publish-consume statements

Consider a publish-consume statement S_P of the form

$$(p_1[\mathbf{E_1}(\mathbf{y})], \ldots, p_n[\mathbf{E_n}(\mathbf{y})]) \leftarrow A[\mathbf{y}].g(e_1, \ldots, e_m)$$

where $P = \{p_1, \ldots, p_n\}$, and the $\mathbf{E}$ symbols represent vectors of well-formed expressions of the `foreach`-bound variables $\mathbf{y}$.

Then the evaluation of S_P proceeds as follows. Consumed value subscript expressions $e_1, \ldots, e_m$ are evaluated, as in 3.8.2. Then, the object expression $A[\mathbf{y}]\{\vec{i}/\mathbf{y}\}$ is evaluated, as in 3.8.2, yielding α. If any of the consumed values evaluates to $\emptyset$, i.e. the location corresponding to the value in the semantic store has not yet been published, an **error** results, and the computation terminates. If all consumed values are well-defined, the function g is invoked with arguments α, and the evaluated consumed value expressions, namely

$v_1, \ldots, v_m$. Formally,

$$\frac{\begin{array}{ll}\langle e_1\{\vec{i}/\mathbf{y}\}, \sigma, X\rangle \rightarrow_A v_1 & \langle \mathbf{E_1}\{\vec{i}/\mathbf{y}\}, \sigma, X\rangle \rightarrow_A \vec{j_1} \\ \ldots & \ldots \\ \langle e_m\{\vec{i}/\mathbf{y}\}, \sigma, X\rangle \rightarrow_A v_m & \langle \mathbf{E_n}\{\vec{i}/\mathbf{y}\}, \sigma, X\rangle \rightarrow_A \vec{j_n} \\ \multicolumn{2}{c}{\langle A[\mathbf{y}]\{\vec{i}/\mathbf{y}\}, \sigma, X\rangle \rightarrow_O \alpha}\end{array}}{\langle S_P\{\vec{i}/\mathbf{y}\}, \sigma, X\rangle \rightarrow \langle \text{skip}, \sigma', X'\rangle}$$

In the above, we have used the following notation:

1. $g_{seq}(\alpha, v_1, \ldots, v_m) = \alpha'$, i.e. the user-defined serial method g modifies the state of the object on which it is invoked, according to C++ semantics.

2. $X'_A = X_A[\alpha'/\vec{i}\,]$, i.e. the above change in state is recorded in the set of all object states, for the appropriate member of the collection A.

3. $\forall k \in \{1, \ldots, n\}$. $g_{pub,\,k}(\alpha, v_1, \ldots, v_m) = \beta_k$, i.e. the invocation of serial method g yields n published values (due to invocations of the `produce`/`reduce` methods on `PublishedValue`'s and `ReducedValue`'s). The k-th value published by this invocation is denoted β_k.

4. If $\exists k.\ \sigma_{p_k}(\vec{j_k}) \neq \emptyset$, then an **error** results, halting the computation. The rationale behind this rule is that we want to prevent *simultaneous* updates to a particular value. Recall that we had cleared all value spaces published by the enclosing `foreach` statement prior to executing any instances of the enclosed publish-consume statement. Therefore, any non-$\emptyset$ values within p_k must have resulted from a publication performed by a preceding instance of the current publish-consume statement.

5. $\forall k \in \{1, \ldots, n\}$. $\sigma'_{p_k} = \sigma_{p_k}[\beta_k/\vec{j_k}]$, i.e. the published values are used to modify the appropriate locations in the value store, σ, thus yielding σ'.

3.9 Examples of applications written in *Charisma*

Now that we have an understanding of the programming constructs provided by *Charisma*, we examine the expression of two HPC applications

with data-independent data flow. We aim to demonstrate that code written in *Charisma* is compact and abstract, and most importantly, is able to capture the overall parallel structure of the expressed algorithm.

3.9.1 Dense LU decomposition

First we consider an example of section multicasts. The *Charisma* code below captures the global flow of data and control for the dense LU decomposition algorithm [48]. It uses a different object index space argument in each iteration of the algorithm to address the upper and lower active panels, as well as the objects corresponding to the trailing matrix.

```
parameter N, g : int;

class LUclass;
objects A : LUclass[N/g,N/g];

value du, dl : Matrix;
values u, l : Matrix[N/g];

for(K = 0:N/g-1){
  (du, dl) <- A[K,K].factorize();

  foreach(j in {K+1:N/g-1}){
   (u[j]) <- A[K,j].utri(du);
   (l[j]) <- A[j,K].ltri(dl);
  }

  foreach(i,j in {K+1:N/g-1}*{K+1:N/g-1})
    A[i,j].update(l[i],u[j]);
}
```

We first declare an object collection A, whose members encapsulate portions of the matrix to be factorized, as well as the functionality for doing so. The granularity of these objects can be controlled using the parameter `g`. The decomposition is phrased as a series of iterations. In iteration K, the current diagonal object $A[K, K]$ performs a factorization of the diagonal tile, and produces two triangular tiles, *du* and *dl*. These values are consumed

by the upper and lower active panel objects, respectively. The result is an implied multicast of *du* from $A[K, K]$ to $A[K, j]$, and of *dl* from $A[K, K]$ to $A[j, K]$, where j goes from $K + 1$ to $N/g - 1$ in either case. The active panel objects then produce values that are consumed by the trailing matrix objects. Each $A[i, j]$ in the trailing matrix, where $(i, j) \in \{K+1 : N/g-1\}^2$, consumes the tile in its column of the upper active panel, which is generated by $A[K, j]$. This means that the value published by a single upper active panel object $A[K, j]$, is consumed by several trailing matrix objects $A[i, j]$, where $i \in \{K+1 : N/g-1\}$. Therefore, $A[K, j]$ multicasts its tile to objects $A[K+1 : N/g-1, j]$, for each $j \in \{K+1 : N/g-1\}$. Similarly, lower active panel tiles are multicast by $A[j, K]$ to objects $A[j, K + 1 : N/g - 1]$, again for each $j \in \{K + 1 : N/g - 1\}$.

The *Charisma* code for the LU algorithm is compact and clearly captures its overall flow of control and data. However, we one cannot express the data-dependent *pivoting* operation which is essential to numerical stability. For this, one would have to apply the interoperation techniques developed in § 6 and write the pivoting code externally to *Charisma*.

3.9.2 Cutoff-based Molecular Dynamics

Next, we present a molecular dynamics code that serves as a good example of the applicability of the language to "real-world" HPC. The code is written after the algorithm of Phillips *et al.* [7]. It performs a fine-grained decomposition of both the input data (i.e. ensemble of simulated atoms) and the parallel work (of computing the Newtonian interactions between atoms). The code highlights an important advantage of the object-based paradigm of *Charisma*, namely that it allows the separation of the entity encapsulating work (i.e. objects) from the processing element (e.g. thread, processor, etc.) on which it is executed.

As input, the code accepts a large ensemble of atoms (in two-dimensional space) subject to pairwise interactions amongst themselves. These atoms are decomposed over a two-dimensional collection of objects called `Patches`. Each patch contains the atoms that lie within its spatial extent.

Force computations are performed by a four-dimensional collection of objects, called `Computes`. Only those `Computes` exist that are necessary to cal-

culate all the pairwise interactions in the system. In the cutoff-based scheme used for short-range forces, only the interactions between atoms of geometrically neighboring patches are computed. Therefore, the collection `Computes` is sparse, containing object `Computes[x1,y1,x2,y2]` iff `Patch[x1,y1]` and `Patch[x2,y2]` are proximal.

The algorithm proceeds as follows: In every iteration, each `Patch[x1,y1]` multicasts its atoms to all the computes that calculate forces on its atoms. Therefore, the set of computes to which `Patch[x1,y1]` multicasts its atoms, comprises all and only those `Computes[x1,y1,*,*]` and `Computes[*,*,x1,y1]` that are present in the object collection. When a compute receives the atoms from its two patches, it calculates their pairwise interactions, yielding a vector of forces on each atom in the two sets. These forces are reduced over all computes associated with a patch, so as to obtain the net force on each atom in the path. The patch then uses this net force to integrate the trajectory of each of its particles over a short span of time.

```
parameter N, NIterations : int;

class PatchClass, ComputeClass;
type Atom, Vector;

objects Patch: PatchClass[N,N];
sparse objects Computes : ComputeClass[N,N,N,N];

values atoms : Atom[N,N];
values f1, f2 : Vector[N,N];

initialize {
  foreach (x,y in {0:N-1}*{0:N-1}){
    insert(Computes[x, y, ((x-1) + N)%N, ((y-1) + N)%N],
           Computes[x, y, ((x-1) + N)%N, y],
           Computes[x, y, ((x-1) + N)%N, (y+1)%N],
           Computes[x, y, x, y],
           Computes[x, y, x, (y+1)%N]);
  }
}
```

In the *Charisma* code for the algorithm, we begin with the declaration of

a dense two dimensional `Patches` collection, and a sparse, four-dimensional collection `Computes`. Recall that initially, a sparse collection has no member objects. We use the *Charisma* `initialization` section to insert one `Computes` object for each pair of interacting `Patches`. The code shows that `Patches[x,y]` only inserts `Computes[x,y,*,*]`. However, the neighbors of `Patches[x,y]`, i.e. `Patches[x-1,y]`, `Patches[x+1,y]`, etc. insert `Computes` with indices of the form `[*,*,x,y]`. The implementation of this insertion is distributed, so that insertion does not become a bottleneck for large simulations.

```
orchestrate {
  for (I = 0 : NIterations-1){
    foreach (x,y in {0:N-1}*{0:N-1})
      (atoms[x,y]) <- Patches[x,y].sendAtoms();

    foreach (x1,y1,x2,y2 in ispace(Computes))
      (+f1[x1,y1],
       +f2[x2,y2]) <- Computes[x1,y1,x2,y2].forces(atoms[x1,y1],
                                                   atoms[x2,y2]);
    foreach (x,y in {0:N-1}*{0:N-1})
      Patches[x,y].integrate(f1[x,y], f2[x,y]);
  }
}
```

Every iteration of the main simulation code begins with the publication of `atoms` position information by the `Patches`. Each `Patches[x,y]` publishes value `atoms[x,y]`. This value is read by `Computes[x,y,*,*]`, and also by `Computes[*,*,x,y]`, so that the *Charisma* compiler infers two multicasts emanating from each `Patches[x,y]`.

The next `foreach` statement invokes the `forces` method on all `Computes`, signified by the object index space `ispace(Computes)`. A `Computes` object with index `[x1,y1,x2,y2]` waits for the receipt of two `atoms` values, and thereafter computes the pairwise interactions of the two sets of atoms. In this manner, each `Computes` object obtains a `Vector` of forces on each atom in `atoms[x1,y1]` and `atoms[x2,y2]`. The `Computes` object contributes these forces to two reductions. Respectively, the forces on the first set of atoms are reduced over all `Computes[x,y,*,*]`, i.e. the set of computes that were inserted by `Patches[x,y]`, and the second reduction occurs over the set of all

`Computes[*,*,x,y]`, namely the computes that were inserted by neighbors of `Patches[x,y]`. These forces are received by the patches, and used to update atom trajectories.

3.10 A look at prevalent programming languages for HPC

Having specified the *Charisma* language, we briefly compare it to prevalent programming languages in the HPC arena.

3.10.1 MPI

The message-passing model, as embodied by MPI [49], has been a mainstay of large-scale parallel programming. MPI features a simple, process-centric execution model (although the semantics of some of its communication routines can be subtle), portability, and access to low-level features. However, it has poor support for modularity: (Tightly-coupled) multi-module computations require temporal- or spatial-decomposition of the underlying hardware resources. By contrast, message-driven execution of translated *Charisma* code enables efficient, and transparent composition of parallel modules. Moreover, *Charisma* leverages Charm++'s object-migration-based dynamic load balancing infrastructure; a similar dynamic scheme would have to be written from scratch for MPI.

Exploiting intra-node parallelism is an issue for both MPI and *Charisma*, although with MPI-2 and beyond, the standard has seen greater support for multithreading. Nevertheless, a common idiom when programming with MPI is to exploit intra-node parallelism using OpenMP or pthreads, and perform inter-node communication using MPI calls.

3.10.2 PGAS languages

UPC and CAF are languages for the Partitioned Global Address Space (PGAS) paradigm. As such, they provide productivity improvements over MPI by simplifying communication and by incorporating data distributions

as first class entities. Given their SPMD structure, programs written in PGAS languages maintain an unfragmented view of the global flow of control, although one often finds differentiation of behavior for threads based on their identities. Something similar can be achieved with *Charisma*'s `foreach`-associated filter expressions.

UPC [6] is basically an augmentation of C with an explicitly parallel execution model, operations for efficient RMA [50] between partitioned global address spaces, shared and global data pointers, and synchronization operations.

CAF [51] is a basic enhancement of Fortran, including an SPMD model, so-called co-arrays, and barriers. A version of CAF from Rice University, named CAF 2.0 [5], is a more thorough redesign of Fortran. It includes support for process subsets (or teams), global pointers, asynchronous copying and collectives, synchronization at various levels: cofence (local barrier for asynchronous operations) and finish (team-wide asynchronous operation barrier).

In comparison, *Charisma* provides asynchrony transparently; it has a simpler semantics for synchronization between tasks through publication and consumption. Therefore, it does not require constructs such as fences and barriers, which can have complex semantics, depending on the relaxation assumptions.

3.10.3 Chapel

Alongwith X10, Chapel [4] is one of the two major *asynchronous* PGAS languages. Chapel has a fork-join execution model, unlike *Charisma*'s coarse-object-based, data-driven model. It provides data parallelism through the `forall` construct, whereby granularity of iterations assigned to each thread is controlled by the runtime system, and a `coforall` construct, which should be used when the granularity of individual iterations of the loop is known to be coarse. Chapel enables dynamic task parallelism through `begin` and `cobegin` constructs. This is unlike in *Charisma*, where tasks or threads cannot be created dynamically. Similarly to *Charisma*, Chapel also provides `sync` variables for publish-consume dependencies. From the point of view of semantics, Chapel's `sync` variables resemble the M-structures of Id [52],

whereas *Charisma*'s publication and consumption is more akin to the scheme provided by Id's I-structures [53]. Like *Charisma*, Chapel provides language-level support for reductions. A rich set of idioms based on the intrinsic *scan* (inspired by ZPL [54]) construct is also available.

3.10.4 X10

X10 [3] adopts a task-based approach to the expression of parallelism. These tasks may be fine-grained, are spawned asynchronously (possibly remotely), and are typically synchronized by means of a distributed, completion detection algorithm. These task-centric constructs apply especially well to tree structured computations, as evidenced by recent results showing impressive scaling with the UTS benchmark. Locality of tasks can be specified using the `at` clause; provisions are made for atomic tasks as well. X10 also provides a host of lower-level, performance-oriented features, atomic remote-memory operations, and *teams* for processes. The X10 `foreach` is used to launch parallel instances of an enclosed statement, in much the same way as the *Charisma* `foreach` causes invocations of multiple, concurrent instances of an enclosed publish-consume statement. However, X10 has more flexibility and expressive power, since it allows the free nesting of these constructs. The *Charisma* language is restrictive in this regard, with the intent to provide a concise language with very few constructs and simple semantics.

3.10.5 CnC

Intel's CnC [55] framework provides a model of execution that has some marked similarities to that of *Charisma*. In CnC, serial methods called *steps* encapsulate stateless computation. Steps can publish and consume collections of immutable values (just as in *Charisma*), and can *prescribe* other steps by publishing tokens called *tags*. Data dependencies are resolved dynamically, making CnC more like Linda than *Charisma* in this regard. Unlike the object-based model of *Charisma*, which encourages locality-aware programming from the ground up, CnC incorporates add-on features for locality of data access. The CnC programmer may avail of the underlying TBB [56] runtime system's ability to do work stealing, or manually place tasks on

processors. Although a distributed memory implementation is available for CnC, performance results on such machines have been less impressive than those obtained on multicore systems [57].

3.11 Compiling global *Charisma* flows into local, message-driven specifications

Charisma code is translated into message-driven Charm++ by a compiler infrastructure based on ANTLR4 [58] and Java. The ANTLR system allows for a sufficiently flexible specification of the grammar of the language, and encourages a principled approach to compiler construction. The result is a modular source-to-source translator that comprises several phases, each of which is independent of every other. The first three phases are written in ANTLR4, and correspond closely to the sequence of activities performed during translation, namely (1) lexing, parsing and AST construction, (2) symbol table generation, (3) construction of the control flow graph (CFG). The next two phases, namely (4) dependency analysis and CFG annotation, and (5) Code generation, are expressed in Java. Since this is not meant to be an ANTLR tutorial, we will skip phases (1) and (2) of the compilation procedure. Below we discuss the remainder of our translation strategy, namely phases (3)–(5), considering, as necessary, the data structures and algorithms used therein.

For the remainder of this section, we will conduct our discussion of the *Charisma* translation strategy in the context of the example code shown in Figure 3.2. Although the code is simplistic, it serves to illustrate some interesting issues that arise when translating global data and control flow specifications into local ones. In the code, we have two object collections, `A` and `B`, whose members are instances of the class `MyClass`. In the outermost scope, namely that of the program itself, there are three statements: two `foreach` publish-consume statements invoked on members of object collection `A`, separated by a `while` statement, within which is nested a `foreach` publish-consume statement invoked on objects in collection `B`.

Intuitively, the translation procedure should emit code that causes the publication of values by members of `A` through the invocation of the sequential method `MyClass::f1()`. This statement *produces* values in value space p,

```
parameter N : int;
parameter ERR_TOL : double;

class MyClass;
objects A,B : MyClass[N];

values p : double[N];
value e : double;

foreach (i in {0:N-1})
  (+e, p[i]) <- A[i].f1();

while(e > ERR_TOL)
  foreach (i in {0:N-1})
    (+e, p[i]) <- B[i].f2(p[i]);

foreach (i in {0:N-1})
  A[i].f3(p[i]);
```

Figure 3.2: Example code. We discuss the compilation of this code fragment in this section.

and *reduces* a single value in value space e. The reduced value e determines whether, and for how many iterations, the `while` loop executes. If the `while` loop fails to execute even one iteration, the p-values published by members of `A` in the first `foreach` statement should be consumed by the `foreach` statement invoked on members of `A` after the `while` statement. On the other hand, if the `while` loop's continuation predicate evaluates to *true* even once, members of `B` should consume, and thereafter produce, p-values iteratively, until the predicate `e > ERR_TOL` doesn't hold anymore. Finally, the p-values published by the last invocation of `MyClass::f2()` on objects of `B` should be consumed by the invocation of `MyClass::f3()` on `A`. In the following, we present a series of procedures that that achieve such functionality in a general-purpose manner.

3.11.1 Syntax-directed CFG construction

We use standard, syntax-directed techniques to construct a control flow graph (CFG) from the *Charisma* orchestration source. Our CFG construction algorithms are straightforward adaptations of elementary compiler techniques,

and as such we do not describe them here.

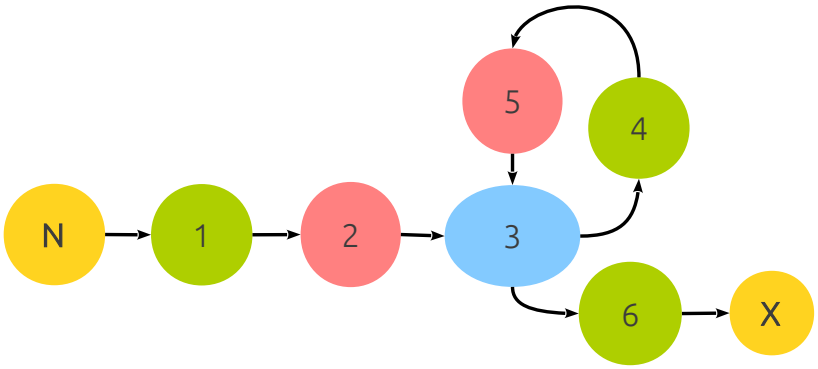


Figure 3.3: Control flow graph obtained from our example code in Figure 3.2. Notice the addition of the pink *value fetching* nodes: these are used to collect all the values required for the computation of the predicate associated with a control flow construct, which in our case is a (blue) `while` loop. Green nodes are `foreach` statements, and yellow ones are the entry (labeled N) and exit (X).

We do note, however, the introduction of so called *value-fetch* nodes for control flow statements whose predicates are dependent on the centralized evaluation of values (`if-else` and `while` statements). An example of such a node is shown in Figure 3.3.

3.11.2 Preparing the CFG for dependency analysis

The control flow graph is annotated with information to help match published values (defs) to consumed values (uses). Several publish-consume statements may publish to the same value space at different points in the program. Therefore, we must determine which *version* of the value space is used at each point of consumption in the program. For instance, in Figure 3.3 the two green nodes labeled 1 and 4 represent the publish-consume statements `A.f1` and `B.f2`, respectively. Both statements publish values to value space '`p`'. Node 6 (`A.f3`) consumes value space '`p`', but which *version* of '`p`' it consumes depends on the particular control flow path taken. That is, if the `while` loop's continuation criterion evaluates to `true` even once, `A.f3` (6) will consume the value published by `B.f2` (4). Otherwise, it will consume the value published by `A.f1` (1). So how can we automate this procedure of

inferring which version of a value to use at each consumption point in the program?

In general the problem we are trying to solve is the following: we are given a node n that lies on several control flow paths in the CFG. If n consumes a value v, and v is published along multiple paths that intersect at n, then which version of v should we use to feed the consumption at n? Another way to think of this is that given a path from the start node of the CFG to a node n, several nodes, both on and off this path might publish to a value space that is consumed by n. The compiler must determine, given the global state of the program, which version of the published value should be consumed by the publish-consume statement represented by n. There is no such problem if there exists only one node that publishes v between the start and n. Based on this insight, we use a straightforward adaption of the *static single assignment* (SSA) notation [59] to ensure that for each consumer of a value, there is only a single node in the CFG that publishes the value. This allows a single publishing statement to be determined (at run time) for each consumed value, and greatly simplifies the code generation procedure. We perform the following steps:

1. Dominator analysis.

2. Immediate dominator relation.

3. Computing dominance frontiers.

to obtain program metadata that enable us to match versions of published values either to consuming publish-consume nodes, or to so-called ϕ-nodes [59]. The *Charisma* compiler uses the strategy of Cooper, Harvey and Kennedy [60] (the CHK algorithm) for computing dominance frontiers. Briefly, dominator analysis is performed using standard data flow techniques [61], leading to the construction of the *immediate-dominator* tree. This tree is used by the CHK algorithm to iteratively grow the dominance frontiers of nodes in the CFG. The chief difference in our approach is that instead of adding new ϕ-nodes to the graph, we simply replace existing `while` or `if-else` nodes with their ϕ-variants.

Applying these techniques to the CFG in Figure 3.3, we obtain the ϕ-node-augmented CFG shown in Figure 3.4. The next step is to identify the

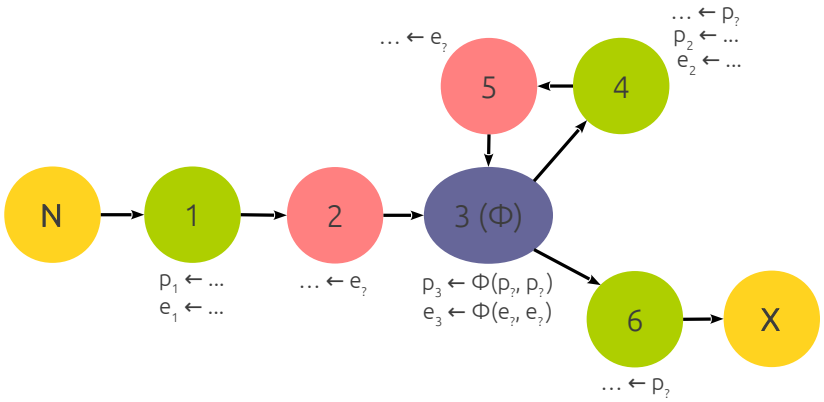


Figure 3.4: CFG after augmentation with ϕ-nodes. After applying the ϕ-node-placement algorithm to the annotated CFG in Figure 3.3, we see that a single node, 3, needs a preceding ϕ-node. In our formulation, we simply relabel node 3 as a "ϕ-`while`" node. Subscripts for consumed values are determined by the dependency analysis phase (*cf.* 3.11.3).

identities of the published values that may be used at every consumption site in the CFG.

3.11.3 Dependency Analysis

We use the concept of *reaching definitions* [61] to determine the (possible) data dependencies in a *Charisma* program. We say that a published value *reaches* a node n in the CFG if: (i) there exists a node p that publishes the value, and (ii) there is a control flow path from p to n that does not involve any intervening nodes that also publish a value with the same name. If these conditions hold, the value published by p *might* be the one consumed by node n. Notice that following the placement of ϕ-nodes, each non-ϕ node n will have only one value (which may have been published by a ϕ-node) that reaches it. Only for a ϕ-node is it the case that several values bearing the same name reach it. These values are the ones that must be merged, and the true published value that leaves the ϕ-node can only be determined at run time.

These ideas lead to Algorithm 1 for computing the dependencies in a *Charisma* program. We first compute the reaching definitions for each node in the CFG. The set of reaching definitions for node n is denoted $\mathrm{In}(n)$. The algorithm is structured as a data flow algorithm, with a fixpoint solution

Algorithm 1: *Charisma* dependency analysis.

ComputeDataDependencies(G, T)
Input: idom tree T for CFG G; def-use information for G.
Output: D, the set of data dependencies in the program corresponding to G.
begin
  $D \leftarrow \emptyset$;
  for $n \in N$ **do**
    $\text{In}(n) \leftarrow \emptyset$;
    $\text{Out}(n) \leftarrow \{(d, n) : d \in \mathit{defs}(n)\}$;
  end
  repeat
    for $n \in N$ **do**
      $\text{In}(n) \leftarrow \bigcup_{p \in \mathit{pred}(n)} \text{Out}(p)$;
      for $(d, k) \in \text{In}(n) : d \notin \mathit{defs}(n)$ **do**
        $\text{Out}(n) \leftarrow \text{Out}(n) \cup \{(d, k)\}$
      end
    end
  until *convergence*;
  for $n \in N$ **do**
    for $(d, k) \in \text{In}(n)$ **do**
      if $\exists\ u \in \mathit{uses}(n)\ :\ \mathit{Name}(d) = \mathit{Name}(u)$ **then**
        if *IsPhi*(n) **then**
          $D \leftarrow D \cup \{$**new** PhiDependency$((d, k), (u, n))\}$;
        else if *InterveningFlowDivergence*$(n, p, T, G) \vee$ *IsPhi*(k) **then**
          $D \leftarrow D \cup \{$**new** DynamicDependency$((d, k), (u, n))\}$;
        else
          $D \leftarrow D \cup \{$**new** StaticDependency$((d, k), (u, n))\}$;
        end
      end
    end
  end
end

InterveningFlowDivergence(n, k, T, G)
Input: nodes $n, k \in G$; idom tree T for CFG G; def-use information for G.
Output: whether there is an *unmatched* control flow node on idom path $k \rightsquigarrow n$.
begin
  $p \leftarrow \mathit{FindNodeInIdomTree}(T, n)$;
  $q \leftarrow \mathit{FindNodeInIdomTree}(T, k)$;
  foundMatchingMerge $\leftarrow$ *false*;
  while $p \neq$ nil **and** $p \neq q$ **do**
    if *IsMergeNode*(p) **then** *foundMatchingMerge* $\leftarrow$ *true* ;
    else if *IsControlFlowNode*(p) **then**
      if *foundMatchingMerge* **then** *foundMatchingMerge* $\leftarrow$ *false* ;
      else return *true* ;
    end
    $p \leftarrow \text{idom}(p)$;
  end
  return *false*;
end

obtained by iterating until convergence. The In set of a node is simply the union of all the Out sets of its predecessors. Moreover, In(n) is initially empty for all n, whereas Out(n) is initialized to a set of *augmented* defintions of n, i.e. the set of tuples (d, n) that mark the definition and also the node n that published the corresponding value. In each round of the iteration, a node is made to *filter* out the values that it receives in its In set from its predecessors: a value published by some control-flow ancestor with the same name as a definition made by the node itself, is *not* included in the Out set of the node. The rationale behind this exclusion is that the definition made by node n interferes with that made by its control flow ancestor, so we favor the more recently made definition over the old one.

Upon convergence of this computation, we obtain a stable set of definitions that reach each CFG node. The next phase of the algorithm matches these reaching definitions with the uses made by each node. Three kinds of data dependency are generated as a result of this process of matching:

Φ dependency. If the target of a data dependency is a ϕ-node, we term a data dependency a ϕ dependency. As we will see in § 3.11.7, such dependencies are not true data dependencies, and as such do not result in the communication of data between the source and target at run time. Instead, they are used to perform some book-keeping actions by an entity that manages the global state of the program. Such a dependency is used to determine, at run time, which among the consumed values of the target ϕ-node is to be fed to nodes that are downstream of the ϕ-node and dependent on the merged value published by it.

Dynamic dependency. A *dynamic* data dependency is one for which the target of the data dependency is *control* dependent on a node that is strictly dominated by the source of the data dependency. In our analysis we do not compute control dependencies, but are able to identify such dynamic dependencies all the same, using the *idom* tree. The *InterveningFlowDivergence*() routine traverses the *idom* tree in a bottom-up fashion in Algorithm 1, testing for the occurrence of a control flow node but no merge node, between the target of a dependency (where the traversal begins) and its source (where the traversal ends.) If (call this condition (a)) we *do* find an intervening control flow node on the *idom* tree path from the the source p to the target n, it means

that we are *not* guaranteed to visit n having visited p.[1] For the structured CFGs that we generate, invoking the *InterveningFlowDivergence*() routine is equivalent to looking for the most deeply nested control flow constructs that enclose a consuming node, and the node that has been identified as the publisher that feeds it, and testing for the equality of the two.

The other condition (call it condition (b)) under which we designate a dependency as dynamic is when the source of the dependency is a ϕ-node. In this situation the true identity of the statement that publishes the value appropriate for consumption at a given point in the CFG is determined at run time, so that we must wait until both the target of the dependency has been reached, and its source has been determined. The occurrence of either one of conditions (a) and (b) implies that upon publishing the corresponding value, the source must buffer it until it has been ascertained that the control flow path involving the target of the dependency will be executed. We will return to this issue in § 3.11.7.

Static dependency. If there are no intervening control flow nodes between the source and target of a dependency along the path that connects them in the *idom* tree, and the source of the dependency is not a ϕ-node, then we generate a *static* dependency. As the name suggests, these dependencies can be resolved at compile-time. This situation exhibits two characteristics: (i) if the target is visited, then the source must also have been visited (ii) the source is not a ϕ-node. Together these conditions imply that as soon as the source node has published its value, it can be communicated to the site of its consumption in the CFG. Although program correctness doesn't require that we distinguish this case from the dynamic one above, the distinction certainly improves efficiency of execution: Static dependencies do not require mediation by an entity that tracks global program state.

By applying Algorithm 1 to our example program, we obtain the CFG in Figure 3.5. Here the only non-ϕ CFG nodes that consume values, are 4 and 6, and the set of nodes that make definitions are node 1, which publishes value p_1, node 4, which publishes value p_2 and ϕ-node 3, which publishes p_3. It is evident that ϕ-node 3 will filter out the values published by nodes 1 and 3, only propagating its own value, p_3, to nodes that are downstream of it.

[1] This is precisely the condition for *postdominance*, which we do not discuss in this thesis. The interested reader is referred to the work of Cytron *et al.* [59] for a thorough discussion of the subject.

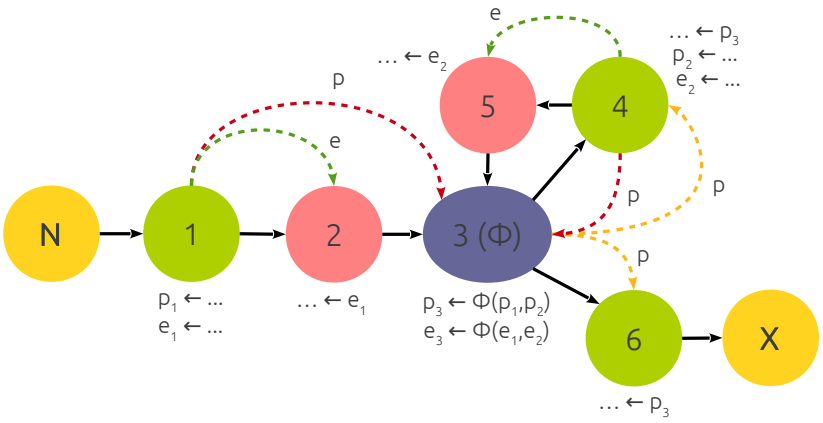


Figure 3.5: CFG annotated with data dependencies. Dependencies are shown here as colored, dashed arrows: Φ dependencies (due to the consumption of values published by 1 and 4 by the ϕ-node, 3) are shown in red, and dynamic dependencies (due to the consumption of values published by the ϕ-node, 3) in yellow. Static dependencies occur due to the publication of e by straightline statements, and are shown in green.

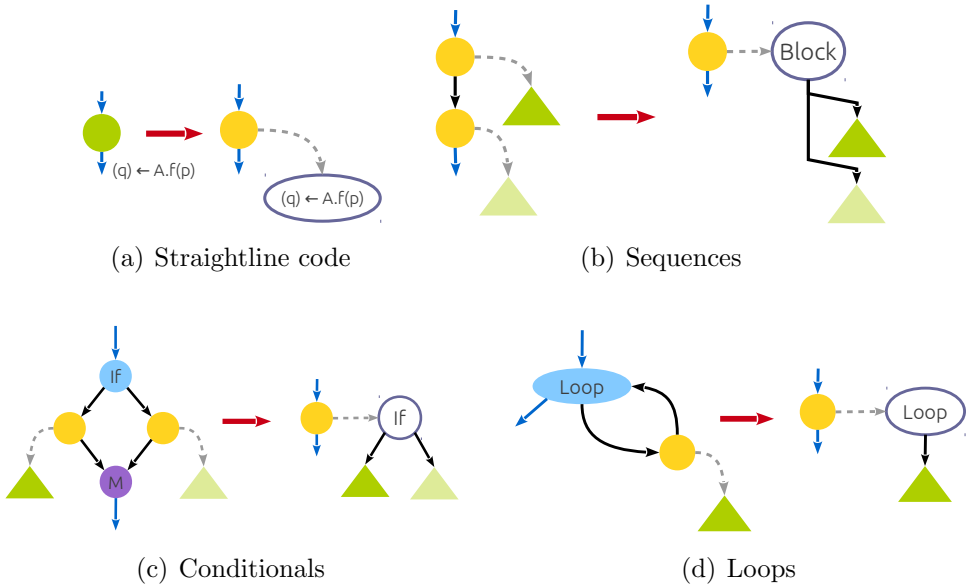
Therefore, only definition $(p_3, 3)$ reaches nodes 4 and 6. Since the source of the dependency is a ϕ-node in each case, the dependencies from ϕ-node3 to 4 and 6 are both marked as dynamic (yellow, dashed lines).

Now consider the definitions made by nodes 1 and 4. These reach ϕ-node 3, but do not propagate any farther, because ϕ-node 3 publishes its own value that *interferes* with the value of either one. Since the target of these dependencies is a ϕ-node, we annotate the graph with ϕ-dependencies, shown as red dashed lines between nodes 1 and 3 and 4 and 3, respectively.

3.11.4 Graph contraction: preparing for code generation

For reasons of performance, compactness and analyzability of generated code (as described in earlier work by the author [62]), we are interested in the generation of *structured* SDAG (*cf.* § 2.5 and the work of Kale and Bhandarkar [32]) code to describe the message-driven behavior of *Charisma* objects. Therefore, we must first reconstruct the original hierarchical representation of the *Charisma* program from its augmented CFG. One can imagine this phase as doing the reverse of the syntax-directed CFG construction. Whereas in the latter we obtained a CFG from the AST, here we recover a nested hierarchy of statements from a (different) CFG.

Our algorithm is structured as a series of iterations, in each of which we check whether the application of a certain *contraction rule* is permissible. The application of a contraction rule is allowed if all the nodes in the subgraph that it aims to contract, are *ready*. Initially, we mark all nodes as *not-ready*; the application of allowed contraction rules changes the readiness state of nodes, and allows the application of other rules, and so on. Once a node becomes ready for contraction, it never returns to the not-ready state again. If we identify a contraction rule applicable to a certain subgraph within the CFG, we replace that subgraph with a single, ready node. We make specific updates to the hierarchy of statements in order to record the statement represented by the subgraph that has just been removed from the CFG.



(a) Straightline code (b) Sequences

(c) Conditionals (d) Loops

Figure 3.6: CFG contraction rules. Each panel shows on the left hand side of a red rightward arrow, the state of the graph before the application of the corresponding rule, and on its right hand side, the structure of the graph after the application of the rule. Nodes that are ready for contraction are marked yellow. Initially, only publish-consume and foreach nodes are ready. Each application leads to the *contraction* of a subgraph within the CFG, and produces a subtree of the code hierarchy. Repeated application of these contraction rules yields a single node, which encapsulates the code hierarchy for the entire program.

Applying these rules to the CFG for our running example (Figure 3.5), we obtain the statement hierarchy shown in Figure 3.7. In this tree represen-

tation, sibling nodes form, reading from left to right, a sequence of (nested) statements within a block.

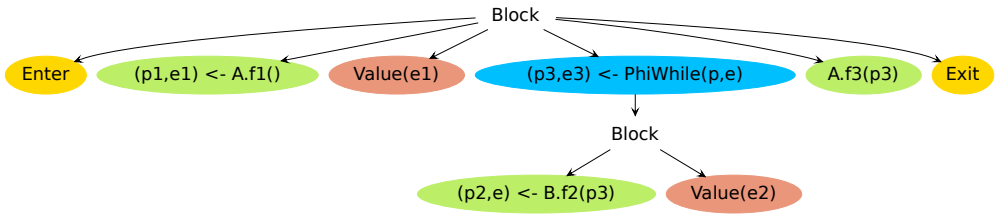


Figure 3.7: The statement hierarchy obtained by the repeated application of the contraction rules in Figure 3.6, on the graph in Figure 3.5.

3.11.5 Code generation

There are three key components to the code generation phase:

1. Obtaining the local control flow of each object collection, which encapsulates both the data dependencies on values consumed by members of the collection, and the program order dependencies implied by the sequential occurrence of multiple method invocations on a single object collection.

2. Generating a centralized entity called the *global state manager*, which records the particular control flow path being followed in the program. This is done by managing dynamic dependencies and determining the true identity of values published by ϕ-nodes.

3. Generating messaging code, in the form of asynchronous invocations of Charm++ entry methods on objects, so as to communicate published values generated within sequential code, to appropriate targets. Targets are determined either statically, or dynamically, by the global state manager.

Translating global control flow into local control flow

The contraction algorithm above yields a hierarchy of statements representing the *Charisma* program. In order to obtain from this *global* specification,

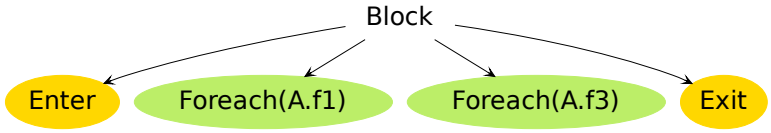
a representation suitable for locally-specified, structured, message driven object code, we *project* the global hierarchy onto the object collections that constitute the program.

We use a simple algorithm for this purpose, based on the following inductive rules: (i) as the base case, a publish-consume statement only projects onto the object collection on which it is invoked; (ii) in order to project a block of statements, we examine each statement in the block. This process of examination can be performed recursively, to account for statements embedded within control flow constructs. If there are no statements in the block that project onto the object collection in question, then the block itself doesn't project onto the object; (iii) an `if-else` construct is projected onto an object collection if at least one of the `then` and `else` blocks projects onto the collection; (iv) similarly, a loop construct (`for` or `while` loop) projects onto an object collection if and only if the body of the loop projects onto the object. The algorithm also accounts for the projection of statements that fetch values for the evaluation of control flow predicates. Whereas such a statement projects onto every object, if no other statement in a block projects onto the object collection at hand, then the block does not project onto the collection either.

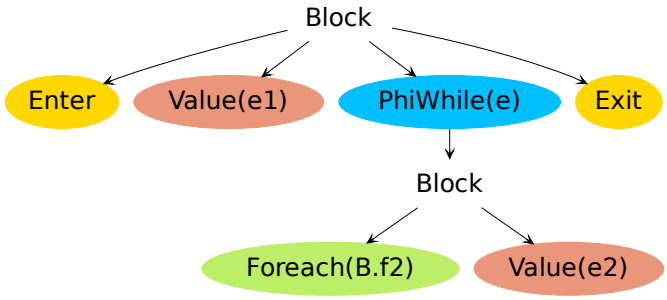
Returning to our running example, projections of the statement hierarchy in Figure 3.7 onto the two object collections used in the *Charisma* program, are shown in Figure 3.8. It is worth noting that since no *substantive* statement (other than the one that fetches the values required for predicate evaluation) within the `while` loop projects onto collection `A`, the `while` loop doesn't project onto it either.

3.11.6 Emitting SDAG from statement hierarchies

Once we have the hierarchical representation of the local control flow of each object collection in the program, we can proceed to emit *structured*, message-driven code for the classes corresponding to these collections. As mentioned previously, we emit SDAG clauses, which embody an unfragmented and compact representation of the message-driven control flow local to each object collection. The procedure for this translation is described pictorially in the following figures.



(a) Collection '`A`'



(b) Collection '`B`'

Figure 3.8: Statement hierarchy projections, as obtained by projecting the hierarchy in Figure 3.7 onto object collections '`A`' and '`B`'.

Figure 3.9 shows the SDAG code generated for a tree node corresponding to a publish-consume statement in the original *Charisma* source. In SDAG, data dependencies are expressed through the `when` construct. Dependencies can be conjoined by simply including a comma-separated list of method invocations that occur upon the receipt of the corresponding messages. In the figure at hand the SDAG code states that in order to invoke `serial` function `f()`, the object in question must first receive n messages in the form of entry method invocations $recv\_p_1, \ldots, recv\_p_n$. Serial method '`f`' may itself publish values, which are represented by the "handles" passed into the method along with the values that the method consumes. These handles are instances of the class template `Charisma::PublishedValue<T>`.

Figure 3.10 shows the code emitted for a a `foreach` statement. The SDAG `if` statement in the generated code allows the embedded publish-consume statement to be invoked on only those objects whose indices satisfy two conditions simultaneously: (i) the identifying index of the object must belong to the index space referenced in the `foreach` construct, and (ii) the *filter* predicate Q of the `foreach` construct must evaluate to true for that object.

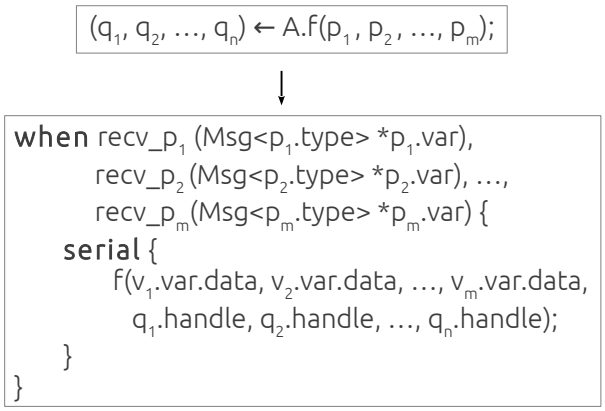


Figure 3.9: Translating a publish-consume node in the statement hierarchy to SDAG.

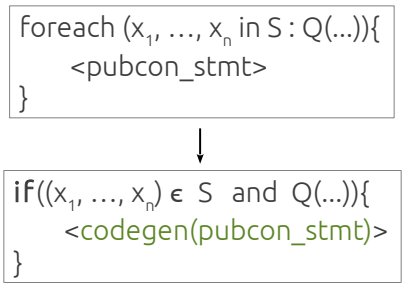


Figure 3.10: A `foreach` clause with an embedded publish-consume statement.

Recall that this predicate can be an expression over the indices of the object, and within-scope `for`-loop index variables. Therefore, it may evaluate to different Boolean values on different objects.

As shown in Figure 3.11 the generation of code for a block of statements entails the sequential composition of the SDAG code generated for each subtree in the block in turn.

The code generation procedure is called recursively on the body of a `for` loop. The resulting generated code is enclosed within an SDAG `for` loop that iterates over an appropriate range of integers.

We must be careful in generating code for `while` and `if-else` subtrees. Recall that the predicates associated with these control flow constructs may use values published by other statements. In general, these published values are not available at all objects that depend on the value of the predicate. Therefore, we task a special entity called the global state manager (GSM, *cf.* § 3.11.7), with the centralized evaluation of all control flow predicates.

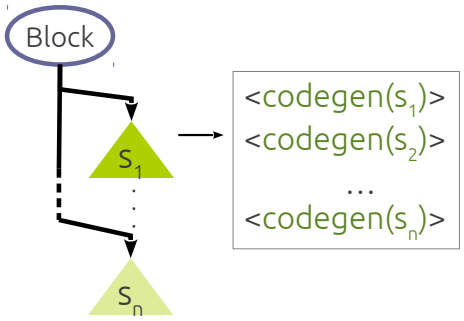


Figure 3.11: Sequential generation of SDAG for a block of statements.

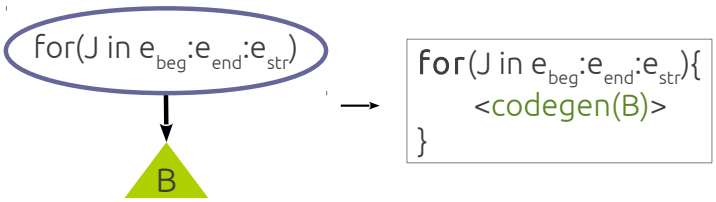


Figure 3.12: A `for` loop and its body of statements, `B`.

Objects that require the most recent Boolean value of the predicate are simply sent this value by the global state manager. We will discuss the GSM shortly. Here we concern ourselves with code generation for object collections that require evaluated data-dependent control flow predicates. Figure 3.13 shows the SDAG code generated when a "Value" subtree is encountered. It specifies a dependency on the evaluated predicate, which is received from the GSM. Upon receipt, this value is stored for later use in determining the appropriate local control flow path.

The predicate value associated with a `while` loop is obtained from the GSM, as seen above. All that remains, then, is to generate the SDAG code for the body of the `while` loop, and an enclosing, object-local SDAG `while` loop.

A similar strategy is followed for `if-else` subtrees.

3.11.7 Maintaining global program state

We saw in § 3.11.3 that in a *Charisma* program, certain data dependencies could be identified as being *static*. Intuitively, this means that as soon as the publisher (i.e. the source of the dependency) has finished publishing

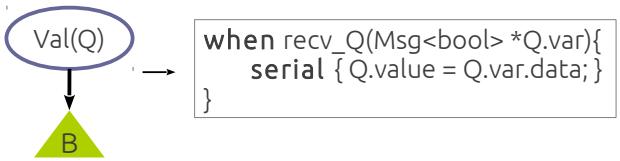


Figure 3.13: The SDAG for a *Value* statement declares a data dependency on the evaluated predicate associated with a nearby control flow construct.

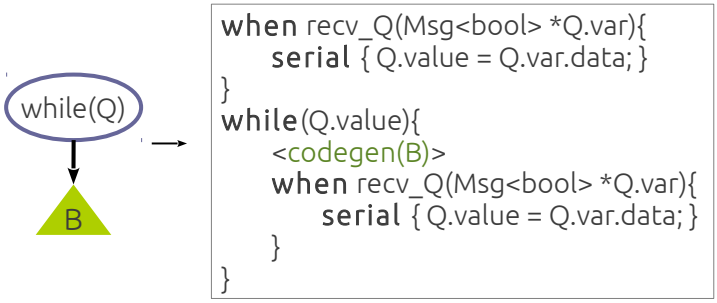


Figure 3.14: A `while` loop and the block of statements, `B`, that it encloses.

the value, it can be sent to the consumer (the target of the dependency). However, dependencies of the *dynamic* variety can only be resolved at run time. That is, only when it is ascertained that the target of the dependency will indeed execute, can the source send its published value to the target. Therefore, information regarding the global state of the program, namely which control flow paths have been activated in the past, must be maintained.

For the purpose of the discussion that follows, we recall that in the data dependency analysis phase, every publication of a value space was given a unique numeric label, denoting the *version* of the value space published by it. We will simply use the term 'version' when the value space referred to is clear from the context. The GSM performs book-keeping in order to perform three tasks related to program correctness and efficiency:

1. Informing the source of a dependency that its source has become activated, so that the source may send to the target the published value conveyed through the dynamic dependency. This case occurs when we encounter a control flow node between the target and the source. It is not guaranteed that the target of such a dependency will become activated following the source. The activation of the target could depend, for instance, on the successful evaluation of an `if-else` predicate, and

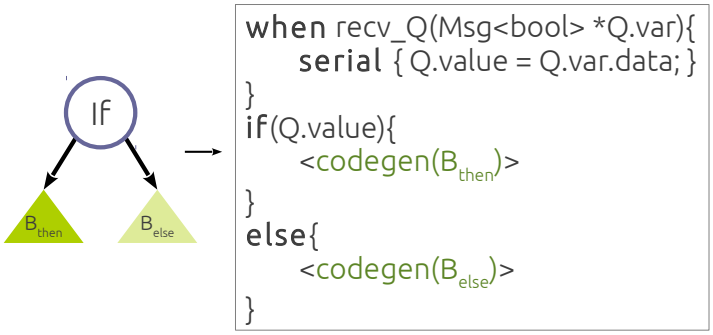


Figure 3.15: An `if-else` clause with embedded `then` and `else` blocks.

value of the predicate is known only to the GSM.

2. Recording the *most recently published* version of a value space, so that it can be decided, at run time, which of the multiple merged versions of a ϕ-node's consumed values can be supplied to dependency targets. This case occurs when multiple control flow paths intersect at a given CFG node, allowing several definitions of the same value space to reach the intersection point. In order to ensure that each consumption (other than the consumptions of ϕ-nodes) has only one reaching definition that can feed it, we placed ϕ-nodes in the original CFG of the program. In essence, the GSM must merge the incoming definitions by choosing the *most recently* published one, and propagating that to downstream nodes.

3. Recall that the source of a dynamic dependency must buffer its published value until the GSM routes it to a particular target. Therefore, a natural question is, *when is it safe for the source to discard/recycle the memory allocated for this value?* The GSM maintains metadata that allow it to inform the source of a dynamic dependency that, given the current global state of the program, its published value cannot be consumed by any node in the CFG before the source publishes it again. This allows the dependency source to *kill* that value, thereby freeing the memory resouces allocated to it.

3.12 Inferring communication patterns from publications and consumptions

Previously, we defined data dependencies as occurring between *collections* of objects referenced within `foreach` statements. Here, we discuss dependencies at the level of *individual* objects. Consider the general form of two `foreach`-embedded, publish-consume statements, the first of which publishes a value space that is consumed by the second.

$$\text{foreach}(\vec{x} \in \psi_1)\ (p[\vec{\alpha}(\vec{x})]) \leftarrow A[\vec{x}].f()$$
$$\text{foreach}(\vec{y} \in \psi_2)\ B[\vec{y}].g(p[\vec{\beta}(\vec{y})])$$

In this section, we determine, for each $\vec{x_0} \in \psi_1$ those $\vec{y_0} \in \psi_2$ such that $A[\vec{x_0}]$ sends a message to $B[\vec{y_0}]$. For the time being, we consider dependencies on produced, not reduced values.

In the above code $A[\vec{x_0}]$ publishes $v[\vec{\alpha}(\vec{x_0})]$, and $B[\vec{y_0}]$ consumes $v[\vec{\beta}(\vec{y_0})]$. Our inference procedure iteratively matches the component index expressions of the produced value, $v[\vec{\alpha}(\vec{x_0})]$, with those of the consumed value, $v[\vec{\beta}(\vec{y_0})]$, to obtain a target expression referencing an object, or a number of objects, in B. This is shown in Algorithm 2, where $\vec{x}^{(i)}$ denotes the i-th component of vector $\vec{x}$.

Algorithm 2: Inferring message targets from a publication/consumption pattern.

GenerateTargetObjectIndex$(\vec{x_0}, \vec{y})$
Input: Source object index expression $\vec{x_0}$ and target object index expression, $\vec{y}$.
Output: Target object index expression $\vec{y_0}$, to which the source sends a message.
begin
  $\vec{y_0} \leftarrow \vec{y}$
  for $i \in |\vec{\alpha}(\vec{x_0})|$ **do**
    $e \leftarrow \vec{\beta}(\vec{y_0})^{(i)}$
    if *e is a wildcard expression* **then skip** ;
    $k \leftarrow$ *FindPosition*$(e, \vec{y})$
    if $\vec{\alpha}(\vec{x_0})^{(i)}$ *is a range expression* **then** $\vec{y_0}^{(k)} \leftarrow \vec{\alpha}(\vec{x_0})^{(i)}$;
    else $\vec{y_0}^{(k)} \leftarrow r_i$;
  end
end

Briefly, this algorithm operates as follows. We start with no information about the specific $B[\vec{y_0}]$ to which $A[\vec{x_0}]$ is to send a message. Then, we iterate over the components of the published (and consumed) value's subscript. For

each index e expression in that subscript, if e is a wildcard expression, then it doesn't specify the identity of a particular value consumed by $B[\vec{y}]$, so we skip it. Otherwise, e can only be a `foreach`-bound identifier (*cf.* 3.7). In this case, we determine the position in $\vec{y}$ at which e appears; call this k. Then, we set the k-th component of $\vec{y_0}$ to the i-th component of the published value's subscript, $\vec{\alpha}(\vec{x_0})$. However if $\vec{\alpha}(\vec{x_0})^{(i)}$ is a range expression, then we handle the implied scatter operation at the publisher by placing an identifier r_i in $\vec{y_0}^{(k)}$. This identifier is a placeholder in the generated code for the index variable of the loop over all published range elements.

Intuitively, we exploit the fact that the value published by the message source, and the value consumed by its target must have identical subscripts for a message send to occur. In particular, the i-th index expression within these subscripts must match. What if, at the end of this procedure, there remains an index expression e' at position i of $\vec{y_0}$, such that e' is bound by the consumer's enclosing `foreach` statement? Then the following must be the case: Identifier e' is `foreach`-bound and *not* present in the consumed value's subscript, $\vec{\beta}(\vec{y_0})$, for otherwise, it would have been replaced by a corresponding component of $\vec{\alpha}(\vec{x_0})$. Therefore, there are multiple consumer objects consuming the same value, $v[\vec{\beta}(\vec{y_0})]$. That is, a *multicast* operation is implied, and so $A[\vec{x_0}]$ must multicast its message to all objects along the i-th dimension of B. Let us consider an example code fragment:

```
foreach (x1,x2 in {0:N-1}*{0:N-1})
  (p[(x1+x2)%N, (x1-x2+N)%N]) <- A[x1,x2].f();
foreach (y1,y2 in {0:N-1}*{0:N-1}){
  B[y1,y2].g(p[y1,y2]);
```

Given the above dependency, we have $\vec{x_0} = (x_1, x_2)$, $\vec{y} = (y_1, y_2)$,

$$\vec{\alpha}(\vec{x_0}) = ((x_1 + x_2) \bmod N, (x_1 - x_2 + N) \bmod N)$$

and $\vec{\beta}(\vec{y}) = (y_1, y_2)$. Starting with $i = 1$, we have $\vec{\beta}(\vec{y})^{(i)} = y_1$, which appears in position $k = 1$ in $\vec{y}$. Therefore, we set $\vec{y_0}^{(k)} = \vec{\alpha}(\vec{x_0})^{(i)} = (x_1 + x_2) \bmod N$. Similarly, for $i = 2$, we have $\vec{\beta}(\vec{y})^{(i)} = y_2$, which appears at position $k = 2$ in $\vec{y}$, so that $\vec{y_0}^{(k)} = (x_1 - x_2 + N) \bmod N$. Therefore, object $A[x_1, x_2]$ sends a message to object $B[(x_1 + x_2) \bmod N, (x_1 - x_2 + N) \bmod N]$.

However, what if a consumed value does *not* contain all of the `foreach`-bound identifiers? This situation implies the multicast operation, as dis-

cussed previously, and is illustrated below:

```
foreach (x1 in {0:N-1})
  (p[x1]) <- A[x1].f();
foreach (y1,y2 in {0:N-1}*{0:N-1}){
  B[y1,y2].g(p[y1]);
```

Using our simple inference procedure, we determine that each $A[x_1]$ sends a message to $B[x_1, y_2]$. Here, y_2 is an identifier expression that is bound by the consumer's `foreach` statement. This identifier does not appear in the subscript of the consumed value, $p[x_1]$. Therefore, a multicast from $A[x_1]$ to $B[x_1, 0 \ldots N-1]$ is inferred.

Reductions. In order to determine the *set* of publishers whose collective action results in the publication of a *reduced* value, we note the following. First, in *Charisma*, a reduced value's subscript is of a strictly lower arity than the subscript of the object collection that publishes it. Therefore, all identifiers bound by the publishing `foreach` statement, that do not appear in the subscript of the reduced value, constitute the dimensions of A over which a reduction occurs. Let us clarify this with an example:

```
objects A : Aclass[N,N,N];
values r : MyType[N];

foreach (x,y,z in {0:N-1}*{0:N-1}*{0:N-1})
  (+r[x]) <- A[x,y,z].f();
```

In the above example, `A` is a three-dimensional collection of objects, whereas `r` is a one-dimensional value space. The `foreach`-enclosed publish-consume statement above is interpreted as follows: invoke method f on each $A[x, y, z]$ such that $(x, y, z) \in \{0 \ldots N-1\}^3$; each such invocation contributes to the reduction of a value, named $r[x]$. Since y, z are `foreach`-bound, but do not appear in the reduced value, $r[x]$, the reduction to publish it occurs over all objects $A[x, 0 : N-1, 0 : N-1]$. There are N such reductions implied, since the arity of the reduced value is 1. The *Charisma* compiler generates code that adheres to this semantics.

It is easy to chain reductions together with arbitrary consumption patterns. Let the members of collection A reduce a value space r, and those of B consume it. Conceptually, the compiler treats the values in r as being

consumed by the members of an intermediary collection, C, whose arity is the same as that of r. Therefore, the result of each reduction over members of A is (conceptually) sent to a single member of C. The members of C then produce the same values that they consumed. These newly produced r values are consumed by objects in B. This reorganization serves to decouple the reduction operations over A from the subsequent consumptions by B, and we use Algorithm 2 to determine the particular objects in B to which each C must send the reduction result that it received.

3.13 Support for modularity

A piece of *Charisma* code can interact with another if it is declared as a `module`. This provision also allows for *Charisma* code to be incorporated into other Charm++ programs, as discussed in § 6.3. A piece of *Charisma* orchestration code is declared as a module by including the directive `module <moduleName>;` as the first line of the orchestration specification. This causes the compiler to generate code allowing the module to be incorporated into external code, instead of forcing control to originate within the code itself. The compiler also generates a *descriptor* of the module so that the incoporating code can instantiate and interact with the module as required. The descriptor provides the incorporating code access to all of the `objects` and `parameters` within the *Charisma* module. In§ 6.4, we see how external code can use this descriptor to interact with individual `objects` within *Charisma* module.

We will conduct our discussion in the context of an illustrative example. Suppose that we have written a simple *Charisma* module that performs a one-dimensional FFT opertion. The module splits up the butterfly computation into two local phases, which are mediated by a transpose operation (i.e. the transpose-based one-dimensional FFT [39].) We assume a blocked decomposition of the input data onto a one-dimensional object collection F in the module. We will defer the question of how extra-module code can ship data into and out of a *Charisma* module, until § 6.

```
module fft1d;
parameter N : int;

class FFT;
objects F : FFT[N];

values p : double[N];

foreach (i in {0:N-1}){
  (p[0:N-1,i]) <- F[i].butterfly_near();
  F[i].butterfly_far(p[i,*(N)]);
}
```

Basically, the code performs a transpose operation (the conjunction of a scatter in the first publish-consume statement, with a gather in the second one). The module includes a declaration of object collection F, and an integer parameter, N. The compiler generates the following C++ module descriptor for the above code:

```
struct CharismaModule_fft1d {
  struct Descriptor{
    int N;
    CProxy_FFT F;
    CProxy_Main_fft1d main;
  };

  static Descriptor instantiate(int N, const CkCallback&);
  void start(const CkCallback&);
};
```

The module class `CharismaModule_fft1d` encapsulates a descriptor, whose attributes include a proxy to the object collection F declared in the module, and an integer representing the parameter N. It also contains a proxy to the object that serves as the GSM for this module. The `instantiate` method is a factory method, and returns a descriptor to a new instance of the module.

```
CharismaModule_fft1d::Descriptor
CharismaModule_fft1d::instantiate(int N,
                                  const CkCallback &cb){
  Descriptor desc;
  desc.N = N;
  desc.F = CProxy_FFT::ckNew(N);
  desc.main = CProxy_Main_fft1d::ckNew();

  desc.F.moduleDescriptor(desc);
  desc.main.moduleDescriptor(desc, cb);

  return desc;
}
```

This method takes as argument the value of the parameter `N` to be used by the *Charisma* module. Other parameters, had they been present in the module, would have been similarly obtained. The factory method then instantiates the constituent objects of the module, and broadcasts the module descriptor to these objects. The callback argument provided to the factory method allows control to return to the instantiating context once the *Charisma* module has been initialized. Our approach allows the programmer to create multiple instances of each module.

```
void
CharismaModule_fft1d::start(const CkCallback &cb){
  main.start(cb);
  F.start();
}
```

Finally, the descriptor includes a `start` method, which is invoked by external code in order to commence module execution. When the module has finished execution, it invokes the callback argument provided. Invoking the `start` method and waiting for completion of the module is not the only way in which the incorporating code can interact with the module. Once the incorporating code obtains a descriptor to the module instance, it can directly communicate with the constituent objects of the *Charisma* module. This method leads to a *tight* coupling between the incorporating code and the module, and is discussed further in § 6.3.2.

3.14 Comparing performance and productivity with hand-written codes

In this section, we consider the performance of the code generated by the *Charisma* compiler. We compare the performance of *Charisma*-generated code with hand-written, Charm++ code for three simple algorithms that are easily expressed in the static data flow paradigm. We compare the SLOC of both versions and their parallel performance on up to 4096 processor cores. Through these results we illustrate that *Charisma* offers good performance, even as it makes code more comprehensible and concise.

Most of the following performance experiments were done on Argonne National Laboratory's *Intrepid*, which is an IBM Blue Gene/P supercomputer. Each *Intrepid* node has four 850 MHz PowerPC 450 processor cores and 512 MB of main memory per core. The nodes are connected by several networks, but the one utilized for application communication by the Charm++ runtime system is a proprietary three-dimensional torus network. Each Blue Gene/P node has a total of 5.1 GB/s bidirectional bandwidth, which is shared between its four processor cores.

The molecular dynamics simulations were performed on a Blue Gene/Q machine, *Vesta*, also at the Argonne National Laboratory. Each *Vesta* node comprises 16 4-way SMT compute cores, with 16 GB of DRAM shared among them. A 2 GB/s link connects nodes in a five dimensional torus configuration.

3.14.1 Jacobi relaxation

Our first benchmark is a three-dimensional stencil calculation representing a Jacobi relaxation procedure. Its inherent *halo exchange* is an important pattern of structured communication, forming the basis of applications such as fluid dynamics, PDE solvers and lattice QCD solvers.

Figure 3.16 compares the performance of the stencil calculation benchmark written in Charisma vs. Charm++. It is noteworthy that both implementations scale well up to 4096 cores. The *Charisma* version of the code was faster across the board by about 5-10%.

We saw a 38% reduction in the total (parallel + sequential) number of lines of code written, from 339 for the Charm++ version to 209 for the one written in *Charisma*.

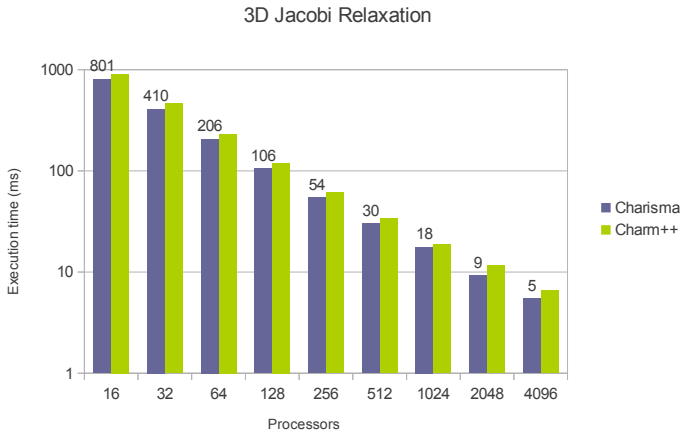


Figure 3.16: Performance comparison of hand-written Charm++ code and *Charisma*-generated code for the three-dimensional Jacobi successive over-relaxation scheme.

3.14.2 Parallel matrix-matrix multiplication

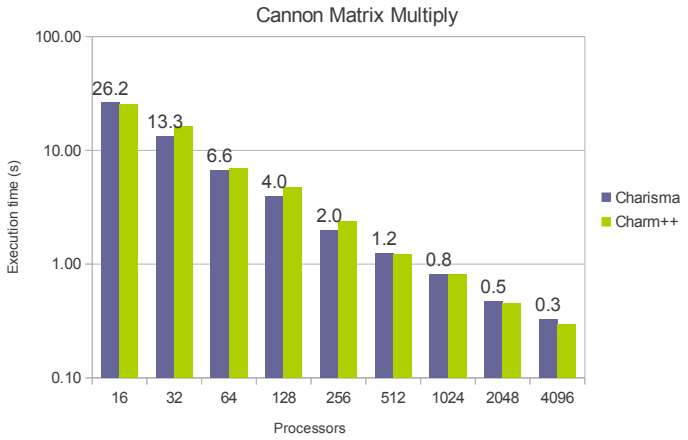


Figure 3.17: Performance comparison of hand-written Charm++ code and *Charisma*-generated code for the Cannon matrix multiplication algorithm.

We studied the performance of Cannon's iterative matrix multiplication algorithm [63, 64]. Inputs A and B are decomposed into tiles, which are encapsulated within objects in a two-dimensional collection called '`Cannon`'. The work of computing the result C ($= A \times B$) is decomposed in a similar manner: In each iteration of the algorithm, each `Cannon[x,y]` receives a tile of A and one of B, and computes an incremental sum accumulated in its C tile. It then sends the A tile to its left neighbor, and the B tile to the object

above it in the worker array, and proceeds with the next iteration.

In addition, the algorithm requires a tile-skewing phase to ensure that appropriate inputs are available to begin the computation of each tile of the result. The *Charisma* code for this reads:

```
foreach (x,y in {0:N-1}){
  (A[x,(x+y)%N], B[(x+y)%N,y]) <- Cannon[x,y].produce();
  Cannon[x,y].mult(A[x,y], B[x,y]);
}
```

The main iterative computation is shown below:

```
for (i in {0:N-1})
 foreach (x,y in {0:N-1} * {0:N-1}){
  (A[(x-1+N)%N,y], B[x,(y-1+N)%N]) <- Cannon[x,y].produce();
   Cannon[x,y].mult(A[x,y], B[x,y]);
 }
```

Even so, there was a 64% reduction in SLOC over the Charm++ version. Thus, in this case, not only was the *Charisma* version more concise, it also benefited from optimizations done by the orchestration compiler.

3.14.3 Three-dimensional FFT

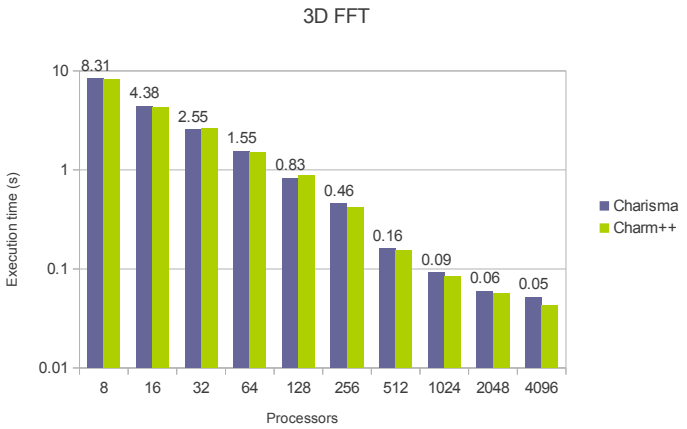


Figure 3.18: Performance comparison of hand-written Charm++ code and *Charisma*-generated code for a transpose-based, three-dimensional FFT algorithm.

We consider the three-dimensional FFT with *pencil* decomposition [39] onto a two-dimensional collection of objects. This leads to three line FFT's with two intervening transposes. Figure 3.18 compares the performance of the Charm++ and *Charisma* versions with 512^3 double-precision elements. Both versions used the FFTW library [65] for (serial) one-dimensional line FFTs. The results show that the Charm++ code performs better, especially at scale. In fact, at 4096 cores, the *Charisma* code obtains a speedup (relative to performance at 16 cores) of 158 compared to the Charm++ speedup of 191. However, neither version scales particularly well beyond the 2048 core mark: parallel efficiency is close to 40 per cent for the Charm++ code, and about 30 per cent for the *Charisma* code. The reason is that the small message overhead for the underlying Charm++ messages is significant. In fact, in order to scale the all-to-all operation inherent in the parallel transpose, dynamic aggregation and routing techniques, as used by Kale *et al.* [26] would be required.

SLOC were reduced by 37% for this program. In this specific benchmark, sequential code dealing with local FFT computation constitutes a significant portion of the program, so that the reduction of SLOC is not as impressive as achieved with some of the simpler programs. On the other hand, complicated applications still stand to benefit from the clarity of global control flow expression afforded by *Charisma*.

3.14.4 Lennard-Jones molecular dynamics

As a final benchmark, we present a molecular dynamics application that performs the fine-grained decomposition of both space (data) and force calculations (work). This decomposition technique follows the strategy of Kale and colleagues [66], and is described in greater detail in § 3.9.2. Here, we compare the performance of the three-dimensional extension of the *Charisma* code from § 3.9.2, with a Charm++ code that has been written as a representative *mini-app* for molecular dynamics simulators. Both codes simulate three-dimensional ensembles of particles subject to the short range, Lennard-Jones potential. The simulation volume is divided into a structured grid of 1Å patches, and the particles in a patch interact only with those within a two-patch neighborhood in the x and y dimensions, and a one-patch neigh-

borhood in the z dimension. This is called a 2-away XY, 1-away Z simulation. Particle migration between patches was disabled for the purpose of benchmarking performance.

The Charm++ code uses the SDAG notation to express control flow local to each object, and so enjoys a clean separation of parallel structure from serial computations (namely the equations that describe the physics of the simulation). As a result, we were able to borrow all of the serial code for the *Charisma* version from its Charm++ counterpart. The *Charisma* code ran to a total of 477 lines, much of it sequential code, and the Charm++ version was 854 lines long. A more detailed analysis of the code for this benchmark is presented in § 3.14.5.

Both versions used a two-tiered strategy (built into the Charm++ load balancing infrastructure) for the placement of objects on processors. The first twenty iterations of the benchmark were performed with a blocked assignment of objects to processors. Performance metrics collected during that time were used to perform a *greedy* assignment of heavy objects to underloaded processors. The simulation was then resumed, and from that point onward, a *refinement* of the greedy mapping was periodically performed.

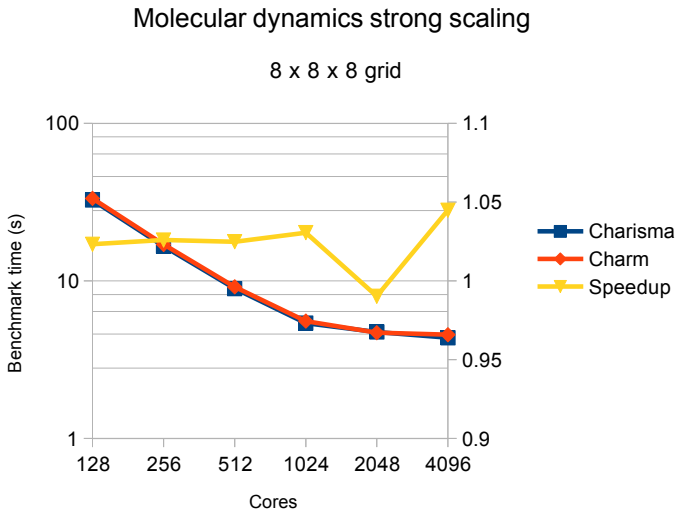


Figure 3.19: MD Performance comparison for an $8 \times 8 \times 8$ patch grid.

Figure 3.19 shows how the Charm++ and *Charisma* versions of the MD benchmark scale. The "speedup" curve shown in that figure plots the ratio of benchmark time for the Charm++ version, to that taken by the *Charisma*

versions. The figure demonstrates that *Charisma* code is comparable (to within 5 per cent) to hand-written Charm++ in terms of performance Neither code scaled particularly well beyond 1024 cores on the smaller configuration of about $89,000$ particles decomposed over an $8 \times 8 \times 8$ patch grid. For a 2-away XY, 1-away Z simulation, we obtained 19456 computes.

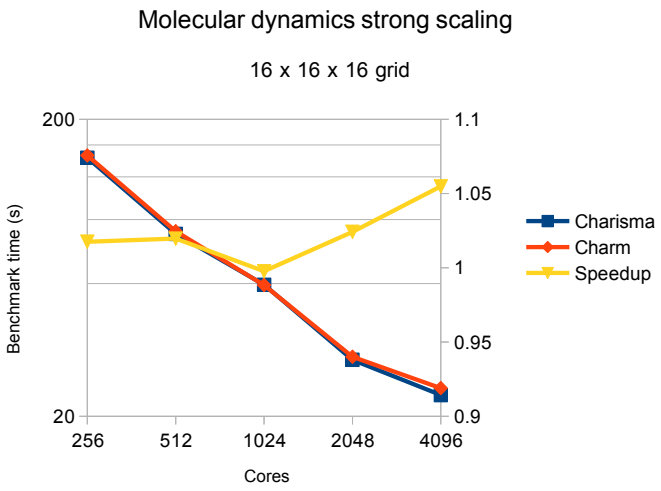


Figure 3.20: MD Performance comparison for an $16 \times 16 \times 16$ patch grid.

Better scalability was observed on the larger, $715,000$ particle data set, decomposed over $16 \times 16 \times 16$ patches (307200 computes). The two versions gave comparable performance on up to 4096 cores of *Vesta* (Figure 3.20).

3.14.5 Summary of productivity results

Figure 3.21 presents a summary of the reduction in percentage lines of source code (SLOC) for the four benchmarks that we have examined. The percentage reduction is obtained by dividing the absolute difference in SLOC by the number of lines in the Charm++ implementation. We see that the reduction in SLOC is consistent and significant across the different benchmarks.

Figure 3.22 provides a deeper analysis of these reductions in SLOC for the particular case of the molecular dynamics benchmark.

The graph plots several categories of code on the x axis, for both versions of the MD benchmark. On the y axis is the number of lines of code present in each category. The shorthand names for the categories denote the following,

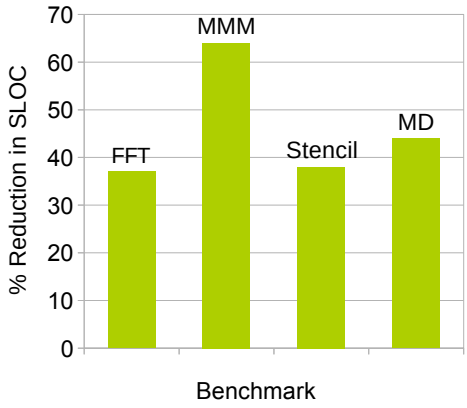


Figure 3.21: Percentage reduction in source lines of code for the *Charisma* version of each benchmark, over its Charm++ counterpart.

in order:

- *Decls.* declarations of variables relating to the parallel structure of the program. For example, in Charm++ programs, one must declare *proxies* to collections of objects. In *Charisma*, this would be code present in the declaration preamble of the orchestration code.
- *Par.* Parallel structure of the application. In the *Charisma* code, this is the section following the declaration and initialization sections in the orchestration file. In the Charm++ version, it is the SDAG code that represents object-local flow of control.
- *Ser.* Serial code, namely that present in the serial methods of objects, in both the *Charisma* and Charm++ versions.
- *Phys.* Simulation physics, i.e. code to compute the Lennard-Jones potential.
- *Obj.* Definition of classes representing coarse-grained objects (patches and computes).
- *DS.* Serial data structures, e.g. particles and vectors.
- *LB.* Code required for the serialization/deserialization of object state during migration-based load balancing.

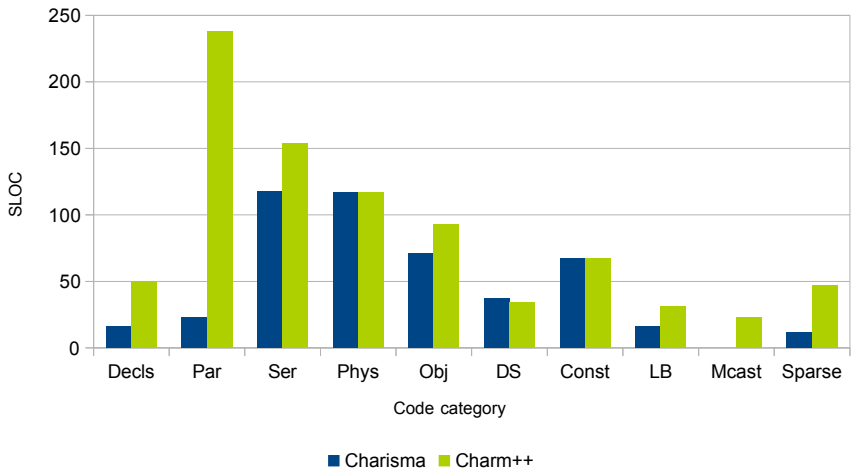


Figure 3.22: Break-up of MD benchmark code for *Charisma* and Charm++ verisons, into a number of categories. The key for the shorthand categories on the x axis is provided in the text. The graph shows that *Charisma* programs are compact in the expression of their parallel structures, and benefit from automatic generation of much of the code required for load balancing, insertion into sparse object collections, and multicasts/reductions over sparse arrays.

- *Mcast*. Creation and maintenance of sparse subsections of object collections, the members of which may be spread over multiple processors. This code is automatically generated in the *Charisma* version.

- *Sparse*. Insertion of objects into the sparse collection of *computes*. The *Charisma* code for this comprises a total of 12 lines in the initialization section of the orchestration file.

The graph shows that *Charisma* succinctly captures the parallel structure of applications, and automates the generation of code related to certain routine tasks such as the creation and maintenance of sections of object collections, and serialization/de-serialization required during load balancing. Unsurprisingly, the amount of code in the serial portion of the computation, as well as data structure definitions, are almost identical in the two versions.

We list some of the more intangible productivity benefits of *Charisma* below:

1. The ability to capture global flows of data and control. This allows the

reader of a program to examine the *unfragmented*, overall structure of communication and flow of data in the program.

2. An abstract *publish-consume* communication model. We believe that this model yields a simple means to achieve communication between the objects of a program. As seen in § 3.12, it is flexible enough to express a number of commonly used communication patterns.

3. Clean separation of parallel and sequential components. Not only does this separation allow users to employ familiar C++ syntax to achieve the bulk of their computation, it also allows for a separation of concerns for collaborative application development. Specifically, the parallelization strategy of the algorithm is largely separate from the sequential methods used therein, allowing their concurrent development by computer scientists and domain specialists, respectively. Moreover, the simple syntax of *Charisma* enables one to quickly assess first-order performance ramifications of restructuring of the overall parallel algorithm.

Of course, *Charisma* is a specialized language, and as such only supports the expression of a limited subset of possible parallel programs. However, we believe that the ability to succinctly express global data-independent communication patterns is important to HPC programmers and domain specialists alike. Furthermore, the expressive gaps in *Charisma* can be mitigated by other specialized languages (e.g. *cf.* § 4 and § 5), or through interoperation with a more general-purpose language, such as Charm++ (*cf.* § 6).

CHAPTER

4

DIVIDE-AND-CONQUER

4.1 Introduction

Divide-and-conquer is a powerful programming technique and one that is closely attuned to our natural understanding and expression of algorithms. Many algorithms can be expressed in a compact and elegant manner through this technique of recursive specification. Such an expression leads to programs that have an inherently parallel structure embodied by a recursion tree, every node of which is a function invocation. The programmer needn't be aware of this *task* parallelism implicit in the program structure – it can be extracted (with the aid of a compiler) at run-time by an intelligent runtime system.

These observations have inspired extensive research on the use of the divide-and-conquer paradigm as a medium for the expression of parallel programs. For example, the work of Loidl *et al.* [67] provide a survey of different parallel dialects of the Haskell functional language [68], as well as a comprehensive review of various other approaches to parallel functional programming. Much work has also been done in the management of grain size of

tasks generated by the parallel execution of such programs [69, 70], the load balancing problems that they face [71], and increased efficiency through prioritization [72]. Most work that shows good scaling on distributed memory systems pertains to the solution of state space search problems (for example, see the work of Dinan *et al.* [73]). However, in general, it has been difficult to achieve good speedups on large scale distributed memory machines (although good results have been obtained for relatively small, shared memory machines) for divide-and-conquer programs, especially for algorithms that involve large, distributed data.

4.1.1 This chapter

The objective of this chapter is to develop a parallel programming language called *DivCon*, designed specifically for the expression of divide-and-conquer algorithms operating on large sets of input data. In particular, we aim to provide a simple language that allows for the expression of divide-and-conquer algorithms for distributed memory machines, especially for such algorithms that exhibit *generative* recursion, and therefore nominally involve the movement of large amounts of data over the network. In such an algorithm, each invocation of a function receives as input a set of data elements. The function call, in turn, performs some work to *generate* new sets of data. Each new set of data is then passed as input to a recursive function call. The book of Felleisen *et al.* [74] describes this pattern as *generative recursion*. The general form of such an algorithm is presented below:

Algorithm 3: *DivCon*(A)

```
Data: Set A of data elements
begin
    if isBaseCase(A) then
        return solveBaseCase(A)
    (A_1, ..., A_m) ⟵ createSubProblems(A)
    return combineSubSolutions(DivCon(A_1), ..., DivCon(A_m))
```

Typically for such algorithms, the application of the *partitioning function* *createSubProblems* on a given element of A, is independent of its application on any other. Therefore, the partitioning of the input collection A into multiple sub-collections, is in fact data parallel.

Let us examine this pattern more closely, in the context of distributed

memory machines. If the input A to invocation $DivCon(A)$ is assumed to be very large, it must be decomposed over several processors (call this set of processors P_A) on the distributed memory computer. Similarly, each recursive invocation $DivCon(A_i)$ receives its input A_i as a set decomposed over several processors (P_{A_i}). Therefore, data elements generated due to the application of *createSubProblems* must be partitioned onto processor sets P_{A_i}. Typically, this involves an expensive, each-to-many exchange over the network between the members of P_A. Thereafter, each invocation $DivCon(A_i)$ performs a similar partitioning operation, this time over a smaller set of processors, P_{A_i}, and so on, until sequential computations are invoked.

Bearing this pattern in mind, our objective is two-fold: (i) To provide an abstraction that efficiently manages data parallelism in generatively recursive divide-and-conquer applications. (ii) To design a simple language, called *DivCon*, which serves as a vehicle for this abstraction.

DivCon is suited for the expression of divide-and-conquer computations as a sequence of expression evaluations, including function invocations. The language we develop is influenced by the design of Cilk [75], in that the programmer explicitly declares certain tasks to be concurrently executable with others through a *spawn* construct. As such, even though the underlying runtime techniques used by our language differ from the work-stealing based approach of Cilk [76], we do not claim our design to be particularly novel. On the other hand, the *DivCon* compiler performs data dependency analysis, so that a construct corresponding to `cilk_sync` is not needed in our language. Task parallel work is managed by the runtime using the techniques of Kale *et al.* [77]: First, we dynamically agglomerate fine-grained tasks into larger tasks, thereby reducing parallel overhead. Second, we employ *seed-based* dynamic strategies to distribute load across the processors of the parallel machine.

DivCon also provides a construct called the *DivConArray*, through which the programmer specifies data-parallel operations. We outline the design of a module within the *DivCon* runtime system that manages such data-parallel computations. The chief purpose of this module is to avoid the each-to-many operation nominally required at every invocation of a generatively recursive function. In order to do this, redistribution of partitioned data is *delayed*. Moreover, messages that initiate data-parallel operations are combined and issued *en masse*. As we shall see, this allows the cost of data communication

to be amortized over multiple invocations of a generatively recursive function. We will analyze the impact of these optimizations on a few applications that typify tree-structured computations and generative recursion. We will end with a comment on programming productivity.

4.2 Design principles

Our design of the *DivCon* language is rooted in the desire to provide a minimal set of constructs for expressing task and data parallelism within the divide-and-conquer paradigm.

4.2.1 Task parallelism

Let us begin by enumerating some of the principles that guide the design of *DivCon* constructs for task parallelism.

Fine-grained tasks

We provide a simple core language in which divide-and-conquer computations are expressed in terms of fine-grained, concurrently executable functions, called *tasks*. Thus, a task is the basic unit of work in *DivCon*. A task evaluates a sequence of expressions, and may in turn invoke other tasks. The programmer explicitly *spawns* tasks, in a manner similar to Cilk.

Dependencies between tasks

The expressions evaluated by a task may depend on the results of other tasks. Such dependencies between tasks are specified by the production and consumption of values. Tasks can produce values, and can consume values produced by other tasks. We perform simple compiler analysis to determine the data dependencies between tasks. If there are no such *explicit* data dependencies between two tasks, then we assume that the tasks may be executed in parallel. This means that a *DivCon* analogue of the explicit *sync* operation of Cilk is not required: synchronization points are automatically detected by the *DivCon* compiler.

4.2.2 Separation of parallel and serial code

We aim to limit the syntactic repertoire of the language, so as to keep the language simple (thereby reducing the cognitive burden as well as novelty). This simplicity has an added benefit, in that it allows for precise analysis of dependencies by the compiler. In the absence of language constructs such as pointers (coupled with a call-by-value semantics of function invocation), the *DivCon* compiler can extract all the dependencies that exist in the program. This is in contrast to, for example, Cilk, wherein artifacts such as memory aliasing through pointers, and array expressions, can obscure the flow of data in the program.

In this vein, *DivCon* provides only basic evaluation of expressions within the language, but allows the invocation of serial, C++ functions from it. We believe that this feature makes *DivCon* programs simple to understand (both for the programmer, and for the compiler), while not unduly restricting language expressiveness. The compiler does not attempt to parse or extract parallelism from serial functions: to the compiler, such a function is a black box.

4.2.3 Program order semantics

DivCon has a sequential semantics of execution. Each task contains a sequence of expression evaluations (which may in turn invoke other tasks), and control flows from one such evaluation to the next as dictated by program order. Recall that tasks may produce and consume values. A data dependency can only exist from a *DivCon* expression that produces a value, to one that succeeds it in program order. As a result, *DivCon* has an imperative, rather than a declarative flavor.

4.2.4 Explicit data parallelism

We enable the explicit expression of *data parallelism* through a novel distributed data structure called the *DivConArray*. We pay particular attention to an important class of divide-and-conquer applications, namely those that exhibit the so-called *generative recursive* pattern [74] over *large sets of data elements*. Examples of this pattern include quicksort, median-finding,

quickhull, Delaunay mesh generation, etc.

Real-world divide-and-conquer applications are characterized by the large size of input problems. This means that the input data set for each recursive invocation must typically be distributed over several processors. As described in § 4.1.1, the partitioning phase of divide-and-conquer applications with generative recursion over sets of elements, is characterized by the movement of data from one set of processors, to several different, and smaller sets of processors. The associated *each-to-many* operation is expensive in terms of data movement (communication) cost for distributed memory systems, and must be incurred for each invocation of the function.

Our solution to this problem involves the amortization of data movement costs over several invocations of the recursive function. In this thesis, we develop a distributed array abstraction, namely the *DivConArray*, that realizes this amortization technique. That is, the *DivConArray* provides efficient support for basic operations required by divide-and-conquer applications that exhibit generative recursion over sets of data elements.

The *DivConArray* is available in the *DivCon* language as a first class construct, thereby allowing programmers to leverage this abstraction using a simple notation.

4.2.5 Dynamic optimizations

Our design leverages run-time strategies such as task agglomeration and communication avoidance (using *DivconArrays*) to optimize performance of the executing *DivCon* program. The *DivCon* runtime system is based on Charm++, and as such uses its dynamic load balancing, and message combining infrastructures.

Many of *DivCon*'s language constructs find echoes in previously developed languages for recursive or divide-and-conquer computations. For this reason, we do not claim the design of the *DivCon* language to be particularly novel; instead, we consider its placement within the context of object-based, message-driven and runtime-centric interoperability to be an important contribution. In addition, as discussed in § 4.6.2 the *DivConArray* abstraction provides a convenient and efficient means of expressing data parallelism in generatively recursive divide-and-conquer algorithms.

4.3 Examples of *DivCon* code

To set some context for the description of *DivCon* that is to follow in § 4.4, we present two well-known divide-and-conquer algorithms in *DivCon*.

4.3.1 Computing the i-th Fibonacci number

A *DivCon* program consists of a number of declarations, followed by the definitions of *functions*. Declarations typically include `extern`al, immutable symbols, whose values are set extraneously to the program (e.g. `Tau` below); and (C++) `serial` functions (`seqFib`), which are meant to encapsulate serial computations of a coarse grain size.

```
extern int Tau;

serial seqFib(int i) : int;

fib (int i) : int {
  if (i <= Tau) return seqFib(i);
  else {
    int i1, i2, n;

    i1 = spawn fib(i-1);
    i2 = spawn fib(i-2);
    n = i1 + i2;
    return n;
  }
}

main enter(int n) : int {
  return fib(n);
}
```

Types of functions

A function is a sequence of expression evaluation statements. Control flows from one such statement to the next in program order. A function receives

input arguments, performs some computations using these and other variables, and `returns` a number of results to its caller. The function `fib` takes an integer `i` as input, and returns the `i`-th Fibonacci number as its only result. When the invocation of a function is preceded by the `spawn` keyword, it indicates that the invocation will be executed in a lightweight *task* that executes concurrently with all other tasks, subject to data dependencies. An expression (perhaps a function invocation) that depends on the results of preceding expressions (in program order) is not evaluated until all such results are available. (The notion of *availability* is discussed more fully below.

Typically, tasks are fine-grained, and spawn other tasks. In the example above, the task `fib(i)` spawns two other tasks, namely `fib(i-1)` and `fib(i-2)`. On the other hand `seqFib` is declared as a `serial` function. A `serial` function is a C++ function, i.e. it is specified separately from the *DivCon* code, and it may not contain *DivCon* statements. As a result, a `serial` function may not `spawn` any *DivCon* tasks in its body. Therefore, the invocation of a `serial` function results in the execution of some C++ code to completion. One may `spawn` a new task to evaluate such a `serial` function. Such tasks allow the overhead of parallel task creation to be amortized over serial computations. It is expected that the programmer will encapsulate a coarse-grained serial computation within a `serial` function. In our simple example, the `seqFib` `serial` function might serially evaluate an appropriately large member of the Fibonacci sequence.

Program execution

Control enters a *DivCon* program in the function labeled as `main`. The body of a *DivCon* function consists of sequence (block) of statements. These statements execute in program order, making for an *imperative* style of programming. A statement may execute once the result of every one of its embedded expressions has been evaluated. In turn, an expression is evaluated once all of its subexpressions have been evaluated.

Given the presence of asynchronously executed tasks (via the `spawn` construct), we must define *when* the result of an evaluated expression may be used by the expressions that depend on it. For this purpose, we define the notion of *availability* of results within a *DivCon* function. The result of an identifier expression is available immediately if the identifier corresponds to

an `extern` constant, or is a formal argument of the function. On the other hand, if the identifier has been assigned the result of an expression (through an assignment statement of the form $v = e$), then it becomes available when the result of the expression on the RHS becomes available. The result of an arithmetic or logical expression becomes available when the results of its operands are available, and the corresponding arithmetic/logical operation has been performed. The result of a function invocation (whether `spawn`ed or not) is available when it has `return`ed a number of results to the calling context. In this sense, *DivCon* employs *strict* evaluation.

More concretely, the task `fib(i)` results in the following sequence of computations. We first check whether `i` is less than `Tau`. If so, we evaluate `seqFib(i)`, thereby executing a block of serial, C++ code. When the (C++) body of `seqFib` has been fully evaluated, its result is returned to its calling context, which in turn returns the result to the task that invoked it.

On the other hand, if `i` is larger than `Tau`, task `fib(i)` spawns a task `fib(i-1)`, and upon receiving its result, stores it in scope-declared variable `i1`. Next, `fib(i-2)` is spawned, and its result (when it becomes available) is saved in `i2`. The sum of these intermediate results is stored in variable `n`, and returned to the caller of `fib(i)`.

Although this is the programmer's view of the sequence of computations, the *DivCon* compiler constructs a CFG of interdependent sub-tasks from the specification of each task. Consquently, execution is data-driven, and more dynamic than suggested by the above discussion. Given the semantics of evaluation, we could have specified `fib` more compactly:

```
fib (int i) : int {
  if (i <= Tau) return seqFib(i);
  else return spawn fib(i-1) + spawn fib(i-2);
}
```

Serial code

We now see how `serial functions` serve as an interface between *DivCon* and C++ code. The C++ definition of `seqFib` is present in a different file from the *DivCon* code above. This file is not analyzed by the *DivCon* compiler:

```
void seqFib(const Divcon::ConsumedValue<int> &i,
                  Divcon::ProducedValue<int> &result)
{
  if(i == 0 || i == 1) result.produce(i);
  else{
    int i1, i2, i3;

    i1 = 1; i2 = 0;
    for(int j = *i; j >= 2; --j){
      i3 = i1 + i2;
      i2 = i1;
      i1 = i3;
    }
    result.produce(i3);
  }
}
```

A `serial` function's consumed values are encapsulated by `ConsumedValue`'s. The associated data can be accessed using the '`*`' operator. The C++ signature of the serial function identifies it as having a `void` return type. Produced values are encapsulated within `ProducedValue<>`'s, and can be passed to *DivCon* code via the `produce` method.

4.3.2 Quicksort

Next we present the quicksort algorithm to introduce the use of *DivconArrays* in expressing data parallelism. The code begins by declaring the user-defined type, `Record`. As in *Charisma*, types are not defined within *DivCon*. They appear instead in separate C++ files. Next, the externally defined threshold for switching over to serial sorting is given by the immutable `extern int Tau`. The `serial` function `seqSort` performs coarse-grained, serial sorting work in order to amortize the cost of parallel tasks. Here, by coarse we mean on the order of a few hundred to a thousand `Record`s.

```
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));
  }
}
```

Now, consider the *DivCon* function `qsort`. It receives as its only input a *DivConArray* whose elements are of type `Record`, denoted `Array<Record> A`. *DivConArray* `A` is an indexed collection of elements that supports some basic operations such as `length`, `read` and `concat`. We check whether `A` is small enough (using `length`) that we can sort it serially. Otherwise, we `read` a particular index within `A`, and assign the obtained value to the variable `pivot`.

Next comes the application of the data-parallel, out-of-place partitioning operation of the quicksort algorithm. Using the `foreach` construct of *DivCon* we iterate over each element `a` in the *DivConArray* `A`. The `foreach` construct provides an *unordered* iteration over all the elements in a *DivConArray*. For our present discussion, we treat the '`+=`' operator in statements of the form `LT += a` as adding a single element `a` to the *DivConArray* `A`. The precise meaning of the '`+=`' operator is discussed in § 4.4.1. Therefore, depending on the value of the current element `a` relative to the `pivot`, we place it in one of the `LT`, `GT`

or `EQ` *DivconArrays*. Finally, we spawn `qsort(LT)` and `qsort(GT)` functions, `concat`enate their results appropriately, and `return` the result.

Below is shown the C++ code for the `serial` function `seqSort`:

```
void seqSort
(const Divcon::ConsumedValue< Divcon::Array<Record> > &in,
 Divcon::ProducedValue< Divcon::Array<Record> > &out)
{
  int n = *in.size();
  std::vector< Record > v(n);
  std::copy(*in.first(), *in.last(), v.first());

  if(n >= 2) std::sort(v.first(), v.last());
  out.produce(Divcon::Array<Record>(v));
}
```

In the code above, arguments consumed by the C++ function are encapsulated `Divcon::ConsumedValue`, whereas data produced by the function are encapsulated within the `Divcon::ProducedValue` class template. We first copy the elements of the consumed *DivConArray* `in` into a C++ `vector` and sort it using the C++ standard library. The sorted result is funneled to *DivCon* via the `produce` call.

4.3.3 Discussion

From the examples above, we can make some preliminary observations about the *DivCon* language. First, the simple notation used to specify functions, forms a fine-grained scaffold into which are placed serially executing, coarse-grained, `serial functions`. The latter can leverage the vast functionality afforded by C++ standard libraries, as well as other sequential libraries plugged into the program by the user. Therefore, we believe that *DivCon* provides a good mixture of simplicity of notation, and expressive power.

These `serial` functions provide a means to amortize the cost of `spawn`ing function invocations over coarse-grained chunks of work. In cases where it is not possible to statically determine a good cut-off criterion for `serial` work, *DivCon* performs task agglomeration to mitigate parallelization overheads. (*cf.* the *unbalanced tree search* benchmark, § 4.8.1.)

Notice that the language conceals from the programmer the fact that *DivconArrays* may be distributed across several processors. These arrays support simple functionality such as `read`, `length`, `concat`, `scan` as well as the '`+=`' operator to examine and manipulate *DivconArrays*. Additionally, *DivconArrays* can be `produced` as a result of arbitrary C++ computations encapsulated within `serial` functions. Such *DivconArrays* are not distributed over multiple processors.

However, we do not believe these operations to be universally applicable. Indeed, they provide very restricted functionality, and other languages and frameworks provide richer distributed array abstractions (*cf.* ZPL [78], the PGAS languages [5, 6], the APGAS languages [3, 4], the multiphase shared array [79] and Global Arrays [80]). However, we believe that in conjunction with the `foreach` construct, the operations supported by *DivCon* capture an important set of manipulations performed in the divide-and-conquer paradigm. Later, we exploit the specialized nature of our *DivConArray* abstraction in order to improve performance in a distributed memory setting.

The imperative nature of the *DivCon* language, and the explicit expression of parallelism is reminiscent of Cilk [75]. However, *DivCon* performs the necessary dependency analysis so that the programmer doesn't have to identify `sync` points in the program. We believe that this automatic analysis leads to *DivCon* programs that are less prone to data races than Cilk programs. *DivCon* and Cilk also differ in the manner in which they interact with an otherwise-serial language: whereas Cilk `spawn`/`sync` operations are embedded within C, *DivCon* enforces a strict separation between function code and C++ code. Finally, *DivCon* makes provisions for the explicit expression of data parallelism with *DivconArrays* and the `foreach` construct. The `foreach` construct is a restricted form of the list comprehension operator of NESL [81]. NESL programs enjoy a great deal of generality in expressing data parallelism on shared memory machines. However, the restrictions we impose on the `foreach` construct of *DivCon* are necessary to adapt to a distributed memory setting.

4.4 Language design

Let us now take a closer look at the syntactic and semantic structure of *DivCon*.

4.4.1 Base expression evaluation language

In the following, we discuss the syntax of *DivCon* in terms of its EBNF.

Top-level structure

$$\langle program \rangle \quad ::= \quad \langle decls \rangle \langle functions \rangle$$

A *DivCon* program consists of a number of declarations, followed by function definitions.

Declarations

The declaration preamble contains individual declarations, each of which declares a `extern` constant, a user-defined type, or a serial function written in C++.

$$\begin{aligned} \langle decls \rangle &::= (\langle decl \rangle \text{ ";" }) * \\ \langle decl \rangle &::= \langle externType \rangle \mid \langle externConst \rangle \mid \langle serialFunctionDecl \rangle \end{aligned}$$

Constants declared as `extern` are defined externally to the *DivCon* program, e.g. via the command line. Type declarations inform the *DivCon* compiler of all non-primitive, user-defined types that are to be considered valid in the *DivCon* program. Serial function declarations indicate pieces of C++ code that are invoked by a *DivCon* program, but which cannot themselves spawn *DivCon* tasks.

Types. A type declaration specifies a valid, externally-defined C++ type that can be used in the *DivCon* program.

$$\langle externType \rangle \quad ::= \quad \text{“}\textbf{type}\text{”} \ \langle ident \rangle$$

Types used in the *DivCon* program are either declared (as above), or primitive C++ types, e.g. `int`, `bool`, etc. Aggregate types are allowed, but these cannot be defined within *DivCon*. A function witha `void` return type can not `return` an evaluated expression.

External constants. An external constant is a C/C++ style declaration of an externally defined parameter that is referenced (in read-only mode) in *DivCon* code.

$$\begin{aligned} \langle externConst \rangle \quad &::= \quad \text{“}\textbf{extern}\text{”} \ \langle type \rangle \langle idents \rangle \\ \langle type \rangle \quad &::= \quad \langle cppType \rangle \mid \ \text{“}\textbf{Array}\text{”} \quad \text{“<”} \ \langle cppType \rangle \ \text{“>”} \end{aligned}$$

Examples of such declarations are shown below:

```
extern int seqThreshold;
extern double maxAllowedError;

type MyType;
extern MyType myTypedConstant;
extern Array<MyType> someArray;
```

External constants may have user-defined types, and may even be *DivconArrays*. External constants may be defined either on the command-line, or as shown in § 4.7, in an external module that invokes *DivCon* code. In fact, `extern` *DivconArrays* form the basis for interoperation with Charm++, and the other languages of this thesis. *DivCon* does not support `Array`s of `Array`s.

Serial functions. Serial functions provide a way for *DivCon* to interface with serially executed C++ code. A serial function is specified separately from the *DivCon* part of a program. We believe that this separation helps to

keep the *DivCon* notation simple, while allowing sufficient flexibility in the expression of serial computations.

A `serial` function may receive input parameters from (parallel) *DivCon* code. The `seqFib serial` function in § 4.3.1 is an example of serial code returning a single value, via the `Divcon::ProducedValue<>::produce` call, to its *DivCon* calling context. In general, multiple values may be returned to *DivCon* in the same way. The reader will note the similarity of this mechanism with the one used in *Charisma* (*cf.* § 3).

$$
\begin{aligned}
\langle \textit{serialFuncDecl} \rangle &::= \text{“}\textbf{serial}\text{”} \\
&\quad \langle \textit{ident} \rangle \text{ “(” } [\langle \textit{paramDecls} \rangle] \text{ “)” } \text{ “:” } \langle \textit{returnType} \rangle \\
\langle \textit{paramDecls} \rangle &::= \langle \textit{paramDecl} \rangle (\text{ “,” } \langle \textit{paramDecl} \rangle) * \\
\langle \textit{paramDecl} \rangle &::= \langle \textit{type} \rangle \langle \textit{ident} \rangle \\
\langle \textit{returnType} \rangle &::= \text{ “(” } \langle \textit{type} \rangle (\text{ “,” } \langle \textit{type} \rangle) * \text{ “)”}
\end{aligned}
$$

Serial functions are provided so that *DivCon* code may interface with external, general-purpose code. For example, arithmetic operations that are not supported by *DivCon* may be wrapped within `serial` functions.

```
// in DivCon file:
serial NewtonZero(Functor) : double;

// in .cpp file:
void NewtonZero(const Divcon::ConsumedValue<Functor> &f,
                      Divcon::ProducedValue<double> &y){
  double root = 0.0;
  while(fabs(f.evaluate(root)) > 0.001)
    root = root -
        (*f).evaluate(root)/(*f).derivative().evaluate(root);

  y.produce(root);
}
```

The above `serial` function `NewtonZero`, when invoked from within *DivCon*, receives as input a `Functor`, `f`, and uses Newton's method to find a

root of the corresponding (mathematical) function. The function was declared (in *DivCon*) as consuming a `Functor` and returning a `double`. In the actual C++ code, both input and output are encapsulated within *DivCon* classes. The result is returned to the calling *DivCon* context by means of the `produce` call on the corresponding `Divcon::ProducedValue`. The body of `NewtonZero` makes free use of standard C/C++ library functions (e.g. `fabs`) and constructs (e.g. `while`).

DivCon functions

A *DivCon* function allows the specification of recursive computations in the form of a textually-expressed CFG.

$$
\begin{aligned}
\langle functions \rangle &::= \langle functionDef \rangle * \\
\langle functionDef \rangle &::= [\ \text{“}\textbf{main}\text{”}\ ]\langle ident \rangle\ \text{“(”}\ \langle paramDecls \rangle\ \text{“)”}\quad \text{“:”} \\
&\quad \langle returnType \rangle \langle stmtBlock \rangle
\end{aligned}
$$

The input parameters to a function are given by a number of typed, formal arguments. The body of a function is a sequence of statements. *DivCon* is lexically (and in particular, block-) scoped, so that the input parameters of a function are in-scope for all statements in its body. Statements within a *DivCon* function may invoke other *DivCon* functions, and may make assignments to in-scope variables. These variables, in turn, may be the input parameters of subsequent (in program order) function invocations, thereby specifying data dependencies. Conceptually, *DivCon* function invocations are *blocking*.

A single *DivCon* function may be marked with the `main` qualifier, indicating that control enters *DivCon* through that function. Finally, unlike in functional languages, *DivCon* functions are *not* first class entities, in that functions cannot be passed as arguments to functions.

Expressions

DivCon expressions are simple in structure and not particularly different from the kinds provided by other, well-established programming languages such as C and C++.

$$
\begin{aligned}
\langle expr \rangle &::= \langle atomicExpr \rangle \mid \langle attrAccess \rangle \mid \\
&\quad \langle arithExpr \rangle \mid \langle boolExpr \rangle \mid \langle invocation \rangle \\
\langle atomicExpr \rangle &::= \langle lit \rangle \mid \langle ident \rangle \\
\langle attrAccess \rangle &::= \langle expr \rangle . \langle ident \rangle \\
\langle invocation \rangle &::= [\ \text{“}\textbf{spawn}\text{”}\ ][\langle expr \rangle .]\langle funCall \rangle \\
\langle funCall \rangle &::= \langle ident \rangle\ \text{“(”}\ [\langle exprs \rangle]\ \text{“)”} \\
\langle exprs \rangle &::= \langle expr \rangle (\ \text{“,”}\ \langle expr \rangle) *
\end{aligned}
$$

The most basic *DivCon* expressions are the *atomic* expression, (i.e. external constants, numeric or Boolean literals, and identifiers corresponding to in-scope, declared variables), and the *attribute access* expression (of the form `x.attr`, denoting a reference to an attribute named `attr` of object `x`). *DivCon* also supports binary arithmetic expressions and binary Boolean expressions. Negation of Boolean expressions is supported via the '`!`' operator. Given their structural similarities to their C++ counterparts, we do not discuss these types of expressions further.

Invocations of functions (whether `serial` or not) are valid *DivCon* expressions. Each argument of an invocation is itself a *DivCon* expression. Scalars are passed into function invocations *by value*, whereas `Array`s are passed *by reference*. As indicated by the EBNF, *DivCon* supports the invocation of C++-style instance methods (e.g. `x.f()`, for some object `x` and its declared instance method, `f`) as well as "normal" functions (e.g. `f(x)` for some in-scope object `x` and some declared function `f`).

Finally, an invocation may be preceded by the keyword `spawn`. From the programmer's point of view, this annotation dictates that a new lightweight task be created for the evaluation of the associated invocation. This provides an opportunity to the programmer to specify task parallelism. An invocation

that is not `spawn`ed is evaluated *in-line*. In § 4.6.1 we will see that the *DivCon* runtime system agglomerates spawned lightweight tasks into chunks of work that have a reasonable grainsize.

Statements

As noted previously, the body of a function is a sequence of statements. We identify the various kinds of *DivCon* statement here.

Statement blocks. A statement block creates a curly-brace-enclosed scope, which comprises a *variable-declaration* preamble, followed by a number of statements.

$$
\begin{aligned}
\langle block \rangle &::= \langle stmt \rangle \mid \text{“\{”} \ \langle varDecls \rangle \ (\langle stmt \rangle) * \ \text{“\}”} \\
\langle varDecls \rangle &::= (\langle varDecl \rangle) * \\
\langle varDecl \rangle &::= \langle type \rangle \langle idents \rangle \ \text{“;”}
\end{aligned}
$$

The variables declared within such a statement block scope are available to all statements therein. Statements within a block are executed in program order. Consider the following fragment of code, which shows the `else` block of the quicksort example from § 4.3.2:

```
if(A.length() <= Tau) ...
else{
  Array<Record> LT, EQ, GT;
  Record pivot;
  ...
}
```

The preamble of the `else` block declares a scalar `pivot`, of type `Record`, and three *DivconArrays* of `Record`s, namely `LT`, `EQ` and `GT`. Note that `pivot` is a value object as would be familiar to C++ programmers, and not a reference (as might be expected by Java programmers). *DivconArrays* are initially empty.

Next, we discuss the four types of *DivCon* statement that may occur in a block.

$$\langle stmt \rangle \;::=\; \langle conditional \rangle \mid \langle assignment \rangle \;\text{“;”} \mid \langle forall \rangle \mid \langle foreach \rangle$$

Conditional execution. An `if-then-else` statement specifies the execution of *either* its *then* branch *or* its *else* branch, depending on the result of the evaluation of an associated Boolean predicate expression.

$$\langle conditional \rangle \;::=\; \text{“}\textbf{if}\text{”}\;\text{“(”}\;\langle expr \rangle\;\text{“)”}\;\langle block \rangle\;\text{“}\textbf{else}\text{”}\;\langle block \rangle$$

Variable assignment. The *DivCon* assignment statement takes the form “`LHS = RHS`”. `LHS` contains a list of *in-scope* variables. `RHS` specifies a comma-separated list of *DivCon* expressions.

$$\langle assignment \rangle \;::=\; [\langle idents \rangle\;\text{“=”}\;]\langle exprs \rangle$$

If variables are present in the LHS, their number must equal the aggregate number of results returned by the expressions in the RHS, as defined next. An identifier expression returns a single result, namely its most recently assigned value. Similarly, arithmetic and Boolean expressions are single-valued. Finally, a function invocation returns a number of results that is implied by the declaration of the corresponding function. The assignment of results to variables occurs in a one-to-one, left-to-right manner.

Statements that express *dynamic* task parallelism

So far, we have not discussed a looping construct within which the *DivCon* programmer may arbitrarily enclose function invocations and task creations. Indeed the absence of such constructs would restrict the scope of the language to the expression of *data-independent* task parallelism, i.e. computations in which the number of (children) tasks spawned, i.e. number of functions in-

voked, by a given (parent) task is determined independently of the input data. Quicksort and related algorithms are an example of this data-independent task parallel pattern: the number of recursive calls made is two, regardless of the input data.

In order to allow a more *dynamic* expression of task parallelism in *DivCon*, we provide the `forall` construct. Our intention is to provide functionality akin to *list comprehensions*, an idea prevalent in many functional languages (e.g. Haskell and OCaml) and more recently, Java (the *for-each* construct) and Python. Such a facility is crucial for algorithms that exhibit *data-dependent* task parallelism, wherein the number of child tasks spawned by each parent depends on the input data of the parent. Examples of this pattern abound in the state space search domain, including N-Queens, unbalanced tree search, etc.

The `forall` construct executes a set of associated statements, once for every element in a C++ data structure object that supports a list-like interface.

⟨*forall*⟩ ::= "**forall**" "(" ⟨*type*⟩⟨*ident*⟩ "**in**" ⟨*expr*⟩ ")" ⟨*forallBlock*⟩

Iteration occurs over a typed, so-called *iteration variable* that is dynamically bound to elements of the list-like data structure specified by the `in` clause. This iteration variable is in-scope for the enclosed block of statements. More specifically, an expression is evaluated, whose result must be a type that implements the *DivCon* list interface. This interface specifies two methods: `hasNext()` and `getNext()`. The *DivCon* runtime system invokes these methods on the list to iterate over the elements of the list. The iteration variable is bound dynamically to each element in the list, thereby creating an instance of the `forall`'s body. In this way, the `forall` loop body is executed once for each list element.

The syntax of the `forall` construct is reminiscent of Java's *for-each* loop, and to similar provisions in the C++-11 standard. The similarity runs a little deeper: the order of iteration over the result of `expr` is determined by the programmer's implementation, and is nominally unspecified. This means that `forall` statements are specified to be non-deterministic. Next, we discuss the structure of the body of a `forall` statement.

Statements embedded within the `forall` *construct.* A `forall` statement

may only enclose a block of statements of restricted type, as represented by the EBNF non-terminal *forallBlock*. These restrictions are placed so as to prevent the expression of dependencies between different instances of the `forall` loop's body, while catering to the requirements of common divide-and-conquer applications.

$$\begin{aligned} \langle forallBlock \rangle &::= \langle forallStmt \rangle \mid \text{“\{”} \ (\langle forallStmt \rangle) * \ \text{“\}”} \\ \langle forallStmt \rangle &::= \langle forallConditional \rangle \mid \langle forallUpdate \rangle \ \text{“;”} \end{aligned}$$

A `forall` statement's body can only consist of *forallConditional* or *forallUpdate* statements. The intent is to allow only such changes to the program's state that associate and commute with each other. Given this restriction, a `forall` statement cannot nest another `forall` statement.

The *forallConditional* statement executes one of two blocks, depending on the value of the associated predicate expression.

$$\begin{aligned} \langle forallConditional \rangle ::= \ & \text{“}\textbf{if}\text{”} \ \text{“(”} \ \langle expr \rangle \ \text{“)”} \ \langle forallBlock \rangle \\ & \text{“}\textbf{else}\text{”} \ \langle forallBlock \rangle \end{aligned}$$

The *forallUpdate* statement comprises a LHS and a RHS separated by the “`+=`” operator. The LHS specifies a comma-separated list of in-scope variables, whereas the RHS specifies a list of evaluated expressions.

$$\langle forallUpdate \rangle ::= [\langle idents \rangle \ \text{“+=”} \ ] \ \langle exprs \rangle$$

Just as in the assignment statement, the number of variables on the LHS must equal the number of results returned by the expressions on the RHS.

The `forall` construct allows one to express data-dependent task parallel patterns such as the N-Queens algorithm, a fragment of which is presented below. (To be discussed fully in § 4.5.1):

```
type Board, BoardQueue;
serial placeNextQueen(Board b) : BoardQueue;
NQueens (Board b) : int {
    if(...) ...
    else{
      int nQueens;

      nQueens = 0;
      forall(Board c in placeNextQueen(b))
        nQueens += spawn NQueens(c);

      return nQueens;
    }
}
```

Above, the programmer-defined `BoardQueue` C++ class must define methods `hasNext()` and `getNext()`. The `serial` function `placeNextQueen` accepts as input a chess `Board` configuration, and returns a `BoardQueue` object with several configurations for the next level of recursive invocations of the `NQueens` function. Each configuration in the `BoardQueue` corresponds to a chess board configuration with a queen in the next row, placed at a viable position. Therefore, for every feasible configuration, '`c`', that results from a current configuration, '`b`', the above code spawns a new task to compute whether any configuration that results from `c` (eventually) yields a solution to the N-Queens problem.

4.4.2 *DivconArrays*

We have already introduced *DivconArrays* as one-dimensional, indexible collections of data elements. Although these collections are distributed over multiple PEs, the abstraction is designed to hide from the programmer, to the extent possible, the artifacts of data distribution.

Scope of application

The *DivConArray* is *not* a general-purpose, distributed shared array. It is meant for the narrowly-defined purpose of communicating large sets of data

between a parent task and its children in generatively recursive computations.

To be more concrete, consider a generatively recursive algorithm such as quicksort, in which a parent task t evaluates an invocation $f(A)$. Here A is a large, distributed array of data elements. In the divide-and-conquer paradigm, t prepares a static number of smaller, but still distributed, arrays of elements $A_1, \ldots, A_k$, based on the elements of A. Typically, each element of A is considered in turn, and is used to *generate* elements within $A_1, \ldots, A_k$. Task t then creates new tasks $t_1, \ldots, t_k$, each t_i of which recursively evaluates $f(A_i)$. The *DivConArray* provides an efficient, and abstract means of generating A_i from A, thereby achieving efficient communication between task t and its children t_i.

As detailed below, the *DivConArray* supports a number of operations that are useful in the context of large-scale, generative recursion. However, not every supported function may be invoked in any given calling context. The set of operations that may be invoked from within `serial`, C++ functions, subsumes the set that may be invoked from within *DivCon* code. In this way, commonly-used operations for which it is easy to provide efficient, parallel implementations, have been made available to *DivCon* code. On the other hand, array operations that are not common to generatively-recursive codes, but which are necessary for interfacing with C++, are available in `serial` functions.

Operations supported within *DivCon*

The following functions may be invoked on a *DivConArray*. Each such invocation is a valid *DivCon* expression.

1. **length.** Returns the number of data elements present within the *DivConArray* on which it is invoked. Example usage:

```
if(A.length() <= Tau) ...
else ...
```

2. **read.** The `read` function accepts a single, integral argument, i, and returns the element at the i-th position of the *DivConArray* on which it is invoked. Example usage:

```
Record r;
Array<Record> a;
int i;

r = a.read(i);
```

3. **scan.** The `scan` function applies the scan operation on the elements of a *DivConArray* A. The resulting *DivConArray*, B, is returned, where:

$$B[i] = \bigoplus_{j<i} A[j]$$

In the above, $\bigoplus$ represents a commutative-associative operation that is applied to elements of A in order to get elements of B. The `scan` function accepts as argument a single, user-specified functor, that encapsulates this operation. Compiler-generated code invokes this functor with the appropriate arguments so as to realize the intended semantics of the scan operation.

```
type Data, ScannedData;
type MyAToBScanFunctor;

Array<Data> A;
Array<ScannedData> B;
MyAToBScanFunctor f;

B = A.scan(f);
```

In the above, we perform the `scan` operation on *DivConArray* `A`, whose elements are of type `Data`. The scan-functor `f`, of user-defined type `MyAToBScanFunctor`, is applied to elements of `A` to obtain elements of `B`, which are of type `ScannedData`. For this purpose, the scan-functor class `MyAToBScanFunctor` must define the following:

```
const ScannedData &operator+=(const Data *in, int numIn) const;
const ScannedData &identity() const;
```

The `operator+=` determines the result of the application of the $\bigoplus$ operator on a set of consecutive elements from the operand *DivConArray*, to

get a partial result for an element within the scan result *DivConArray*. The `identity` function determines, in the terminology of Blelloch [82], the *identity* element of the $\bigoplus$ operator.

4. **concat.** A *DivConArray*, A, may be concatenated with another *DivConArray*, B, by passing the latter as an argument to the `concat` function invoked on the former. This invocation returns as its result a third *DivConArray*, C, whose length is $|C| = |A| + |B|$, and whose elements are as follows:

$$C[i] = \begin{cases} A[i] & \text{if } 0 \leq i < |A| \\ B[i] & \text{otherwise} \end{cases}$$

The following code fragment from the quicksort example in § 4.3.2 illustrates the usage of the `concat` function:

```
type Record;

qsort(Array<Record> A) : Array<Record> {
  ...
  Array<Record> LT, EQ, GT;
  return (spawn qsort(LT)).concat(EQ).concat(spawn qsort(GT));
}
```

5. **free.** When invoked on a *DivConArrayA*, this function frees up all resources associated with A, including all memory allocated to store its data elements. An attempt to access a `freed` *DivConArray* results in a dynamic error.

Unordered iteration over *DivconArrays*

Now, we discuss statements that enable iteration over *DivConArray* elements. These statements enable the expression of intra-task data parallel operations.

The *DivCon* `foreach` construct executes an enclosed block of statements for every element present within some *DivConArray*. The structure of the `foreach` statement is given below:

$$\langle foreach \rangle \quad ::= \quad \text{“}\textbf{foreach}\text{”} \quad \text{“(”} \ \langle type \rangle \langle ident \rangle \ \text{“}\textbf{in}\text{”} \ \langle expr \rangle \ \text{“)”} \\ \langle foreachBlock \rangle$$

As in the case of the `forall` loop, iteration occurs through the binding of *DivConArray* data elements to a typed iteration variable that is in-scope for the block of statements enclosed by the `foreach` construct. An expression is evaluated, whose result must be a *DivConArray*. Only one iteration variable is allowed, i.e. *zippered* iteration is *not* supported. For each element in this resulting *DivConArray*, the associated block of statements is executed once. The ordering between individual executions of the block is unspecified.

Statements embedded within the `foreach` *construct.* As in the case of the `forall` loop, a `foreach` statement may only enclose a block of statements of restricted type, as represented by the EBNF non-terminal *foreachStmts*.

$$\langle foreachBlock \rangle \quad ::= \quad \langle foreachStmt \rangle \mid \ \text{“\{”} \ (\langle foreachStmt \rangle) * \ \text{“\}”} \\ \langle foreachStmt \rangle \quad ::= \quad \langle foreachConditional \rangle \mid \langle foreachUpdate \rangle \ \text{“;”}$$

A *foreachStmt* may either be a conditional, or an update statement. The conditional statement executes one of two blocks, depending on the value of an associated predicate expression, *foreachExpr*.

$$\langle foreachConditional \rangle \quad ::= \quad \text{“}\textbf{if}\text{”} \quad \text{“(”} \ \langle foreachExpr \rangle \ \text{“)”} \ \langle foreachBlock \rangle \\ \text{“}\textbf{else}\text{”} \ \langle foreachBlock \rangle$$

The update statement comprises a LHS and a RHS separated by the “`+=`” operator. The LHS specifies a comma-separated list of in-scope variables, whereas the RHS specifies a comma-separated list of evaluated expressions.

⟨*foreachUpdate*⟩ ::= [⟨*idents*⟩ "+="] ⟨*foreachExprs*⟩

The number of variables on the LHS must equal the number of results returned by the expressions on the RHS. The RHS is constrained to be have a structure given by the EBNF non-terminal *foreachExprs*, which is a comma-separated list of *foreachExpr* non-terminals. These non-terminals represent expressions with the same structure as the previously defined non-terminal *expr*, except that any they must be `spawn`-*free*: A `serial` function is `spawn`-free. A non-`serial` function is `spawn`-free if its body does not contain the keyword `spawn`, *and* every function that is invoked in its body, is itself `spawn`-free. A *foreachExpr* is `spawn`-free if it is either a non-invocation expression, or it invokes a `spawn`-free function. One may think of the `foreach` construct as *mapping* a function onto the elements of a *DivConArray*. This mapped function, then, has a simple form, and in particular, may not initiate other *DivCon* tasks.

The semantics of the '`+=`" operator depends on whether the LHS variable is a scalar or a *DivConArray*. Say that the i-th LHS variable is a scalar. Then, the i-th result on the RHS is accumulated into the scalar variable. For non-primitive types, the accumulation operation is programmer-defined, and given by the C++ `operator+=()` for the type of the i-th LHS variable.

On the other hand, if the i-th LHS variable is a *DivConArray*, the i-th value among the list of returned values of the RHS function is *added* to the *DivConArray*. The type of the i-th result on the RHS must correspond to the array's base type. Since there is no ordering enforced on the execution of different instances of a `foreach` statement block, this addition of elements to the *DivConArray* also occurs in an unspecified order. Thus, one cannot write individual elements within *DivConArray*, but only 'add' to it using `+=`. This scheme avoids the question of coherence of updates. In our experience, this relaxed mode of *DivConArray* update suffices for the divide-and-conquer paradigm and permits optimizations, as discussed in § 4.6.2.

Previously we saw that just like the `foreach` construct, the `forall` construct could also embed update statements with the `+=` operator. However, the programmer is allowed to place *DivConArray* variables in the LHS of an update statement only when the update is embedded within a `foreach`

statement. The reasons behind this restriction are discussed in § 4.6.2.

We illustrate the `foreach` statement by revisiting the parallel partitioning code from the quicksort example of § 4.3.2.

```
Record pivot;
Array<Record> LT, EQ, GT;
foreach(Record r in A){
  if      (r < pivot)  LT += r;
  else if (r > pivot)  GT += r;
  else                 EQ += r;
}
```

Whereas earlier we had explained the appearance of the '`+=`' as an operator that *adds* elements to an array, we can now see that it is part of a `foreach`-enclosed update statement. Since the sole identifier in the LHS of each of the above udpate statements is a *DivConArray*, each `Record` '`r`' in the input *DivConArray* `A` is appended to one of `LT`, `GT` or `EQ`, depending on how it compares with the `pivot`.

Interaction with `serial` C++ functions

DivconArrays may be passed as arguments into `serial`, C++ functions. We have seen an example of this in the quicksort algorithm from § 4.3.2. When a *DivConArray* is passed into a `serial` function in this way, it is *consolidated*. That is, all of its data elements are collected into a single, contiguous local buffer that is accessible to the `serial` function. Given their inherently serial nature, `serial` functions can invoke a larger set of operations on *DivconArrays*. In addition to the `length`, `read`, `concat`, `scan`, `free` and '`+=`' operations, which are also available in *DivCon* code, the following *DivConArray* operations may be invoked exclusively from `serial` functions:

1. **resize.** This function accepts a single argument, n, and reshapes the *DivConArray* on which it is invoked, to allow up to n elements to be stored within it. Any required reallocation of underlying memory buffers is performed by the function.

2. **write.** The `write` function accepts two arguments, the first an integer, i, and the second, a C++ object t, of the same type as the elements of

the *DivConArray* on which it is invoked. The function overwrites the i-th location of the *DivConArray* to t.

4.5 More examples of *DivCon* code

We now discuss two more examples of *DivCon* code. The objective of this section is to present in a concrete context the language constructs discussed previously.

4.5.1 N-Queens

The N-Queens problem is to find, given a chess board of size $N \times N$, the number of configurations of N queen pieces that can be placed on the board such that no queen *attacks* any other. A queen is said to attack another if the two pieces share the same row, column or positive/negative diagonal on the board. There are no known polynomial-time algorithms to solve this problem. Therefore, a brute-force approach with back-tracking is often adopted, in which every possible feasible configuration is explored.

The brute-force algorithm is simple. The *current* row is initialized to the first row, and the board is initially empty. Therefore, any square in the first row is a feasible one. We place a queen at a feasible square on the current row, record the set of feasible squares for the next row, and recursively explore the feasible squares for the next row. This procedure is carried out for each feasible square in the current row, and continues until we encounter a row with no feasible positions, or we finish placing queens on the N rows.

Although there are few practical applications in which one encounters this problem, there are three reasons why it is an important representative of the broader class of combinatorial *state-space search* problems. First, the computation generated by such an algorithm is tree-structured, where each node of the tree is a task (invocation of a recursive function), and a directed edge exists between two nodes if the source of the edge spawned the target. Second, the amount of work done per node is very small: in the case of N-Queens, it is the simple matter of generating a number of children configurations given a parent configuration of the chess board. Third, it is hard to predict the amount of work that will be done in exploring the search tree

rooted at a particular task node (although Kale [83] provides a surprisingly effective heuristic for the particular case of N-Queens).

The *DivCon* rendition of this algorithm is presented below:

```
extern int Tau;
type Board, BoardQueue;
serial seqNQueens(Board b) : int;

NQueens (Board b) : int {
  if (b.isSolution()) return 1;
  else if (b.numRemainingRows() <= Tau) return seqNQueens(b);
  else{
    int nQueens;

    nQueens = 0;
    forall(Board c : b.nextConfigurations())
      nQueens += spawn NQueens(c);

    return nQueens;
  }
}
```

The overall structure of the algorithm matches quite closely the verbal description above. Given a `Board` '`b`', if `b` is a valid solution (`isSolution`), then we notify the caller of a single solution beneath the subtree rooted at `b`. If `b` has fewer than a statically set number (`Tau`) of empty rows, we explore all possibilities in that subtree serially (via `seqNQueens`). Otherwise, we place a queen in each of the currently viable squares using the `serial` C++ function `nextConfigurations`. This function returns a `BoardQueue` containing children `Board`s '`c`', each of which is the same as `b`, except that in `c` the first non-empty row of `b` has a queen in a viable position. A new task, `NQueens(c)`, is spawned for each such configuration `c`. Every spawned task recursively returns the number of solutions found underneath the corresponding node in the search tree.

In the example above, only the fine-grained, parallel part of the computation is expressed in *DivCon*. The definition of the function that `serial`ly explores sufficiently small search subtrees, namely `seqNQueens`, and the generation of child configurations from a parent, i.e. `placeNextQueen`, are both

present in the serial, C++ code. We show the latter below:

```
void placeNextQueen(const Divcon::ConsumedValue<Board> &b,
                          Divcon::ProducedValue<BoardQueue> &q){
  BoardQueue children;
  std::bitset<NUM_QUEENS> mask = b.markedCols |
                                 b.markedNegDiags |
                                 b.markedPosDiags;
  for(int i = 0; i < NUM_QUEENS; ++i)
    if(!mask.test(i)){
      Board child;
      std::bitset<NUM_QUEENS> colMark(1 << i);
      child.markedCols     = (b.markedCols|colMark);
      child.markedPosDiags = (b.markedPosDiags|colMark) >> 1;
      child.markedNegDiags = (b.markedNegDiags|colMark) << 1;
      child.row = b.row + 1;

      children.add(child);
    }
  q.produce(children);
}
```

A bitvector representation of `Board` rows is used, in which set bits denote infeasible squares on the current row. The `placeNextQueen` function iterates over all possible columns, and for every feasible column, marks it as occupied, and updates the bitvector for the next row. We account for diagonal effects of previous rows through the left- and right-shift operations. A `Board` '`child`' is created in this way, and added to the `BoardQueue` '`q`'. This queue is returned to *DivCon* code via the `produce` method. The `BoardQueue` class implements the `getNext` iterator method. This method is invoked by generated *DivCon* code on the returned `BoardQueue` to get arguments for the recursive `NQueens` calls within the `forall` statement.

4.5.2 Oct decomposition

The Oct decomposition strategy is frequently used in scientific and engineering applications to efficiently partition data elements with spatial attributes.

The basic idea behind the algorithm is simple. We begin with a simulation volume (we call this a *voxel* after the terminology of the computational graphics community) that encloses a large number of particles. These particles are arbitrarily equidistributed over a number of processors. Therefore, a voxel's enclosed particles may be distributed over several processors. If a voxel v contains more than τ particles, we divide it geometrically into two non-intersecting sub-voxels, v_l and v_r, whose combined volume is the same as that of v. The procedure is applied recursively to v_l and v_r. Following the precedent of N-body simulators such as ChaNGa [8], we have described the algorithm as dividing each voxel into *two* sub-voxels. However, eight-way division of voxels is equally commonplace; hence the name *Oct*-decomposition.

The following code shows an implementation of this algorithm in *DivCon*, beginning with the declarations.

```
extern int Tau;
type Particle;
type Key;

serial SendParticles(Key k, Array< Particle > P) : void;
serial ToSplitter(Key k) : Key;
```

The variable `Tau` is the previously described voxel size threshold. Particles enclosed within voxels are of C++ type `Particle`. Each `Particle` has a `Key`, which is a 64-bit integer unique to the particle. It is obtained from the position of the particle, using the bit-interleaving technique described by Warren and Salmon [84] and Aluru and Sevilgen [85]. The `Key` defines the location of each particle in the Z space-filling curve (SFC) that runs through a three-dimensional unit cube. The `serial` function `ToSplitter` takes a voxel's `Key` as its only input, and returns the `Key` of the *first* particle in the SFC that lies within the voxel. This is used to determine which particles are within the voxel, and which of them lie outside it.

The `serial` function `SendParticles` is used to transfer particles to members of a collection of Charm++ objects (actors). It accepts the `Key` "`k`" of a voxel, and sends the descriptor of the *DivConArray* that contains its enclosed particles to the object whose identifying index is `k`. This collection of objects is used in the actual gravitational simulation of particles. Since the simulation is not amenable to expression in *DivCon*, we do not discuss it here. (This is done in § 5 and § 6.)

Now for the recursive specification of the Oct-decomposition algorithm:

```
Oct(Array<Particle> P, Key k) : void {
  if(P.length() <= Tau)
  {
    SendParticles(k, P);
    return;
  }

  Array<Particle> Pleft, Pright;

  foreach(Particle p in P)
    if(p.key < ToSplitter(k))
      Pleft += p;
    else
      Pright += p;

  spawn Oct(Pleft, 2*k);
  spawn Oct(Pright, 2*k+1);
}
```

The `Oct` function takes as input a distributed array of particles in *DivConArray* `P`, and the `Key`, `k`, of the voxel that encloses these particles. The following `if` statement checks whether the voxel encloses fewer than τ particles. If so, we invoke the `serial` function `SendParticles`, so as to send the particles in `P` to the Charm++ object that is charged with computing forces on all particles within the voxel.

On the other hand, if the voxel encloses too many particles, we split its contents into a *left* and *right* set of paritcles, namely `Pleft` and `Pright`, respectively. The `foreach` construct is used to iterate over all particles within `P` in parallel. We recursively invoke the `Oct` procedure on these children *DivconArrays*.

4.6 The *DivCon* runtime system

Having familiarized ourselves with the constructs of the language, we now consider the dynamic optimizations performed by the *DivCon* runtime system. These optimizations can be grouped into two categories: (i) techniques

to manage the concurrent execution of fine-grained tasks, and (ii) techniques that enable efficient generative recursion over large sets of input data.

4.6.1 Adaptive grain size control through task agglomeration

In *DivCon* each `spawn`ed function invocation is treated as a task that is independent of all others, subject to data dependencies. In principle each task could be executed in a separate parallel context. However, in order to mitigate the overheads associated with such fine-grained parallelism, the *DivCon* runtime system performs dynamic task agglomeration. In our scheme, multiple tasks are agglomerated into a single *parallel context*, which in the Charm++-centric vocabulary of this thesis, is a *chare* (*cf.* 2).

In this context, good performance can be obtained by carefully considering the following factors:

1. Given that tasks spawn other tasks, how many tasks should we assign to each chare; that is, at what point should the agglomeration procedure create new chares from an existing one? In the terminology of the tree-structured computation literature, *when* should we split the recursion stack?

2. When creating new, children chares from an existing parent, how should the parent's tasks be partitioned among its children? That is, *how* should we split the recursion stack?

3. On which processor in the parallel machine should the newly created children be placed? Equivalently, *where* should we place the pieces of the split task?

There is a significant body of work relating to the efficient execution of tree-structured computations (which are characteristic of divide-and-conquer algorithms) on both shared- and distributed memory machines.

Grama and Kumar [86] present a comprehensive survey of stack-splitting techniques for efficient exploration of the tree structures of recursive computations. A similar compendium in the context of parallel logic programming has been prepared by Gupta *et al.* [87]. Particular paradigms in parallel combinatorial search have also been explored extensively – see the work of

Li and Wah [88] and Lai and Sahni [89] for discussions of techniques and heuristics in branch-and-bound, [90] for parallel iterative deepening A*, [91] for game-tree search and [92] for parallel search of AND-OR trees.

A discussion of the impact of granularity on the performance of OR-parallel programs has been given by Furuichi *et al.* [93]. That paper also provides a multi-level load balancing scheme for multiprocessor systems.

The use of work-stealing as a load balancing strategy was first described by Lin and Kumar [94] and subsequently popularized by the Cilk system [75]. The work on Cilk also provides asymptotic bounds on the performance of the work-stealing approach. More recently, the Scioto framework [95] was designed as a bridge between the task-parallel, and distributed, global address space paradigms. The Scioto runtime system uses differential enqueuing strategies to achieve *locality-aware* stealing of task-based work. Saraswat *et al.* [96] have developed a low-overhead, and scalable variant of work-stealing called *lifeline-based* stealing. Their technique is applicable in the *async-finish* paradigm of X10 and related languages, and precludes the repeated rounds of *active* termination detection that are otherwise required. Finally, Lifflander *et al.* [97] have examined work stealing in the context of large-scale, distributed memory machines. The key observation in their work is that *retention* based stealing can significantly improve the scalability of the work-stealing scheme.

The use of priorities in a variety of parallel search contexts has been outlined by Kale *et al.* [77]. They conclude, as does later work by Loidl and Hammond [69] that a static cutoff criterion for sequential task creation does not suffice in general, especially for computations with irregular tree structures. Kale *et al.* perform adaptive grain size control using the assumption of nearly constant per-node work. The work of Duran *et al.* [98] applies similar findings to the context of OpenMP.

In our work, we adapt the grain size control techniques of Kale *et al.* (more recent work along these lines has been done by the author and colleagues [70]) to enable efficient, task-parallel execution of *DivCon* programs.

Figure 4.1 depicts the decomposition of work over multiple PEs of a parallel machine. Each PE hosts several objects (namely chares). Each chare contains and manages the execution of a number of fine-grained `spawn`ed *DivCon* tasks. The presence of several chares per PE allows the runtime system to dynamically overlap useful computation with communication latency. Ad-

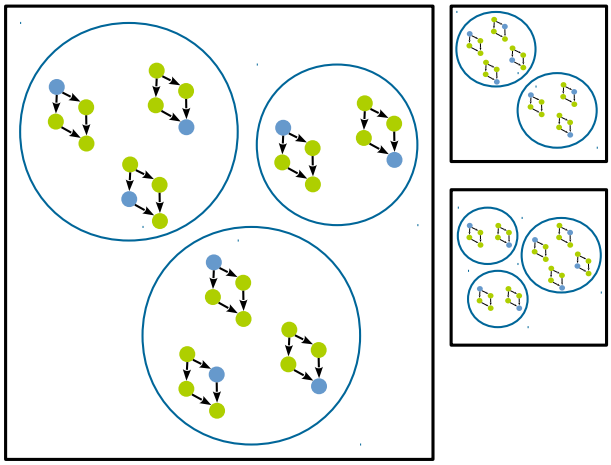


Figure 4.1: Decomposition of work in *DivCon*. Lightweight tasks are depicted as CFGs. A darkly shaded node in a CFG depicts the current state of execution of the corresponding task. Tasks are embedded within parallel contexts called chares (unfilled circles), and there are several chares per PE (unfilled squares).

ditionally, Charm++ provides the wherewithal to migrate chares across PE boundaries, thus enabling dynamic load balancing.

Task execution

We use simple compiler analysis to encode the data and control dependencies in *DivCon* functions into CFGs. Conceptually, the state of each task t can be represented by (1) a stack of nodes that can be processed without violating any control or data dependencies (i.e. the *ready* nodes of t, $ready(t)$); (2) a stack of *activation records*, $vars(t)$, one for each function invocation made by t; and (3) a stack of the number of *pending dependencies*, $pending(t)$ for each invocation in the current call-stack of t. Stacks $vars(t)$ and $pending(t)$ are synchronized, in that for each function invoked by t, a single element is pushed onto the two stacks, and when the function invocation completes, both stacks are popped. The elements in $pending(t)$ map each node m of a particular function's CFG, onto the number of dependencies (both control- and data-) that must be satisfied before m can be executed.

Initially, for each task t, $ready(t)$ contains only the entry node of t's CFG, and $vars(t)$ and $pending(t)$ are empty.

Execution without `spawn`

Let us first discuss task execution in the absence of any `spawn`ed function invocations. This will help us to describe the overall framework first, and then address the question of dynamically spawned tasks. A newly created chare contains a number of tasks t, all of which have been appropriately initialized. A chare iterates over its assigned tasks, and for each task t, pops $ready(t)$ to get n, and executes n. If n represents an assignment statement, the expressions in its RHS are evaluated first.

For each invocation of a function f embedded in the RHS of the assignment statement, an activation record v is pushed onto $vars(t)$ and a map p is pushed onto $pending(t)$. Activation record v encapsulates all the variables of f. All variables except for the formal parameters of f are initially undefined in v. Then, activation record variables on the LHS of the assignment are updated using results obtained from the function invocations. Map p holds the number of pending dependencies for each node m in the CFG of f: $p[m]$ is initially one more than the number of data dependencies in f's CFG with target m. The "+1" provides an implicit test for whether the node on which m is control-dependent, has been executed. Finally, the entry node of f's CFG is pushed onto $ready(t)$.

During the evaluation of an invocation of function f, a node corresponding to a `return` statement causes some expressions to be evaluated, and the results to be assigned to locations within the activation record beneath the current top of $vars(t)$. Then $vars(t)$ and $pending(t)$ are popped, returning control back to the context that invoked f. At this point, the invocation has finished executing.

For any node n that has finished executing, $ready(t)$ is popped. Let $p \leftarrow top(pending(t))$. Then, $p[m]$ is decremented for each m that is control-dependent on n. Additionally, $p[m]$ is decremented for each m that is the target of a data dependency whose source is n. Finally, all m such that $p[m] = 0$ and there exists a directed edge (n, m) in t's CFG, are pushed onto $ready(t)$. If $ready(t)$ is empty, the chare moves to the next t in its list of tasks. Once all the tasks in a chare have been processed in this way, the next chare on the PE is scheduled for execution. Since we assume that no task is allowed to `spawn` children tasks, each chare is scheduled only once.

Handling dynamically `spawn`ed tasks

In the presence of dynamically `spawn`ed children tasks, we must modify the above scheme. Each chare now maintains a *new task queue*. Suppose that chare c is executing a *ready* node n of task t, and the execution of n results in a new task t' being `spawn`ed. Then, t' is placed in the new task queue of c. Moreover, c is marked as *dirty*. This indicates that c must be re-scheduled for execution at a later point in order to process newly `spawn`ed tasks.

The results of a `spawn`ed invocation are not available to the parent task immediately. Therefore, the parent's activation record variables that hold the results of the `spawn`ed invocation are updated at a later point in time. Let $p \leftarrow top(pending(t))$. Then, the associated decrementing of $p[m]$ for those m that are targets of data dependencies from n, are similarly delayed. However, $p[m]$ *is* decremented for nodes m that are control-dependent on n.

When a CFG node corresponding to a `return` statement is executed, the action taken is dependent on the nature of the executing invocation. If the invocation of a function was not `spawn`ed, then the `return`ed expression's evaluated results are used to update the appropriate variables within the parent's activation record. Otherwise, a message encapsulating these results is sent to the chare containing the parent task. As an optimization, if the `spawn`ed task and its parents are on the same PE, the parent's activation record is updated directly.

Creating parallel contexts from collections of tasks

We now examine the question of when it is fruitful to create a number of new parallel contexts (i.e. chare) given a pool of newly `spawn`ed tasks. We adopt a two-phase strategy, similar to the one used by Sun *et al.*, to split the current recursion stack of a chare among concurrent, children chares.

1. In order to maximize processor utilization, the top levels of the recursion tree are eagerly and randomly disseminated across the machine.

2. PE utilization is measured dynamically through repeated reductions. As global utilization increases, the system switches to *adaptive grain size control*.

3. In the adaptive grain size control regime, newly created tasks are placed on the new task queue.

4. Execution time for tasks is measured. Given the fine-grained nature of tasks, a single call to the system timer per spawned task adds too much overhead. However, we assume a low variance of the branching factor across tasks, and therefore amortize the cost of timer calls over several tasks. Thus, we obtain a reasonable estimate of the amount of work done per task.

5. When the cumulative execution time for a chare, measured across all the tasks executed by it thus far, matches a user-provided threshold, the chare creates a number of new, children chares. The number of tasks assigned to each child is such that the *expected* grain size of each child chare is roughly the same as the user-provided threshold. We use the previously calculated average work per task to determine the expected grain size of each child chare.

Distributing chares across PEs

As described above, `spawn`ed tasks are agglomerated into chares to dynamically manage the grain size of computations. We now consider the question of *where* on the parallel machine newly created chares must be placed. The key challenge here is to overcome global load imbalance, while minimizing the amount of communication required for coordination and transfer of work between PEs. We will be brief, since these techniques are not contributions of this thesis. They were developed in the earlier work of Kale *et al.* [71, 99] and Sun *et al.* [70].

DivCon allows the programmer to select (at run-time), one of three techniques for balancing dynamically created chares across PEs. In all of these strategies, once a chare has been assigned to a PE, it never leaves the PE.

1. *Randomized scattering.* Each newly created chare is sent to a randomly chosen PE. This scheme incurs a large communication overhead, especially at scale.

2. *Randomized work-stealing.* We use a basic version of work-stealing, in which a PE becomes a *thief* when it goes idle. A thief picks a randomly

chosen *victim*, and queries it for chares. This process is repeated until the thief obtains some work. While the performance of this scheme is better than that of randomized scattering, the algorithms of Saraswat *et al.* [96] and Lifflander *et al.* [97] could further enhance scalability.

3. *Neighborhood diffusion.* A virtual topology, such as a dense graph, or a k-dimensional torus, of PEs is created. Each PE places its chares on a local queue. This queue is queried periodically by the PE's topological neighbors. Neighboring PEs' queues are equalized by chare migration, so as to reduce the amount of spatial variance in load across the system. Like randomized scattering, but unlike work-stealing, this scheme is *proactive*: PEs exchange chares before they become idle, so as to prevent delays on the critical path. A *time-to-live* (TTL) counter is attached to each chare. A chare's TTL is decremented each time it is migrated during load equalization. When a chare's TTL drops to zero, it is no longer eligible for migration. This helps to limit the amount of communication overhead incurred by the load balancing algorithm.

4.6.2 A distributed array specialized for generative recursion

We now describe the dynamic optimizations performed by the *DivCon* runtime system when operating on *DivconArrays*. As mentioned in § 4.4.2, *DivconArrays* are designed for the purpose of efficiently communicating large, indexed collections of data generated by a parent task, to the children that it `spawn`s. Let us recapitulate this usage scenario. Let f be a recursive function call, that takes as input an indexed collection of data elements, A. Assume that in the generatively recursive computation given by f the task t, evaluating $f(A)$, *generates* elements of $A_1, \ldots, A_k$ *based on* the elements of A. Collections $\langle A_i \rangle$ are processed through recursive calls to f. That is t `spawn`s children $\langle t_i \rangle$ to evaluate $\langle f(A_i) \rangle$, respectively. This situation is depicted in Figure 4.2.

In that figure, rectangular cells represent pieces of distributed arrays. For instance, the collection of 10 cells labeled A_1 represents an array named A_1, which is distributed over 10 pieces. These pieces may be on different PEs. In the following, we assume that arrays are distributed over non-intersecting ranges of PEs. This allows for a compact representation of the PEs that host

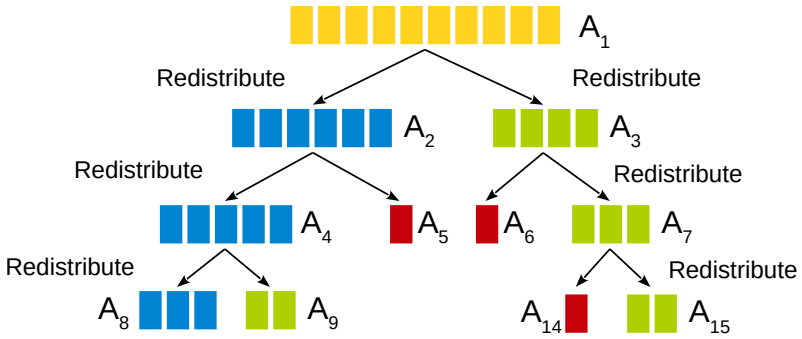


Figure 4.2: Repeated movement of data over network in a typical generatively recursive program.

any given distributed array. We refer to the range of PEs that collectively hold A_i, as P_i. The subarray of data elements of A_i held by PE $p \in P_i$ is denoted $A_i[p]$.

Figure 4.2 depicts the recursion tree for a function f that makes $k = 2$ recursive invocations. Therefore, invocation $f(A_i)$ uses the elements of A_i to generate data elements for A_{2i} and A_{2i+1}. It is convenient to refer to A_i as the parent and A_{2i} and A_{2i+1} as its children. Let the program-specific actions taken by the generative process applied to the parent to obtain its children, be denoted g. Children arrays A_{2i} and A_{2i+1} are inputs to invocation invocations $f(A_{2i})$ (darkly shaded/blue in the figure) and $f(A_{2i+1})$ (lightly shaded/green), respectively.

Eager redistribution

In the following, we describe a typical implementation of generatively recursive computations on distributed machines, e.g. the *team-parallelism* scheme of Hardwick [43].

In every invocation $f(A_i)$, each $p \in P_i$, applies g to every data element $d \in A_i[p]$. The result, $g(d)$ is placed in one of $k = 2$ local buffers, $L_p^{(2i)}$ and $L_p^{(2i+1)}$. Next, a parallel scan operation is performed on all $p \in P_i$, with each p contributing the lengths of its $k = 2$ local buffers, $L_p^{(2i)}$ and $L_p^{(2i+1)}$. The result of the scan determines the ratio for splitting P_i into $k = 2$ sub-partitions, P_{2i} and P_{2i+1}. The scan result also determines the PEs of P_{2i} and P_{2i+1}, to which each $p \in P_i$ scatters elements from $L_p^{(2i)}$ and $L_p^{(2i+1)}$, thereby adding to A_{2i} and A_{2i+1}, respectively. We call this the *redistribution* operation.

Typically, redistribution is performed *eagerly*, i.e. at every invocation of the generative recursive function. When the redistribution has finished, P_{2i} and P_{2i+1} are two disjoint partitions of P_i, holding distributed arrays A_{2i} and A_{2i+1}, respectively. The elements of the arrays are equidistributed over their respective processor partitions. At this point, $f(A_{2i})$ and $f(A_{2i+1})$ can be invoked concurrently. The same process continues recursively, until distributed arrays become so small that it is more profitable to operate upon them serially, than in a data-parallel manner.

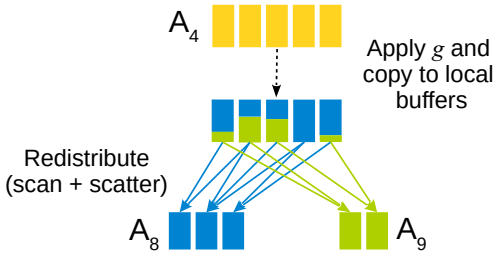


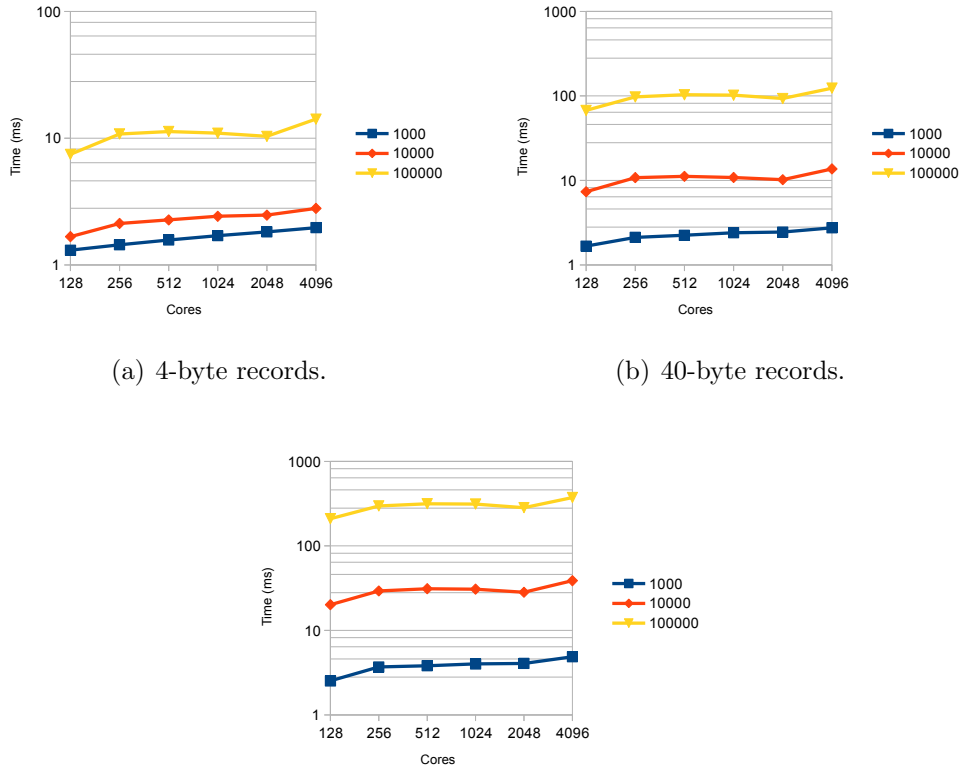
Figure 4.3: The each-to-many scatter inherent in the redistribution of data from a parent invocation to its children.

Figure 4.3 depicts the application of g onto elements of a particular invocation, $f(A_4)$ from Figure 4.2. For this invocation, A_4 is distributed over a range P_4 consisting of five PEs. The (parallel) application of g on each $p \in P_4$ yields differently shaded local buffers on each p. This is followed by a scan operation (not depicted), and an each-to-many scatter operation, resulting in the creation of arrays A_8 and A_9, distributed over partitions P_8 and P_9, respectively. Partitions P_8 and P_9 are sized approximately in the ratio of the arrays that they hold.

The cost of eager redistribution

A natural question to ask is, *what is the cost of repeatedly redistributing data over partitions of processors*? We answer this question by means of a simple experiment. We begin with an array of records, A, equidistributed over a large set of processors, P. Array A represents the input of some recursive invocation $f(A)$. We model the cost of moving data from A to subarrays A_1 and A_2, distributed over equally sized partitions of P, i.e. P_1 and P_2, respectively. We control for time spent in the actual partitioning of $A[p]$

by each $p \in P$, by splitting $A[p]$ into two halves, without actually iterating through and operating upon its individual elements.



(a) 4-byte records.

(b) 40-byte records.

(c) 120-byte records.

Figure 4.4: Average time required to conduct one round of data redistribution, for various sizes of data elements, and various processor counts.

Figure 4.6 graphs the time taken to run a single iteration of this benchmark, averaged over 1000 iterations, for three sizes of data elements, and values of data density (i.e. number of data records per core) Our experiments were conducted on 128 through 4096 cores of the Blue Gene/Q machine *Vesta* at Argonne National Lab. The results indicate that the cost of moving large volumes of data across the network can be significant, reaching almost 400 ms per redistribution for a density of 100000 120-byte records per core, on 4096 cores. These are not contrived values for data density and record size. Similar (and even higher) values have been used by others [100].

Delayed Redistribution

The above benchmark presents a somewhat skewed view of communication cost in generatively recursive divide-and-conquer algorithms. In reality, successive redistributions typically take place over smaller and smaller partitions, so that we would never incur the worst case cost shown above. Nonetheless, one would do well to avoid these costs when possible. In the following, we describe an optimization called *delayed redistribution*, which amortizes this communication cost over multiple recursive function invocations.

We build upon the following observation: If a child distributed array, A_j, is distributed over the *same* range of PEs as its parent, A_i then we needn't generate any network traffic in the creation of the former from the latter. As before, each $p \in P_i$ applies g to $d \in A_i[p]$, and creates a local buffer for each $L_p^{(j)}$, which holds all those data elements $g(d)$ that are to be added to A_j. However, the each-to-many scatter operations on P_i are *not* performed (right away); i.e. redistribution is *delayed*.

As a result, all children arrays A_j are distributed over the same PE range as the A_i, i.e. $P_j = P_i$. Note that the distribution of an *individual* child array over P_j is uneven. However, in the absence of any information about the data elements, the probability distribution of the *total* number of data elements *per PE* in P_j is expected to have low variance. That said, when delaying redistribution, we must perform additional work to determine the mapping of array elements to PEs. Specifically, we must perform a parallel scan operation on the lengths of $L_p^{(j)}$, where $p \in P_j$, to obtain this mapping.

This process can be applied for several steps of recursion, thereby avoiding the redistribution cost at every step. We call this scheme *delayed data redistribution* (DDR).

Figure 4.5 pictorially depicts the operation of the delayed redistribution scheme for the same set of function invocations as in Figure 4.2. The first redistribution is delayed, so that following the generation of A_2 and A_3 from A_1, all three *DivconArrays* are distributed over the same partition of PEs, P_1. Similarly, the redistribution of the children of A_2 and A_3, namely A_4 and A_5, and A_6 and A_7 is also delayed, so that $A_1, \ldots, A_7$ are all distributed over P_1. Thereafter, the system determines that it is beneficial to redistribute $A_4, \ldots, A_7$ to individual sub-partitions of P_1. This results in an each-to-

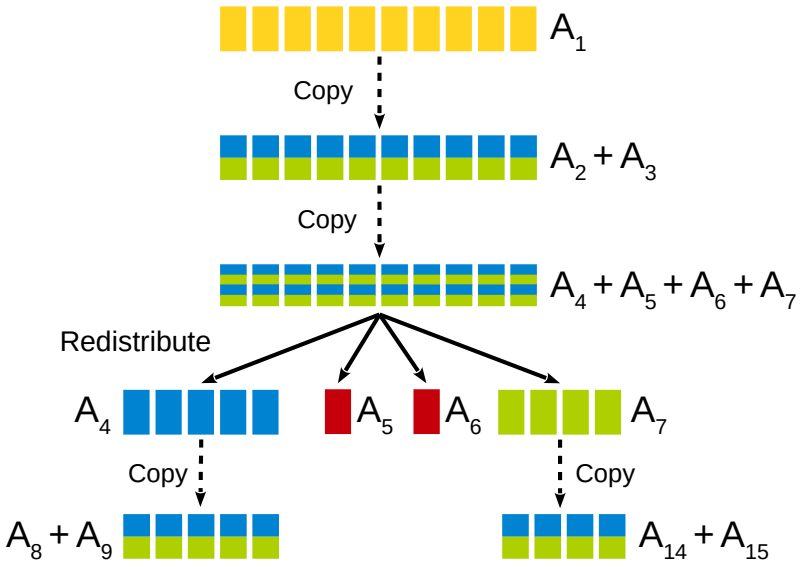


Figure 4.5: Avoiding communication by *delaying* the redistribution of generated data elements.

many scatter, following which, invocations on each of $A_4, \ldots, A_7$ proceed concurrently.

Operation Agglomeration

DDR can be complemented with one further optimization, namely the agglomeration of *DivConArray* operations (e.g. `read`, `foreach` data-parallel operations, `scan`'s, etc.). Put simply, since each partition of processors contains multiple distributed arrays, we collect the operations acting on these arrays, and multicast them to the processors of the partition *en masse*.

Our strategy for agglomeration is as follows: every time a *DivConArray* operation is issued by a *DivCon* task, it is placed in an appropriate (based on the type of the operation), fixed-size buffer. There is one buffer for each type of operation per processor range.

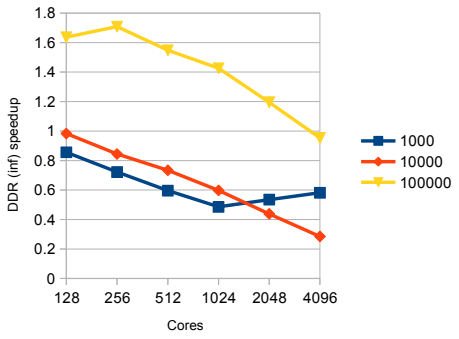
Whenever this buffer fills up, the operations are dispatched in a single message to the corresponding processor range. A conservative scheme to count blocked tasks is employed so as to guarantee progress. That is, when the system determines that it is impossible to issue any more *DivConArray* operations on arrays distributed over a given partition of PEs, all buffered operations for that partition are released.

This agglomeration scheme has two advantages: First, we reduce the communication volume when issuing *DivConArray* operations. More importantly, agglomeration mitigates the problem of load imbalance that arises from delaying array redistributions. This is because even though the number of elements of a *particular* array present on a given processor may vary widely from processor to processor, since there are multiple such arrays distributed over the partition, we would expect that the *total number* of elements assigned to each processor, across all arrays in the partition, would be roughly the same.

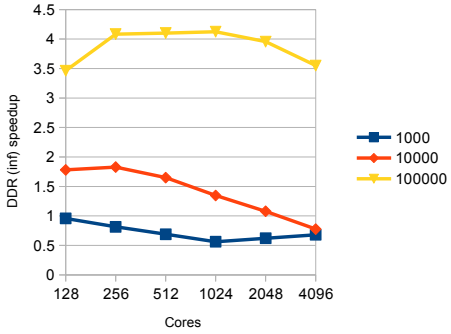
4.6.3 DDR and Operation Agglomeration can improve efficiency

We assess the efficacy of DDR and agglomeration using a benchmark that is more representative of a typical divide-and-conquer workload than the benchmark in § 4.6.2. In particular, we wish to model the cost of redistributions over successively smaller partitions of processors. As in § 4.6.2 we begin with a single, large array of records, A, equidistributed over a set of processors P. As with the previous benchmark, the idea is to split each $A[p]$, for $p \in P$, into two pieces, one meant for each of two children, A_1 and A_2 of A. However, unlike the benchmark in § 4.6.2, each processor p actually iterates through the records within its piece $A[p]$, thereby simulating the work done during the partitioning phase of a typical divide-and-conquer application. With eager redistribution, the code performs the scan+scatter operation to redistribute records meant for A_1 and A_2 from P to its two halves, P_1 and P_2. However, with DDR, A_1 and A_2 remain distributed over the original set of processors, P. This procedure is then recursively conducted on A_1 and A_2, until there are fewer than 500 records per array. We call this the *shuffle* benchmark.

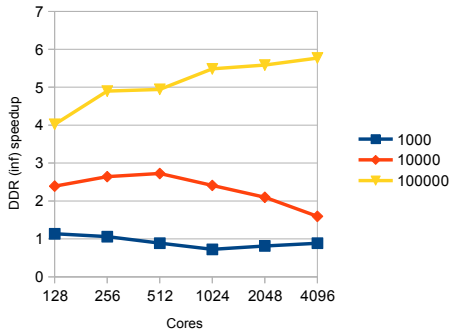
Figure 4.6 compares the performance of the shuffle benchmark on *Vesta* with the eager and delayed redistribution schemes. The three graphs in that figure quantify the relative advantage of avoiding redistributions. Processor counts increase from left to right on the x axis. The y axis plots the ratio $t_{eager}/t_{ddr(\infty)}$, where t_{eager} is the time taken to complete the shuffle benchmark under the eager redistribution scheme, and $t_{ddr(\infty)}$ is the time taken to complete the same benchmark, but avoiding redistributions altogether (i.e.



(a) 4-byte records.



(b) 40-byte records.



(c) 120-byte records.

Figure 4.6: The performance advantage of avoiding redistributions.

redistributing with a periodicity of ∞). As mentioned previously, the DDR scheme is complemented with *DivConArray* operation agglomeration. Each of the three curves in a graph depicts the $t_{eager}/t_{ddr(\infty)}$ ratio for a given data density.

Our results suggest that the combination of DDR and agglomeration is very effective for large record sizes, as well as large data densities. However, for less dense (*sparse*) arrays, or arrays with small records (i.e. *light* arrays), eager redistribution is often better. Even where DDR and agglomeration are effective for sparse/light arrays, their performance advantage diminishes with an increase in the number of processors.

This decrease in the performance of the DDR + agglomeration strategy is due to the following factors:

1. If the per-array cost of metadata management is constant, then the total overhead per PE rises *exponentially* (although with a small con-

stant). Metadata management costs include the calculation of a local memory address corresponding to an element at some index within a *DivConArray*. This operation involves the lookup of a table that maps *DivConArray* keys to local buffer pointers.

2. The message-complexity of mapping `foreach`-embedded code onto elements of *DivconArrays*, decreases more slowly in the delayed redistribution scheme than the eager one. Recall that when we delay redistribution of children *DivconArrays*, they remain distributed over relatively large PE ranges. The application of the generation function g `foreach` element of a *DivConArray* involves a multicast over all the PEs that hold the *DivConArray*. Therefore, the cost is logarithmic in the size of the *DivConArray*'s PE range, and decreases more slowly when redistribution is delayed.

3. Finally, by delaying redistributions indefinitely, processors are unable to fully utilize the increased network bisection bandwidth that becomes available with larger processor configurations. This reduces parallel efficiency.

The performance impact of these inter-related effects is mitigated by the runtime system through demand-driven redistribution of *DivconArrays*. That is, redistribution is delayed by default, but can be triggered on demand. In the current version, the burden of triggering redistribution lies with the programmer. Ideally, the runtime system would dynamically measure the average number of data elements per processor, and automatically trigger redistribution for low-density arrays. We will demonstrate the performance impact of dynamic redistribution in § 4.8.2 and § 4.8.2.

4.7 Provisions for modularity

DivCon code may be compiled into a module by passing an appropriate flag to the *DivCon* compiler at compile-time. The presence of a `main` function in such a piece of *DivCon* code is an error. By compiling the *DivCon* code in this manner, we indicate to the compiler that control flow originates in some other, external, piece of code. This allows the *DivCon* code to be included in the external code as a module, to be instantiated and initiated as required.

In the present work, we only discuss the inclusion of *DivCon* modules into external Charm++ code.

4.7.1 Spawning *DivCon* tasks from external code

In order to begin a *DivCon* computation, one need only `spawn` the root task, providing it the appropriate input arguments. To do this in external code, the following static C++ function template is provided:

```
template< typename T_Task >
void Divcon::Spawn(const Divcon::Task::Config &taskConfig,
                   const Divcon::Array::Config &arrayConfig,
                   const T_Task &root,
                   const CkCallback &callback);
```

The first and second arguments to this function are simple C++ `structs` used to configure the *DivCon* runtime. The first contains attributes relating to task execution, such as the targeted expected grain size of tasks, and the policy used to enqueue newly spawned tasks (LIFO or FIFO). The second configures DDR and operation agglomeration parameters for *DivConArray* management.

The third argument to the `Spawn` function is a C++ object that encapsulates the arguments to the root invocation. The specific type of this object is generated by the *DivCon* compiler based on user code. For every function named `foo` present in a *DivCon* module file, a class named `foo` is defined in the `Divcon` namespace. The constructor for this class has arguments corresponding to the arguments declared in the signature of the *DivCon* function. For instance, assume that the following *DivCon* function is present in a file being compiled as a module:

```
Fib(int n) : int {...}
```

Then, the compiler will generate the following C++ class to encapsulate arguments to the root invocation of `Fib`:

```
class Divcon::Fib
{
  public:
  Fib(int n);
  ...
};
```

An object of the `Divcon::Fib` type can be used to initiate the root invocation of `Fib` from Charm++ code:

```
Divcon::Task::Config taskConfig;
Divcon::Array::Config arrayConfig;

taskConfig.ExpectedGrainSize = 2.0; // in msec
taskConfig.QueueingPolicy = Divcon::Task::Config::LIFO;

// When balancing arrays, split current partition into four
arrayConfig.BalancingSplitPartitions = 4;
// Agglomerate up to 100 operations before flushing
arrayConfig.MaxNumAgglomeratedOperations = 100;

Divcon::Spawn(taskConfig, arrayConfig,
              Divcon::Fib(22),
              CkCallback(...));
```

The above code sets up some configuration parameters for the *DivCon* code, and then instantiates the compiler-generated class `Divcon::Fib` to initiate the computation of the twenty-second `Fib`onacci number. The role of the final argument to the `Spawn` function is illustrated here: When the above computation finishes, control will be transferred to the Charm++ method specified by the `CkCallback` callback argument.

4.7.2 *DivconArrays* in external code

Now let us consider how to initialize and operate upon *DivconArrays* in code that is external to *DivCon*. First we must provide a collection of Charm++ objects to serve as containers for pieces of the distributed *DivConArray*. This is achieved by deriving a Charm++ object collection from the *DivCon*-provided `Divcon::Array::Container` template class:

```
// In .ci file of Charm++ code

// For Divcon Array< Record >,
// where Record is defined by user
class Record;
array [1D] MyContainers : Divcon::Array::Container< Record >
{
  // parallel interface definition
}
```

In this way, the `MyContainers` collection inherits *DivConArray* functionality. `Record`s may be deposited into each container using the `Add` method, and the contents of a container may be examined/modified using the `Get`/`Set` methods, respectively:

```
template< typename T_Data >
void Divcon::Array::Container< T_Data >::
Push(const T_Data &data);

template< typename T_Data >
const T_Data &Divcon::Array::Container< T_Data >::
Get(int idx) const;

template< typename T_Data >
void Divcon::Array::Container< T_Data >::
Set(int idx, const T_Data &data) const;
```

To pass a *DivConArray* reference as an argument to a root function invocation, we use the `Divcon::Array::Descriptor` template class. This class serves as a typed handle to the *DivConArray*. It identifies to the *DivCon* runtime system, the collection of containers over which the corresponding *DivConArray*'s elements are distributed.

```
template< typename T_Data >
class Divcon::Array::Descriptor
{
  public:
  Descriptor(const Divcon::Array::
              CProxy_Container< T_Data > &containerProxy,
             int nContainers);
  ...
};
```

The constructor of the `Descriptor` class expects a proxy to the Charm++ collection of containers over which the *DivConArray* is distributed. Its second argument is an integer that indicates the size of the container collection. More concretely, the code below shows how we would call the `qsort` function defined in § 4.3.2:

```
Divcon::Spawn(taskConfig, arrayConfig,
              Divcon::qsort(Divcon::Array::
                                Descriptor(containerProxy,
                                           nContainers)),
              callback);
```

It assumes that the configuration objects passed as arguments to the `Spawn` function have been initialized. It also assumes that the proxy to the container collection is given by '`containerProxy`', and that the input data to the sorting procedure has been equidistributed over these containers.

4.8 Performance results

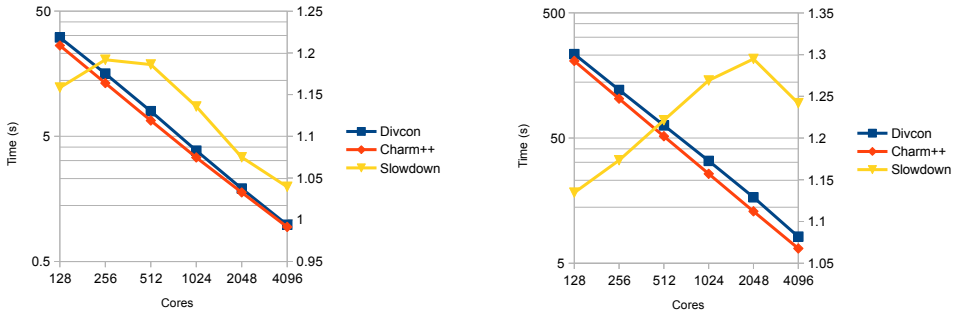
To end this chapter, we consider the performance of a sampling of *DivCon* programs. We present results for several simple benchmarks. These benchmarks are divided into two categories, namely *task-parallel* and *data-parallel* benchmarks. The first of these broad categories emphasizes the efficiency of the *DivCon* runtime system in managing dynamically spawned tasks. The second category highlights the utility of the delayed redistribution technique in the context of generatively recursive algorithms.

4.8.1 Task-parallelism

We present strong scaling results for different instances of the *N-Queens* and *unbalanced tree search* problems. These are standard benchmarks used to assess the efficacy of grain size control techniques in the context of dynamic, tree-structured computations.

N-queens

The objective of this benchmark is to determine the number of queen pieces that can be placed on an $N \times N$ chessboard, such that no queen on the board *attacks* (i.e. is in the same row, column, diagonal or anti-diagonal as) another. A backtracking exhaustive search for this purpose was implemented in *DivCon* in § 4.5.1. That code was a simplification of the variant studied here. The present variant avoids the generation of configurations that are the mirror images of each other, thereby halving the total amount of work performed. This optimization only affects serial code, however. We compare its performance to an equivalent Charm++ program, which has been previously published [70]. Both versions employ a static tree depth cutoff, beyond which each subtree is evaluated as a serial task. In the shallower portions of the tree, adaptive grain size control is performed in either version.



(a) $N = 18$ queens.

(b) $N = 19$ queens.

Figure 4.7: Comparing the performance of *DivCon* and Charm++ code on the task-parallel N-Queens benchmark.

Figure 4.7 shows both versions scale well up to 4096 cores of *Vesta*. Although the performance of the two versions is comparable, the Charm++ version is almost twenty per cent faster on 256 and 512 processors when

solving the 18-Queens problem. This difference diminishes as the core count is scaled up. On the other hand, the Charm++ code demonstrates better scaling on the larger, 19-Queens problem. The gap between the two versions for this particular case rises with the number of cores, going up to a maximum of about thirty per cent on 2048 cores. This difference is likely due to better memory management in the Charm++ version, and a more robust load balancing strategy.

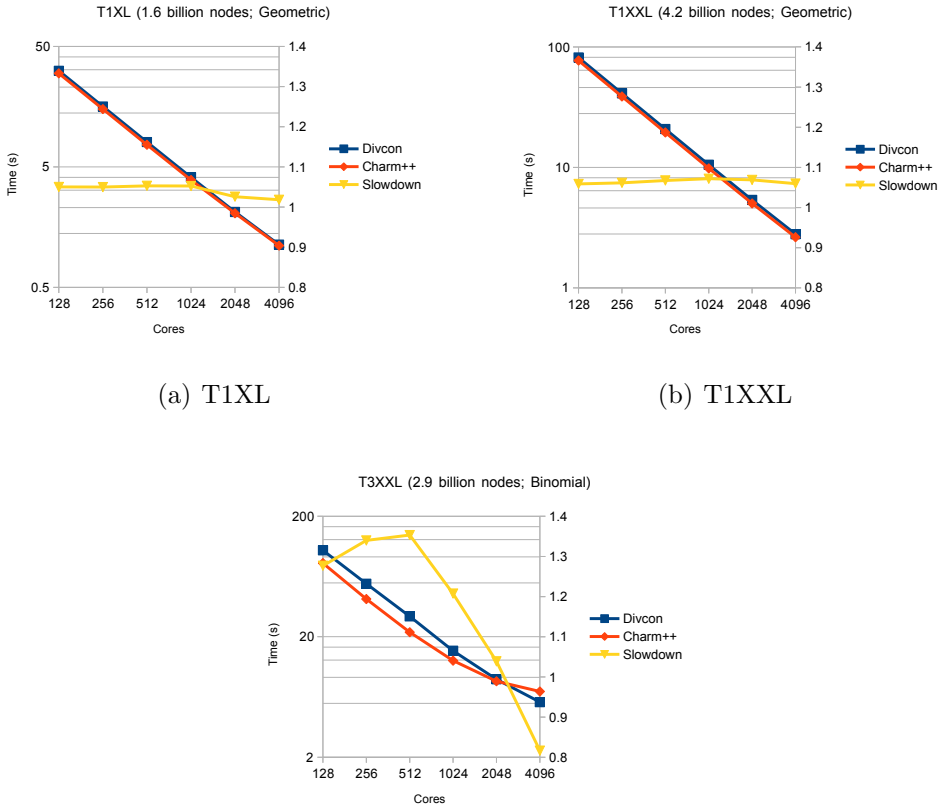
Unbalanced tree search

The unbalanced tree search (UTS) benchmark [101] is a means of testing grain size management in the context of *irregular* tree-based computations. Briefly, the UTS benchmark explores a tree, starting at the root, in which the number of children of a given node is a random variable with a given probability distribution. The benchmark provides a variety of these distributions, and varying their parameters allows one to generate trees of different sizes and structure.

In such computations, the depth of a task bears little or no correlation to the amount of work required to completely explore the corresponding subtree. This is in direct contrast to, for instance, the N-Queens benchmark, wherein the deeper one descends into the computational tree, the lower the expected amount of work in exploring it. In such a case, one cannot rely on a static depth cutoff beyond which all subtrees are explored sequentially. Although it was originally intended to measure the efficacy of different flavors of work-stealing, we use it here as a stress-test for the adaptive grain size scheme of *DivCon*.

As with the N-Queens benchmark, we compare the performance of a *DivCon* code for this benchmark, with a corresponding Charm++ version from previously published work [70].

Figure 4.8 compares the strong scaling performance of the *DivCon* and Charm++ codes on three types of unbalanced tree. The parameters that determine the structure and size of each tree are available elsewhere [70, 101]. Here we simply note that *DivCon* code remains competitive with its Charm++ counterpart on the two trees with a Geometric distribution of children nodes, but is up to thirty five per cent slower on the Binomially distributed tree. On the other hand, at the 4096 core mark, the *DivCon*



(a) T1XL

(b) T1XXL

(c) T3XXL

Figure 4.8: Performance comparison of *DivCon* and Charm++ code on another task-parallel benchmark, namely *unbalanced tree search* (UTS).

code is about twenty per cent faster than its Charm++ counterpart on the Binomial tree search.

4.8.2 Data-parallelism

Next we examine the ability of *DivCon* to effectively manage data parallelism. This data parallelism results from `foreach` constructs within *DivCon* code. Our main aim here is to demonstrate that the DDR and agglomeration techniques developed in § 4.6.2 increase the scalability of divide-and-conquer codes that operate on large, distributed arrays of data.

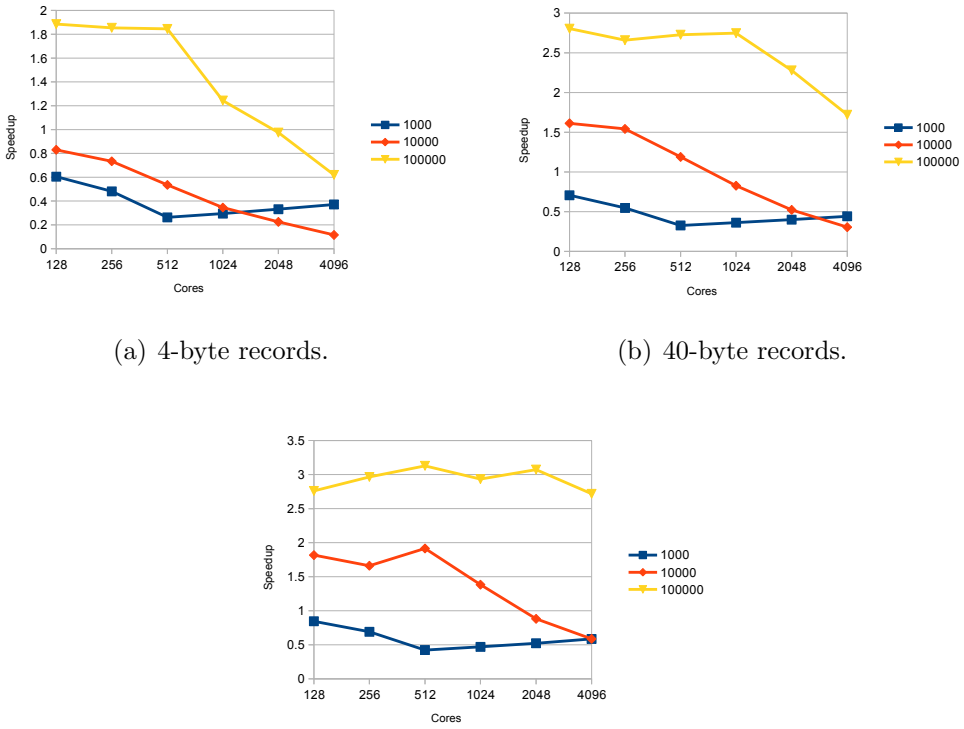
Quicksort

Our first benchmark is the well-known *quicksort* algorithm, adapted to a distributed memory setting. The partitioning of each input array across a pivot is performed out-of-place, resulting in memory transfers as well as network traffic.

The Charm++ version of this benchmark uses the *team-parallel* technique of Hardwick [43], and therefore eagerly redistributes newly created arrays (i.e. `LT`, `EQ` and `GT`). The *DivCon* code for this benchmark adapted from § 4.3.2. We note the use of the `foreach` construct in that code for the out-of-place partitioning operation. The *DivCon* runtime system agglomerates these (as well as the `read`, `free` and `serial` function tasks `seqSort`). The redistribution of newly created *DivconArrays* is delayed indefinitely (*cf.* § 4.6.2). However, one large all-to-all redistribution step was performed at the end, in order to collect distributed data onto single processors for invocations of serial sorting tasks. This redistribution is automatically triggered by the use of *DivConArray* contents by `serial` tasks `seqSort`. As a result, each *DivConArray* that is an input to a leaf task in the recursion tree, is collected onto a single (randomly assigned) processor, where the `seqSort` operation is invoked on it.

Figure 4.10 presents the comparison between the *DivCon* and Charm++ codes on the quicksort benchmark. As before, we present results for different values of record size, data density and processor count. The x axis of each graph in Figure 4.10 plots increasing processor counts, and the y axis, as in § 4.6.3, the ratio $t_{eager}/t_{ddr(\infty)}$. Here t_{eager} is the time taken by the Charm++ version to sort the input array (with eager redistribution), and $t_{ddr(\infty)}$, that taken by the *DivCon* code with infinite delay of *intermediate* redistributions (but including one large all-to-all operation at the end). We see the greatest improvement in performance with arrays that have large data densities and record sizes (Figure 4.10(c)). In fact, for arrays with the smallest sized records and low data densities, the eager redistribution scheme is clearly superior (Figure 4.10(a)). These results are in keeping with the general trends seen in § 4.6.3.

Therefore, one might ask whether judicious redistribution of intermediate arrays can lead to an increase in performance for input arrays with low data densities and record sizes. We endeavor to answer this question next.



(a) 4-byte records.

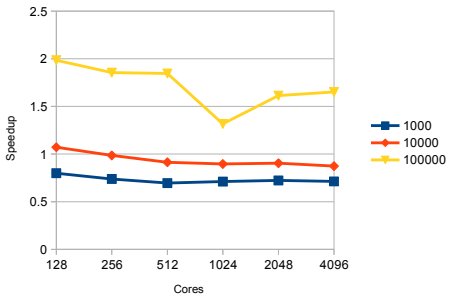
(b) 40-byte records.

(c) 120-byte records.

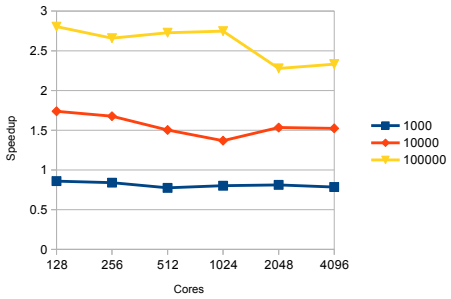
Figure 4.9: Performance comparison of *DivCon* and Charm++ code on *quicksort* benchmark.

To obtain the results in Figure 4.10 we ran the *DivCon* quicksort benchmark with user-triggered redistribution of arrays created at intermediate depths of the recursion tree. In effect, redistribution was triggered periodically at different depths of the recursion tree. For instance, given a period of 4, and starting with an input *DivConArray* distributed over 1024 processors, all *DivconArrays* created at depths 0 (the root, which generated 3 *DivconArrays*, `LT`, `GT` and `EQ`) 1, and 2 remained distributed over the initial 1024 processors. All *DivconArrays* created at depth 3 were redistributed by the runtime system using a greedy assignment of arrays to a fixed number (here, 4) of equally sized partitions of the original 1024 processors. Thereafter, *DivconArrays* created at depths 4, 5, and 6 remained distributed over partitions of size 256 processors, whereas those created at depth 7 were redistributed over partitions of size 64 processors each, etc.

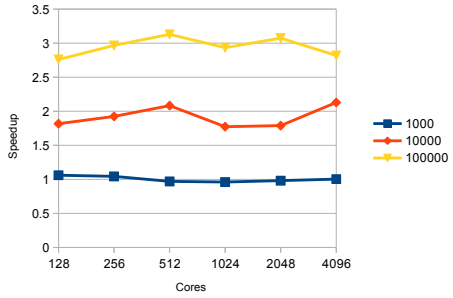
We ran these experiments with various periodicities, and calculated the best time obtained across periodicities for a given data density, record size



(a) 4-byte records.



(b) 40-byte records.



(c) 120-byte records.

Figure 4.10: The same benchmark as in Figure 4.9, except that the *DivCon* code performed user-directed, dynamic delayed redistribution.

and number of processors over which the input array was distributed. This time is denoted t^*_{ddr}. Each graph in Figure 4.10 plots the ratio t_{eager}/t^*_{ddr} for different configurations of input array and processor configuration. Our intent here is to show that if a proactive runtime strategy were available to dynamically determine the profitability of redistribution, one could combine the benefits of communication avoidance with DDR at scale, with those of the eager redistribution strategy for lighter and sparser arrays.

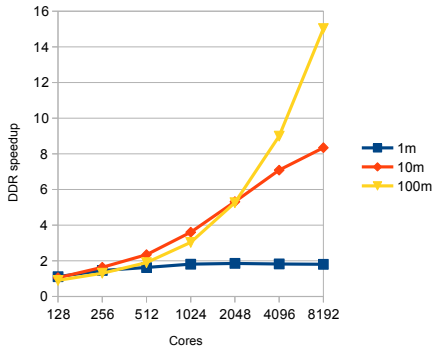
Finally, we remark that in our results, we found that no one statically set value for DDR periodicity outperformed the eager scheme over *all* configurations of input array and processor count. Therefore, a dynamic scheme is essential to ensure that we get good performance across the board.

Oct decomposition

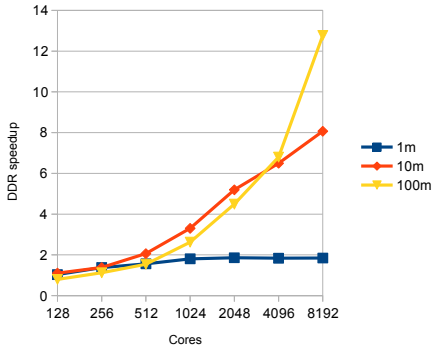
As our last benchmark, we consider the performance of the *DivCon* code for Oct decomposition, introduced in § 4.5.2. The algorithm is widely used in the simulation of particles with spatial location, subject to a force field. To recapitulate the discussion in § 4.5.2, the algorithm geometrically decomposes the simulation volume, and therefore the set of particles, over a set of processors. The input to the algorithm is a set of particles with locations in three-dimensional space, and its output is a set of non-intersecting voxels, each of which contains no more than a set threshold, τ of particles. Beginning with the root voxel, which encloses all particles, the idea is to gradually partition each voxel v with more than τ particles in it, into axis-aligned sub-voxels $\langle v_i \rangle$, such that: the v_i are contained within v, are non-intersecting, and have a combined volume that equals v's.

We compare the *DivCon* code from § 4.5.2 with the implementation of the Oct decomposition strategy used in a previously published Barnes-Hut mini-app [26] written in Charm++. The Charm++ version maintains a list of *pending* voxels, each of which encloses an unknown number of particles. The algorithm is iterative: initially, the root voxel, enclosing all simulated particles, is placed on the *pending* list. In every iteration the algorithm performs the following two operations. First, the list of pending voxels is broadcast to the set of processors. Second, each processor determines the number of particles owned by it, that are enclosed by each voxel in the pending list, and contributes these counts to a reduction over all processors. The reduced counts are used to modify the *pending* list. All voxels enclosing up to $\tau = 500$ particles are simply removed from the list. All voxels enclosing more than τ particles are replaced in the list by two children sub-voxels. The algorithm continues as long as the *pending* list is non-empty.

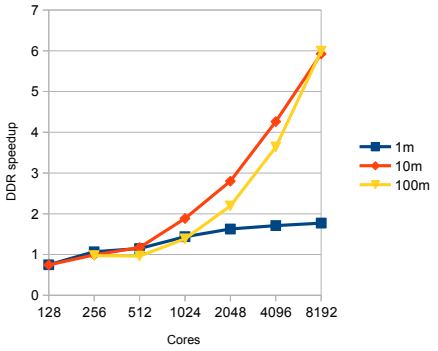
As with the quicksort benchmark, we ran the *DivCon* code with various redistribution periodicities, and collected the best times obtained across all periodicities for a given particle size (64, 96 or 176 bytes), number of particles (1 million, 10 million and 100 million) and processor count (256 through 8192). These numbers resulted in plots of speedup over the Charm++ version, depicted in Figure 4.11. The results are impressive: dynamically redistributing *DivconArrays* can reduce decomposition time by more than an order of magnitude. These gains are more consistent than those seen for the quicksort



(a) 64-byte particles.



(b) 96-byte particles.



(c) 176-byte particles.

Figure 4.11: Performance on the Oct-decomposition benchmark.

benchmark, because the Charm++ version never performs any redistribution at all. However, the cost of redistributing particles grows with particle size, and our gains diminish as we move from 64-byte particles through to 176-byte particles. We have attempted to keep particle sizes realistic, and approximately equal to those used in real applications (e.g. ChaNGa [102]) and benchmarks (e.g. the Barnes-Hut benchmark [26]).

Once more, the point here is that the ability to redistribute *DivconArrays* dynamically, and to delay their redistribution when it is profitable to do so, confers a significant performance advantage. The *DivCon* infrastructure provides this ability. However, we must couple it with a good, dynamic strategy to determine when it is profitable to delay distributed array redistribution, and when to effect it, in a divide-and-conquer setting. Such a facility is currently lacking in the *DivCon* runtime system.

4.9 Productivity

In this section, we compare the previously discussed *DivCon* codes with their Charm++ counterparts, looking for differences in algorithm expression, and hence implied gains in productivity.

4.9.1 Task-parallel benchmarks

We begin with the smaller, task-parallel benchmarks, N-Queens and UTS. The *DivCon* rendition of the N-Queens benchmark comprised a total of 159 lines of code, whereas the Charm++ version comprised 206 lines. It is noteworthy that the Charm++ version was based on a framework for state space search applications [70]. Therefore, code required for efficient grain size management was provided by the framework, and is not included in the SLOC count presented here.

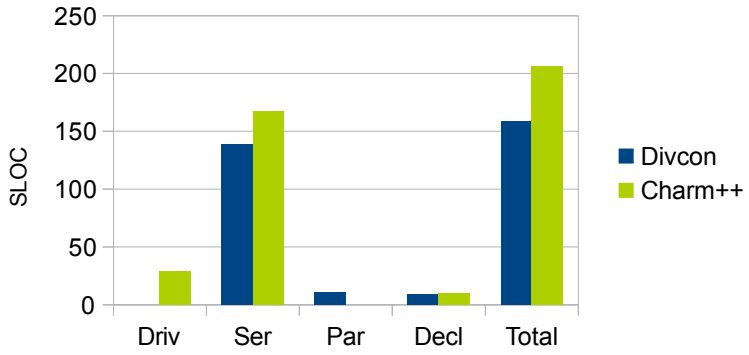


Figure 4.12: Comparison of source lines of code (SLOC) for the N-Queens benchmark.

In spite of this, we saw a modest, 23 per cent, reduction in overall SLOC for the N-Queens benchmark. Figure 4.12 reveals that the majority of the benchmark consists of serial (Ser) code, namely, the construction of child board configurations from the parent. This code remained nearly identical between the two implementations. The Charm++ version also contained driver (Driv) code, in order to interface with the Charm++ runtime system. This code is automatically generated in the *DivCon* version. Importantly, the *DivCon* code embodies a separation of parallel structure (Par) from serial

code. Remarkably, this parallel structure was expressed in a total of 11 lines of *DivCon* code, as shown in § 4.5.1.

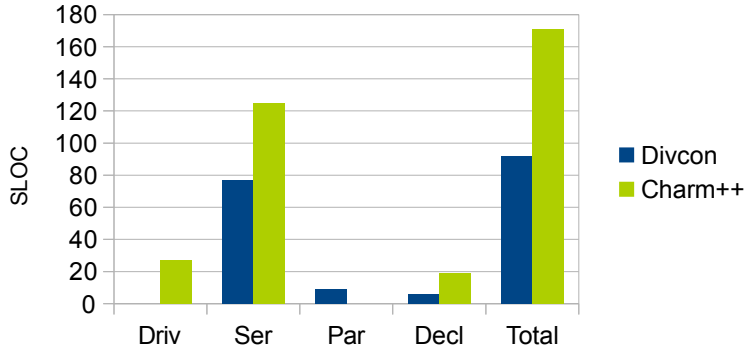


Figure 4.13: Comparison SLOC for the UTS benchmark.

Next, we consider productivity gains for the UTS benchmark. Overall, the *DivCon* version of the benchmark had a total of 96 lines, and the Charm++ version, 171 lines: a total reduction in SLOC of 44 per cent. The break-up of these SLOC into various categories is shown in Figure 4.13. Once more, the Charm++ version was based on the previously mentioned state space search framework. This pushed the responsibility for grain size management, which is a crucial aspect for this benchmark, into the framework/runtime system. This also meant that the parallel structure (Par) of the Charm++ code was implied by the constraints of the state space search framework, and not explicitly specified. On the other hand, the parallel structure of the algorithm can be expressed in all of 13 lines of *DivCon* code:

```
Uts(UtsNode n) : uint64_t {
  uint64_t n;

  n = 0;
  forall(UtsNode c : GetChildren(n))
    n += spawn Uts(c);

  return n;
}

main() : uint64_t {
  spawn Uts(GetRootUtsNode());
}
```

In the above, the definition of the `UtsNode` data structure, and the `serial` functions `GetChildren` and `GetRootUtsNode` are included in the serial code (Ser). As with the N-Queens benchmark, the size of the serial code accounts for the majority of the program.

4.9.2 Data-parallel benchmarks

Now we discuss productivity gains for the larger, data parallel benchmarks.

The Charm++ version of the quicksort benchmark includes code for a lot of the functionality that is handled by the runtime system underlying *DivconArrays*. This includes code for mapping the pivot-partition operation onto the elements of distributed arrays, the underlying containers for the distributed array, and a parallel scan operation that precedes array redistribution. The Charm++ version also implements continuation-based task management, whereas DAG management for tasks is done by the compiler-generated code in the case of the *DivCon* version.

Figure 4.14 shows the composition of the *DivCon* and Charm++ versions of this benchmark. For the Charm++ implementation, we base the Map and Scan operations on existing MapReduce and parallel scan libraries, respectively. The code included as part of the benchmark only contains what is required to interface with these libraries. Regardless, the code for Map/Scan, combined with the implementation of containers for distributed arrays (Arr) accounts for a significant portion (893 lines; 53 per cent) of the Charm++ ver-

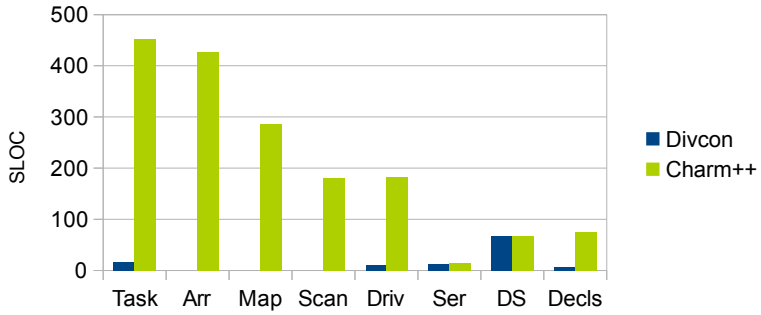


Figure 4.14: Comparison of SLOC for the quicksort benchmark.

sion. Task management adds another 453 lines (27 per cent) to the Charm++ code. In the *DivCon* version, however, task DAGs are managed by the runtime system. In fact, the quicksort task's DAG is generated from a total of 17 lines of *DivCon* code. The two versions are evenly matched in terms of serial code (12 lines for *DivCon*, and 14 for Charm++) and the definition of the sorted record's C++ class (68 lines each). As a result, we see a reduction in SLOC of 93 per cent over the Charm++ version.

Finally, we present productivity results for the Oct-decomposition benchmark. The algorithmic structure of the benchmark was discussed in § 4.5.2, and its Charm++ implementation in a tuned *Barnes-Hut* mini-app was discussed in § 4.8.2. Here, we recall that the Charm++ implementation performed a (data-dependent) number of multicast-reduction iterations, with each iteration dividing above-threshold voxels in two sub-voxels. This relatively simple parallel structure results in a Charm++ code that is only 340 lines long. Even so, its *DivCon* counterpart, at 162 lines long, represents an an overall reduction in SLOC of 52 per cent.

As has been the case with other benchmarks, both versions have similar amounts of serial code (Ser). In Figure 4.15 the iterative broadcast-reduction *histogramming* code is presented as the "Histo" column. Revisiting the code box in § 4.5.2, we see that this histogramming is *implied* in the *DivCon* version: we invoke the `P.length()` function to check whether the *DivConArray* of particles enclosed within a particular voxel has sufficiently few elements.

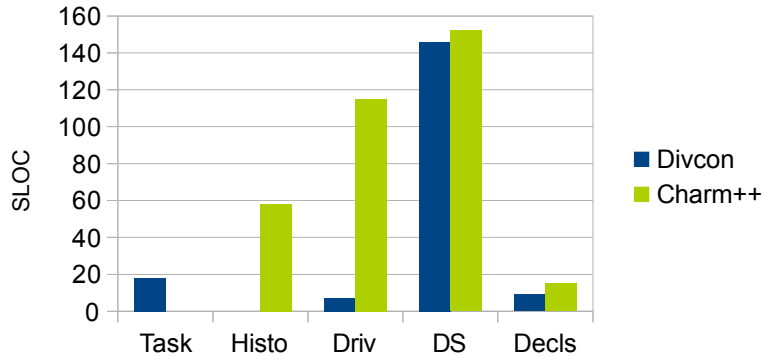


Figure 4.15: Comparison of SLOC for the Oct-decomposition benchmark.

4.10 Conclusion

In summary, the task-parallel benchmarks over which we assessed the *DivCon* language have demonstrated that compiler-generated code can be competitive with hand-written Charm++ code in terms of performance. *DivCon* also provides an automated grain size control mechanism and a succinct expression of recursive algorithmic structure. We believe that these factors contribute to an improvement in productivity. Using SLOC as a crude indicator of productivity, we have shown that task-parallel *DivCon* programs are compact, and allow the programmer to focus on serial code, rather than parallel structure.

For the data-parallel benchmarks, we saw that delayed redistribution and agglomeration performed by the *DivCon* runtime system can lead to better performance, particularly for large data ensembles. We believe that by combining *DivCon*'s infrastructure for delayed redistribution, with a dynamic strategy that assesses the feasibility of redistribution, one can attain good performance across data ensemble sizes, and processor counts. The productivity gains for data-parallel divide-and-conquer programs can be especially noteworthy, given that the *DivCon* language obviates the need to implement generic containers for distributed arrays.

In its current form, the *DivConArray* abstraction has several limitations. First, the strategy for agglomeration of *DivConArray* operations relies on the association of a single array with each task. Second, the dynamic redistribution infrastructure relies on the user to trigger redistribution. Furthermore,

the *DivConArray* abstraction supports a limited set of operations and has a restrictive semantics. It is thus intended for use in the narrow case of divide-and-conquer algorithms with generative recursion on large arrays of elements. It remains to be seen whether the optimizations developed in this thesis can be extended to more general use-cases. More generally, the *DivCon* language would benefit from the provision of other distributed data structures besides arrays. For instance, a distributed graph data structure is required for the implementation of algorithms such as recursive Delaunay mesh generation [103]. We will address these limitations in future work. For now, we conclude by saying that *DivCon* provides a productive means for the expression of task- and data-parallel algorithms. It incorporates a number of optimizations, such as adaptive grain size control, and delayed array redistribution, to improve performance of divide-and-conquer algorithms.

CHAPTER

5

DISTRIBUTED TREES

5.1 Introduction

Tree-based data structures have countless applications in HPC. Such data structures are a natural fit for applications that require the hierarchical arrangement of an underlying set of data elements in order to achieve efficiency. Examples of such applications include N-body problems in computational cosmology, high performance graphical rendering, Lagrangian gas dynamics, granular dynamics, high throughput correlational statistics of large astrophysical data sets, etc.

Especially in HPC settings, it is desirable to distribute not just the underlying set of data elements, but also the tree structure constructed over it, across a number of processors. This means that the programmer must implement parallel operations to build and maintain the distributed data structure. The main purpose of this chapter is to provide a framework that removes this burden from the programmer. We present the design of a framework called *Distree*, which enables the productive expression of object-based, explcitly decomposed, distributed memory applications based on distributed

trees. The *Distree* framework has the following provisions for productivity and performance:

1. It is flexible, in that it allows the programmer to decide the: (i) decomposition of the tree over multiple processors, (ii) the precise structure of the tree's nodes, and (iii) the relevant attributes of tree nodes to be serialized when subtrees are to be communicated over the network.

2. It automates tasks such as data decomposition, tree construction and consolidation of pieces of the tree across SMP processors.

3. It presents an abstract model of traversals over the distributed tree, and separates them from the computational operations to be performed during tree traversal. This modular separation allows the reuse of traversals across different tree-based applications. The framework provides implementations of common traversal types.

4. *Distree* supports fine-grained computational operations on the tree data. The runtime system associated with the framework is based on Charm++, and optimizes performance through data and communication aggregation techniques.

We first describe the tree data structure that forms the basis of the *Distree* framework. We also consider its distribution over a set of PEs.

5.2 A tree data structure for parallel HPC applications

Programmer-defined *data elements* form the basic entitities from which the distributed tree is built. Such data elements could be, for example, particles in the case of an N-body simulation, or triangles in a tree-based rendering engine. Data elements are arranged hierarchically by placing them in the tree structure. In the code box below, we provide the definition of the node data structure as a C++ template class. The template has as its sole parameter a `TreeDataDescriptor`. This is provided by the programmer when instantiating the tree. The `TreeDataDescriptor` class collects the definitions of appropriate data types (e.g. the types of the key, data elements, and node payload). These types are chosen by, and therefore known only to the programmer, but required by the framework to create and manage the tree data structure.

```
template <typename TreeDataDescriptor>
class Node {
  TreeDataDescriptor::Key key;
  vector<Node<TreeDataDescriptor> *> children;
  TreeDataDescriptor::DataElement *data;
  int nData;

  TreeDataDescriptor::NodePayload payload;
};
```

Each node has a *key* that identifies it uniquely among all nodes in a distributed tree. We use the scheme of Warren and Salmon [84, 104]) to assign keys to nodes. The root of the tree is assigned a key of 1: $key(root) = 1$. For all $n \neq root$, $key(n) = (key(parent(n)) \ll k) + i$, where $i \in \{0 \ldots k-1\}$ is the rank of n among its siblings.

A node also contains a list of pointers to each of its children, as well as a pointer to the data elements that are associated with it. This association is typically decided by some aspect of the tree-based algorithm. For instance, in an N-body computation, the data elements associated with a node are the particles that are contained within its extent. Each node has a programmer-provided *payload* (a class encapsulating the attributes that summarize the node for the purpose of the programmer's tree-based algorithm). Continuing with the N-body example, the payload associated with each node is typically a set of so-called *multipole moment* expansions, which compactly summarize the distribution of mass within the tree node. An example of such a tree is shown in Figure 5.1.

5.2.1 Decomposition

Tree T (comprising nodes $N(T)$ and edges $E(T)$ between them) must be decomposed (explicitly, either by the programmer, or using one of the *Distree* decomposition procedures) into a number of subtrees called *tree pieces*. Each tree piece is encapsulated within a message-driven Charm++ object called a *chare* (*cf.* § 2). Typically, there are several hundred to a few thousand data elements (depending on the computational cost of the calculations associated with each element) per chare, and several (on the order of ten) tree pieces

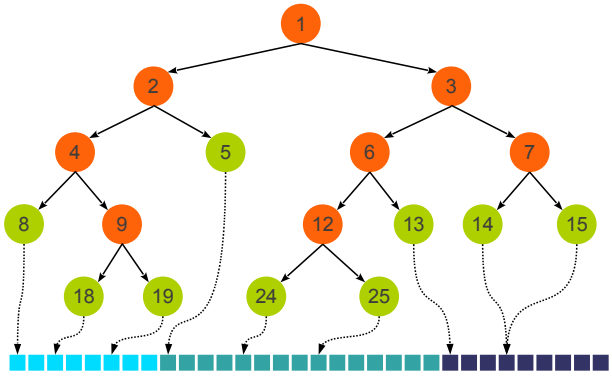


Figure 5.1: Schematic of a tree constructed over a set of data elements. Data elements are shown as squares at the bottom. Each node contains a pointer to the contiguous set of data elements that has been assigned to it, although only the pointers from leaf nodes to data elements are shown (as dashed lines) for clarity.

per PE. As mentioned in § 2.2 this coarseness is necessary to amortize the cost of scheduling message deliveries to objects.

A tree *leaf* is the unit of data decomposition over pieces: each leaf (containing several particles) in $N(T)$ is assigned to a single tree piece. Suppose that the set of all leaves assigned to p is given by $L(p)$. Then, the set of nodes contained within p, $N(p) = E(p) \cup R(p) \cup S(p)$, with each of the components described below:

Exclusively owned nodes, $E(p)$

If $n \in L(p)$ then $n \in E(p)$, i.e. all leaves assigned to a piece p are owned exclusively by it. This makes sense given the assignment of each leaf to a single tree, so that $n \in L(p) \wedge n \in L(p') \Rightarrow p = p'$. If there is a node $n \in N(T)$ such that for all children c of n, $c \in E(p)$, then $n \in E(p)$. It is easy to see that for all pieces p and p' with $p \neq p'$, $E(p) \cap E(p') = \emptyset$.

For the purpose of the tree-based algorithm, the properties of an exclusively owned node of p are *completely* determined by locally available tree data, i.e. data elements within leaves assigned to p. Figure 5.2 shows the tree in Figure 5.1 decomposed over three tree piececs. Exclusive nodes are shown in green (leaves) and orange (non-leaves): nodes 4, 8, 9 and 19 in tree piece 0, and 5, 12, 24 and 25 in piece 1 are examples of exclusively owned nodes.

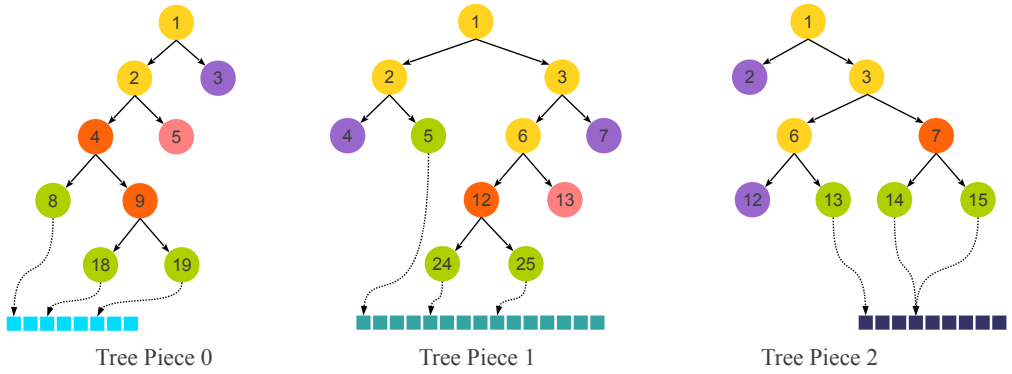


Figure 5.2: Distributing the tree over several pieces. The distribution entails the addition of remote nodes (node 3 in piece 0; 4 and 7 in piece 1 and nodes 2 and 12 in piece 2) and remote leaves (node 5 in piece 0 and node 13 in piece 1).

Remote nodes, $R(p)$

A node $n \in R(p)$ if $n \in E(p')$ for some $p' \neq p$. Therefore no descendant of n is a leaf of p, so that the properties of n are completely determined by tree data that is remote to p. Nodes 3 and 5 in piece 0, and nodes 2 and 12 in piece 2 are examples of remote nodes.

Shared nodes, $S(p)$

A node n is shared by a piece p (among others) if $n \notin E(p)$ but *some* descendant of n is a leaf assigned to p. More formally, $n \in S(p)$ if n is not a leaf of the global tree T, and at least one, but not all, of its children is remote to p. In such an event, the properties of n are *partially* determined by tree data that is local to p. Also, if $n \in S(p)$ then $\wedge n \in S(p')$ for some $p \neq p'$. Nodes 1, 2, 3 and 4 in piece 1, and nodes 1, 3 and 6 in piece 2 are examples of shared nodes.

5.3 An example tree application: the *Barnes-Hut* algorithm

To motivate the main thrust of this chapter, let us consider the structure of the well-known *Barnes-Hut* algorithm [42]. Our objective here is not so much

to provide a precise and comprehensive listing of the code for this particular algorithm, but rather to assess the stylistic and semantic requirements of tree-based algorithms in general. We will use this alogrithm to guide the design of *Distree*, as detailed in § 5.4. Later, we will show that our design generalizes well to other types of tree-based algorithms (*cf.* § 5.8.2).

The *Barnes-Hut* algorithm provides an efficient solution to the N-body algorithm for long-range forces such as gravity. It is considered to typify the behavior of applications with highly irregular, and dynamic patterns of computation and communication. In the field of computational astronomy, variants of the algorithm are used to compute the evolution of large, self-gravitating systems of point particles with mass.

Since gravity is a long-range force, we must compute the gravitational interaction between every pair of particles. Naively, this leads to an $\mathcal{O}(N^2)$ complexity, where N is the number of particles being simulated. Therefore, the all-pairs method is considered infeasible for large ensembles (billions to trillions of particles [105]).

The *Barnes-Hut* scheme uses a tree-based algorithm to compute the *approximate* interactions in such large ensembles. To give a brief description of the algorithm, it considers the cumulative effect of a set of masses, instead of their individual effects, at those points in space where an approximation is valid. The algorithm embodies a tradeoff between accuracy and computational cost: an increase in desired accuracy means that a greater number of pairwise evaluations must be performed. The theoretical complexity of the method is $\mathcal{O}(N \log N)$.

5.3.1 The parallel *Barnes-Hut* algorithm.

Here, we present the iterative structure of the parallel version of the *Barnes-Hut* algorithm. We assume that the particles have already been read to memory from an input file, and each PE holds some arbitrary set of particles initially. Then, the following steps are performed in each iteration.

Discretization of particle positions

Particle positions are discretized onto a fine, three-dimensional grid, yielding the integral *key* of each particle. This process is akin to applying a unique

hashing function to particle positions. The outcome is that each simulated particle (bearing a unique three-dimensional position) is assigned a unique key.

Particle decomposition

Typically, a distributed memory sorting algorithm is used to sort particles by their keys, and create n contiguous partitions of these sorted particles. Each one of these n partitions is used to construct a *piece* of the distributed, *Barnes-Hut* tree (as below). In the terminology of distributed memory sorting, each partition contains a set of particles whose keys are bound by the so-called *splitter keys* for the partition.

The *Barnes-Hut* tree

We construct a tree whose leaves contain the simulated particles. The tree has a uniform branching factor, 2^k ($k = 1$ in this treatment). A key is assigned to each node based on the lexicographic ordering of Warren and Salmon (*cf.* § 5.2). The tree is such that each node either has no children, or 2^k children.

Tree construction

As mentioned above, each piece is assigned a range of particles given by its splitter keys. Conceptually, the tree encapsulated within a piece is constructed by iterative insertion of particles within the range of the piece. Each piece begins with a tree that has only a copy of the shared root node. It then inserts its particles into the tree, beginning at the root. Each particle is propagated down to some leaf of the tree, while maintaining the *prefix property*: For each particle π, the leaf l that it is placed in, is such that $key(l)$ is a prefix of $key(\pi)$. No leaf is allowed to have more than a programmer-determined, threshold number of particles. A leaf that contains too many particles by this measure is split into 2^k children, and is converted into an internal node.

A bottom-up pass is then performed over the tree, so as to compute the multipole moments of each node. The moments of a node give an approximate summary of the distribution of mass enclosed by it, and are valid only

past a certain distance from the node. For a leaf, moments are computed from the particles that it contains. The moments of a non-leaf node are computed from the moments of its children. Finally, nodes are given labels *local*, *remote*, or *shared*, in a manner similar to § 5.2.1.

Remote frontier annotation

A piece has no multipole moment information about its remote nodes. Therefore, it uses splitter key information to determine the list of pieces that share the node, and requests the multipole moment information from one of those pieces. We refer to this process of requesting the payload of remote nodes from the appropriate pieces, as *remote fronter annotation*. Remote frontier annotation is not a trivial operation, and requires hierarchical book-keeping, since a tree piece requesting the moments of a node might itself have received a request for a (different) node that it shares with other tree pieces.

Tree traversal

Each tree piece traverses the global *Barnes-Hut* tree in order to compute forces on the particles encapsulated within its local leaves. One traversal is performed for each leaf, l, in the piece, and each traversal begins at the root (which is *shared* by all tree pieces, and therefore available to each one). For each node, n, encountered during a traversal, we check whether n is distant enough from l that we can reasonably approximate n's mass distribution by its multipole moments at l. If so, the force computation is performed, and we recursively return up the tree. Otherwise, each child of the node is visited recursively. Given the distribution of the global tree over multiple pieces, the computation may attempt to access portions of the tree that are not available locally, thereby generating communication requests and responses.

5.4 Design considerations for a tree code framework

The *Barnes-Hut* algorithm is representative of a large and important class of irregular HPC algorithms. Here, we reflect on the mechanics of the algorithm,

and synthesize a basis for the design of a framework for the expression of tree-based codes.

Appropriate division of responsibilities between programmer and framework

We ask whether it is the responsibility of the programmer, or the system to perform each one of the following tasks: data decomposition, local tree building, distributed tree construction, remote frontier annotation, and tree traversal. Although programmer productivity increases as the system subsumes a greater number of these phases, in general, flexibility and generality suffer. A good compromise is to provide a toolkit of common methods for each one of the above steps, but to allow the programmer to plug-in custom strategies.

Support for fine-grained parallelism

The framework must provide an opportunity for the expression of computations in a fine-grained manner. We believe that this affords the ability to express tree-based algorithms in their natural form. For instance, in the *Barnes-Hut* algorithm, the traversal of the tree (which dominates execution time) for each piece is best expressed as a series of independent traversals over the global tree, one for each leaf local to the piece. It should be the responsibility of the framework to transparently perform aggregation of computation and communication to increase efficiency in the face of such fine-grained pieces of work.

Seamless, asynchronous access to remote data

For programming productivity, the programmer's expression of a tree-based algorithm should be agnostic to details such as the decomposition of tree data over PEs, or whether a given subtree is local to the current PE. Moreover, the interface between programmer code and framework must be such that continuations can be spawned transparently for traversals that require remote data. Such traversals would result in the generation of asynchronous communication events, whose latency can be overlapped with traversals that

are currently performing computations on locally available data. A similar method has been adopted, though in the narrower context of a single application variant, by Zhang *et al.* [106].

Explicit differentiation of local and remote data

The performance-conscious HPC programmer must be given access to information about data locality. This allows the programmer to exploit algorithm-specific knowledge to increase performance in a manner that cannot be done by the framework alone. For instance, in the *Barnes-Hut* algorithm above, the programmer could initiate two traversals (instead of a single one) for each leaf local to a piece. One traversal would be given priority over the other, and would operate only on remote data; the other traversal would operate with lower priority, and operate only on local data. This allows a greater proportion of the communication latency (suffered by the first traversal) to be overlapped with local computation (performed by the second traversal).

Intuitive and relaxed semantics for update of remote data

Algorithms that require *write-through* or *accumulate* access to remote data should be able to use these modes efficiently, without much effort to understand update semantics on the part of the programmer.

5.5 The *Distree* framework

Motivated by our observations in the previous section, we describe the design of the *Distree* framework.

5.5.1 Programming model

We first give the reader a feel for *Distree*'s programming model.

1. There is a set of spatially organized *data elements* on which the tree algorithm operates. A *distributed tree* is constructed over these data elements.

2. The tree is distributed coarsely over so-called *tree pieces*. Each tree piece is a container for a partition of the underlying set of data elements, as well as the minimal subtree that exists over this set. The decomposition is such that the shallower levels of the tree are shared across several tree pieces (*cf.* § 5.2.1). Typically, there are several (tens of) tree pieces on every PE. This *overdecomposition* of the tree into more pieces than there are PEs, allows for flexibility in mapping the work and data to PEs.

3. The algorithm begins with the user instantiating a *collection* of tree pieces. These tree pieces are typically loaded with the underlying data elements, e.g. by reading input files.

4. Tree pieces inherit functionality from the *Distree* system. The programmer leverages this functionality by invoking methods on the pieces, thereby performing tasks such as key-based decomposition of data elements over pieces, and distributed tree construction.

5. A *Distree* program includes programmer-defined collections of coarse-grained, message-driven *work units* in the form of *chares* (*cf.* § 2.2). The ARTS assigns to each work unit a PE on which it executes. However, work units are not bound to PEs: they may move due to dynamic, migration-based load balancing. The collection of work units may be the same as the collection of tree pieces.

6. Regardless of the PE it is on, a work unit has access to (at least) the root of the global tree. The programmer directs work units to initiate concurrent, fine-grained *traversals* over the global tree.

7. The crux of a *Distree* algorithm is embodied by the *traversal*. A traversal is a combination of (i) a *walk*, which provides a loose, partial ordering over nodes in the tree, and (ii) a *visitor*, which performs some programmer-defined computational actions each time a node is visited by the walk. The visitor *directs* the traversal, by signaling to the walk whether the children of a visited node should in turn be visited. Therefore, the structural recursion inherent in tree-based algorithms is driven by the actions of the visitor. Different traversal orderings over

tree data may be suitable for different algorithms, and each algorithm may require a unique visitor behavior.

8. During tree traversal, the visitor may implicitly request the *Distree* framework to fetch a remote subtree to the PE on which it is executing. The framework uses the mapping of nodes/data elements to tree pieces in order to fetch the required subtree. Meanwhile, the traversal continues to operate on other, available nodes of the tree; a continuation is created and stored, so as to resume the traversal at the missed node when it is finally fetched. Therefore, the execution model is inherently and implicitly (i.e. without the programmer's knowledge) asynchronous.

We can now discuss the salient features of the *Distree* framework in greater detail.

5.5.2 Explicit decomposition of work and data

The *Distree* programmer explicitly partitions the global tree into a number of distributed units, called *pieces*. The principle purpose of these pieces is to serve as containers for distributed tree data. The mapping of tree nodes to pieces can be obtained (incrementally) on any PE, so that any computation may discover the tree piece that contains a given node/data element, and hence request that data from the container. Each piece is *coarse* in the amount of tree data it contains. Tree pieces are user-defined entities, but every tree piece class must inherit from the system class template `distree::Piece`. This class encapsulates several important functions related to decomposition, tree building, serialization/deserialization and communication of subtrees across processors, saving the programmer the trouble of having to implement them.

As with other chares, the programmer may construct an indexed collection of pieces, using Charm++-generated factory classes. The size of the collection is generally independent of the number of processors on which the program is to run, though typically tens of pieces are present on each PE. The mapping of pieces to processors is handled by Charm++.

Although it is not necessary to do so, the chare-based decomposition of *Distree* encourages the programmer to decompose parallel *work* over a dif-

ferent collection of chare from the tree pieces (which hold *data*). This ability is useful in algorithms such as Dehnen's momentum-conserving scheme for gravity calculation [107], which is most naturally expressed in terms of interactions between *pairs* of spatial extents (i.e. tree pieces). Both work and data units (tree pieces) are amenable to migration by the Charm++ ARTS so as to effect dynamic load balancing.

5.5.3 Division of tasks between the programmer and *Distree*

Data decomposition

Distree provides three levels of support for data decomposition onto pieces, and the programmer may use any one of these, depending on the requirements of the algorithm.

Built-in strategies. A suite of commonly used decomposition strategies is provided by the *Distree* framework. Currently, these include Oct- and Space-filling curve (SFC) based strategies. The programmer specifies an identifying key for each data element. Conceptually, the key of a data element determines its position along a linearization of the data elements. Based on this assumption, the framework splits the linearized data elements among pieces. The particular partitiong that results depends on the strategy used by the programmer. the SFC strategy orders data elements using the Peano curve, and evenly partitions elements among pieces (up to a programmer-specified tolerance). The Oct strategy performs a similar ordering of elements, but ensures that the deepest common ancestor node of all the elements assigned to a piece p is exclusively owned by p. This constraint is helpful in tree codes that require that each piece have a *compact* extent: it ensures that if the data elements are particles with positions, then the bounding box of each tree piece is non-intersecting with that of any other tree piece.

Pluggable comparison-based strategies. The Oct and SFC decomposition strategies above are based on a more general-purpose, parallel histogramming strategy, into which may be plugged programmer-provided serial comparator functions. The algorithm proceeds in a manner similar to the comparison-based parallel sorting algorithms of Kale and Krishnan [108] and Solomonik and Kale [100].

The histogramming occurs as a series of coordinated histogramming iterations involving a master and several workers; there are as many workers as there are PEs, and each worker holds a partition of the set of data elements. In each iteration, the master broadcasts a list of sorted keys to the workers. In the first iteration, the key of the root node (1) is broadcast. For each key k in the broadcast list, a worker counts the number of data elements for which k is a prefix of the data element's key. One may think of two consecutive keys k_i and k_{i+1} as forming a *bin*. Then, a programmer-provided comparator may be used to efficiently determine the bin into which each data element falls. These counts are reduced to the master, so as to construct a histogram of the number of data elements in each bin. The master examines this histogram, and attempts its equalization by splitting some bins.

The decision of which bins to split is made based on a programmer-provided, binary *static load balancing* parameter. If the parameter is switched on, the master obtains and broadcasts the sorted list of keys of the children of every previous key. This continues until the list of bins can be halved evenly (up to a programmer-provided precision) among the input set of pieces. The master then performs the histogramming recursively for each half of the set of pieces, and the corresponding set of keys. The recursive procedure terminates when a list of keys is obtained for each individual piece. This strategy does not provide optimal partitioning of data elements over pieces, but results in a reasonably small load variance for large numbers of input data elements and pieces.

On the other hand, if the static load balancing parameter is switched off, then the histogramming proceeds on a per-bin basis, each bin being split until it has fewer than a (programmer-provided) threshold number of data elements. This mode provides poor load balance between pieces, but guarantees that each piece can have no more than a certain number of data elements. In this case, the Charm++-prescribed overdecomposition of the problem into more pieces than PEs can be leveraged to perform measurement-based, dynamic load balancing.

Programmer-defined strategies. If the comparison-based decomposition of *Distree* does not suffice for the programmer's algorithm, she may write her own Charm++ code to determine the partitioning of data elements over pieces. In this case, *Distree* allows individual pieces to submit their data

elements to the framework post decomposition for tree building (described next).

5.5.4 Building the local tree

Distree supports the building of the local tree for each piece, from the data elements assigned to it. This can be done at one of two levels of abstraction.

Comparison-based tree construction

Once a piece p has received the data elements assigned to it by the decomposition procedure, it uses the key of each data element to insert it into a local tree structure. A procedure similar to the one described in § 5.3.1 is used, thereby ensuring the prefix property of keys, and preventing leaves from enclosing too many particles. A programmer-provided comparator may be plugged into the insertion procedure, so as to customize its behavior.

The tree construction procedure also assigns a type to each node. *Remote* nodes, as well as the list of pieces that share each such node, are identified. The identification of remote nodes is depicted in Figure 5.3. Let $[f_i, l_i)$ be the range of data element keys assigned to piece i (i.e. its splitter keys). Therefore, we can use the key prefix property to identify each node ν in the local tree of i whose key *cannot* be a prefix of *any* particle assigned to piece i, i.e. $Key(\nu)$ is not a prefix of *any* key in the range $[f_i, l_i)$. Therefore, $Key(\nu)$ can only be the prefix for a paricle assigned to some piece *other than* i. Therefore, ν is remote to tree piece i.

Each node n is shown to cast a *shadow* representing the range of keys such that $Key(n) = k$ is a prefix for all keys in that range. We call this the *prefix shadow* of n, Π_k. Therefore, if Π_k intersects with the range of keys represented by the j-th green box, i.e., $\Pi_k \cap [f_j, l_j) \neq \emptyset$, then the node is *at least* shared by piece j. This is because the intersection implies that at least some keys with prefix k are found in $[f_j, l_j)$.

On the other hand, if Π_k is completely contained within box j, i.e. $\Pi_k \subseteq [f_j, l_j)$, then piece j is the *exclusive* owner of the node with key k. Intuitively, this is because *all* keys with prefix k are contained within $[f_j, l_j)$. If n is owned exclusively by j, then all its descendants will be owned by j, so that we will never find a remote node beneath n. Therefore, when computing the

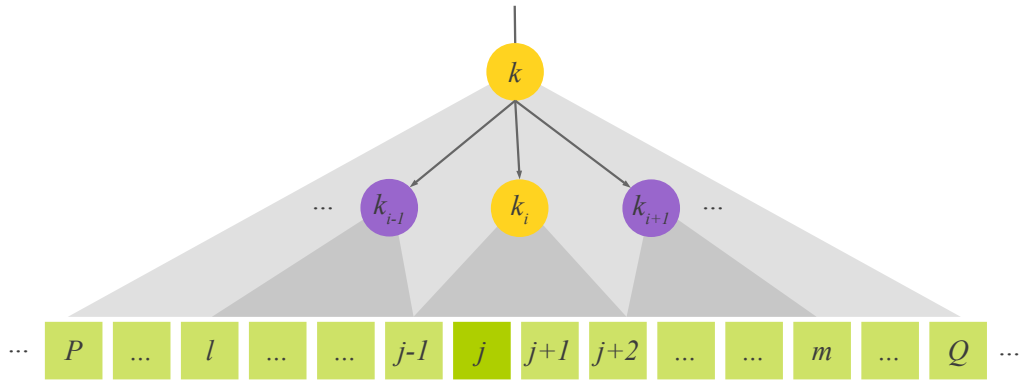


Figure 5.3: The inductive procedure for determining the owner(s) of the i-th child of tree node k. Boxes represent the sorted list of splitter keys assigned to all tree pieces. Array element j represents the range $[f_j, l_j)$ of keys assigned to the j-th piece. The grey shadow cast by a node n represents the range of data element keys for which $Key(n)$ is a prefix.

remote frontier of j (i.e. the set of nodes in $R(j)$), we need only consider nodes n such that n is shared by a set of pieces that includes j. The figure shows k to be such a node: $\Pi_k \cap [f_j, l_j) \neq \emptyset$, so that k must be shared by j, among other pieces. However, j is not the exclusive owner of k, since $\Pi_k \not\subseteq [f_j, l_j)$. Considering the children of k, we find that child k_i is the only one whose prefix shadow intersects $[f_j, l_j)$. For other children of k, e.g. k_{i-1}, we have $\Pi_{k_{i-1}} \cap [f_j, l_j) = \emptyset$, so that no key with prefix k_{i-1} can be in $[f_j, l_j)$. Therefore, node k_{i-1} is remote for tree piece j. Applying this logic recursively, we identify all the remote descendants of k_i.

We can augment this procedure to discover the owners of nodes in the remote frontier. Notice that $\bigwedge_{r=P}^{Q}(\Pi_k \cap [f_r, l_r) \neq \emptyset)$. That is, node k is shared by pieces $P, P+1, \ldots, j-1, j, j+1, \ldots, Q$. We write this is as $\Pi_k \cap [F_P, L_Q) \neq \emptyset$. Now, consider the children of k. In particular, $\Pi_{k_{i-1}} \cap [F_l, L_{j-1}) \neq \emptyset$, so that node k_{i-1} is shared by pieces $l, \ldots, j-1$. Similarly, $\Pi_{k_i} \cap [F_{j-1}, L_{j+2}) \neq \emptyset$, etc. Note that j is one of the owners of k_i, as we had identified earlier. Again, if this procedure is applied in a recursive fashion, one can find the owners of all nodes in the remote frontier of piece j. In this way, the tree held by piece j can be augmented so that for every node n, either n is exclusively owned by j, so that j has access to the entire subtree underneath it, or n is remote and j knows the piece from which to request portions of the subtree underneath n.

Arbitrary, programmer-defined tree structures

On the other hand, the programmer may choose to construct a tree customized tree, based on a different strategy. The only requirement is that each node in the tree be manually augmented with ownership information (which is automatically inferred when the previous, built-in strategy is used for tree construction).

5.5.5 Annotating the local tree

A remote node's ownership information is used by the *Distree* framework to obtain information about the structure of the subtree rooted at that node. Algorithm 4 presents a distributed algorithm that automates this task of *remote frontier annotation*. It performs the following basic functions: (i) requesting the payload of each remote node from some tree piece that exclusively owns it, or shares it with other pieces; (ii) tracking requests from other pieces for nodes either exclusively owned by the current piece, or shared with others; and (iii) upon receiving the payload of some remotely requested node, recursively moving up the tree and fulfilling pending requests as possible.

The algorithm is structured as a series of interdependent "when" clauses to reflect its message-driven nature. Here, the expression "when m S" denotes that when a message m is received, statements S are performed. Algorithm 4 is executed by every tree piece.

Let us first consider the initialization of data structures. The *requests* table of piece p stores requests received from other pieces for nodes owned (either exclusively or in a shared manner) by p. The *missing* table stores the pointers to those nodes of the piece p that have been identified as *remote*. Next, there is a key-indexed *nodes* table that maps the key of each node to a pointer to that node. (It could be a local node, or a remote node that has already been been fetched during the course of the algorithm.) This table is used to respond to requests from other pieces for nodes local to this tree piece. Finally, we maintain a *Ready* list that stores the keys of all nodes for which we have payload information. This list is initialized with the keys of all local nodes, since their payload information is known once the local trees have been built.

When a piece p receives a REQUEST message for the payload of a node

Algorithm 4: Annotating the remote frontier of a consolidated local tree.

```
Initially do
    requests ← ∅;
    Ready ← ∅;
    for n ∈ P do
        k ← Key(n);
        if Type(n) = Local then
            Ready ← Ready ∪ {k};
            nodes[k] ← n;
        else missing[k] ← n ;
    end
end

when REQUEST(k, i) do
    if k ∈ Ready then
        n ← nodes[k];
        Assert(Type(n) = Local);
        SEND to i, REPLY(k, Payload(n));
    else requests[k] ← requests[k] ∪ {i} ;
end

when REPLY(k, p) do
    n ← missing[k];
    Payload(n) ← p;
    Up(n);
end

when Up(n) do
    k ← Key(n);
    nodes[k] ← n;
    for i ∈ requests[k] do SEND to i, REPLY(k, Payload(n)) ;
    requests[k] ← ∅;
    Ready ← Ready ∪ {k};
    p ← Parent(n);
    if p ≠ nil then
        if ⋀_{c∈Children(n)} c ∈ Ready then Up(p) ;
    else DoneFrontierAnnotation() ;
end
```

with key k, it checks whether k is in its *Ready* list. If so, the payload of that node is available to p, and it responds to the requestor with that payload. Otherwise, p adds the request to $requests[k]$. The fact that the node with key k is currently unavailable indicates that at least one successor of k is remote to p, and that its payload has not yet been fetched. However, k *cannot* itself be remote to p, since the requestor must have determined p to be an owner of k before sending it the REQUEST message.

Recall that for each node that was initially identified as remote, the pointer

to its node data structure was placed in the *missing* table, in the entry corresponding to its key. Upon receiving a REPLY message, this table is used to obtain the pointer to the node corresponding to the received payload. Its payload is then set to the received value, and the node, n, is passed into a recursive procedure, *Up*.

The *Up* procedure does four things: (i) It associates the key of n with a pointer to it in the *nodes* table; (ii) It checks whether there are any outstanding requests for n; if so, it sends out the payload of n to each one of its requestors; (iii) It places n in the *Ready* list; and (iv) It then checks whether the newly set readiness of n has resulted in its parent, p, becoming ready as well. That is, we check whether every child of p is in the *Ready* list. If so, it is safe to update the payload of p using its children. This is done through a function provided by the programmer, and is not shown here for conciseness. Once the payload of p has been set, it is passed into a recursive invocation to *Up*. This results in an upward traversal of the consolidated tree, where at each node outstanding requests for it are sent to the requestors of that node. When we reach the root of the tree in this manner, the algorithm terminates.

5.6 Traversing the tree

We consider a *computation* over a distributed tree to be equivalent to a structured iteration over some subset of the nodes and data elements in the global tree. Such a computation is said to *traverse* the tree, and in what follows we do not distinguish between the computation and the traversal of the tree that it entails.

In explicitly parallel distributed-memory tree codes, concurrency is generally expressed in the form of concurrent traversals over the tree. The *Distree* framework employs a *visitor pattern* over the tree, The grain size of computation encapsulated by each visitor is defined by the programmer, and is typically very fine. For instance, in the *Barnes-Hut* algorithm, a single traversal might consist of a top-down traversal for a handful of particles.

Visitors are not aware of the structure of the tree, and do not rely on information about whether tree nodes and data elements have been fetched from a local or remote source although they may use this information to optimize performance.

5.6.1 Tree traversal equals a *walk* with a *visitor*

Distree enforces a separation of the tree data structure from the traversal algorithms that are employed over it. As a result, the programmer phrases each traversal as the composition of (i) a *tree walk*, or simply, a *walk*, and (ii) a *visitor*.

The *walk visits* nodes of the distributed tree, and invokes particular methods on the *visitor* (*cf.* § 5.6.3) as it does so. The programmer may employ different types of walks, so as to change the *order* in which the visitor receives tree nodes to process. If the framework determines that a node required for a traversal is not present locally, it is requested from its remote source. A corresponding continuation is created and saved by the framework, for invocation upon receipt of the remote data. Meanwhile, other traversals may be resumed or initiated.

The *visitor* encapsulates the programmer-provided logic to process nodes of the tree as they are encountered. Moreover, it *guides* the walk, by indicating to it whether or not a visited node should be expanded into its children. This separation makes it easy for the programmer to write code for tree traversals. It also simplifies the process of adding new, independent traversals to the tree-based algorithm.

5.6.2 Types of tree walks

A *walk* over a tree is a loosely ordered iteration over its nodes and data elements. The *Distree* framework provides *walk objects* that encapsulate the details of traversing the distributed tree. The walk object insulates the user from details of traversal, e.g. the *chunking* scheme used to amortize the cost of fetching required remote nodes. The *Distree* framework provides two types of walk, each of which visits nodes in a different order.

The *top-down* walk

The top-down walk is a *best-effort*, recursive, depth-first descent into a tree. The ordering has been relaxed from a strict depth-first in order to exploit *locality* of access and to leverage *asynchrony* in fetching remote subtrees. A top-down walk guarantees only that the children of a node are visited after

the node itself.

The traversal adheres to depth-first ordering when possible. In particular, local nodes are visited in depth-first order. However, if a remote node, r, is encountered, the traversal continues to the next available (local) node in depth-first order, skipping the nodes underneath r temporarily. A message requesting a subtree beneath r is sent, and a continuation of the traversal at r is created. When r becomes available locally, all pending traversals for it are invoked, thereby resuming depth-first visits over the fetched subtree underneath r. Therefore, traversals are allowed to process remote data opportunistically, i.e. as soon they become available locally. Of course, this means that the precise ordering in which nodes are visited, is dependent on the order in which the traversal receives the messages containing them. As such, the ordering of visited nodes is inherently non-deterministic.

The top-down walk is useful in such applications where the regions of the distributed tree that are of interest to a given traversal, are known before the start of the traversal. An example of such a scenario is the calculation of gravitational forces in tree-based cosmology codes. Recall from § 5.3 that we traverse the tree once for each leaf, and that each traversal descends deeply into only those subtrees that are proximal to the leaf.

Another example is the calculation of the radial two-point autocorrelation function of a point distribution. This technique offers a quantitative measure of the *lumpiness* of the distribution, and can be approximated based on the following principle. Given two sufficiently distant and geometrically compact sets S_1 and S_2 of spatially distibuted points, the distance between each pair of points (p_1, p_2) such that $p_1 \in S_1$ and $p_2 \in S_2$, will be similar in magnitude.

Therefore, if we consider the traversal as searching the tree, the top-down walk is useful in situations where we know the *tree pruning* condition prior to beginning the traversal. Such algorithms have the interesting property that the number of nodes processed per traversal is independent of the order in which the walk visits those nodes. Instead, the number of nodes processed is determined completely by the specifics of the visitor algorithm, and the set of data elements underlying the tree structure.

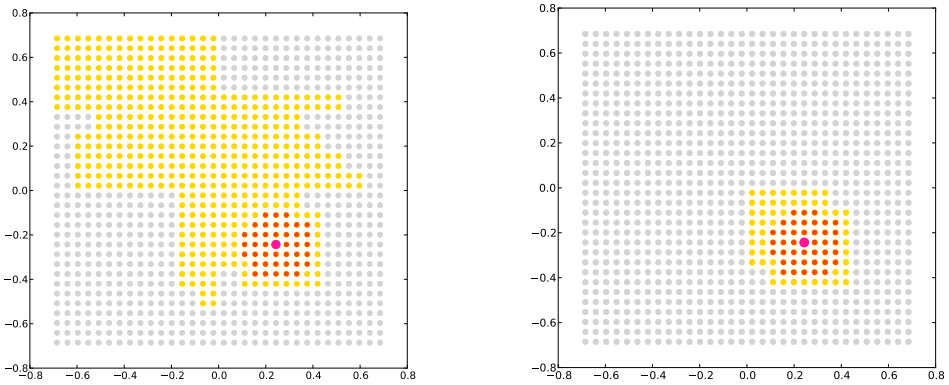
The *up-and-down* walk

There are certain tree-based algorithms in which the pruning condition for a traversal cannot be determined prior to the start of the traversal. In fact, the pruning condition is gradually refined as the traversal progresses. This characteristic of a *dynamic* pruning condition can cause the number of nodes processed by a traversal to depend very strongly on the *order* of visited nodes. Therefore, a bad ordering of visited nodes can significantly increase the amount of computation performed in achieving a given result through the traversal.

Let us consider a concrete example of this situation. Suppose that we are given a spatial distribution of points, and would like to compute the k points nearest (by some user-defined metric) to each point, where k is a fixed positive integer. This is the *k-nearest neighbors* problem, and given the spatial layout of points, is typically solved by arranging the points in a tree. Now, let us consider the application of a top-down traversal to find the k closest points to a target point. For the purpose of explication, assume that we already have a list of k candidate nearest neighbors for the target. We will return shortly to the issue of initiating the traversal without this list of k candidates. So, we have a list of k points, arranged in descending order of distance from t. However, we have not finished processing all points in the distribution yet, so we are not sure that these k are the *closest* neighbors of t.

What should be our pruning condition given these k points? By imposing a tree structure on the points, we have ensured that each node of the tree represents a spatial extent within which are bounded some potential neighbors for the target. Now, let d_{max} be the distance from t to the farthest of its k neighbors. Therefore, whenever we encounter a node so far from t that it does not intersect with a sphere of radius d_{max} centered at t, we can safely prune our search for close neighbors at that point in the tree. However, if the node is close enough to intersect the aforementioned sphere, then we *could* find points closer to t than its current list of neighbors. We must recursively descend into such a node, and if we do find a point p closer to t than its current farthest neighbor f, we must update our neighbor list to include p and exclude f. We must also shorten the pruning radius for the traversal: That is, we set d_{max} to the distance from t to its *new* farthest neighbor.

This sounds reasonable, but how do we initiate the traversal for target t? That is, what should our pruning condition be when we have fewer than k candidate closest neighbors? Obviously, if it doesn't have k candidates already, the traversal must accept *every* point offered to it, as a candidate for one of the k closest to t. Correspondingly, the pruning condition is very conservative at the start of the algorithm, and will attempt to recursively descend into nodes that might (in simulated space) be very distant from the target t. Therefore, we require a walking strategy that first visits those parts of the global tree that are *likely* to be closer to t before searching more distant parts of it.



(a) top-down ordering

(b) up-and-down ordering

Figure 5.4: For certain traversals, node ordering can significantly impact the amount of computational work done. This is shown for a k-nearest neighbor search. The target point is shown as the large, off-center pink dot. Its 40 nearest neighbors for a simple, uniform distribution of particles, are shown in red. Those nodes that are processed by the walk are shown in yellow (and red), and the ones that remained untouched by the walk are shown in light gray. Panel (a) shows the result for the top-down (depth-first) traversal, whereas panel (b) shows the same computation, peformed using far fewer point accesses with the up-and-down walk.

The *Distree* framework provides the *up-and-down* walk for such scenarios. The walk begins at a node of the user's choosing, n, which is referred to as the current node. The walk then travels up to the parent of n, p, and performs a top-down walk on each child of p except for n. Thus, the siblings of n are visited in top-down order. Thereafter, the parent of n becomes the current node, and the siblings of the new current node are visited in top-down order,

etc. This process continues all the way up to the root of the tree. Therefore, if the starting node for a walk is a leaf, the up-and-down walk first visits the siblings of the leaf, then visits, in a top-down fashion, the subtrees of its uncles, its grand-uncles, and so on. This ordering uses the heuristic that *topologically* proximal nodes represent regions of the simulation domain that are physically proximal, and therefore interact strongly.

This ordering heuristic helps to reduce the total number of nodes processed in a traversal algorithm that progressively refines its pruning condition. Figure 5.4 demonstrates this effect for the k-nearest neighbor search algorithm. As can be seen, the number of points accessed (yellow circles) by the top-down walk (Figure 5.4(a)) is much greater than that accessed by the up-and-down walk (Figure 5.4(b)). Of course, the above scheme is not the only way to calculate the k nearest neighbors of a point – Fukunaga and Patrenahalli [109] have developed a so-called *ball*-tree to efficiently compute the same relation.

5.6.3 A visitor pattern fosters separation of concerns

In *Distree* the actions that are taken whenever a node is visited by a walk, are dictated by user code, and encapsulated by a *visitor* object. Visitors may be stateful, and the actions they perform are expected to be reentrant. The walk iterates over nodes in the tree, adhering to some loose ordering constraints. For each node visited, the walk invokes a suitable visitor method, together with appropriate data arguments as context for the visitor, to drive the computation forward. For instance, whenever the walk visits a node of the global tree, it forwards the node to the visitor by calling the its `node()` method. This method takes as argument the visited node, and returns a Boolean value, which indicates to the *Distree* framework whether the visitor requires that the visited node be *opened*, i.e. its children be visited in turn. If a leaf is opened by a visitor, the walk fetches the *contents* (data elements) of the leaf. When the walk has obtained the contents of the leaf, it invokes the visitor's `leaf` method. In this way, the visitor can *steer* the traversal according to its program logic.

This scheme allows visitor code to access the global tree in a *seamless* manner. Consider what happens when the visitor wishes to open a node whose children are present on a remote processor. In this case, the walk generates

a request message for the remote data and saves a pointer to the visitor along with other context about the traversal. This saved context enables the walk to resume the traversal when the remote data become available. If, on the other hand, the data have already been fetched from a remote processor when the visitor asks to descend into the associated remote subtree, the data is simply fed to the visitor via its appropriate `node`/`leaf` method. In either case, the only way that the visitor can determine the origin of a visited node or leaf, is by examining the attributes of the node itself. Therefore, by adopting the visitor pattern, *Distree* allows the programmer to leverage message-driven execution without having to write fragmented code that is phrased as a collection of asynchronous message sends and associated completion callbacks.

A visitor written by the programmer must implement the following interface of functions.

Computational actions

Each visitor must define a `node()` and a `leaf()` function, which are invoked by the walk on the visitor when it visits, respectively, a non-leaf node, or a leaf of the distributed tree.

State update actions

Distree provides a declarative and asynchronous model of computation. Thus, calls to fetch remote data do not block. Therefore, *Distree* provides certain hooks that allow the visitor to do book-keeping when specific events occur (e.g., a node is missed, a node is received, etc.) A visitor may define the `miss()` and `hit()` functions for this purpose. Using these methods, the programmer's code can determine when all computation associated with a certain traversal has finished.

The orthogonalization of the walk and visitor is a realization of the visitor design pattern [110]. The ensuing separation of concerns [111] allows us to achieve reuse of each walk class across multiple, suitable traversals, and insulates the user from the mechanics of tree traversal.

5.7 Run-time optimizations

Let us now discuss some performance optimizations incorporated into the *Distree* framework.

5.7.1 Local tree pieces are consolidated automatically

In order to increase the amount of local tree data available to traversals, *Distree consolidates* tree pieces present on a PE. The programmer can even instruct *Distree* to consolidate all pieces across an entire SMP domain. As detailed by Gioachin *et al.* [8], such consolidation can lead to a considerable reduction of communication volume. Our work differs from their algorithm in that *Distree* couples consolidation and remote frontier annotation, thereby leading to further reductions in communication volume, both during tree construction and tree traversal.

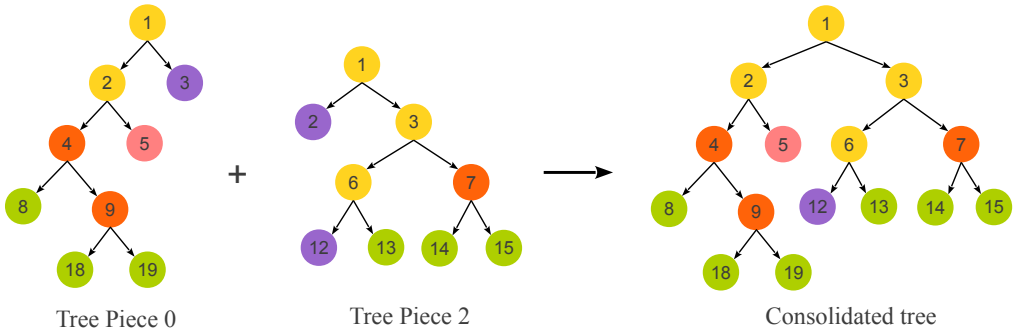


Figure 5.5: Schematic of the tree piece consolidation algorithm. The algorithm receives as input a list containing the roots of two co-resident tree pieces. The procedure recursively considers the corresponding children of these listed nodes, until it encounters the base case: (i) either only a single node is listed, or (ii) none of the listed nodes has any children. Note that a listed node without a child is not necessarily a leaf – it could just be a remote node for all of the tree pieces on the PE/SMP.

In a nutshell, Algorithm 5 descends into the pieces present on a PE (or SMP) recursively, and for every remote node n of a tree piece t_1 encountered, it checks whether there exists another tree piece t_2 on the PE, for which n is *not* remote. If this is the case, then the subtree underneath t_2's copy of n can be used for both t_1 and t_2.

In slightly greater detail, *Consolidate* algorithm operates as follows. It

Algorithm 5: Consolidation of tree pieces.

```
Consolidate(L)
Input: list L of tree pieces rooted at node with key k
Output: root with key k of consolidated piece
begin
    if |L| = 1 then
        return L[0];
    end
    pick ← BestNode(L);
    for i ∈ {0 ... |Children(pick)| − 1} do
        L' ← ∅;
        for n ∈ L do
            if n is not a leaf then L' ← L' ∪ {Child(i, n)};
        end
        if L' ≠ ∅ then Child(i, pick) ← Consolidate(L');
    end
    if IsRemote(pick) then
        REQUEST Payload(pick) from piece p ∈ Owners(pick);
    end
    return pick;
end

BestNode(L)
Input: list of nodes, L
Output: best candidate from L for root of merged tree
begin
    best ← nil;
    for n ∈ L do
        if EltsBeneath(n) > EltsBeneath(best) or IsRemote(best) and IsLocal(n)
        then
            best ← n;
        end
    end
    return best;
end
```

receives as input a list of tree nodes, each one from a different tree piece on the PE (or SMP processor). Each one of the nodes in the list has the same key (call it k), a property that is maintained in the recursive call. The output of the algorithm is a single node, which represents the *merged* tree. The merged tree is obtained by consolidating the subtrees rooted at nodes from the input list. The algorithm picks the *best* suited node as the root of the consolidated tree beneath k. In this context, the best node is the one with the most data elements underneath it. If all nodes have an equal number of data elements underneath, we pick the one that is exclusively owned by a tree piece on the current PE. For each i from zero to one short of the number of children of the best node, the procedure recurses on the list containing the

i-th child of each one of the listed nodes. An example of the result of this operation is shown in Figure 5.5.

The process of consolidation leads to significant savings in execution time for both the tree construction, and tree traversal phases. It is especially relevant in the strong scaling regime, where the amount of computation performed on the tree *per core* decreases with the addition of more cores, so that the time taken for distributed tree construction can become comparable to the time taken to complete the computation on the tree thus constructed.

5.7.2 A software cache promotes remote data reuse

Data reuse can be critical in determining the performance of tree-based algorithms [8]. Furthermore, modern SMP-based supercomputers offer several levels at which data sharing can be effective. Requests for the same remote elements from two visitors on a PE can be merged. When the requested data are received at the PE, they can be shared among those objects. Similarly, PEs in the same SMP domain can share remotely fetched data. In the following we describe a two-level caching scheme that enables the data reuse across computational units on a PE, as well as across PEs on an SMP processor. This caching mechanism is transparent to the programmer's visitor code.

Algorithm 6 gives the outline of our two-level caching scheme. Each PE on the SMP has a *private* cache, which stores pointers to the remotely fetched data that has been requested by traversals on that PE. There also exists one cache at the level of the SMP processor that is *shared* by all the PEs in the SMP. The shared cache contains the union of all the entries in the private caches of these PEs.

Briefly, the algorithm funnels all requests for remote data through the cache. If the data are found in the private cache, then they are immediately passed into the requesting traversal's visitor code. If the data are not found on the PE, we check whether some other piece on the PE has requested them previously. If so, a lightweight continuation is created to resume the traversal at the requested node upon its receipt. Otherwise, the more expensive, SMP-wide table lookup is performed.

Two schemes have been devised to manage concurrent accesses of the shared, SMP-wide cache table. The first version funnels all requests for

Algorithm 6: Searching for remote data with a two-level cache.

```
FetchData(k, owner, traversal)
Input: Key k of remote data; remote data owner; and requesting traversal.
Output: pointer to the requested data if available on SMP processor of request;
        or nil otherwise.
begin
    e ← privateCache[k];
    if e = nil then
        e ← new PrivateCacheEntry(k, owner, traversal);
        Requestors(e) ← {traversal};
    end
    if Data(e) ≠ nil then return Data(e) ;
    else
        if RequestSent(e) then
            Requestors(e) ← Requestors(e) ∪ {traversal};
            return nil;
        else
            data ← LookupSharedCache(k, owner);
            Data(e) ← data;
            if data = nil then
                Requestors(e) ← {traversal};
                RequestSent(e) ← true;
            end
            return data;
        end
    end
end
```

remote data generated by traversals on the SMP processor through a single core, which is termed the *fetcher* for that SMP processor. Cheap, intra-node messaging between PEs is used for efficiency. We call this the *single-fetcher* version. The second implementation follows a *first-touch* policy for fetching remote data. That is, the first core to request the data corresponding to a particular key, is responsible for sending the message to its owner to initiate its fetching. This scheme uses fine-grained locking to prevent read-write conflicts on cached entries. Experiments (not shown here) suggest that the second approach is better suited for SMP processors with 4-8 cores/SMP, whereas the first one is better for wider SMPs.

The caching mechanism allows *delayed* write-throughs

As we shall see in § 5.8.2, certain traversals require *write-through* access to remote data. That is, when a remote subtree is fetched from its source, the

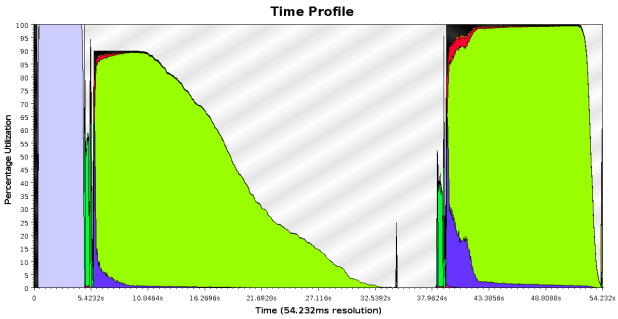


Figure 5.6: Load imbalance in a 512-core run of the *Barnes-Hut* algorithm with a highly non-uniform spatial distribution of about five million particles. Time increases along the horizontal axis, whereas colored regions indicate cumulative work performed across all 512 cores at each instant in time. The left panel profiles the execution of an iteration without load balancing, and the right panel, with load balancing switched on.

local traversal might update the subtree through its computations. For such traversals, the programmer may configure walk objects to be *write-through*. When a write-through walk finishes, the software cache module flushes out all changes accrued in the cached data to the owner of that data. The programmer will find this feature useful in implementing algorithms such as Dehnen's $\mathcal{O}(n)$ momentum conserving tree scheme [107]. Currently, we only support the application of a commutative-associative operation on the writethrough data. This restriction allows us to provide a simple consistency semantics, and can be implemented efficiently.

Remotely fetched data is organized into subtrees, so that the cost of sending node and data element updates to their owners can be amortized over several nodes. However, the write-through cache flush operation is still very communication intensive. As such, it can benefit from a message aggregation optimization provided by the TRAM module of Charm++. This module enables the dynamic aggregation of short messages in order to optimize network bandwidth usage [26, 112]. A completion detection algorithm [29] is used to detect when all updates have been applied to data sources.

5.7.3 A heuristic for dynamic load balancing

Tree-based methods are well-suited to situations in which data elements are inhomogeneously distributed over the simulation domain. As such, these al-

gorithms are subject to severe load imbalance, as demonstrated in Figure 5.6.

Distree provides a communication-heuristic based load balancing strategy that attempts to balance both computational load and communication volume. Previous work [102] has shown that strategies that map pieces onto PEs based only on the computational load of each piece, can achieve good load balance, but simultaneously incur significant communication overhead. Our algorithm, called *Orb3dLB*, allows the programmer to provide hints about the location of each tree piece in simulation space. The *Distree* framework treats proximity in simulation space as a strong determinant of the volume of communication between tree pieces. Together with this heuristic, an orthogonal recursive bisection strategy is used to map tree pieces onto PEs. Figure 5.7 shows the improvement in performance with Orb3dLB in the context of a *Barnes-Hut* simulation of a realistic, non-uniform distribution of 5 million particles.

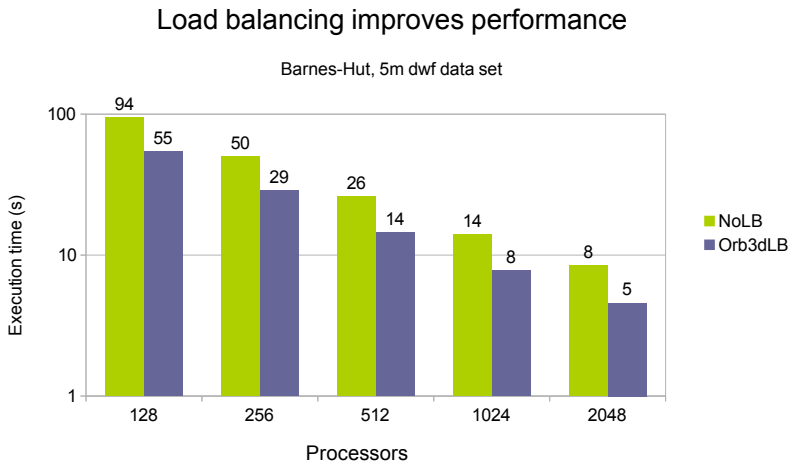


Figure 5.7: Performance improvement due to the communication heuristic load balancing strategy of *Distree*. For each core count, the left column shows performance without load balancing, and the right column, performance with load balancing switched on. Performance numbers are given for a *Barnes-Hut* simulation of a non-uniform, five million particle input distribution.

5.8 Composing the elements of the *Distree* framework

Figure 5.8 summarizes the interactions of the various components of *Distree*.

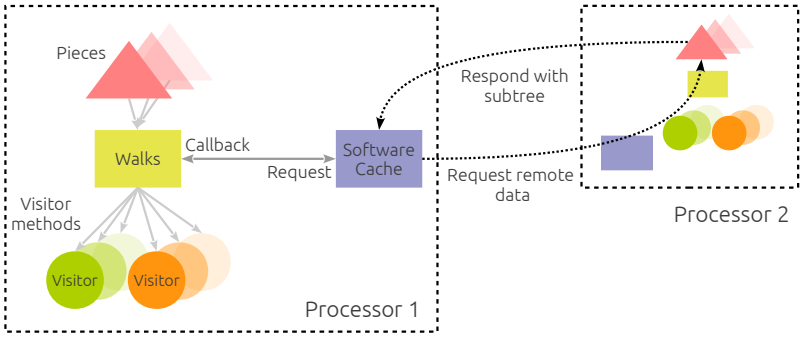


Figure 5.8: Schematic of the overall flow of control in a *Distree* traversal. The participating entities are visitors, walks, and the software cache. The walk provides an ordered iteration over tree nodes, and is guided by the visitor, which uses the nodes it receives for computations. A traversal over the tree generates remote data requests, which are mediated by the software cache. Lightweight continuations are created on data misses, to be resumed upon receipt of remote data.

The tree walk hides the implementation of the distributed tree data structure, and performs an ordered iteration over the nodes and leaves of the tree. The visitor processes the nodes visited by the walk, in a manner specified by the programmer. It dictates which nodes should be expanded by the walk into their children. The amount of local data is increased by combining the subtrees present on an SMP. The software cache enables remote data to be reused by co-located traversals with similar data access patterns. Finally, the load balancer takes into account the computational load exerted by tree pieces, as well as their spatial layout, to map them to PEs in an intelligent manner.

We now provide two examples of tree codes written using *Distree*. These examples serve not just to give concreteness to the constructs discussed previously, but also demonstrate their aptness for tree-based codes.

5.8.1 *Barnes-Hut*

We have already discussed the *Barnes-Hut* algorithm in some detail in § 5.3. Here we will use components provided by the *Distree* framework to implement this algorithm. Let us begin by identifying the various entities used in the program.

Tree data

```
class Particle {
  Vector3d<Real> position;
  Vector3d<Real> velocity;
  Vector3d<Real> acceleration;
  Real mass;
};

class Payload {
  Vector3d<Real> centerOfMass;
  Real mass;
};
```

(a)

```
class TreeDataDescriptor {
  typedef uint64_t Key;
  typedef Particle DataElement;
  typedef Payload NodePayload;
};
```

(b)

Figure 5.9: Data structures required for *Barnes-Hut* algorithm, and a `TreeDataDescriptor` that is used to instantiate the `distree::Node<>` template. The `TreeDataDescriptor` defines the key, node payload and data element type for the tree.

Figure 5.9(a) shows the *Distree* particle and node data structures to be used in the tree. The particle and node data types are used to create a *tree data descriptor*. As mentioned in § 5.2, the tree data descriptor collects in one place types that are of interest to *Distree*, but defined by the user. In our case, the `TreeDataDescriptor` class states that each node/data element has a 64-bit key, data elements have the type `Particle` and each node has a payload of type `Payload`.

Tree pieces

Functionally, a tree piece is a container for a portion of the global, distributed tree. From the point of view of the execution model, a tree piece is a message-driven chare that may be migrated across PEs by the load balancer. A tree piece must derive from the `distree::Piece` class, so that it inherits functionality for decomposition, tree building, and tree traversal.

```
class BarnesHutTreePiece :
  public distree::Piece<TreeDataDescriptor> {

  vector<TreeDataDescriptor::DataElement> myParticles_;
  vector<distree::Node<TreeDataDescriptor> *> myLeaves_;

  void load(...);
  void decompose(...);
  void build(...);
  void gravity(...);
  void integrate(...);
};
```

For our *Barnes-Hut* application we define the `BarnesHutTreePiece` class. Each piece contains a number of particles, and an (initially empty) list of pointers to the leaves that will hold the particles decomposed onto it. The list `myLeaves_` is properly initialized by the *Distree* framework during tree building.

The tree piece class defines a number of methods. Particles are loaded in parallel with other tree pieces using the programmer-defined `load` method. The `decompose` method is provided by the `distree::Piece` parent class, and decomposes the input particles over the set of tree pieces using either Oct or SFC decomposition. Similarly, the `distree::Piece`'s `build` method constructs an Octree over the particles assigned to each piece after decomposition. After building the tree, the *Distree* framework also performs tree piece consolidation and remote frontier annotation.

In our example, both data and work are distributed over the pieces array. We compute the net gravitational force on each particle of a tree piece by invoking the `gravity` method on it. The `gravity` method instantiates traversals by using *Distree* constructs, as we shall see later. Finally, particle trajectories are integrated through the user-provided `integrate` method.

Tree visitor

Next, let us examine the tree visitor class for gravity computation. Recall that traversal happens in parallel on each tree piece. Each tree piece initiates a single, fine-grained traversal for every one of its local leaves. Call such a

local leaf the *target* (since forces are being evaluated on its particles). The traversal's visitor maintains a pointer (`myPiece_`) to the tree piece that initiated it, as well as a pointer to the target leaf (`myTargetLeaf_`). The visitor defines computational, and state-update actions, as shown below:

```
class GravityVisitor {
  // state
  BarnesHutTreePiece *myPiece_;
  distree::Node<TreeDataDescriptor> *myTargetLeaf_;

  // behavior
  template<typename NodeType> bool node(NodeType *n);

  void leaf(TreeDataDescriptor::Key sourceLeafKey,
            TreeDataDescriptor::DataElement *sources,
            int nSources);

  void miss(TreeDataDescriptor::Key missedNodeKey);
  void hit(TreeDataDescriptor::Key hitNodeKey);
};
```

The visitor's `node` method is invoked once for every *source* tree node that is visited during the traversal for the target leaf. The `node` method of the visitor encapsulates the essence of the *Barnes-Hut* algorithm: if the target leaf is far enough away from a source, we compute the approximate *Barnes-Hut* interaction between the two, and return `false`. This signals to the top-down walk that the source node should not be expanded into its children for the current traversal. Otherwise, the target and source are too close, and we consider the children of the source in turn.

```
bool GravityVisitor::node(distree::Node<TreeDataDescriptor> *n){
  if(!Physics::open(myTargetLeaf_, n)){
    Physics::forces(myTargetLeaf_, n);
    return false;
  }
  return true;
}

void GravityVisitor::leaf(TreeDataDescriptor::Key sourceLeafKey,
                          TreeDataDescriptor::DataElement *sources,
                          int nSources){
  Physics::forces(myTargetLeaf_, sources, nSources);
}
```

For every source visited that is a leaf, the visitor computes the pairwise forces between the target leaf and the particles within the source leaf.

```
void GravityVisitor::miss(TreeDataDescriptor::Key missedNodeKey){
  myPiece_->oneMoreOutstanding();
}

void GravityVisitor::hit(TreeDataDescriptor::Key hitNodeKey){
  myPiece_->oneLessOutstanding();
}
```

Finally, the `hit` and `miss` methods are used to tabulate data hits and misses, so that we can tell when the algorithm has terminated.

Overall control flow

We now show the overall flow of control in the parallel application, beginning with the constructor of the main object, `Main`. Thish is where control first enters the user's program.

```
distree::TopDownWalk walk;
CProxy_BarnesHutTreePiece pieces;
distree::Tree<TreeDataDescriptor> tree;

Main::Main(CkArgMsg *m){
  // process parameters
  ...
  pieces = CProxy_BarnesHutTreePiece::ckNew(10*CkNumPes());
  tree = distree::Tree<TreeDataDescriptor>::instantiate(pieces);

  walk = distree::TopDownWalk::instantiate(readonly, readonly);

  auto nodeCache =
    distree::Cache<TreeDataDescriptor>::instantiateNodeCache();
  auto leafCache =
    distree::Cache<TreeDataDescriptor>::instantiateLeafCache();

  nodeCache.addReadonlyClient(walk);
  leafCache.addReadonlyClient(walk);

  thisProxy.commence();
}
```

We begin by instantiating a collection of tree pieces using the Charm++ factory method `ckNew`. It directs the Charm++ ARTS to create ten times as many pieces as there are PEs. This method returns a handle to the tree piece collection.

We then inform the *Distree* framework of this newly created tree piece collection through the `Tree::instantiate` call. In turn, we receive a handle to the global tree, which is used later in the program. Next, we instantiate the top-down walk object, requesting read-only access to remotely fetched, cached elements during traversal. Finally, we instantiate software caches for tree nodes and leaves (both in read-only mode), and `commence` the iterative *Barnes-Hut* algorithm.

```
void Main::commence(){
  tree.initialize(pieces);
  walk.initialize(tree);

  pieces.load(...);

  for(int i = 0; i < NumIterations; i++){
    pieces.assignKeys(...);
    pieces.decompose(...);
    pieces.build(...);

    walk.sync();
    pieces.gravity(...);
    walk.done();

    pieces.integrate(...);

    tree.free();
  }

  CkExit();
}
```

The code box above shows the driver code for the *Barnes-Hut* algorithm. Each method invocation therein is a blocking call, and returns only after the corresponding computations have finished on all members of the corresponding chare collections. As such, the code captures the overall structure of the algorithm. However, the data-driven computation of the traversal is hidden within the *Distree* framework. After some initialization, we load particles from disk into tree pieces, and enter the iterative part of the algorithm.

In each iteration, we: (1) Determine a key for each particle from its position (`assignKeys`); (2) Decompose the simulated particles onto the tree pieces (`decompose`); (3) Build the local tree for each tree piece (`build`). This operation is automatically succeed by tree piece consolidation, and remote frontier annotation; (4) Initiate the gravity traversal (`gravity`, sandwiched between `sync` and `done` calls to the *Distree* top-down walk object); and (5) use the calculated forces to integrate particle trajectories over a small time interval (`integrate`).

We can see that much of the parallel implementation of the *Distree* framework is hidden from the programmer behind calls to `decompose`, `build`, `sync` and `free`. Moreover, save their instantiation, the code includes no mention of software caches. Therefore, a program that uses *Distree* is able to very clearly demarcate the lines between framework code and user code.

This theme of separation of concerns is evident even in the manner in which the user-provided *visitor* interfaces with the *Distree* walk. To see how, let us look at the `BarnesHutTreePiece::gravity()` method, which is invoked in step (4) above.

Traversal initiation

```
distree::TopdownWalk walk;
distree::TreeHandle<TreeDataDescriptor> tree;

void BarnesHutTreePiece::gravity(...){
  gravityOutstanding() = 0;

  for(int i = 0; i < myLeaves_.size(); i++){
    GravityVisitor *g = new GravityVisitor(&myLeaves_[i], this);
    walk.instance()->go(tree.root(), g);
  }
}
```

In § 5.8.1, we noted that a fined-grained traversal is instantiated for each local leaf of a tree piece. The definition of the `gravity` method of the tree piece class above shows how this is done. An instance each of `GravityVisitor` and `TopdownWalk` are obtained, together constituting a traversal over a particular leaf local to the tree piece (`myLeaves_[i]`)). The traversal is initiated by invoking the `go` method on the walk, with the visitor as an argument. The point at which the traversal starts is given by the first argument of the method (the `root` of the tree).

At this point, the *Distree* framework takes over, invoking various methods on the programmer-provided visitor objects, and fetching subtrees of the global tree that are remote to the current SMP processor via the cache, as necessary.

5.8.2 Smoothed particle hydrodynamics

The smoothed particle hydrodynamics (SPH) algorithm [113] is a Lagrangian (mesh-less) method for the computation of a given kernel over the neighborhood of each point in an input distribution. It is used extensively in computational astrophysics to simulate the interactions of the dynamics of gaseous distributions [114], including galaxy and star formation. It is also a popular method for computing the motion of fluids [115].

At the core of the algorithm is the k nearest neighbor search operation, which is used to create a *cloud* of $k \approx 32 - 64$ neighbors around each particle p. This cloud is used to determine a weighted average of the density at each particle p, whereafter pressure gradients (and hence force) can be evaluated.

Below, we phrase the SPH algorithm as a composition of two successive traversal phases. The first, an up-and-down walk, calculates the density distribution, whereas the second, a top-down walk, uses the calculated densities to calculate forces due to pressure gradients. The code to initiate data decomposition and tree building procedures is nearly identical to that in the *Barnes-Hut* example, and so we do not discuss it here. The `SphTreePiece` class used to encapsulate *Distree* pieces, is also similar to the `BarnesHutTreePiece` class from our previous example. We present only those portions of the SPH code that are significantly different from the previous example.

```
class Particle {
  Vector3d<Real> position;
  Vector3d<Real> velocity;
  Vector3d<Real> acceleration;
  Real mass, density, pressure;
  SphParticleData *sphData;
};

class Payload {
  Vector3d<Real> centerOfMass;
  Box<Real> boundingBox;
  Real radius2;
  Real mass;
};
```

(a)

```
class TreeDataDescriptor {
  typedef uint64_t Key;
  typedef Particle DataElement;
  typedef Payload NodePayload;
};
```

(b)

Figure 5.10: The user-provided node and data element structures for the SPH algorithm; and the tree data descriptor class.

Tree data descriptor

The particle and node data structures for the SPH algorithm are similar to those of *Barnes-Hut*. However, an SPH `Particle` has additional `density` and `pressure` attributes. Each node also has a `radius2` attribute. This attribute defines the sphere within which a visited source particle p must lie, in order for p to be a viable candidate for one of the k-nearest neighbors of *any* target particle enclosed by n. Finally, a per-particle, distance-sorted priority queue (`SphParticleData`) is maintained in order to efficiently construct the k nearest neighbors list. The tree data descriptor provides *Distree* these concrete classes as template parameters.

Visitors

```
class DensitySphVisitor : public BaseSphVisitor {
  bool node(distree::Node<TreeDataDescriptor> *n);
  void leaf(TreeDataDescriptor::Key sourceKey,
            TreeDataDescriptor::DataElement *sources,
            int nSources);
};
```

(a)

```
class DensitySphVisitor : public BaseSphVisitor {
  bool node(distree::Node<TreeDataDescriptor> *n);
  void leaf(TreeDataDescriptor::Key sourceKey,
            TreeDataDescriptor::DataElement *sources,
            int nSources);
};
```

(b)

Figure 5.11: The programmer-defined visitor classes, and its two children, one each for the density, and pressure gradient walks.

The SPH walk defines two concrete visitor classes, both of which derive from the common `BaseSphVisitor` class (not shown). The base class maintains a pointer to the tree piece that initiated its traversal, as well as the local target leaf of particles. In addition, it maintains per-target-particle SPH data structures such as distance-sorted source-particle priority queues. These are used to find neighboring source particles for individual target particles within `myTargetLeaf_`. The `DensitySphVisitor` calculates density at every target

particle by finding its k nearest neighbors, whereas the `PressureSphVisitor` computes pressure gradients. We show the `node` and `leaf` methods of the `DensitySphVisitor` class to outline its behavior.

```
bool node(distree::Node<TreeDataDescriptor> *source){
  return open(myTargetLeaf_,
              myTargetLeafSphData_,
              source);
}
```

Every time we visit a node of the tree, we ask *Distree* to recursively visit its children only if the *opening criterion* (`open`) for the target leaf is satisfied, given the current set of neighbors (k or fewer) for each of the leaf's enclosed particles.

```
bool open(distree::Node<TreeDataDescriptor> *leaf,
          SphLeafData *leafSphData,
          distree::Node<TreeDataDescriptor> *source){

  Real rLeaf = leafSphData->size() + leafSphData->maxDist();
  Sphere<Real> leafSphere(leafSphData->center(), rLeaf);
  if(!intersect(source->boundingBox, leafSphere)) return false;

  for(auto target : leaf->getParticles()){
    if(target->sphData.numNeighbors() < MAX_SPH_NEIGHBORS ||
       intersect(source->boundingBox,
                 Sphere<Real>(target->position,
                              sqrt(target->sphData.maxDist2())))
      return true;

  return false;
}
```

The opening criterion checks whether the source node is so distant that its bounding box doesn't intersect with even the *union* of the target particles' search spheres. If so, we discard the source right away. Otherwise, we consider the individual search spheres of the target particles, and if any one of them intersects the source's bounding box, then we direct the walk to consider the children of the source in turn (by returning `true`).

```
void leaf(Key sourceKey,
          TreeDataDescriptor::DataElement *sources,
          int nSources){
  for(int i = 0; i < nSources; i++)
    Physics::Sph::adjustNeighbors(&sources[i], leaf(), data());
}
```

For every leaf of source particles encountered during the traversal, the `DensitySphVisitor` performs an all-pairs comparison between the target particles and the source particles (in `adjustNeighbors`). The idea is to determine, for every source particle s and every target particle t, whether s is closer to to t than any of t's k current neighbors. If t currently has fewer than k neighbors, we add s to the priority queue of t without doing any distance comparisons. We omit the code for `adjustNeighbors` in the interest of brevity.

Overall control flow

We now briefly discuss the initialization of the SPH algorithm, and its overall flow of control.

```
distree::UpAndDownWalk densityWalk;
distree::TopDownWalk pressureWalk;
CProxy_SphTreePiece pieces;
distree::Tree<TreeDataDescriptor> tree;

Main::Main(CkArgMsg *m){
  // process parameters
  ...
  pieces = CProxy_SphTreePiece::ckNew(10*CkNumPes());
  tree = distree::Tree<TreeDataDescriptor>::instantiate(pieces);

  densityWalk = distree::UpAndDownWalk::instantiate();
  pressureWalk = distree::TopDownWalk::instantiate();

  auto nodeCache =
    distree::Cache<TreeDataDescriptor>::instantiateNodeCache();
  nodeCache.addReadonlyClient(densityWalk);
  nodeCache.addReadonlyClient(pressureWalk);

  auto leafCache =
    distree::Cache<TreeDataDescriptor>::instantiateLeafCache();
  leafCache.addReadonlyClient(densityWalk);
  leafCache.addWritethroughClient(pressureWalk);

  thisProxy.commence();
}
```

As before a collection of tree pieces is created using the appropriate factory method provided by Charm++. Then, a `distree::Tree` is instantiated with the algorithm's tree data descriptor, and up-and-down and top-down walks are instantiated. Although both walks access remote nodes (through the `nodeCache`) in read-only mode, the pressure gradient walk accesses leaf data in *write-through* mode. This is done to ensure symmetry of forces: a cached source particle fetched from some remote tree piece itself experiences reactive forces due to its interactions with targets (by Newton's Third Law). Reactive forces are accumulated on remotely fetched particles, and at the end of the walk, these particles are flushed back to their owner tree pieces.

```
void Main::commence(){
  tree.initialize(pieces);
  densityWalk.initialize(tree);
  pressureWalk.initialize(tree);

  pieces.load(...);

  for(int i = 0; i < NumIterations; i++){
    pieces.assignKeys(...);
    pieces.decompose(...);
    pieces.build(...);

    densityWalk.sync();
    pieces.density(...);
    densityWalk.done();

    pressureWalk.sync();
    pieces.pressure(...);
    pressureWalk.done();

    pieces.integrate(...);

    tree.free();
  }

  CkExit();
}
```

The overall flow of control of the algorithm is similar to that of *Barnes-Hut*: after loading particles from an input file, they are decomposed onto tree pieces, and the distributed tree is built using *Distree*-provided methods. We initiate the density computation walk first, and when it has completed, follow up with a walk to calculate pressure gradients. It is when the `pressureWalk.done()` invocation is made, that the cache writethroughs occur.

Traversal initiation

Finally, we present the initiation of the two SPH walks.

```
distree::UpAndDownWalk densityWalk;
distree::TopDownWalk pressureWalk;
distree::Tree<TreeDataDescriptor> tree;

void SphTreePiece::density(...){
  for(int i = 0; i < myLeaves_.size(); i++){
    DensitySphVisitor *d;
    d = new DensitySphVisitor(&myLeaves_[i], this);
    densityWalk.instance()->go(&myLeaves_[i], d);
  }
}

void SphTreePiece::pressure(...){
  for(int i = 0; i < myLeaves_.size(); i++){
    PressureSphVisitor *p;
    p = new PressureSphVisitor(&myLeaves_[i], this);
    pressureWalk.instance()->go(tree.root(), p);
  }
}
```

Note that in the code above, each up-and-down density-finding traversal begins at a particular *leaf* of the distributed tree, and not at its root.

5.8.3 Discussion

Given the similarity of the above code to that present in the *Barnes-Hut* example, we call attention to the following. When expressed in *Distree*, the high-level structures of markedly different algorithms (in our case, the *Barnes-Hut* and SPH algorithms) appear very similar. In our estimation, this points to the goodness of the abstractions provided by *Distree*.

The differences in algorithms are not structural, but are instead encapsulated within specializations of visitors, and the types of traversals that are initiated on the tree. Algorithms can be pieced together by appropriately composing different combinations of walks and visitors. One can imagine a

component-based approach to tree code construction using our framework. A library of commonly-used walks and visitors would be provided, and programmers could assemble a variety of tree codes using these building blocks.

5.9 Performance results

Let us now consider the performance of *Distree* code. For the *Barnes-Hut* algorithm, we compare performance against a manually tuned code written entirely in Charm++. However, such a point of comparison was not available for the SPH code. As such, we only demonstrate good scaling numbers for that algorithm.

We obtained strong scaling results on the Blue Gene/P machine at Argonne National Lab. For the *Barnes-Hut* benchmark, we used two datasets. The first, *dwf*, is a 5 million particle data set representing the formation of a dwarf galaxy [102]. This data set has a wide density variation, with a very dense center. The second data set, *hrwh*, has a more uniform distribution of mass, and contains 16 million particles. It has been adapted from the work of Heitmann *et al.* [116] and our own previous work [102]. Both simulations suffer from load imbalance due to density variations, though these imbalances are much more marked in the first data set.

For the SPH experiments, we used the *dwf* data set and a synthetic one-million particle data set with Plummer statistics [117]. It models, in a semi-realistic way, the mass distributions of certain star cluster formations, and is popular among N-body practitioners for its analyzability. The mass distribution is clustered near the center of the data set, and falls off rapidly in the radial direction, leading to load imbalance.

For both applications, the load balancing strategy of § 5.7.3 was used to dynamically reassign tree pieces to PEs based on load measurements.

5.9.1 *Barnes-Hut*

We compare performance of the *Distree* version of the *Barnes-Hut* algorithm with a hand-tuned, comparable, Charm++ code. The latter was part of the winning entry to the HPC challenge [26], and has previously demonstrated good scaling results on up to 32K cores.

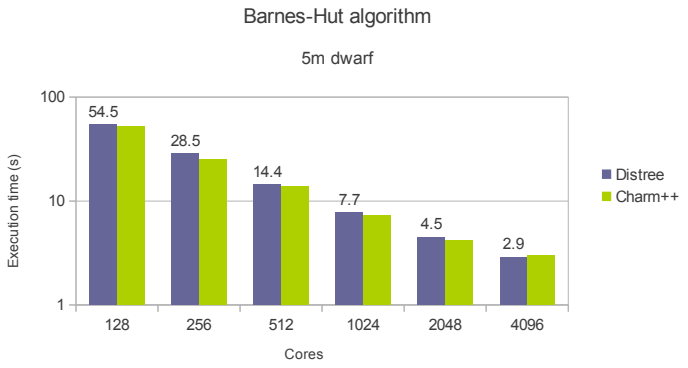


Figure 5.12: Performance comparison of *Distree* code (blue) with Charm++ code for the *Barnes-Hut* algorithm. The input data set was the clustered, *dwarf* simulation.

Figure 5.12 compares the performance of *Distree* code with its highly optimized Charm++ counterpart for the 5 million particle, clustered data set, *dwarf*. The blue (left) bars are labeled with the average time taken by the *Distree* code to complete one iteration of the dominant force computation phase of the algorithm. As we can see, the *Distree* code performs comparably with the Charm++ version, especially as the code is scaled up. The parallel efficiency at 4K cores relative to 128 cores is about 59 per cent. This is a remarkable result, given that at that scale, we have only about a thousand particles per core.

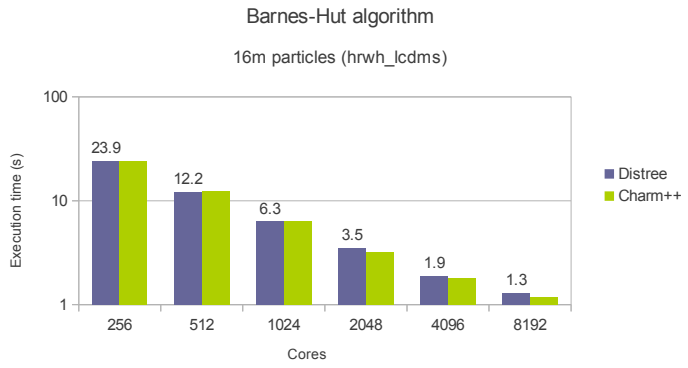


Figure 5.13: Performance comparison of *Distree* and Charm++ for the 16 million particle, *hrwh* data set.

In Figure 5.13 we compare the performance of the two versions of the

Barnes-Hut algorithm on the uniform, 16 million particle data set, *hrwh*. This figure shows even better scaling results than Figure 5.12. We get good speedups all the way up to 8192 cores. However, in this particular simulation, we see that the optimized Charm++ has a slight performance edge over its *Distree* counterpart. Although the difference in performance between the two versions is negligible on up to 1K cores, it grows as we scale to 8K cores. The parallel efficiency at 8K cores relative to 256 cores is about 57 per cent for the *Distree* version, and about 65 per cent for the Charm++ version.

5.9.2 SPH

We now assess the performance of the SPH benchmark on the same, Blue Gene/P machine on two data sets. The first set of tests used the one-million particle Plummer distribution. For the second set of tests, we used the same highly non-uniform, five-million particle *dwf* data set as previously.

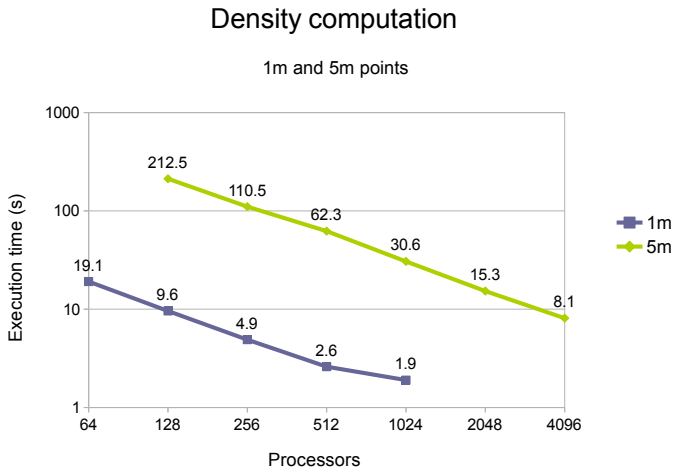


Figure 5.14: Strong scaling for SPH benchmark on two data sets.

The *Distree* SPH algorithm scales well on up to 512 cores with the Plummer distribution. However, performance starts to level off at 1k cores. This is because at that scale, there are fewer than one thousand particles on each core (and fewer than a hundred per tree piece), whereas standard benchmarking tests for production-quality simulators normally feature ensembles of at least ten thousand particles per core (For instance, in recent results Warren [118] uses about 100,000 particles per core at scale.) We obtain linear scaling

for the larger, *dwf* data set on up to 4k cores. There was no hand-tuned, Charm++ version of the SPH code with which to compare performance and program length. However, we believe that given the asynchronous nature of the *Distree* model, the performance of codes written in this framework will be comparable to corresponding codes written in Charm++. Indeed, our claim is supported by the similarity in scaling profile of the Charm++ and *Distree* versions of the *Barnes-Hut* algorithm.

5.10 Productivity

Let us make a comparison of the number of lines of code written by the user when using *Distree*, and when using only Charm++. We caution that the estimates used here are subject to coding styles, as well as the presence of some comments. Therefore, the numbers here are only a first-order approximation of the amount of effort expended in writing code.

Overall, the *Distree* version of the *Barnes-Hut* algorithm comprised 2199 lines, whereas its Charm++ counterpart consisted of 4603 lines. Therefore, by basing our application on *Distree*, we saw a reduction in SLOC of about 52 per cent. Figure 5.15 examines the causes for this reduction.

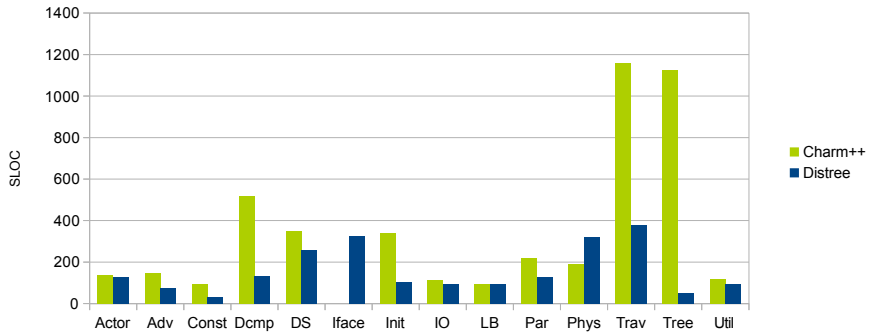


Figure 5.15: A comparison of SLOC for the Charm++ and *Distree* versions on the *Barnes-Hut* benchmark.

Most notably, the Charm++ version contains a significant amount of code for tree building (Tree; 1127 lines or 24 per cent of the Charm++ code), traversal (Trav; 1158 lines, 25 per cent) and decomposition (Dcmp; 518 lines; 11 per cent). The majority of the code for these phases is pushed

into the framework in the *Distree* version. The only code present in the *Distree* version is that required to configure and invoke the corresponding *Distree* routines. This results in saved development and debugging effort, and increased reusability across instances of tree-based algorithms.

The contributions of code for I/O to load particles (IO), load balancing (LB), physics routines (Phys), utilities for bit-word manipulations (Util), and data structures describing the nodes and leaves of the distributed tree (DS), are similar between the two versions. The Charm++ version does not interface (Iface) with *Distree*. However, it manually sets up (Init) the objects (actors) of the simulation, which is otherwise done internally by the *Distree* framework. The amount of C++ code required to describe the behavior of the main set of coarse-grained objects (Actor), i.e. *tree pieces*, is nearly identical in the two versions. Finally, the code to update particle trajectories by advancing (Adv) their positions, is comparable in the two versions. However, the Charm++ version requires some additional parallel code in order to achieve this.

No point of comparison was available for the SPH algorithm. However, we note that its implementation in *Distree* contained 3317 lines of code. So as to gain some insight into the relative contribution of different phases to the codebase, we compare the SPH algorithm with the *Barnes-Hut* algorithm in Figure 5.16.

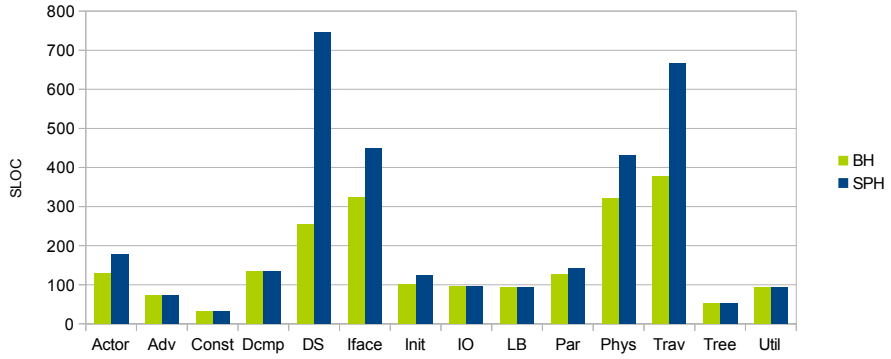


Figure 5.16: Break-up of SLOC for the *Distree* SPH benchmark.

The break-up shows that the SPH algorithm has a nearly identical count of SLOC for some categories that relate to interfacing with *Distree*, e.g. Dcmp, Init and Tree. However, there are some categories for which the SPH al-

gorithm requires slightly more code (e.g. Actor, Init, Par and Trav). This is chiefly due to the fact that the SPH algorithm incorporates two traversals, compared to the single one in the *Barnes-Hut* algorithm. There are other categories still, for which the SPH code requires significantly more code than does *Barnes-Hut*. These categories, namely DS, Iface, Phys, and Trav, are larger than their *Barnes-Hut* counterparts because the SPH algorithm requires more sophisticated data structures, and more code to simulate physical interactions between particles. Therefore, the size of this last set of categories is determined mainly by the demands of the application.

Thus, we believe that *Distree* provides a flexible, productive and performance oriented means for writing tree-based algorithms. Our eventual aim is to provide a larger toolkit of commonly used tree walks, and visitor classes, which can be used by programmers to rapidly develop and experiment with tree codes. We believe that *Distree*'s ethos of assembling algorithms from components is especially well-suited to this paradigm.

5.11 Related work

Let us close this chapter with a look at some approaches that provide a counterpoint to *Distree*. The Global Trees (GT) framework [119] provides a shared memory view of a distributed, abstract tree data structure. Similar to our own proposals, GT considers the chunking of tree data, both for layout in memory and coarse-grained communication over the interconnect. It also provides a separation of concerns between tree traversals and the actions performed on each visited node. However, efficiency of tree manipulation operations is provided through relaxed memory consistency models. We consider the associated use of barriers and fences to be decidedly low-level in nature, and hard to program with and reason about. By contrast, we provide an object-oriented, phase-based approach to tree accesses, which allows an efficient implementation (through, for example, bulk communication of accrued updates) without sacrificing programmer productivity.

Jo and Kulkarni [120] consider the use of compiler-directed *tiling* of recursive tree traversals to enhance locality in shared memory machines. We have used similar techniques designed for use in the ChaNGa N-body code [8,102] to increase cache performance both for remotely fetched data and locally

traversed portions of trees. Subsequently, these techniques have been incorporated into *Distree* as well.

The *Distree* framework has a strong grounding in computational cosmology, chiefly due to the author's involvement in the ChaNGa project. Other scalable, distributed memory tree codes for the efficient calculation of gravity include GADGET-2 [121] and the MPI-based predecessor of ChaNGa, PkdGrav [122]. In previous work, Singh *et al.* [123] have studied aspects of parallel performance, namely data locality and load imbalance, of tree-based codes. That study was conducted in the context of shared memory systems, and represents some of the earliest computer-science oriented investigation of these algorithms. Shan and Singh [124] provide a discussion of algorithms for the parallel construction of trees on shared memory multiprocessors. More recently Zhang, Behzad and Snir [106] have built on that work, and have developed a PGAS variant of the algorithm. Much in the vein of Gioachin *et al.* [8], they provide a systematic discussion of optimization techniques that enable irregular, tree-based codes to scale on distributed memory machines.

However, distributed trees are not the preserve of computational cosmology. The distributed systems community has developed several tree-based abstractions for loosely coupled concurrent systems. Among these are a search tree for peer-to-peer applications (Brushwood [125]), a distributed, fault-tolerant, B-tree that employs migration-based load balancing [126], and a distributed tree that monitors communication between its constituent nodes in order to dynamically optimize its topology [127]. These tree data structures support a richer set of functionality than do the trees of *Distree*. However, they have not been designed for the demands of tightly coupled parallel simulations. As a result, these data structures have not received wide consideration in the HPC realm.

CHAPTER

6

INTEROPERATION

Synopsis. Having explored the productivity and performance benefits of three specialized languages, we now turn our attention to the matter of interoperation between them. Essentially, we wish to address the question of how a module written in one specialized language can communicate with a module written in another. Recall that the languages discussed in Chapters 3, 4 and 5 were *incomplete*: Each language was useful in expressing a certain strict subset of all possible parallel programs. Here, we devise a framework to regain completeness of expression through *interoperation*. This chapter provides the final ingredient of our multi-paradigm approach, and illustrates it through a case study. We consider a *Barnes-Hut* application composed of four pieces, one written in each of Charm++, *Charisma*, *DivCon*, and *Distree*. We show that the performance of this multi-paradigm code is competitive with a corresponding, hand-tuned, Charm++ application. We also argue that the multi-paradigm code is more succinct, and presents a global view of control flow where possible. To end, we discuss some of the shortcomings of our approach to interoperation.

6.1 Scope of interoperation

Interoperation between languages has various connotations, each leading to a different formulation of the overall problem of enabling communication between entities within multiple modules. Therefore, we first clarify the scope and context of interoperation in this work. To do this, we identify five broad kinds of interoperation supported by existing work:

6.1.1 Interface description

The model of defining language-neutral interface specifications between modules written in different languages, has been a particularly successful one. Since its inception in 1991, CORBA [128] has influenced several enterprise-related efforts to enable interoperability between languages, e.g. Java's RMI. Objects export methods that may be remotely invoked through an *interface definition language* (IDL). An associated compiler generates glue code to enable remote access to objects through handles of some sort.

This approach has been adopted in the realm of HPC as well, with projects such as SIDL (Scientific IDL) [129]. SIDL and its associated toolkit, Babel [130], enable modules written in C++, Fortran, Java and Python, among other languages. Unlike CORBA, SIDL provides its own language-neutral multidimensional arrays, and supports complex data types. The Common Component Architecture (CCA [131]) elaborates on this idea, allowing objects to define and modify their interfaces at run-time. In P-COM^2 [132], a component is modeled as a state machine with transitions on external *interaction* (communication) events. Components are assembled into a data flow graph, and a compiler generates the necessary MPI (among other targets) code to realize the application.

Protocol Buffers [133] makes similar, IDL-based provisions for distributed applications. Just as with tools aimed at HPC applications, Protocol Buffers automates serialization and deserialization of data.

6.1.2 Component frameworks

This approach is popular in the multiphysics simulation community. Modules describing different regimes or dynamical systems are created within a

single framework, which typically makes provisions for the orchestration of modules, data management, and expression of concurrency. The framework also generally provides common building blocks that can be incorporated into more sophisticated algorithms.

A good example of this approach is the Roccom [134] framework. The primary view of data is structured and unstructured meshes, thereby limiting its application, in general, to PDE solvers. POOMA [135] provides a more generic data model, wherein so-called *fields* and *particles* are supported. PETSc [136] takes a lower-level approach to the provision of algorithmic building blocks for PDE solvers. It provides vectors and matrices as its primary data structures, and contains scalable implementations of a number of important numerical methods. As such, it has seen widespread adoption in a variety of application domains where PDE solvers are employed. The Uintah Computational Framework (UCF [137]) is based on CCA, and provides a toolbox of libraries and components to develop PDE solvers using structured AMR grids on distributed memory machines. Instead of specifying object interfaces explicitly, in UCF the user describes his application as a *taskgraph* of acyclic dependencies between tasks.

6.1.3 Complementary or compatible paradigms

More recently, the use of *hybrid programming models* has gained prominence. In this scheme, the SPMD model of MPI is complemented with another language, with the purpose of simplifying the expression, and in some cases, improving the performance of shared memory communication inherent in the user's application. For instance, Tang *et al.* [138] and Jones *et al.* [139] have used MPI to express inter-node communication, and OpenMP to express intra-node parallelism in different application domains, and to very good effect. Of more general interest is the study of Cappello and Etiemble [140], who demonstrate good results on the NAS benchmarks using the same MPI+OpenMP scheme.

Dinan *et al.* [141] have examined the different problem of coupling two SPMD language, namely MPI and UPC. They propose that MPI be used to express communication between subgroups of UPC threads that themselves span multiple nodes. The idea is that users can intra-group communication

productively, using the PGAS primitives of UPC. This allows them to limit the size of processor groups that engage in distributed shared memory communication, and therefore improves efficiency. Similarly, the Global Arrays abstraction [80] can interoperate with MPI, so as to combine distributed shared memory accesses with irregular communication patterns. Finally, there has been a recent proposal to include explicit support for shared memory programming in the MPI 3.0 standard itself, leading to the so-called *MPI+MPI* hybrid programming model [142].

6.1.4 Embedding of specialized languages

The Delite [15] project has goals that are very similar to those of this thesis. That project is developing higher-level DSLs that can be transformed into high-performance, lower-level components. The Delite project has a broader scope than this thesis, however, in that it targets heterogeneous architectures. The languages described in this thesis are not DSLs, since they capture interaction patterns between units of computation. Therefore, they are more broadly applicable than DSLs. However, if one were to draw a comparison, the specialized languages described herein would be akin to *external* or *stand-alone* DSLs, since they are parsed independently, and have individual compiler and runtime substrates. On the other hand, the Delite project focuses on *internal* DSLs, i.e. those that are embedded within a host language, which in their case is Scala. The Delite framework also makes provisions for *lightweight modular staging* [143], so that DSL-specific optimizations can be made in the host language's compiler infrastructure.

6.1.5 Multi-paradigm languages

Finally, we consider languages that themselves allow for programs to be expressed in multiple paradigms. By virtue of being written in such a language, modules expressed in different paradigms would at least nominally be interoperable. Even though such modules may be interoperable, it is non-trivial to compose them into a functioning program, given the different semantic domains of each paradigm. Nonetheless, the single language substrate on which the paradigms are based does simplify the process of interoperation.

In this regard, such multi-paradigm languages are similar to the specialized languages of this thesis, which are all based on the Charm++ runtime system.

A prime example of this approach is the Oz [13] language, and its accompanying runtime system, Mozart [14]. The language contains constructs for object-oriented, functional, logic, and message-passing, and data-driven thread programming. It is primarily intended to be a language for the development of distributed programs, and not tightly coupled parallel programs.

6.1.6 This thesis

In this chapter, we will examine a limited form of interoperation between *specialized languages that are based on a single runtime substrate*. We will see in § 6.3 that for modules written in *Charisma* and *DivCon*, we require require a limited form of interface definition, similar to the model adopted in § 6.1.1. In a manner similar to the component frameworks of § 6.1.2, the modules expressed in our specialized languages will often require so-called *glue* code to compose different modules. Although we show in § 6.6 that such glue code makes a minor contribution to overall application code, a limitation of the present work is that it must be written by the programmer. Unlike MPI and UPC (§ 6.1.3), the languages of this thesis are not semantically compatible, nor are they applicable in complementary settings (e.g. intra-node OpenMP and inter-node MPI). This is one of the challenges to interoperation that we address in the following section. Another feature that distinguishes our work from that of § 6.1.4 is that our specialized languages are *not* embedded, and have greater applicability than DSLs However, we share the objective of embedded DSLs in attempting to provide performance *and* productivity through specialization. Finally, the mechanism that enables multi-paradigm programming in this thesis, is similar to that employed by the Oz language. Namely, *Charisma*, *DivCon* and *Distree* are all based on the same runtime system, thereby facilitating interoperation. Whereas Oz incorporates constructs for multiple paradigms in a single language, we advocate a separation between individual paradigms in the form of language boundaries.

6.2 Challenges to interoperation

There are two chief obstacles that we must overcome in order to develop a means of interoperability between the languages discussed in this thesis. First, the languages have very different domains of application: whereas *Charisma* expresses data-independent control and data flows, *DivCon* is specifically designed to express recursive, tree-structured computations on distributed sets of data. By contrast, the *Distree* framework is meant to express algorithms based on distributed trees. In providing a mechanism for interoperation, we must somehow reconcile the disparate domains of applications of the three languages.

Second, if modules written in these three languages are to interoperate in an efficient manner, they must effectively share the processors of the parallel machine. Moreover, processors must be shared in a manner transparent to the programmer. We discuss the two issues of semantic disparity, and resource sharing in greater detail below.

6.2.1 The problem of effectively sharing processors between modules

Let us first consider a general concern relating to the execution of multi-module applications on parallel machines. Namely, one must efficiently divide processors among the modules of the application.

We first examine two traditional approaches to processor sharing in the context of SPMD programming models, such as MPI. The problem here is twofold: (i) Messages may only be sent to processes or ranks, and messages are *tagged* so as to distinguish between messages addressed to individual modules; and (ii) The control flow of the entire process, and not just an individual module executing within a process, must be specified by the SPMD application code. This leaves us with two ways to write a multi-module SPMD application:

- Phrase the control flow of each process/rank as a scheduler, which repeatedly posts "receives" for messages with any possible tag. When such a "receive" eventually completes, the message is dispatched to the intended module within the process. This is tantamount to implementing a message-driven infrastructure within an SPMD framework.

Although acceptable from the performance standpoint, this approach implies that for the addition of each module into the application, one must modify the scheduler code. Furthermore, one must be aware of tags used by other modules when adding a new one to the application. Clearly, language-level support for modularity would be superior to this scheme.

We must also consider the question of which module instances are brought together on the same rank, and its implications for locality of data access. Consider an astrophysics application with two modules, one for tree-based gravity computation, and the other performing an *Eulerian*, unstructured mesh-based hydrodynamics computation (unlike the *Lagrangian* method described in § 5.8.2). Typically, a graph partitioner [144] is used to decompose the hydrodynamics mesh over the set of available ranks. In this case, the k-th piece of the mesh has no correspondence to the k-th rank's gravity subtree, leading to poor locality of data access, since gravity computations will involve the hydrodynamics particles, but not vice versa. Some dynamic, run-time instrumentation is required to measure communication volume (i.e. the degree of coupling between instances of different modules).

- A second approach to executing multi-module applications is to divide the set of processors among modules. In the following subsections, we enumerate the two basic ways to accomplish this. In the process we recapitulate some of the arguments made previously in the work of Gursoy and Kale [145].

Space-division

Typically, the set of available processors is *a priori* divided among the various modules of the program. The proportion of processors assigned to each module is decided by a static load balancing scheme, taking into account the anticipated, or previously measured, computational load exerted by each module. This can be problematic: First, the load of each module must be estimated accurately. In practical terms, one must repeatedly sample application performance with different static partitionings of the machine, and iteratively refine the partitioning. This tuning must be done to some

extent for each instance of the application's input, and every time a code is run on a new parallel machine.

Time-division

By contrast, we may divide the processors of the machine among modules *in time*. That is, each module runs exclusively on the entire parallel machine, and is then replaced by the next module, as determined by a globally known schedule. This technique is feasible only when the application's dynamics permit such a lock-step execution between modules. As a counterexample, consider a coupled multi-scale simulation of a biophysical simulation, which is gorverned chiefly by Newtonian dynamics (*a la* NAMD [66]), except that at so-called *active sites* of binding, and interaction between molecules, quantum-mechanical effects, as simulated by a QM module (e.g. OpenAtom [9]), become important. It might happen that there are several such localized interactions, and therefore it is inefficient to yield the entire machine to their execution while the molecular dynamics computations are suspended. Even if the application permits time-division of processors, there is often idle time during the execution of a particular module. This idle time arises due to a number of effects, including load imbalance. If we allowed the concurrent execution of multiple modules on processors of the parallel machine, idle time in one module could be overlapped with useful work in another (subject to data dependencies).

Multi-module execution in Charm++

Four provisions are made by the Charm++ programming model to enable the efficient composition of parallel modules.

First, Charm++ allows the explicit definition of modules at the programmatic level. The basic computational units within modules, namely coarse-grained objects (called *chares*), can be uniquely identified across modules and across processors. A chare within a module that wishes to communicate data to another chare, possibly in a differently module, simply sends it an *active* message, which specifies both the recipient of the message, and the work to be performed upon its receipt. Conversely, a chare that must synchronize with another chare (perhaps in a different module) before per-

forming a given computation, need only specify that the computation is to be performed upon the receipt of a corresponding message from the other chare.

Second, the Charm++ scheduler on each processor continually dequeues messages received from other processors, and schedules their execution on the appropriate chares. Therefore, computations occur in a message-driven fashion, subject to the availability of data.

Third, the user's application is decomposed into the activities of objects. Thus, there is an explicit distinction made between processors and the units of work performed on them. Objects can therefore be *migrated* to effect more efficient mappings of work to processors. This is an especially powerful technique when combined with Charm++'s ability to dynamically instrument objects for load and communication. In the context of the two-module astrophysics example discussed in § 6.2.1, we could dynamically determine, based on the degree of coupling of objects, a good mapping of objects to processors.

Fourth, Charm++ allows modules to be dynamically instantiated and destroyed. The objects within a module can then be distributed across processors according to a strategy provided by the user. This capability is critical in adapting to the requirements of dynamic applications, such as the one discussed in § 6.2.1.

We contend that these features allow Charm++ to efficiently and productively compose parallel modules. In this chapter, we leverage Charm++'s support for multi-module programs to develop a means for modules to initiate, exchange data, and synchronize with modules written in different languages. By virtue of Charm++'s message-driven execution model, a parallel program that is composed in this way can automatically overlap idle time with useful work across modules, subject to the availability of data.

6.2.2 The problem of disparity in programming models

We have considered some general issues when running multi-module applications on a parallel machine, and also proposed to leverage the message-driven model of Charm++ to mitigate some of them. The programming models of our three specialized languages present the following confounding factors.

Different units of computation

The basic unit of computation in each language differs significantly across languages.

Charisma is a language that orchestrates computations performed on indexed collections of *explicit, coarse-grained objects* through the publication and consumption of values (messages). As such, its basic units of computation are objects that consume (i.e. wait for) and produce messages. These explicit computational objects also have unique, identifying names. With the right hooks in place, this should allow for a particular *Charisma* module's objects, to be accessible from a different module, whether it is written in *Charisma*, or in a different language such as Charm++.

DivCon describes computations in terms of *implicit, fine-grained tasks*, each one performing the typically small amount of work associated with a recursive function invocation in a divide-and-conquer program. Therefore, the basic unit of a *DivCon* computation is a task that can spawn other tasks. Unlike *Charisma*, these units of *DivCon* computation do not have explicit names. As such, it is not possible for code outside a given *DivCon* module to reference tasks within a *DivCon* computation.

Finally, tree traversals in *Distree* are independent, loosely ordered iterations over subsets of tree nodes and leaves. These traversals perform fine-grained calculations on tree data, either available locally, or fetched from a remote source. *Distree* traversals, similarly to *DivCon* tasks, are anonymous, and their execution is driven by the availability of tree data.

So, the first question we must address is, how can we instantiate the basic computational units of one module from another, in a uniform manner? When addressing this issue, we would like to avoid the *pairwise interoperation problem*, i.e. having to develop pairwise schemes for interoperation between specialized languages. For instance, we would like to have a uniform way of instantiating a recursive *DivCon* computation, regardless of the language in which the instantiating module was written.

Different data and synchronization models

The basic units of computation in each language employ different modes of synchronization and data exchange. For instance, *Charisma* objects consume

values. As discussed in § 3.11, these consumption specifications are translated into message-driven code, whereby an `object` performs a specified coarse-grained, sequential computation when it has received all the messages on which that computation depends. This ability to wait for messages means that a *Charisma* `object` may synchronize with other `objects` at various points in its lifetime.

On the other hand, *DivCon* tasks receive non-array data as arguments to recursive function calls. These arguments may include references to distributed *DivconArrays*. In Chapter 4, we saw that the data within such *DivconArrays* is generated by the `+=` operator embedded within `foreach` invocations on other *DivconArrays*. Here, we must develop a means to load such arrays from external code, and thereafter pass them into *DivCon*. As concerns synchronization, *DivCon* tasks cannot synchronize with each other unless they share a parent-child relation. Even then, a parent task may only (asynchronously) spawn a child, and implicitly synchronize with it when it reads the results generated by the child task. Similarly, if a child invocation receives as arguments values generated by the parent, its spawn is delayed until those values are ready.

Finally, the *Distree* framework expects tree data elements (e.g. particles) to be submitted on a per-processor basis, following decomposition. It then constructs a tree based on these decomposed data elements. Thereafter, concurrent traversals over the tree access local and remote tree data. These traversals are independent, in that they do not synchronize with each other at all. Indeed, a traversal may only be instantiated, and thereafter iterates over tree nodes and data elements. The order in which the iteration takes place is guided by the behavior of the visitor associated with the traversal. The lifetime of an individual, fine-grained traversal is implied by the behavior of its visitor as it processes nodes and data elements in the tree.

In order to avoid the pairwise interoperation problem here, we require that each language expose an interface through which it receives data, *and* that it may be invoked from *every* other language. For the particular case of *Charisma*, we also require a means to explicitly transfer control into and out of *Charisma* code.

6.3 Mechanisms for interoperation

We avoid the the pairwise interoperation problem by using Charm++ as the common denominator for our specialized languages. That is, each language provides a Charm++/C++ interface for instantiation and data communication. Each language also has *sequential components* from which such Charm++ code can be invoked. Therefore, if a module m_1, written in one language, is to instantiate or communicate data to module m_2, possibly written in a different language, it does so by invoking the Charm++/C++ bindings for m_2. The invocations themselves are a part of the sequential components of m_1.

Next, we describe the techniques developed in this thesis to facilitate interoperation between *Charisma*, *DivCon*, *Distree* and Charm++. As we have hinted previously, three capabilities are required to enable interoperation between any set of languages: First, the external code must be able to *instantiate* a new module uniformly, i.e. independently of the language in which the instantiating module is written. Second, the external code must be able to *communicate data* to the basic computational units of a module, and thereafter be able to extract results from it. Finally, in cases where it is feasible to do so, external code must be able to *synchronize* with the computational units of modules.

In the following subsection, we develop these ideas more fully. First, we provide C++ bindings for module instantiation and initiation. These bindings are only required for *Charisma* and *DivCon*, since *Distree* is itself a Charm++-based framework, and by extension a C++ framework. Additionally, we expose C++ bindings for *DivconArrays*, thereby allowing user code to initialize and operate upon these arrays, externally to *DivCon*. Similar arrangements are made to expose *Charisma* values as C++ objects to Charm++ code. As a result, message-encapsulated data can flow between *Charisma* modules and external code. In this way, Charm++ code can instantiate, initiate and communicate with modules written in the specialized languages.

6.3.1 Initiating a new module

Charisma

A *Charisma* module that can be instantiated from external (Charm++) code must be declared as a `module` instead of a `program`. This causes the *Charisma* compiler to generate a C++ *descriptor* class (*cf.* § 3.13) that encapsulates the *Charisma* module. The descriptor contains attributes corresponding to *Charisma* `param`s and handles to the `object` collections present within the *Charisma* module. As detailed in § 3.13, the descriptor also has factory methods for instantiating and initiating execution of the module. A callback is provided in each of these factory methods, to signal completion of module initialization, and module execution, respectively. An example demonstrating the incorporation of *Charisma* modules into Charm++ code has already appeared in § 3.13.

The above scheme allows *Charisma* modules to be incorporated into external, Charm++ code. But what about Charm++ modules that are to be incorporated into *Charisma* code? Suppose that we want to incorporate a Charm++ module named `MyCharmModule` into a *Charisma* `program` called `MyCharismaPgm`. For simplicity, assume that each module contains a single object. Suppose that the singleton within the Charm++ object is of type `Foo`, and has the name `foo`. We would like to invoke a method named `two` on the Charm++ object from *Charisma* orchestration code. Let us see what this entails.

For each externally defined module that is to be incorporated into *Charisma* code (which could itself be a `program` or a `module`), the user must:

1. Declare the Charm++ module as an `extern module`. This declaration serves to inform the *Charisma* compiler of symbols from the external module that are used in the *Charisma* code.

   For our example, here is the complete *Charisma* orchestration code. It includes the `extern module` declaration to inform the compiler of externally defined symbols:

```
program MyCharismaPgm;

extern class Foo;
extern module MyCharmModule
{
  object foo : Foo;
} moduleInstance;

orchestrate
{
  // can refer to 'moduleInstance.foo' here
}
```

A single instance of the external `MyCharmModule` is defined in the program, namely `moduleInstance`. We can then refer to the object `foo` within it as `moduleInstance.foo`.

2. Define a C++ module container class, similar to the kind automatically generated for *Charisma* modules. A descriptor class embedded within the module container must at least define as attributes proxies to the collections of objects (*chares*) to which the *Charisma* code makes a reference. If the *Charisma* code references global, read-only variables within the Charm++ module, these must also appear as attributes of the descriptor.

   For our example, we define the following module container and (embedded) descriptor class:

```
struct MyCharmModule
{
  struct Descriptor
  {
    CProxy_Foo foo;
    Descriptor(const CProxy &f) :
      foo(f)
    {}
  };

  static Descriptor instantiate()
  {
    return Descriptor(CProxy_Foo::ckNew());
  }
};
```

The `instantiate` method of the module allows generated *Charisma* code to obtain an instance of `MyCharmModule`, and bind it to the orchestration name `moduleInstance`.

DivCon

In § 4.7, we saw that *DivCon* provided C++ bindings for the invocation of the root task of a *DivCon* computation from external code. To recapitulate, the `Divcon::Spawn` function initiates a recursive *DivCon* computation, and accepts three sets of arguments: (i) configuration parameters for the *DivCon* runtime system, (ii) an object instance that encapsulates the arguments to the root function invocation, and (iii) a callback that is invoked when the computation finishes. The result of the computation is embedded within a message passed to the callback.

The invocation of external modules from *DivCon* can be achieved through `serial` *DivCon* functions. Since these are plain C++ functions, we can make appropriate function calls to, for instance, initiate another *DivCon* computation within an existing *DivCon* computation.

Distree

Finally, § 5.8.1 presented code for the initiation of traversals over tree data in the *Distree* framework. Recall that this is a simple, three-step procedure: (i) obtain a reference from *Distree* to a walk object of the appropriate type, e.g. a top-down walk for *Barnes-Hut*; (ii) create a tree visitor to process the nodes and data elements of the tree, e.g. for the *Barnes-Hut* example, a visitor that computes the gravitational interaction between particles and nodes; and finally, (iii) initiate the traversal by invoking the `walk()` method on the walk object, passing into the call the node at which to begin the traversal, and a reference to the visitor object. Since *Distree* is a Charm++ framework, we do not require any special mechanisms to initiate *Distree* traversals. A few lines of C++ suffice.

6.3.2 Communicating data, and synchronizing

We now address the question of how external code can synchronize, and exchange data with the basic computational units of the three specialized languages.

Charisma

Collections of `objects` in *Charisma* publish and consume `values`. Recall that the sequential, C++ methods of *Charisma* objects receive consumed values encapsulated in `ConsumedValue`'s. They bind results to published values by invoking the `produce`/`reduce` methods on `ProducedValue`'s. This scheme can be extended to interface with methods invoked on coarse-grained, Charm++ objects (which are extraneous to *Charisma*). First, notice that *Charisma* implicitly expects user-defined sequential methods to publish results (if the method is declared as having any) *before* control returns from the method invocation.

However the situation is different when *Charisma* orchestration code invokes a method on an object that is defined in an external module. To wit, we cannot assume that the values published by that method will be ready when we return from its invocation. Indeed, when such a generic Charm++ method is invoked on an extra-*Charisma* object, it might asynchronously

spawn new computations. The result of this computation might become available to the object only much later, via a callback.

Let us return to our simple example, wherein `MyCharismaPgm` referenced an external, Charm++ module called `MyCharmModule`. The complete orchestration code for `MyCharismaPgm` follows:

```
program MyCharismaPgm;

extern class Foo;
extern module MyCharmModule
{
  object foo : Foo;
} moduleInstance;

class Bar;
object bar : Bar;
value p,q : int;
orchestrate
{
  (p) <- bar.one();
  (q) <- foo.two(p);
         bar.three(q);
}
```

The *Charisma* code defines a single `object`, `bar` of type `Bar`, and a `value` that enables communication between `bar` and `foo`. The orchestration code dictates the following sequence of events: sequential method `Bar::one` is invoked on `bar`, leading to the production of `p`. This produced value is consumed by `foo` via `Foo::two`. This method publishes `q`. Finally, `bar` consumes the value `q` through `Bar::three`. We note that `foo` being an externally defined object, its method `Foo::two` may not publish `p` before returning. For instance, `Foo::two` could invoke another parallel library, whose eventual result is to be bound to the value `p` that it publishes:

```
// This is a Charm++ method,
// external to Charisma and invoked by it.
void Foo::two(const Charisma::ConsumedValue< int > &p,
              Charisma::ProducedValue< int > &q)
{
  myLibrary->Invoke(*p,
                    CkCallback(CkIndex_Foo::result(NULL),
                               thisProxy));
  // returning without publishing q!
}

void Foo::result(ResultMsg *msg)
{
  int libraryResult = msg->result;
  // want to publish 'libraryResult' to
  // 'q' at this point.
}
```

To allow for cases such as the one above, we must provide a way for external code to delay returning control to *Charisma* until it has actually bound results to the values published by it. That is, control should not be transferred back to *Charisma* implicitly, but rather explicitly, as dictated by external code. To do this, we introduce the C++ class `Charisma::Token`. External code can explicitly transfer control back to *Charisma* by invoking the `Charisma::Token::Advance` method on an object of this type. Returning to our illustration, the following rewrite of the above code would serve our purpose:

```
void Foo::two(const Charisma::ConsumedValue< int > &p,
              Charisma::ProducedValue< int > &q,
              Charisma::Token &token)
{
  myLibrary->Invoke(*p,
                    CkCallback(CkIndex_Foo::result(NULL),
                               thisProxy));
  // returning without publishing q!
  // save token, so that we can explicitly
  // transfer control back to Charisma when ready.
  m_Token = token;
  m_Q = q;
}

void Foo::result(ResultMsg *msg)
{
  int libraryResult = msg->result;
  // want to publish 'libraryResult' to
  // 'q' at this point.
  m_Q.produce(libraryResult);
  m_Token.Advance();
}
```

Notice the appearance of the `Charisma::Token` as the final argument to the `Foo::two` method. We now initiate the parallel library as before, and expect control to return to `foo` in its `Foo::result` method (as dictated by the callback argument to the library invocation). We save the `token` and `ProducedValue` container in instance attributes `Foo::m_Token` and `Foo::m_Q`, respectively. This allows us to `produce` the saved `q` value only when we have the appropriate result (`libraryResult`) to bind to it. Following this, we return control to *Charisma* by invoking the `Advance` method on the previously saved token.

We present a concrete example of the inclusion of external, Charm++ code into a *Charisma* module in § 6.4.

DivCon

In § 4.7 we covered techniques by which to pass arguments (both scalars and references to *DivconArrays*) to the root invocation of a recursive *DivCon* function. Here we only recall that compiler-generated argument wrappers are used for this purpose. The *DivCon* function that spawns such a computation also expects a callback argument, which dictates the point at which control re-enters external code, having finished the *DivCon* computation. The result of the computation is available to this callback in the form of a Charm++ message.

We have noted previously that *DivCon* supports a very limited form of synchronization between tasks, namely that a parent may asynchronously spawn children, and synchronize with them only to receive their results, at which point the children no longer exist. Therefore, it is not possible to synchronize with tasks from extra-module code. However, external code may asynchronously spawn the root task, and synchronize with its completion in an associated callback.

Distree

In *Distree*, tree data is held in a collection of coarse-grained objects that serve as containers, called *tree pieces*. Typically, data elements are loaded into tree pieces from an input file. Since tree pieces are Charm++ *chares*, extra-module code may remotely invoke entry methods on them. Thereafter, as outlined in § 5.5.3, they are deposited into the *Distree* framework. The framework then performs data decomposition, tree building and consolidation automatically. § 5.8.1 shows how, thereafter, traversals can be instantiated using a C++ API. Once initiated, these traversals execute to completion, based on the availability of requested tree data. It is not possible to synchronize with these fine-grained tasks from external code. In fact there is no explicit provision for traversals to communicate/synchronize even amongst themselves.

6.4 An example of interoperation

So far, we have discussed the basic requirements, and compiler support provided to enable interoperability between the languages described in this thesis. We now give a concrete example of this interoperability. In this section, we will construct a simple, parallel simulator of particles moving under the force of gravity. We will base this example on the detailed exposition of the *Barnes-Hut* algorithm given in § 5.3. Broadly speaking, the application is composed of the following components:

1. A *Charisma* module that expresses the data-independent flow of control and data in the algorithm. The parallel invocation of sequential methods on tree pieces in such phases as particle loading, local tree building, and trajectory integration, is especially well-suited to expression in *Charisma*.

2. A *DivCon* Oct-decomposition module, identical to the code seen in § 4.5.2.

3. A *Distree* module that performs tree building, consolidation and traversal, as discussed in § 5.3.

4. A Charm++ module that serves to *glue* together the above three modules. This module instantiates the *Charisma* module, and thereafter serves as a conduit between the code encapsulating overall control flow (in *Charisma*) and the *DivCon* and *Distree* modules that are invoked to implement particle decomposition and tree traversal, respectively.

6.4.1 Structure of the multi-paradigm *Barnes-Hut* application

In this subsection, we take a closer look at the structure of the multi-paradigm *Barnes-Hut* application. We begin with its overall control flow, which is expressed in *Charisma*. The preamble tells us the module's constituent collections of `objects`, and the `values` that the publish and consume in order to communicate.

```
module BhControlFlow;

class Dummy;
class BoundingBox, TraversalData;

value box : BoundingBox;
value d : Dummy;
value stats : TraversalData;

class Glue, TreePiece;
class Config;

extern module GlueModule
{
  param config : Config;
  object glue : Glue;
  objects pieces : TreePiece[];
} Bh;
```

The control flow module is named `BhControlFlow`. We declare a few externally defined classes, named `BoundingBox`, `TraversalData` and `Dummy`. These classes are used to define individual `value`s, which in turn help to describe the flow of data in the application. The `value` named `box`, of type `BoundingBox`, encapsulates the physical extent of the simulation volume, and is used to assign integer keys to particles based on their positions. The `value` named `stats`, of type `TraversalData`, is used to communicate aggregated statistics about the tree traversal. Finally, `d` is used to express synchronization between different `objects` within the application. The `Dummy` class doesn't contain any useful data. It is solely used as a means to explicitly specify an ordering on methods invoked within the *Charisma* code. These `value`s are discussed more concretely following the declaration of the external module `GlueModule`.

`GlueModule` encapsulates entities that are not defined within *Charisma* code, but are nonetheless referenced within it, for the purpose of orchestration. For our purpose, we include as its members a singleton `object`, `glue`, of type `Glue`, a collection of `TreePiece objects` `pieces`, and finally, `config`, a structure encapsulating several simulation-related parameters. These symbols are all defined and instantiated externally. (As we will see later, they are

instantiated in the Charm++ piece of this code.) We create an instance of this `extern module` by suffixing it with the identifier `Bh`. This identifier refers to an instance of the external module in the rest of the orchestration code. As mentioned earlier, the idea behind declaring externally defined (e.g. in Charm++) objects is that they can perform tasks that cannot be expressed in *Charisma*, e.g. tree traversal.

Let us now consider the main `orchestration` block of the module. We have broken up this code into several pieces, so as to facilitate a simpler exposition. In the first part of this code, we initialize the *Distree* framework, by invoking the `InitDistree` method on object `glue` of the external module `Bh`:

```
orchestration{
  Bh.glue.InitDistree();
  ...
}
```

The `Glue` class (defined in C++) is used to instantiate a singleton Charm++ object. It defines `InitDistree()` as a remote method that, when invoked, runs to completion. The body of this method initializes the *Distree* framework, as shown below:

```
void Glue::InitDistree(const Charisma::Token &token){
  m_Token = token;

  // CREATE TREE CONTAINER
  m_TreeHandle = MyTreeHandleType::
                  instantiate(m_GlueModule.pieces,
                              m_Config.cacheChunkDepth);

  // CREATE TRAVERSAL MANAGER
  CacheAccessType ro = CacheAccessType::GetReadonly();
  m_GravTravHandle = GravityVisitor::
                      TraversalType::instantiate(ro, ro);

  // CREATE CACHES
  NodeCacheClientsType nodeCacheClients;
  nodeCacheClients.add(m_GravTravHandle);
  MyGravityCacheManagerType::Handle nodeCacheHandle =
    MyGravityCacheManagerType::instantiate(0, nodeCacheClients);

  LeafCacheClientsType partCacheClients;
  partCacheClients.add(m_GravTravHandle);
  MyGravityCacheManagerType::Handle particleCacheHandle =
    MyGravityCacheManagerType::instantiate(0, partCacheClients);

  // SETUP CACHES FOR TRAVERSAL
  m_GravTravHandle.setNodeCache(nodeCacheHandle.proxy());
  m_GravTravHandle.setLeafCache(particleCacheHandle.proxy());

  CkCallback callback(CkIndex_Glue::InitializeGravityTraversal(),
                      thisProxy);
  // initialize tree: control will return to this object's
  // InitializeGravityTraversal method after tree initialization.
  m_TreeHandle.initialize(callback);
}
```

The `Glue::InitDistree` method accepts a single argument, in the form of the previously discussed `Token` class, which facilitates explicit control transfer between *Charisma* and external code. In this context, its purpose is to allow

the called function to inform *Charisma* when it is safe to execute to the next orchestration statement of the `Bh.glue` object. To recapitulate, this explicit synchronization is required because from the point of view of *Charisma*, methods invoked on externally defined `objects` behave differently from those invoked on `objects` defined within *Charisma*. Specifically, control returns immediately to the orchestration code upon exiting a sequential method of a *Charisma* `object`.

On the other hand, external objects are message driven, and the invocation of a method f on such an object might trigger parallel work. In general, the results of that work will *not* available immediately when we return from f. Therefore, external code must explicitly pass control back to *Charisma* through `token`.

The body of the `Glue::InitDistree` method sets up the tree container, traversal and node/leaf caches that will be required for the *Distree* part of the computation. Thereafter, we create a callback, which is invoked by the *Distree* framework when it has finished its initialization. The actual initialization of the framework is triggered by invoking the `initialize` method on the previously created tree container's handle.

Eventually, *Distree* invokes the callback given to it in the `initialize()` call. This directs control to the `Bh.glue` object's `InitializeGravityTraversal` method. This method in turn triggers the initialization of *Distree*'s top-down traversal manager on all processors. Once again, control leaves the `Bh.glue` object for the *Distree* framework, to finally return to it in the `Glue::Advance()` method:

```
void Glue::Advance()
{
  m_Token.Advance();
}
```

The `Glue::Advance` method returns control to *Charisma* following the successful initialization of the *Distree* framework. This is done by calling the `Advance` method on the token that was previously saved in `Bh.glue`'s `Glue::InitDistree` method. Compiler-generated code then transfers control to the `foreach` statement following the `InitDistree()` invocation on `Bh.glue`:

```
orchestration{
  Bh.glue.InitDistree();

  foreach(x in ispace(Bh.pieces))
    (+box) <- Bh.pieces[x].Load(Bh.config);

  for(I = 1 : Bh.config.NumIterations){
    ...
  }
}
```

Each `TreePiece` in the collection `Bh.pieces` then `Load`s its share of particles, using the filename embedded within the `Config` `struct`, `Bh.config`. The code for the `TreePiece::Load` method is shown below:

```
void TreePiece::Load
(const Charisma::ConsumedValue< Config > &config,
 Charisma::ReducedValue< BoundingBox > &box,
 Charisma::Token &token)
{
  loadTipsy(config);
  box.reduce(m_Box);
  token.Advance();
}
```

As is the case with sequential methods of objects internal to *Charisma*, the `Load` method invoked on external `Bh.pieces` `objects` receives *Charisma* values as arguments. The first one, `config` is a value consumed by the method, and the second one, a value that is published (reduced) by the method. Since this is a method of an object external to *Charisma*, it must also accept a token object for explicit return of control to *Charisma*.

The `Load` method in turn calls a serial function, `loadTipsy`, to read particles from an input file in the Tipsy data interchange format, and also to calculate the bounding box of the loaded particles in the member attribute `m_Box` of `TreePiece`. This bounding box is then contributed to a reduction of the *Charisma* value `box`. Finally, control is transferred back to *Charisma* by invoking the `Advance` method on the token argument of the method. This Charm++ method `reduce`s data to a *Charisma* value just like a *Charisma* object's sequential method might have.

We re-enter the *Charisma* orchestration code, and now the tree pieces assign integer keys to their particles based on their positions through invocations of the `AssignKeys` method.

```
orchestration{
  ...
  foreach(x in ispace(Bh.pieces))
    (+box) <- Bh.pieces[x].Load(Bh.config);

  for(I = 1 : Bh.config.NumIterations){
    foreach(x in ispace(Bh.pieces))
      (+d) <- Bh.pieces[x].AssignKeys(box);

    (d) <- Bh.glue.OctDecompose(d);
    ...
  }
}
```

Here the `Dummy value`, `d` makes its first appearance. This reduced value contains no data, but is required to inform the *Charisma* compiler of the ordering between the invocation of `AssignKeys` on members of `Bh.pieces`, and the invocation of `OctDecompose` on `Bh.glue`. Therefore, it is guaranteed that all `AssignKeys` invocations will have completed before `OctDecompose` is invoked. Had we neglected to specify this dependency to the *Charisma* compiler, it would have treated these invocations as concurrent.

The `OctDecompose` method of the `glue` object initiates the decomposition of particles in *DivCon*:

```
void Glue::OctDecompose(const Charisma::ConsumedValue< Dummy > &d,
                        const Charisma::Token &token);
{
  Divcon::Task::Config tasks;
  tasks.QueueingPolicy = LIFO;

  Divcon::Array::Config arrays;
  arrays.BalancingSplitPartitions = 4;
  arrays.MaxNumAgglomeratedOperations = 100;

  Divcon::Spawn(tasks,
                arrays,
                Oct(Divcon::Array::Descriptor(m_GlueModule.pieces,
                                              m_Config.nPieces),
                    Key(1)),
                CkCallback(CkIndex_Glue::DoneDecompose(),
                           thisProxy));

  m_Dummy = d;
  m_Token = token;
}
```

The body of this method uses the C++ interface of *DivCon* to `Spawn` the root of the `Oct` decomposition recursion tree. The *DivCon* static function `Spawn` takes four arguments as input: the first two are configuration `structs` for the task-parallel and data-parallel frameworks of *DivCon*. Here, we set the queueing strategy for newly spawned tasks to LIFO, and also set the parameters for delayed redistribution and operation agglomeration in *DivconArrays*. The third argument is an anonymous instance of the compiler-generated `struct Oct`, which encapsulates the arguments of the *DivCon* function `Oct` (*cf.* § 4.5.2). The last argument to the `Spawn` function is a callback object to be invoked when the *DivCon* computation has completed. As with the `Glue::InitDistree` method, we save the *Charisma*-provided token for later, to be invoked in the target of the completion callback (namely, `Glue::DoneDecompose`).

This takes us back to *Charisma* orchestration code, wherein we performs several *Distree*-related tasks: tree building, (`TreePiece::BuildTree`), tree piece consolidation, (`Glue::MergeTrees`) and synchronization of traversal man-

agers (`Glue::SyncTraversal`). Thereafter, we compute the gravitational interactions by instantiating *Distree* traversals in `TreePiece::Gravity`. Again, control is transferred back to *Charisma* explicitly by the Charm++ glue when all gravity traversals have finished. We end the iteration by clearing *Distree* data structures (`Glue::DoneTraversal` and `Glue::DeleteTree`), and updating particle trajectories (`TreePiece::Advance`).

```
orchestration{
  ...
  for(I = 1 : Bh.config.NumIterations){
    ...
    foreach(x in ispace(Bh.pieces))
      (+t) <- Bh.pieces[x].BuildTree(t);

    Bh.glue.MergeTrees(t);
    (t) <- Bh.glue.SyncTraversal();

    foreach(x in ispace(Bh.pieces))
      (+stats) <- Bh.pieces[x].Gravity(t);

    Bh.glue.DoneTraversal(stats);
    Bh.glue.DeleteTree();

    foreach(x in ispace(Bh.pieces))
      (+box) <- Bh.pieces[x].Advance();
  }
}
```

We omit the code for these methods, since it is nearly identical to the code presented in § 5.8.1.

Recall that the code above was declared as a *Charisma* `module`, instead of a `program`, implying that control originates in some other, external piece of code. It is the responsibility of that external code to instantiate this *Charisma* module. The instantiating code invokes a compiler-generated factory method, and passes into this method a C++ `struct` of type `GlueModule`. This `struct` *must* have attributes named `config`, `glue` and `pieces`. (We will discuss the types of these attributes shortly.) The instantiating code must also set the values of these attributes appropriately. We briefly discuss these

issues next. First, consider the definition of the `GlueModule` struct, which enables *Charisma* code to refer to coarse-grained, Charm++ objects external to it.

```
struct GlueModule
{
  Config config;
  CProxy_Glue glue;
  CProxy_TreePiece pieces;
};
```

`GlueModule` is a C++ structure that contains attributes declared as extern to our *Charisma* module. It has a `config` attribute to match the associated `param` in *Charisma*. Furthermore, it encapsulates *proxies* (i.e. handles that are valid references to Charm++ objects across address spaces) to Charm++ objects that were declared as external `object(s)` in our *Charisma* module. Finally, we must address the question of how this entire computation is initiated. The constructor of the `Glue` singleton object is declared (in Charm++) to be the entry point of execution for the application. Here, we have the following code:

```
Glue::Glue(CkArgMsg *msg)
{
  // INITIALIZE SIMULATION PARAMETERS USING
  // COMMAND-LINE PARAMETERS from CkArgMsg
  m_Config = InitializeParameters(msg->argc, msg->argv);
  ...

  GlueModule Bh;
  Bh.config = m_Config;
  Bh.glue = thisProxy;
  Bh.pieces = CProxy_TreePiece::ckNew(m_Config.nPieces);

  Module_BhControlFlow::Descriptor descriptor;
  descriptor = Module_BhControlFlow::instantiate(Bh, ...);
  descriptor.start(...);
}
```

First, we initialize `m_Config`, a `struct` of type `Config`, using command-line parameters. (As seen in the definition of the `Glue::OctDecompose` method

earlier, the attributes of this `struct` are used to instantiate the *DivCon* module as well.) Next, we construct a `GlueModule struct` that can be used by the *Charisma* code to refer to external entities. The proxy to the `glue` object is set to a special, Charm++-provided variable called `thisProxy` (akin to the `this` symbol in C++, but its value is valid across address spaces). Additionally, we create a collection of `TreePiece` objects using a Charm++-provided factory method.

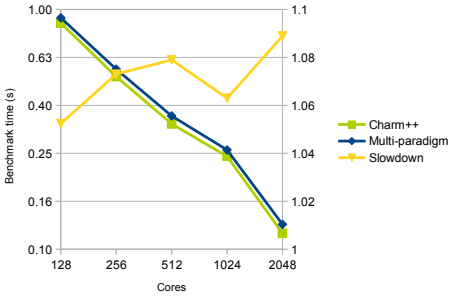
After that, we construct an instance of a `Descriptor` class that encapsulates the *Charisma* module. The definition of this class is generated by the *Charisma* compiler. The variable `descriptor` of this `struct` represents an instance of the *Charisma* module `BhControlFlow`. Notice that the factory method used to instantiate the module takes as input a parameter named `Bh`, of type `GlueModule`. This matches the `extern module` declaration in our orchestration code. In this way, `objects` within *Charisma* gain access to Charm++ objects defined externally. Finally, we initiate the orchestration code by invoking the `start` method on the module descriptor. For simplicity, we have elided the callback arguments to the `instantiate` and `start` methods.

In this way, we are able to express the *Barnes-Hut* algorithm as a tightly coupled, multi-paradigm composition of code expressed in four different languages. The global control flow of the overall algorithm is captured in *Charisma*. Through various techniques developed in this chapter and § 3, the *Charisma* language is accessible to, and is able to access external Charm++ modules. A Charm++ "glue" module initiates computations expressed in the two other specialized languages developed in this thesis, namely *DivCon* and *Distree*. Recursive, geometric decomposition of a large, distributed set of particles onto Charm++ objects called *tree pieces* is done in *DivCon*. The highly irregular and data-dependent traversal of the distributed *Barnes-Hut* tree is expressed succinctly through *Distree* components.

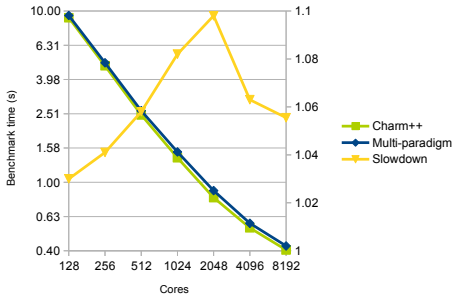
6.5 Performance results

Figure 6.1 compares the performance of our multi-paradigm *Barnes-Hut* code with its Charm++ counterpart. The Charm++ code was a part of the first place entry to the HPC challenge [26], and significant effort has been ex-

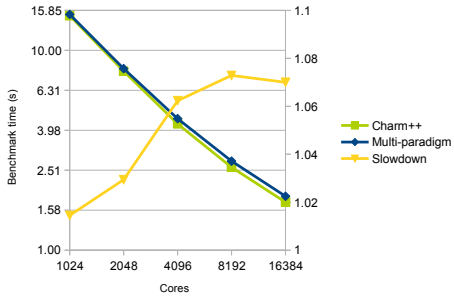
pended in tuning that code. Indeed, this is the same code to which we compared the *Distree Barnes-Hut* implementation in § 5.9.1. Here we compare the performance of the Charm++ version and the multiparadigm version on three synthetic, fairly uniform distributions of particles.



(a) 1 million particles.



(b) 10 million particles.



(c) 100 million particles.

Figure 6.1: Comparing the performance of a hand-written, and well-tuned Charm++ code for the Barnes-Hut benchmark, and its multi-paradigm counterpart, written in a mixture of Charm++, *Charisma*, *DivCon*, and *Distree*.

Our tests were conducted on three data sets, of size 1 million, 10 million and 100 million particles. Both the Charm++ and multi-paradigm versions scaled to 2048 cores of Blue Gene/Q on the smallest, 1 million particle benchmark. Although the multi-paradigm version is slower by 5-10 per cent, the performance of the two versions remains comparable on all processor counts. Similar results are seen with the 10 million particle benchmark on up to 8192 cores. For the largest, 100 million particle data set the difference in performance between the two versions is in the range of 4-8 per cent. On this last data set, it appears that the multi-paradigm version has slightly poorer scalability than the Charm++ version: as we approach the 16384 processor

mark, the gap between the two versions grows.

6.6 Productivity

Overall, our performance results on the *Barnes-Hut* benchmark suggest that performance of the multi-paradigm code is competitive with its hand-tuned, Charm++ counterpart. Let us now look at how the two approaches compare in terms of effort expended in writing code. Once more, we use SLOC to gain a rough estimate of programming effort.

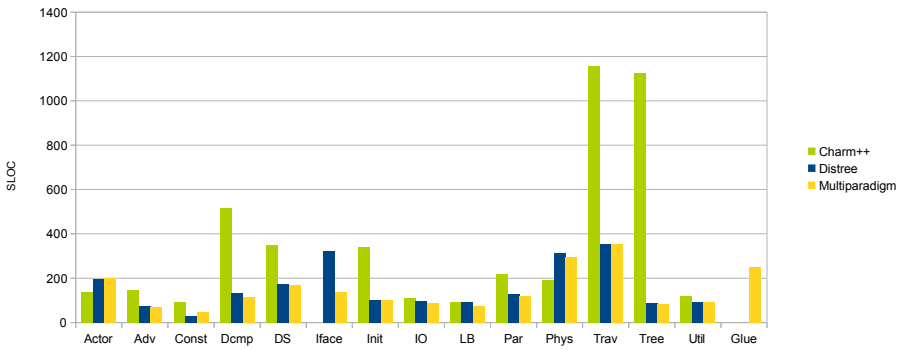


Figure 6.2: A comparison of the break-up of SLOC into different categories for three versions of the *Barnes-Hut* benchmark. This is the same figure as Figure 5.15, but with the addition of data for the multi-paradigm code discussed in § 6.4.

Figure 6.2 compares three versions of the *Barnes-Hut* benchmark: the previously published one written in Charm++ [26]; the *Distree* version described in § 5; and finally, the multi-paradigm code developed in this chapter. The multi-paradigm version borrows heavily from the *Distree* version, so it is unsurprising that the two codes comprise nearly identical numbers of SLOC (2199 vs 2204 lines, respectively; for reference the Charm++ code was 4603 lines long). In fact, the only categories in which they differ significantly are the Iface and Glue categories. (The Iface category includes all code to interface with *Distree*, and the Glue category, that needed in the multi-paradigm version to stitch together the various pieces.) Even so, the Iface and Glue categories account for 385 lines in the multi-paradigm version, which is about 60 lines more than the 323 lines of Iface code needed for the *Distree* version.

Essentially, the SLOC for the two versions has been accounted for differently. More importantly, both codes are significantly more compact than the Charm++ version. This is because much of the code related to tree construction, and book-keeping for cached, data-driven executions of traversals, is subsumed by the *Distree* framework.

Even though the *Distree* and multi-paradigm codes are evenly matched in terms of SLOC, we assert that the higher-level notations used in the latter engender greater productivity. Consider first the *Charisma* code used for orchestrating the simulation. That code captures all the global data-independent flows succinctly and in a single place. A programmer who wishes to understand the application need only look at this orchestration code to comprehend the overall control flow, and from there investigate the building blocks of the application.

Second, the *DivCon* code used for Oct-decomposition is also simple and compact, and leaves the implementation of distributed arrays of particles to the runtime system. The Oct-decomposition algorithm is readily comprehensible in its divide-and-conquer form, and the *DivCon* notation allows us to implement it succinctly and naturally. *DivCon* enforces a separation of the parallel structure of the recursive algorithm from its sequential pieces. This allows the "glue" code for interoperability with Charm++ to be pushed to the sequential methods. We believe that this mode of interoperation, albeit somewhat restrictive, is sufficiently powerful for applications that simply spawn recursive computations, and await their results, instead of attempting to synchronize with individual fine-grained tasks within the tree-structured computation.

Third, the *Distree* framework hides from the programmer much of the routine management of distributed tree data structures. In the context of the multi-paradigm *Barnes-Hut* application, the building of the distributed tree, as well as its consolidation, is handled by *Distree*. The application initiates fine-grained, data-driven traversals over the tree, and the *Distree* runtime manages their efficient execution. Importantly for the performance of applications based on distributed trees, *Distree* automates the reuse of remotely fetched data and provides a heuristic, communication-aware load balancing strategy. All of these features are leveraged in the multi-paradigm application to obtain a level of performance that is competitive with a comparable, hand-tuned application of considerably greater heft.

Finally, tight coupling of these modules was achieved through a common runtime substrate in the form of Charm++. A language such as *Charisma*, which deals in the production and consumption of values (messages) by collections of objects, has a similar programming model to the message-driven, migratable-object one of Charm++. As a result, it is possible to synchronize computations in *Charisma* modules with those in Charm++. However, the fine-grained tasks (recursive function invocations) of *DivCon*, as well as the fine-grained traversals of *DivCon*, have no explicit means of synchronizing with Charm++ objects. These tasks and traversals cannot wait for the receipt of data and control messages from external code. As a result, data transferred into these modules was put in place before tasks or traversals were ever spawned. In the case of the *DivCon* Oct-decomposition module, data to be transferred out of the module, and back to the collection of tree piece objects, was sent via Charm++ calls embedded in `serial` functions. While this serves the present purpose of this work, which is to enable interoperability, this scheme is not very elegant, and is somewhat restrictrive. An interesting extension of this work would be to introduce *futures*, *a la* X10, to this system. We believe that this would increase expressiveness, in addition to making interoperation more elegant.

CHAPTER

7

CONCLUSIONS

In this thesis, we have examined the performance and productivity aspects of a multi-paradigm approach to parallel programming for HPC. Our approach is contrary to the current movement in the community to adopt a single, general-purpose programming language that makes concessions for productivity. Instead, we advocate an approach based on the inter-related themes of *plurality*, *specialization* and *ineroperability*. We argue for a programming system that features a whole ecosystem of specialized programming languages, frameworks and libraries. In such a system, large parallel applications are constructed in a modular fashion. Each module is written in the language or framework that most naturally fits its semantic requirements. This work provides three such specialized programming systems, one addressing each of the following important subclasses of HPC application: (i) applications with data-independent data and control flows; (ii) divide-and-conquer applications, with a special emphasis on those that operate on large, distributed arrays; and (iii) algorithms based on distributed tree data structures.

We have argued that the specialization of these languages fosters productivity through abstraction. At the same time, language specialization restricts the dynamic behaviors of expressible codes. As demonstrated by

our work, the accompanying runtime systems can then make simplifying assumptions about the dynamic behaviors of programs, and therefore optimize performance. Thus, we believe that a multi-paradigm approach provides both abstraction (and therefore programming productivity) and performance.

Interoperability is a key component of this approach. Since no individual language in the system can exhaust the expressive space of parallel programs, we require interoperability across languages. Moreover, we must be able to incorporate a *recourse* language into our scheme, in order to plug the gaps in the expressive scope of the union of our specialized languages. In this thesis, we have made extensive provisions for interoperability between our specialized languages, and the general-purpose, message-driven Charm++ model. We have also demonstrated this interoperability in practice.

We composed the well-studied *Barnes-Hut* algorithm from four modules, each one of which was written in a different language. The first module encapsulated the algorithm's overall global flow of control in *Charisma orchestration* code. The recursive bisection of large, distributed arrays of particles, was expressed as a *DivCon* module. The third module leveraged the components provided by the *Distree* framework to automate distributed tree construction, as well as facilitate the succinct expression of fine-grained, data-driven traversals of the distributed *Barnes-Hut* tree. Finally, the fourth module, written in Charm++, served as the *glue* between modules written in the three specialized languages, thereby allowing them to interoperate.

We have shown that the performance of this multi-paradigm code is competitive with that of a comparable, hand-tuned Charm++ counterpart. By expressing different modules in appropriate specialized languages, we see significant reductions in code size. Finally, codes written in these specialized languages capture the essential structure of their modules' computation. For instance, the *Charisma* module expresses the global, data-independent flows of control in the application. Similarly, *DivCon* succinctly and explicitly captures the recursive nature of the Oct-decomposition algorithm used therein.

Below, we summarize the specific contributions of this thesis.

1. We have undertaken an extensive redesign of the *Charisma* language. We have added new constructs to the language, thereby leading to increased expressive scope. We have also developed a concrete semantics of its publish-consume model. [**§ 3.6 and § 3.8**]

2. We have developed a new compiler infrastructure for the *Charisma* language. We have used optimizing compiler theory to provide a robust and consistent compilation strategy for *Charisma*. Our work also provides new algorithms for dependency analysis and code generation. [§ **3.11**]

3. We provide algorithms for the efficient, dynamic maintenance of global state of a *Charisma* program. The global state is encapsulated in an entity called the GSM, which dynamically matches data dependency sources to their targets, and determines when it is safe for a dependency source to delete a value published by it. [§ **3.11.7**]

4. We have developed a technique to enable explicit transfer of control and data between *Charisma* and other Charm++-based paradigms. In our approach, *Charisma* code can incorporate external modules, and can in turn be incorporated into external code by exposing module *descriptors*. These descriptors expose the *Charisma* module's constituent `objects` and `params` to external code, and vice versa. In order to facilitate interoperation with external, message-driven code, we have developed a scheme whereby control is explicitly transferred between *Charisma* and external code. This approach allows for a *tight* coupling of *Charisma* with external code. [§ **3.13**]

5. We have developed *DivCon*, a simple language for the expression of divide-and-conquer computations. *DivCon* programs are compact and elegant, and allow the programmer to explicitly specify the computations that are performed concurrently. Importantly, *DivCon* incorporates a number of constructs for operating on distributed arrays. However, these are not general-purpose arrays, but rather admit a limited set of operations and semantics, which we believe to be well-suited to the domain of *generative* divide-and-conquer algorithms. [§ **4.4**]

6. The specialized distributed array of *DivCon* incorporates two key dynamic optimizations that enable users to write efficient, array-based divide-and-conquer algorithms on distributed memory systems. First, the *DivCon* runtime allows costly data redistribution operations to be *delayed*. This enables the amortization of communication costs over

multiple recursive invocations. Second, the *DivCon* runtime *agglomerates* operations on arrays distributed over the same partition of processors, thereby improving overall load balance, and reducing communication overhead. [§ **4.6**]

7. We have developed *Distree*, a framework for the productive and performance oriented expression of tree-based algorithms. The key abstraction provided by *Distree* is a flexible, distributed tree data structure. *Distree* provides several built-in data decomposition and tree building algorithms, and also supports more generic parallel algorithms for decomposition and tree construction. It also incorporates performance oriented features, e.g. the merging of individual tree pieces across processor cores and SMP nodes. [§ **5.5**]

8. We have developed a component-based approach to the expression of tree traversals. Our approach encourages component reuse and simplifies the expression of tree traversals. *Distree* encapsulates a (pre-existing) software cache module that fosters the sharing of remotely fetched data during traversal. Furthermore, it supports an asynchronous and fine-grained expression of tree traversals. [§ **5.6.1**]

9. We have extended the software caching mechanism of Charm++ to be *SMP-aware*. This allows data to be reused across all the cores of an SMP node, instead of just across all the coarse-grained objects resident on a processor core. We provide two techniques to achieve this node-wide sharing of tree data, namely *single fetcher* and *first-touch fetching*. [§ **5.7.2**]

REFERENCES

[1] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[2] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* New York, NY, USA: ACM, 2005, pp. 519–538.

[4] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, August 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286120.1286123

[5] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin, "A new vision for coarray fortran," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, ser. PGAS '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1809961.1809969 pp. 5:1–5:9.

[6] T. S. Tarek El-Ghazawi, William Carlson and K. Yelick, *UPC: Distributed Shared Memory Programming.* John Wiley & Sons, Inc., Hoboken, NJ, USA, 2005.

[7] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, "NAMD: Biomolecular simulation on thousands of processors," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, MD, September 2002, pp. 1–18.

[8] F. Gioachin, A. Sharma, S. Chakravorty, C. Mendes, L. V. Kale, and T. R. Quinn, "Scalable cosmology simulations on parallel machines," in *VECPAR 2006, LNCS 4395, pp. 476-489*, 2007.

[9] E. Bohm, A. Bhatele, L. V. Kale, M. E. Tuckerman, S. Kumar, J. A. Gunnels, and G. J. Martyna, "Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L," *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, vol. 52, no. 1/2, pp. 159–174, 2008.

[10] A. Langer, R. Venkataraman, U. Palekar, and L. Kale, "Parallel branch-and-bound for two-stage stochastic integer optimization," in *High Performance Computing (HiPC), 2013 20th International Conference on*, 2013.

[11] K. Asanovic, R. Bodík, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. A. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. A. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[12] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005. [Online]. Available: http://doi.acm.org/10.1145/1118890.1118892

[13] G. Smolka, "The Oz programming model," in *Computer Science Today*, ser. Lecture Notes in Computer Science, vol. 1000, J. van Leeuwen, Ed. Berlin: Springer-Verlag, 1995, pp. 324–343.

[14] P. V. Roy and S. Haridi, "Mozart: A programming system for agent applications," in *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, Nov. 1999, part of International Conference on Logic Programming (ICLP 99).

[15] K. Brown, A. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, 2011, pp. 89–100.

[16] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun, "Optiml: An implicitly parallel domain-specific language for machine learning," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ser. ICML '11, L. Getoor and T. Scheffer, Eds. New York, NY, USA: ACM, June 2011, pp. 609–616.

[17] P. Jetley and L. V. Kale, "Optimizations for message driven applications on multicore architectures," in *18th annual IEEE International Conference on High Performance Computing (HiPC 2011)*, December 2011.

[18] C. Mei, G. Zheng, F. Gioachin, and L. V. Kalé, "Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study," in *TeraGrid'10*, no. 10-13, Pittsburgh, PA, USA, August 2010.

[19] D. M. Kunzman and L. V. Kalé, "Towards a framework for abstracting accelerators in parallel applications: experience with cell," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.

[20] L. V. Kale, D. M. Kunzman, and L. Wesolowski, "Accelerator Support in the Charm++ Parallel Programming Model," in *Scientific Computing with Multicore and Accelerators*, J. Kurzak, D. A. Bader, and J. Dongarra, Eds. CRC Press, Taylor & Francis Group, 2011, pp. 393–412.

[21] D. Kunzman, "Runtime support for object-based message-driven parallel applications on heterogeneous clusters," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2012, http://charm.cs.uiuc.edu/media/12-45/.

[22] S. Kumar and L. V. Kale, "Scaling collective multicast on fat-tree networks," in *ICPADS*, Newport Beach, CA, July 2004.

[23] Y. Sun, G. Zheng, L. V. Kale, T. R. Jones, and R. Olson, "A uGNI-based Asynchronous Message-driven Runtime System
for Cray Supercomputers with Gemini Interconnect," in *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012.

[24] Y. S. Sameer Kumar and L. V. Kale, "Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q," in *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, USA, May 2013.

[25] L. V. Kalé and W. Shu, "The Chare Kernel base language: Preliminary performance results," in *Proceedings of the 1989 International Conference on Parallel Processing*, St. Charles, IL, August 1989, pp. 118–121.

[26] L. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataraman, L. Wesolowski, and G. Zheng, "Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge," Parallel Programming Laboratory, Tech. Rep. 11-49, November 2011.

[27] L. V. Kale, S. Kumar, and K. Vardarajan, "A Framework for Collective Personalized Communication," in *Proceedings of IPDPS'03*, Nice, France, April 2003.

[28] I. Dooley, "Intelligent runtime tuning of parallel applications with control points," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2010, http://charm.cs.uiuc.edu/papers/DooleyPhDThesis10.shtml.

[29] A. B. Sinha, L. V. Kale, and B. Ramkumar, "A dynamic and adaptive quiescence detection algorithm," Parallel Programming Laboratory, Department of Computer Scie nce , University of Illinois, Urbana-Champaign, Tech. Rep. 93-11, 1993.

[30] S. Chakravorty, C. L. Mendes, and L. V. Kalé, "Proactive fault tolerance in mpi applications via task migration." in *HiPC*, ser. Lecture Notes in Computer Science, vol. 4297. Springer, 2006, pp. 485–496.

[31] S. Chakravorty and L. V. Kale, "A fault tolerance protocol with fast fault recovery," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.

[32] 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.

[33] S. Kumar, C. Huang, G. Zheng, E. Bohm, A. Bhatele, J. C. Phillips, H. Yu, and L. V. Kalé, "Scalable Molecular Dynamics with NAMD on Blue Gene/L," *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, vol. 52, no. 1/2, pp. 177–187, 2008.

[34] E. Bohm, S. Chakravorty, P. Jetley, A. Bhatele, and L. V. Kale, "CkDirect: Unsynchronized One-Sided Communication in a Message-Driven Paradigm ," in *Proceedings of International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, August 2009.

[35] L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon, "Converse: An Interoperable Framework for Parallel Programming," in *Proceedings of the 10th International Parallel Processing Symposium*, April 1996, pp. 212–217.

[36] J. Lifflander, P. Miller, and L. Kale, "Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection," in *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*, February 2013.

[37] E. Meneses, G. Bronevetsky, and L. V. Kale, "Evaluation of simple causal message logging for large-scale fault tolerant hpc systems," in *16th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems in 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011).*, May 2011.

[38] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

[39] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms.* Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994.

[40] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A multigrid tutorial (2nd ed.).* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000.

[41] J. A. Board, L. V. Kalé, K. Schulten, R. Skeel, and T. Schlick, "Modeling biomolecules: Larger scales, longer durations," *IEEE Computational Science and Engineering*, vol. 1, no. 4, 1994.

[42] J. E. Barnes and P. Hut, "A hierarchical O(NlogN) force calculation algorithm," *Nature*, vol. 324, 1986.

[43] J. C. Hardwick, "An efficient implementation of nested data parallelism for irregular divide-and-conquer algorithms," in *Proceedings of the First International Workshop on High-Level Programming Models and Supportive Environments*, April 1996, pp. 105–114.

[44] G. D. Plotkin, "A structural approach to operational semantics," 1981.

[45] K. Slonneger and B. Kurtz, *Formal syntax and semantics of programming languages: a laboratory based approach*. Addison-Wesley Pub. Co., 1995. [Online]. Available: http://books.google.com/books?id=HIRQAAAAMAAJ

[46] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, 1991.

[47] N. Carriero and D. Gelernter, "Tuple Analysis and Partial Evaluation Strategies in the Linda Compiler," in *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua, Eds. Pitman, 1990.

[48] J. Dongarra, S. Hammarling, and D. Walker, "Key concepts for parallel out-of-core lu factorization," *Parallel Computing*, vol. 23, no. 1-2, pp. 49–70, April 1997.

[49] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd ed. Cambridge, MA, USA: MIT Press, 1998.

[50] P. Husbands and K. Yelick, "Multi-threading and one-sided communication in parallel LU factorization," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–10.

[51] R. W. Numrich and J. Reid, "Co-arrays in the next fortran standard," *SIGPLAN Fortran Forum*, vol. 24, no. 2, pp. 4–17, Aug. 2005. [Online]. Available: http://doi.acm.org/10.1145/1080399.1080400

[52] P. S. Barth and R. S. Nikhil, "M-structures: Extending a parallel, non-strict, functional language with state," in *Functional Programming Languages and Computer Architecture*, ser. Lecture Notes in Computer Science, J. Hughes, Ed. Springer Berlin Heidelberg, 1991, vol. 523, pp. 538–568. [Online]. Available: http://dx.doi.org/10.1007/3540543961_26

[53] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: data structures for parallel computing," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 4, pp. 598–632, Oct. 1989. [Online]. Available: http://doi.acm.org/10.1145/69558.69562

[54] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, "ZPL: A machine independent programming language for parallel computers," *IEEE Transactions on Software Engineering*, vol. 26, no. 3, pp. 197–211, Mar. 2000.

[55] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tacsirlar, "Concurrent collections," *Sci. Program.*, vol. 18, no. 3-4, pp. 203–217, Aug. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1938482.1938486

[56] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[57] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[58] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995. [Online]. Available: http://dx.doi.org/10.1002/spe.4380250705

[59] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[60] K. D. Cooper, T. J. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," *Software Practice & Experience*, vol. 4, pp. 1–10, 2001.

[61] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[62] P. Jetley and L. V. Kalé, "Static Macro Data Flow: Compiling Global Control into Local Control," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops 2010*, 2010.

[63] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Montana State University, 1969.

[64] A. Gupta and V. Kumar, "Scalability of parallel algorithms for matrix multiplication," *Parallel Processing, 1993. ICPP 1993. International Conference on*, vol. 3, pp. 115–123, Aug. 1993.

[65] M. Frigo and S. Johnson, "Fftw: an adaptive software architecture for the fft," *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, pp. 1381–1384 vol.3, May 1998.

[66] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[67] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, A. J. Rebón, and P. W. Trinder, "Comparing parallel functional languages: Programming and performance," *Higher Order Symbol. Comput.*, vol. 16, pp. 203–251, September 2003.

[68] S. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice-Hall International (UK), 1987.

[69] H.-W. Loidl and K. Hammond, "On the granularity of divide-and-conquer parallelism," in *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 1995.

[70] Y. Sun, G. Zheng, P. Jetley, and L. V. Kalé, "ParSSSE: An Adaptive Parallel State Space Search Engine," *Parallel Processing Letters*, vol. 21, no. 3, pp. 319–338, September 2011.

[71] W. W. Shu and L. V. Kalé, "A dynamic load balancing strategy for the Chare Kernel system," in *Proceedings of Supercomputing '89*, November 1989, pp. 389–398.

[72] A. Sinha and L. Kalé, "A load balancing strategy for prioritized execution of tasks," in *Seventh International Parallel Processing Symposium*, Newport Beach, CA., April 1993, pp. 230–237.

[73] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11.

[74] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to design programs: an introduction to programming and computing*. Cambridge, MA, USA: MIT Press, 2001.

[75] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[76] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, ser. ACM Sigplan Notices, vol. 33, Montreal, Quebec, Canada, June 1998, pp. 212–223.

[77] L. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha, "Prioritization in parallel symbolic computing," in *Lecture Notes in Computer Science*, T. Ito and R. Halstead, Eds., vol. 748. Springer-Verlag, 1993, pp. 12–41.

[78] C. Lin and L. Snyder, "ZPL: An Array Sublanguage," in *Languages and Compilers for Parallel Computing (Proceedings of the Sixth International Workshop)*. Springer-Verlag, 1994, pp. 96–114.

[79] P. Miller, A. Becker, and L. Kal, "Using shared arrays in message-driven parallel programs," *Parallel Computing*, vol. 38, no. 12, pp. 66 – 74, 2012.

[80] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *J. Supercomputing*, no. 10, pp. 197–220, 1996.

[81] G. Blelloch, "NESL: A Nested Data-Parallel Language," School of Computer Science, Carnegie-Mellon University, Tech. Rep. CMU-CS-92-103, April 1993.

[82] G. Blelloch, "Scans as Primitive Parallel Operations," *IEEE Transactions on Computers*, vol. 38, no. 11, November 1989.

[83] L. Kalè, "An almost perfect heuristic or the N-queens problem," *Information Processing Letters*, vol. 34, no. 4, pp. 173–178, April 1990.

[84] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Proceedings of Supercomputing 93*, Nov. 1993.

[85] S. Aluru and F. Sevilgen, "Parallel domain decomposition and load balancing using space-filling curves," in *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*. IEEE, 1997, pp. 230–235.

[86] A. Grama and V. Kumar, "State of the art in parallel search techniques for discrete optimization problems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 28–35, 1999.

[87] G. Gupta, E. Pontelli, K. A. Ali, M. Carlsson, and M. V. Hermenegildo, "Parallel execution of prolog programs: a survey," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 4, pp. 472–602, 2001.

[88] G.-J. Li and B. W. Wah, "Coping with anomalies in parallel branch-and-bound algorithms," *IEEE Transactions on Computing*, vol. 35, no. 6, pp. 568–573, 1986.

[89] T.-H. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms," *Commun. ACM*, vol. 27, pp. 594–602, June 1984. [Online]. Available: http://doi.acm.org/10.1145/358080.358103

[90] V. N. Rao, V. Kumar, and K. Ramesh, "A parallel implementation of Iterative-Deepening-A*," in *AAAI*, 1987, pp. 178–182.

[91] R. Feldmann, P. Mysliwiete, and B. Monien, "Studying overheads in massively parallel min/max-tree evaluation," in *SPAA '94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1994, pp. 94–103.

[92] L. Kalé, "The REDUCE OR process model for parallel execution of logic programs," *Journal of Logic Programming*, vol. 11, no. 1, pp. 55–84, July 1991.

[93] M. Furuichi, K. Taki, and N. Ichiyoshi, "A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi," in *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990, pp. 50–59.

[94] Y.-J. Lin and V. Kumar, "And-parallel execution of logic programs on a shared-memory multiprocessor," *J. Log. Program.*, vol. 10, no. 1/2/3&4, pp. 155–178, 1991.

[95] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan, "Scioto: A framework for global-view task parallelism," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, ser. ICPP '08. Washington, DC, USA: IEEE Computer Society, 2008. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2008.44 pp. 586–593.

[96] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941582 pp. 201–212.

[97] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2287076.2287103 pp. 137–148.

[98] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, pp. 1–11.

[99] W. Shu and L. Kalé, "Chare Kernel - a runtime support system for parallel computations," *Journal of Parallel and Distributed Computing*, vol. 11, pp. 198–211, 1990.

[100] E. Solomonik and L. V. Kale, "Highly Scalable Parallel Sorting," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[101] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "Uts: An unbalanced tree search benchmark," in *Lecture Notes in Computer Sciences*, vol. 4382. Springer-Verlag, 2007, pp. 235–250.

[102] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[103] J. R. Shewchuk, "Triangle: Engineering a 2d quality mesh generator and delaunay triangulator," pp. 203–222, 1996.

[104] M. S. Warren and J. K. Salmon, "Astrophysical N-body simulations using hierarchical tree data structures," in *Proceedings of Supercomputing 92*, Nov. 1992.

[105] V. Springel, S. D. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly et al., "Simulations of the formation, evolution and clustering of galaxies and quasars," *Nature*, vol. 435, no. 7042, pp. 629–636, 2005.

[106] J. Zhang, B. Behzad, and M. Snir, "Optimizing the barnes-hut algorithm in upc," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063485 pp. 75:1–75:11.

[107] W. Dehnen, "A hierarchical $O(N)$ force calculation algorithm," *Journal of Computational Physics*, vol. 179, pp. 27–42, 2002.

[108] L. V. Kale and S. Krishnan, "A comparison based parallel sorting algorithm," in *Proceedings of the 22nd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1993, pp. 196–200.

[109] K. Fukunaga and P. M. Narendra, "A branch and bound algorithms for computing k-nearest neighbors," *IEEE Trans. Computers*, vol. 24, no. 7, pp. 750–753, 1975.

[110] J. Palsberg and C. B. Jay, "The essence of the visitor pattern," in *Proceedings of the 22nd International Computer Software and Applications Conference*, ser. COMPSAC '98. Washington, DC, USA: IEEE Computer Society, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=645980.674267 pp. 9–15.

[111] B. Meyer and K. Arnout, "Componentization: The visitor example," *Computer*, vol. 39, no. 7, pp. 23–30, July 2006. [Online]. Available: http://dx.doi.org/10.1109/MC.2006.227

[112] L. Kale, A. Arya, N. Jain, A. Langer, J. Lifflander, H. Menon, X. Ni, Y. Sun, E. Totoni, R. Venkataraman, and L. Wesolowski, "Migratable objects + active messages + adaptive runtime = productivity + performance a submission to 2012 HPC class II challenge," Parallel Programming Laboratory, Tech. Rep. 12-47, November 2012.

[113] R. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics - Theory and application to non-spherical stars," *Monthly Notices of the Royal Astronomical Society*, vol. 181, pp. 375–389, Nov. 1977.

[114] J. W. Wadsley, J. Stadel, and T. Quinn, "Gasoline: a flexible, parallel implementation of TreeSPH," *New Astronomy*, vol. 9, pp. 137–158, Feb. 2004.

[115] J. Stam and E. Fiume, "Depicting fire and other gaseous phenomena using diffusion processes," in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/218380.218430 pp. 129–136.

[116] K. Heitmann, P. M. Ricker, M. S. Warren, and S. Habib, "Robustness of Cosmological Simulations. I. Large-Scale Structure," *ApJSup*, vol. 160, pp. 28–58, Sep. 2005.

[117] H. Dejonghe, "A completely analytical family of anisotropic Plummer models," *MNRAS*, vol. 224, pp. 13–39, Jan. 1987.

[118] M. S. Warren, "2hot: an improved parallel hashed oct-tree n-body algorithm for cosmological simulation," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503220 pp. 72:1–72:12.

[119] D. B. Larkins, J. Dinan, S. Krishnamoorthy, S. Parthasarathy, A. Rountev, and P. Sadayappan, "Global trees: a framework for linked data structures on distributed memory parallel systems," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413428 pp. 57:1–57:13.

[120] Y. Jo and M. Kulkarni, "Enhancing Locality for Recursive Traversals of Recursive Structures," in *Proceedings of OOPSLA '11*, October 2011, p. to appear.

[121] V. Springel, "The cosmological simulation code GADGET-2," *MNRAS*, vol. 364, pp. 1105–1134, 2005.

[122] M. D. Dikaiakos and J. Stadel, "A performance study of cosmological simulations on message-passing and shared-memory multiprocessors," in *Proceedings of the International Conference on Supercomputing - ICS'96*, Philadelphia, PA, December 1996, pp. 94–101.

[123] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity," *J. Parallel Distrib. Comput.*, vol. 27, no. 2, pp. 118–141, June 1995. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1995.1077

[124] H. Shan and J. P. Singh, "Parallel tree building on a range of shared address space multiprocessors: Algorithms and application performance," in *in: Proceedings of the 12th International Parallel Processing Symposium*, 1998, pp. 475–484.

[125] C. Zhang, A. Krishnamurthy, and O. Y. Wang, "Brushwood: Distributed trees in peer-to-peer systems," in *In Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS05*, 2005, pp. 47–57.

[126] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed b-tree," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 598–609, Aug. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1453856.1453922

[127] M. K. Reiter, A. Samar, and C. Wang, "Self-optimizing distributed trees," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–12, 2008.

[128] *The Common Object Request Broker: Architecture and Specification (Draft)*, 10 December 1991, revision 1.1.

[129] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens, "Divorcing language dependencies from a scientific software library," in *Proceedings of the 10th SIAM Converence on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 2001.

[130] T. G. Epperly, G. Kumfert, T. Dahlgren, D. Ebner, J. Leek, A. Prantl, and S. Kohn, "High-performance language interoperability for scientific computing through babel," *International Journal of High Performance Computing Applications*, vol. 26, no. 3, pp. 260–274, 2012. [Online]. Available: http://hpc.sagepub.com/content/26/3/260.abstract

[131] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a Common Component Architecture for High-Performance Scientific Computing," in *Proceedings of the 1999 Conference on High Performance Distributed Computing*, Redondo Beach, California, August 1999, pp. 115–124.

[132] N. Mahmood, G. Deng, and J. C. Browne, "Compositional development of parallel programs," in *Lecture Notes in Computer Sciences*, vol. 2958. College Station, Texas, USA: Springer-Verlag, October 2003, pp. 109–126.

[133] K. Varda, "Protocol buffers: Google's data interchange format," Google, Tech. Rep., 6 2008. [Online]. Available: http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html

[134] X. Jiao, M. T. Campbell, and M. T. Heath, "Roccom: an object-oriented, data-centric software integration framework for multiphysics simulations," in *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*. New York, NY, USA: ACM Press, 2003, pp. 358–368.

[135] S. Atlas, S. Banerjee, J. C. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn, "POOMA: A high performance distributed simulation environment for scientific applications," in *Supercomputing '95*, 1995.

[136] W. D. Gropp and B. F. Smith, "Scalable, extensible, and portable numerical libraries," in *Proceedings of the Scalable Parallel Libraries Conference*, 1994, pp. 87–93.

[137] S. Parker, "A component-based architecture for parallel multi-physics pde simulation," in *Computational Science ICCS 2002*, ser. Lecture Notes in Computer Science, P. Sloot, A. Hoekstra, C. Tan, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2002, vol. 2331, pp. 719–734. [Online]. Available: http://dx.doi.org/10.1007/3-540-47789-6_75

[138] G. Tang, E. F. D'Azevedo, F. Zhang, J. C. Parker, D. B. Watson, and P. M. Jardine, "Application of a hybrid mpi/openmp approach for parallel groundwater model calibration using multi-core computers," *Comput. Geosci.*, vol. 36, pp. 1451–1460, November 2010. [Online]. Available: http://dx.doi.org/10.1016/j.cageo.2010.04.013

[139] M. Jones, R. Yao, and C. Bhole, "Hybrid mpi-openmp programming for parallel osem pet reconstruction," *Nuclear Science, IEEE Transactions on*, vol. 53, no. 5, pp. 2752–2758, oct. 2006.

[140] F. Cappello and D. Etiemble, "Mpi versus mpi+openmp on the ibm sp for the nas benchmarks," in *Supercomputing, ACM/IEEE 2000 Conference*, Nov 2000, pp. 12–12.

[141] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur, "Hybrid parallel programming with mpi and unified parallel c," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1787275.1787323 pp. 177–186.

[142] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory," *Journal of Computing*, May 2013, doi: 10.1007/s00607-013-0324-2.

[143] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls," *Commun. ACM*, vol. 55, no. 6, pp. 121–130, June 2012. [Online]. Available: http://doi.acm.org/10.1145/2184319.2184345

[144] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, 1996, p. 35.

[145] A. Gursoy and L. Kalé, "Performance and Modularity Benefits of Message-Driven Execution," *Journal of Parallel and Distributed Computing*, vol. 64, pp. 461–480, 2004.