5-1-2009

The application of the key-value-reference model in dynamic irregular parallel computation

Yang Zhang

THE APPLICATION OF THE KEY-VALUE-REFERENCE MODEL IN DYNAMIC IRREGULAR PARALLEL COMPUTATION

By

Yang Zhang

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2009

THE APPLICATION OF THE KEY-VALUE-REFERENCE MODEL IN

DYNAMIC IRREGULAR PARALLEL COMPUTATION

By

Yang Zhang

Approved:

Edward A. Luke Associate Professor of Computer Science and Engineering (Major Professor)	Eric A. Hansen Associate Professor of Computer Science and Engineering (Committee Member)
Yoginder S. Dandass Assistant Professor of Computer Science and Engineering (Committee Member)	David L. Marcum Professor of Mechanical Engineering (Committee Member)
Edward B. Allen Associate Professor of Computer Science and Engineering, and Graduate Coordinator (Committee Member)	Sarah A. Rajala Dean of the Bagley College of Engineering

Name: Yang Zhang

Date of Degree: May 2, 2009

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Edward A. Luke

Title of Study: THE APPLICATION OF THE KEY-VALUE-REFERENCE MODEL IN DYNAMIC IRREGULAR PARALLEL COMPUTATION

Pages in Study: 163

Candidate for Degree of Doctor of Philosophy

This dissertation studies the effects of the "key-value-ref" model in the computational field simulation software development process. The motivation of this study is rooted in addressing the high cost of designing and implementing high-performance simulation software that runs on modern parallel supercomputers. Unlike traditional sequential programming where a number of effective tools exist, parallel super-cluster programming contains many low-level constructs that increase the complexity in the implementation of a software design. More importantly, the dynamic nature of the simulation problems brings additional challenges into the designing stage. Often a designer has to face a number of competing factors and needs to devise strategies to make a trade-off and to find better software structures that can be realized with reasonable performance and flexibility. Proper modeling can help to address many of these issues in the design and implementation stages. Using a two-phase Lagrangian particle-field simulation problem as a case study, this dissertation shows that the "key-space" concept developed in the "key-value-ref" model within this

dissertation is able to model the essential components in available design approaches for parallel computational field simulation, and that the model also helps to expose the design choices in a more sensible way, and also offers certain guidance towards the crafting of a better software structure. In addition, a programming interface is also designed and implemented that allows the development of computational field simulation software utilizing the “key-space” concept. Empirical results show that the current implementation provides a reasonable performance compared to those highly optimized hand-tuned programs.

Key words: parallel programming, software composition, resource management, numerical simulation

DEDICATION

to my parents, for their patience

ACKNOWLEDGMENTS

It is a long long journey to reach this point, and it's good to finally be able to abandon all this work, well, may not be yet, but at least for the moment. Many years ago, I read Matteo Frigo's dissertation. I've always liked his opening for the acknowledgments and I wanted to quote it in here: "This brief chapter is the most important of all. Computer programs will be outdated, and theorems will be shown to be imprecise, incorrect, or just irrelevant, but the love and dedication of all people who knowingly or unknowingly have contributed to this work is a lasting proof that life is supposed to be beautiful and indeed it is." In fact, this sweet little writing is why I've always remembered his thesis.

I am much indebted to Dr. Ed Luke, for being my advisor for such a long time, for all the support (both financially and spiritually) he has given, and for the freedom he has allowed in my study and research schedule. Ed himself is also a brilliant hacker. He has had great influence on my view of computer programming and software development. He also brought me into the Linux and open-source world. All of these I am very grateful to. I also thank Drs. Eric Hansen, Yogi Dandass, Dave Marcum, and Ed Allen for serving on my thesis committee and the excellent feedback they have provided on my research. In the past, I've also had much fun within Dr. Hansen and Dr. Marcum's classes. Dr. Allen also wrote the LaTeX template that I have used to typeset this dissertation. It has greatly simplified the task and has made it much easier to format a standard conforming document.

I also thank Dr. Junxiao Wu for help developing the physics code for the Lagrangian particle tracking case study.

This dissertation was produced on a Ubuntu Linux machine with the help of pdfTEX, GNU emacs, and the Ipe drawing editor. I thank all the people behind these wonderful systems and tools, and I am also grateful that they are made freely available. During these years, I've also enjoyed reading the many articles on the ever expanding `wikipedia.org`, which considerably broadened my knowledge and interest in various topics other than computer science, and this also provided much fun in the spare time (it is currently my most visited website, second only to `google.com`). I appreciate all the people who have contributed their knowledge, effort, and time.

I've been in the grad school for seven years and eight months. I thank all the people who have encouraged and helped me even though it is not possible to list them all at here. I would thank the following friends and former and present research group members, for their friendship and the many joys and frustrations shared together: Jilin Zhang, Qingluan Xue, Krunal Soni, Lei Liang, Qiuying Zhao, Changhe Yuan, Song Zhang, Yangrong Ling, Qiuhan Xue, Eric Collins, Bela Soni, Bijay Shrestha.

I also thank my cousins Chong and Kaiwen, for their care and the many relaxing vacations spent in their places. My parents always deserve a special thanks, for their understanding, comfort, and the constant encouragement, and also to their reminder that life is much more to enjoy than computer programming and reading. I miss very much my dear grandparents, especially at this time. I wish they were able to know that this day has finally come to true.

I have deeply appreciated the following wisdom throughout this period of study. The first one appears to be credited to Fred Brooks: "Dissertations are not finished; they are abandoned." The second one is from Douglas Hofstadter, the Hofstadter's Law: "It always takes longer than you expect, even when you take into account Hofstadter's Law." Finally I think the meaning of this entire process is to experience this journey and to get ready to start a new adventure in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF CODELETS

CHAPTER 1

INTRODUCTION

Large scale software are often identified as the most complex human constructs. Not surprisingly their development is not easy. The classic software engineering article "No Silver Bullet (NSB)" [10] categorizes software difficulties into essence and accidents. In such a view, the essence of a software entity is its conceptual construction that is invariant under different representations. While the accidents are those difficulties arise from the particular approach chosen to realize the abstract design. The central argument of the "NSB" article is that software essence makes a large part in the software difficulties and is inherent and irreducible.

"NSB" has had profound influence over the software community, whether one agrees with its judgement or not. While the theme of this dissertation is not to argue the fundamental points in "NSB," in fact, I have largely agreed with Brooks' assessment, instead this dissertation tries to examine the effectiveness of an abstract model to attack the essential and accidental complexity for a limited domain in software construction. Perhaps the most memorable part in "NSB" is this summary: "that physical laws are designed by God, hence they are inherently simple and elegant; and that software are designed by *different* people, thus they are inherently complicated." This underlines the importance and elegance of proper modeling and abstraction, which is the central theme in theoretical

physics and mathematics. What is so difficult in modeling software essence is that it is essential and diverse, thus it is hard to unify and simplify. "NSB" argues that abstracting away software complexity often abstracts away its essence, making the abstraction largely irrelevant. However modeling does address essence, as acknowledged in the follow up article "NSB refired" [11]. Proper modeling will help crafting and debugging concepts, which is the essential part in software construction.

This dissertation investigates the utility of the "key-value-ref" model in computational field simulation (CFS) software development. By limiting the scope to a rather narrow domain, it is hoped that such a model would be able to help reveal the inherent relationships among the interlocking concepts in CFS software. The intention is not to claim that such a model is a silver bullet, even in the CFS software domain. Rather developing and enriching the model in this dissertation is to try to make an incremental step towards attacking the CFS software essence, if that is ever possible. Constructively, this dissertation shows that the developments described throughout this thesis help to simplify the understanding and identification of the essential parts in CFS software design, provides a more unified view to the cost trade-offs involved (particularly for CFS problems that exhibit certain dynamism), and also offers guidance towards the crafting of a good design. The associated implementation of the model also helps to remove large part of the accidental complexity in realizing CFS on modern high-performance platforms. Surprisingly, despite the myriads of computational tools available to the CFS community, not many address such accidental complexity on a fundamental level. Combining these efforts and developments, it is hoped that the overall productivity for crafting good CFS software will

be greatly improved. The results may not deliver the magic of a silver bullet that even kills the monsters in the CFS space. But it is perhaps a brass one, if used wisely, it certainly has the capability to transform them back to their humanoid form.

The "key-value-ref" model investigated in this dissertation is not a brand new development. In a simpler yet more general form, it has existed as the relational model in relational database theory [23] for a long time. However its application and utility in the high-performance computing (HPC) community has largely not been examined yet. There are only a few previous efforts that consider the relational model in HPC. The most notable example is the Cornell sparse matrix compiler [58] that utilizes the model to synthesize efficient sparse matrix code for a large collection of storage formats. In this case, the model is mostly used to alleviate the accidental complexities in an even smaller domain of well established applications. The work presented here is built on top of the Loci framework [53, 57] (and more importantly its concepts), which is the only work to my knowledge that considers the relational model and enriches (and constrains) it to suit CFS software development. In addition to addressing much of the accidental complexity in the CFS software domain on modern distributed memory architecture, the model created by the Loci framework has great hopes to help address the question how to better structure CFS software, which is an essential step in the design process.

Simply put, the "key-value-ref" model is a view about the relationships of the fundamental objects involved in a problem. In its view, there are only two kinds of attributes attached to an object (a "key"): a "value" field that associates various properties to an object, and a "ref" field that lets an object refer to other objects. Through this model, many

of the components in a CFS problem can be understood in a more simplified manner, yet the essential dynamism among the interacting concepts is largely kept. Hence it has the potential benefit to help craft good designs more productively. Moreover, the simplicity and the connection to relational algebra allow the model to be implemented systematically and has great performance potential. The resulting composability and parallellism obliviousness from the applications developed by the model removes much of the accidental complexity in software development. Although the accidents do not contribute to the fundamental software difficulties, any removal of that is certainly welcomed. In addition, the reusability resulted from these properties promises another direction to address the essence of CFS software design.

One cannot talk about a model without a concrete example. Throughout this dissertation, I use the Lagrangian particle tracking problem as a case study to illustrate the various concepts and developments, and also to provide a convincing case that the "key-value-ref" model developed in the thesis will be helpful in attacking the complexities in building such software. The Lagrangian particle tracking problem is a two-phase dispersed flow problem, which is an important subclass of the more general multiphase flow (MPF) simulation [9, 21] problems. The general setup of the problem consists of discrete particles moving in a continuum field under the influence of the field force, and that the particles and fields also interact. The objective is to follow each individual particle and determine their precise physics information.

There are in general two approaches to solve the problem on a parallel computer. The first approach (termed as "cell-dec" method) decomposes the continuum field (typically

modeled as discrete mesh [78]) to parallel processes. Particles are assigned to the mesh cells that currently contain them. Tracking a particle amounts to transferring its ownership to another mesh cell (potentially on a different parallel process). The difficulty in such a design is that it is hard to manage the balance of computation on each parallel process. The density of particles in the field is usually not uniform, hence, calculation on each cell is not uniform in this design. An unsteady particle movement precludes an optimized partition to parallel processes that balances the computational cost.

Another design is to decouple the particles from their containing mesh cells and also partition them to parallel processes independently (termed as "dual-dec" method). This frees the particles from being bonded to the field geometry and in turn, balanced computation is easily achieved as an optimized partition can be predetermined. The major challenge in this design is that such an optimized data partition on both the mesh and particles may destroy much of the localization of field-particle interaction thereby increasing the frequency and the volume of data exchange during the interaction, resulting in an increased parallel overhead also.

Such design difficulties are prevalent in multi-component computation problems of a dynamic nature because they represent an inherent conflict in the balance of computation and interaction among the many entities and their relations involved. Unfortunately no known algorithms and design approaches can seem to eliminate such an intrinsic difficulty on modern computing architectures. What we can do instead is to devise methods to counteract and *reduce* the conflict and imbalance presented in these problems, while also keeping the cost of these additional efforts small.

We can look at the same problem under the "key-value-ref" model. Under the present Loci incarnation of the model, keys are those objects that can be enumerated in a computation. For example, we can enumerate discretized space while a physics attribute such as pressure is usually not enumerable. Therefore keys are often those objects that have "shape" in a computation, e.g., the geometry part. Thus both the mesh cells and particles can be keys. However there is usually no restrictions to become value of a key. Therefore they both can be values also. This would yield three combinations[1]: mesh cell as key with particle as value, both cell and particle as keys, and particle as key with cells as value. Note they cannot all be values, otherwise, no object would be initiating the computation. The first two formulations correspond to the cell-dec and the dual-dec design respectively. The third design, while feasible, is generally not used. Being keys, the cells usually have many "refs," and being values means that they will lose these refs since only keys can have refs. Converting all these refs through the means of value computation on the particle keys is a complex process. However no one has even discussed this approach in the literature. Either it is obviously impossible, or that no one has ever thought about it.

[1]In fact, with this model view, there are actually more possibilities than just these three. For example, since there will be many particles in the field, then part of them can become keys and part of them can become the value associated with cells. This would yield a hybrid design approach that has great potential. This approach has also not yet been explored in the literature.

Regardless, the "key-value-ref" model captures the essence[2] of the Lagrangian particle tracking problem and indeed offers more approaches. The designing of the cell-dec and the dual-dec methods is coupled with the consideration of parallelization strategies. While the model based design rests solely on the conceptual level, yet the performance trade-off is preserved. What such a model can do to help the software development beyond solution formulation? Firstly in practical engineering applications, it is the cell-dec method that is widely used, the dual-dec design has largely been ignored for complex unstructured mesh applications. Certainly the idea of dual-dec is not new. It is likely that the unpopularity of the dual-dec design is due to the large accidental difficulties in realizing this approach. Developers in the CFS community usually have quality parallel fluid flow solvers already implemented with a well developed data decomposition methodology for the field mesh. The particle tracking capability is usually added at a later stage. In such cases, the effort required to develop another approach with completely different strategy would be significantly greater than to reuse the existing infrastructure. The other possibility is due to belief that the performance of the dual-dec solution will be inferior to the cell-dec solution approach due to its extensive use of parallel communication. This may indeed be true. However without proper evaluation, one cannot be certain that it is true for all cases.

[2]One can argue that the formulation of these solutions is on the accidental level and not the essential part as these formulations of solutions evidently tie in the influence of implementation idioms on modern parallel computing architectures. However as Martin Flower argues in the latest reload of "NSB" [29]: that the essential/accidental separation has become fuzzier and in some cases, the accidental difficulties increasingly appear as the essence as some form of representation is needed to think of the essence of a problem. This is indeed appropriate. There shall be levels of such a separation of essence/accidents in different domains. For example, one can argue that the realization of a solution on the von Neumann architecture is a form of accidents. Yet most software are built on such an architecture, the accidents could then become the essence as most algorithms are designed for such a platform, whether specifically or implicitly.

Thus one of the utilities of the "key-value-ref" model is to offer a rapid prototyping capability for evaluation of further developments. A quality implementation of the "key-value-ref" model maps a design expressed in its domain onto the current mainstream platform with reasonable performance, allowing one to quickly evaluate the effectiveness and possibility of a design. Such rapid prototyping practice is offered in the "NSB" article as one of the promises to attack the conceptual essence.

There are indeed other numerous benefits of the model. However the one that I wanted to focus on is the ability of the model to influence design choice and to provide guidance for better software structuring. To some extent, each programming language and system has such a goal: by using (or enforcing) a particular paradigm (such as Object-oriented paradigm), the users of the tool can be influenced (or enforced) to use a particular design approach that is likely to yield good solutions for certain problems. In the CFS domain, there are not many systems that have such an intention. One of the major goal of this dissertation is to enrich the "key-value-ref" model to incorporate such abilities. Put this in the concrete example, once one decides to use the dual-dec approach, then how would one be able to craft a more detailed design that has the potential to deliver the performance (and not to mention the correctness)? In other words, there are certain complexities (and some I believe are closer to the essence) involved in realizing this process. Merely offering design choice is not enough, there should be some guidance and paradigms for detailing and refining the design choice as well.

The "key-value-ref" model as its present form in the Loci framework does not have such capability yet. First a full implementation for the model is not yet available. Although

it has the potential to model dynamic applications like Lagrangian particle tracking or other MPF problems, it cannot yet support the realization of the modeling on distributed memory machines. Thus it remains as a conceptual tool. Second, to model all the options properly and especially those design choices similar to the dual-dec design outlined before, the "key-value-ref" model needs to be extended. Specifically, I extended the concept of key further in this dissertation. In the new concept, a key is not only an enumerable object in a computation, but also an object that can freely choose process affinity. On top of this, the concept of "key-space" is proposed to help to address the modeling of the full spectrum of solution formulations, and moreover also to provide patterns for better software structuring. As a real case is critically important to understand various trade-offs and conceptual ideas, I devote the entire chapter 4 in this dissertation to the study of the Lagrangian particle tracking problem. Based on the lessons learned from the case study, chapter 5 develops the key-space concept into the "key-value-ref" model and chapter 7 provides a matching implementation on distributed memory platform.

The development of the "key-value-ref" model itself and its implementation is a process of software construction, and more importantly the crafting and reflecting of concepts and ideas. Thus the "NSB" arguments would also apply to the activity of such development if indeed the essence is to be addressed step by step. The developments of the ideas and their realizations mirror the "NSB" insights closely. The previous Loci system represents the first major milestone, in which the more static CFS problems are understood and tested. Indeed, after this many years of developments, we have made numerous small steps along the way: the software framework itself witnessed incremental changes, performance

has been steadily improved, the various design ideas were proposed, rejected, and more importantly turned into lessons. This dissertation can be regarded as the next milestone (at least I hope so) on the road that starts the process of attacking the difficulties in a more dynamic CFS problem domain.

The major contributions of this dissertation can be summarized as the followings:

- Studies the "key-value-ref" model in the CFS problem domain, particularly those problems that involve dynamic components, such as those multiphase flow simulation problems.
- Provides a practical implementation and evaluation for the "dual-dec" Lagrangian particle tracking approach and carefully compares it to the more popular "cell-dec" approach. Although this work results from the case study, it represents an independent contribution to the engineering community.
- Develops the key-space concept into the "key-value-ref" model and provides a practical implementation that is fully compatible with the present Loci framework, and that when used to build applications, the key-space implementation can yield reasonable performance compared to that of the manually tuned code.

CHAPTER 2

LITERATURE SURVEY

This chapter surveys several parallel programming abstractions. Many parallel programming languages and systems aim to only resolve the accidental complexity in parallel software development by supporting easier resource management in a parallel environment. Indeed the "NSB" article argues that software tools can at best remove the accidents while the essence is unlikely to be addressed. However these activities are still important. Unlike the sequential software construction, the accidental complexity in parallel software development remains to be a hard part and is not completely resolved by modern tools, at least their success is not on the same level as modern sequential software tools. These accidents can become a barrier for parallel software development. Tools that address them are certainly improving the development productivity. The distinction between the essence and the accidents also relates to how one considers what is the essential part in a problem formulation. For example, in designing a parallel algorithm, the notion of multiple worker is essential. Thus an implicit parallel language helps to reduce the essential complexity, for it allows one to craft a mental design with fewer parameters. Of course, this does not signify that an implicit parallel programming model resolves the software essence. Software development in large is more than algorithm development. For example, the drafting and debugging of requirements and specifications is likely to be beyond what a conven-

tional programming model can address. Nevertheless, since dramatic improvements is not likely, the best appears to be small steady advances, even in a narrowly defined domain.

In computer science, abstraction is a process to reduce and factor out details so that complex problems can appear simpler and more natural to solve. One way that a programming language can help to find the right abstraction is by providing a variety of ways to organize data and computation. Another way is to make it possible to build program components that capture meaningful patterns in computation [60]. Computer programming is a vast area, the survey presented in this chapter is by no means complete. One of the purposes to perform the survey is to summarize several characteristics that successful programming systems exhibit. Because the model used by the Loci framework is unique in the HPC area, and that it is closely related to the research presented in this dissertation, I devote the entire chapter 3 to describe it in detail.

2.1 General Purpose Parallel Abstractions

Perhaps the most commonly used approach in parallel programming is to explicitly program the parallelism and all low-level resource management by using a sequential language (often FORTRAN or C) with a low-level parallel primitives library such as MPI [59] on distributed memory platforms or Pthreads [13] on shared memory systems. This is perhaps the most flexible style and can yield high performance for finely tuned code , yet it is often considered tedious and error prone and is in fact the major reason to encourage the development of the various higher level parallel programming abstractions.

The High Performance FORTRAN (HPF) [40] and OpenMP [62] compilers provide annotation based programming style for SPMD (Single Program Multiple Data) parallelism. They offer a set of directives that programmers can use to suggest parallel loop constructs, data alignment control, etc. The compiler is then able to generate the actual parallel code based on these directives. HPF is presently fading from mainstream interests due to its lacking of a portable performance model. OpenMP mainly targets shared memory platforms and lacks high-performance support on the more popular distributed memory architectures.

Cilk [8, 41, 65] is a language for multi-threaded parallel programming based on C. It was designed to be simple and efficient and was originated from theoretical studies of thread scheduling. Although it is general purpose, it is particularly suited to dynamic and highly asynchronous problems, such as AI problems like chess [22]. Programming parallelism in Cilk generally consists of using keywords such as `spawn` and `sync` to expose and indicate the parallelism. The compiler and run-time system are responsible for scheduling the computation on a given platform and take care of details like load balancing, paging, and communication. Internally Cilk uses the "work and depth" model to estimate the parallel performance and its randomized work-stealing scheduler can guarantee asymptotically optimal parallel running time. Though successful on shared memory architecture, the "work and depth" model does not appear to fit well on a range of practical architectures including distributed memory machines as it neglects the interprocess communication costs. Another problem is that if locks are used in a Cilk program, then

the run-time system cannot guarantee anything. For some problems, designing a lock-free algorithm can be challenging if not impossible.

In recent years, Java has gained mainstream interest in object-oriented programming and becomes one of the dominant languages today. Its general focus is not on performance, however its other features like safety and portability makes it suitable for software development. Titanium [86] is an explicit parallel dialect of Java to support scientific computing. It inherits all of Java's safety, portability, usability, and expressiveness features. However Titanium is compiled, therefore it has improved performance potential due to the bypass of virtual machines. It incorporates several interesting features. Titanium provides a shared memory abstraction and yet is portable across a range of architectures. However performance tuning may be necessary on distributed memory platforms. Titanium also utilizes static analysis techniques to prevent deadlock on barrier synchronization. It also augmented the type system and utilizes type inference for pointer analysis. In the memory management area, Titanium uses region-based memory management [32] in additional to garbage collection. It also utilizes research results from high-performance communication systems and provides built-in multidimensional arrays, points, rectangles, and general domain types. Several mesh simulation applications have been written in Titanium, including adaptive mesh refinement applications.

Declarative languages such as functional programming has long been considered to be natural and particularly suited for implicit parallel programming. Due to the referential transparency property, compilers are easier to analyze the program and discover independent tasks. SISAL [15, 27] and NESL [6, 7] are two implicit data parallel func-

tional languages designed to support scientific computing and have showed success to some degrees. NESL takes the data parallelism further to be able to discover nested data parallelism, thus made it suitable for implementing parallel nested loops and divide-and-conquer algorithms. NESL also uses the "work and depth" performance model. Programs developed in SISAL and NESL are simple and elegant, and parallelism are largely hidden from the programmer. Reasoning about the program behavior is also easy. However they both lack efficient support on the distributed memory platform, making them less useful in addressing today's needs.

The functional language Haskell [63] has generated many parallel and concurrent extensions [80] such as GpH [79], and distributed implementations GdH [64], and Eden [52]. In such extensions, Haskell is mainly used as the computation language, while the extended language constructs act as coordination languages to manage the parallelism. However such an approach generally do not qualify for a fully implicit parallel language as the coordination languages usually include some forms of explicit concurrency control. Unlike SISAL and NESL, these languages are not designed specifically for scientific computing. Their applicability to high-performance computation is not certain due to the lack of empirical examples.

Another direction in automatic implicit parallel programming is to develop parallelizing compiler techniques to extract the inherent parallelism automatically. The approach could provide a low cost way to convert existing sequential programs to parallel codes that can utilize multiple processing resources. Static program analysis is the most widely used technique in parallelizing compilers. For example, the well known SUIF framework [37]

uses scalar, array, and interprocedural analysis among others to locate coarse-grain parallelism in serial programs that are essential to utilize multiple processors effectively. A drawback of static analysis is that it is not able to consider information only available at run-time. In the other directions, compiler code optimization (parallelization) can be based on information gathered at run-time [67, 72]. Recently techniques such as hybrid analysis [69] is also proposed to combine the classic compile-time and run-time analysis techniques. The major problem for parallelizing compiler technologies is that real world programs may be much too complex for the most sophisticated compilers to analyze. Parallelizing compilers usually work well on small number of processors and may not scale to large parallel system. Another problem is that they often only targeted shared memory architectures.

2.2 Domain-specific Parallel Abstractions

Domain-specific programming languages and systems provide even higher level abstractions. They provide extensive support for a problem domain, often with specialized data type and structures. In scientific applications, domain-specific programming support often takes the form of special library and programming framework. Special libraries have been critical to scientific computing. Well known libraries such as LAPACK [1] (provides collections of parallel solver kernels for dense and banded matrix structures) and PETSc [4] (data structures and routines for the parallel solution of partial differential equations) are used by many applications. These are the *de facto* standards for their ap-

plication domain and represent the very high-performance that can be achieved for these tasks on modern architectures.

It is worth mentioning several attempts to develop portable high-performance scientific libraries in the late 1990s. The libraries produced by those efforts are collectively called active libraries [83]. Unlike traditional libraries, which are usually passive collections of data types and functions, active libraries take active role at compile-time to tune themselves for a target machine. They interact with the compiler and machine environment to generate additional components, specialize algorithms, and reconfigure critical parameters, etc. Well known examples include ATLAS [85] (Automatically Tuned Linear Algebra Software), the Blitz++ array library [82], FFTW [31] for Fourier transforms, the MTL matrix library [73], and the POOMA library for data parallel physics [44], etc. Active libraries blend the traditional boundaries between compiler and library by performing computations and metaprogramming at compile- or run-time to tune themselves, thus achieving portability (by reconfiguring themselves) and performance. At the same time, due to their use of advanced programming features (often object-oriented and generic programming [77]), the programming constructs and interface of most active libraries are quite elegant and indeed very compositional. These activities have however faded out in recent time perhaps due to the intimate interaction between these libraries and the compilers. If the compiler implementations do not have predictable behaviors, then the performance of active libraries may suffer or may not be predictable.

Among the active libraries, the FFTW library uses a unique and innovative approach for self-tuning. FFTW employs a run-time planner that takes problem properties and re-

turns a fast executable structure that can be used to perform discrete Fourier transform. The planner works by recursively breaking the problem to many subproblems and then uses dynamic programming technique to search through the problem space and measures the actual running time of many different plans and selects the best one. Those baseline subproblems are solved by highly optimized special "codelets" generated by a special code generator [30]. The planning may be time consuming. However since a plan can (and is likely to) be reused many times, the cost is thus amortized.

Domain-specific programming framework for computational science often facilitate numerical application development by hiding most of the bookkeeping work in parallel programming. Thus a user can concentrate more on the numerical aspects and be more productive. Examples of these systems include the OVERTURE framework [12, 39] for computations on overset grids, the Cactus framework [34] for Cartesian adaptive mesh refinement solvers as well as collaborative development and computational grid [28] enabling technology. These frameworks typically make specific assumptions about the approaches used to solve given problems. If the numerical strategies are sufficiently similar to the model used in the framework, they can be helpful. However each of them appears to target a rather narrow range of problems. Thus they can be often viewed as "plug and play" systems and often impose detailed component structure and interaction requirements. They can also be thought of as a coordinator for software components written according to their specifications. Note such coordination and composition are usually coarse grained and ad hoc, meaning that they usually lack a clear and fundamental programming model that can succinctly describe the types of computation and problems they intend to

support. However one direction where these frameworks show promises is component reuse. Reuse can resolve much of the complexity in software construction by avoiding the construction at all. But this also means that the concept and interface of a computational framework would need to achieve a prevailing status before reuse is possible.

2.3 Summary

This chapter sampled various parallel programming languages and systems. Although the intention is not to directly compare them to the research work in this dissertation, there are several observations that can be made. First it is obvious to see that for a programming language or system to be useful for scientific computing, it must be able to support the distributed memory architecture well. Although shared memory and other architectures are also used in numerical simulation, distributed memory system remains to be the most performance/cost effective and scalable system today and is generally the preferred choice for running scientific applications. There are many elegant abstractions designed for scientific applications that fail to produce an efficient distributed memory implementation, which limits their interest and real usefulness in the domain of scientific programming. Put this in the perspective of the essence/accidents point of view, these systems do not address the accidental cost in parallel programming well since most of these accidents today appear to be the efficient resource management on distributed memory platforms.

The second important observation is that useful abstractions that are intended for a wide range of problems usually need to incorporate some controls or annotations from the users. Implicit and complete automatic parallel programming can be extremely hard

to have a high-performance implementation. For example it has been argued [60] that declarative programs lack the ability to efficiently utilize the inherent parallelism and it is necessary that they provide some controls to the programmer to specify where parallelism may be beneficial. In fact, this is largely the reason why many parallel declarative languages have added some "extra-declarative" constructs in the end. This observation can also be made in the case of parallelizing compilers. One cause of the problems in parallelizing compiler technologies is that they are usually designed to support traditional sequential programming languages, whose abstractions often ignore any parallel computational aspects. In turn, algorithms expressed with these abstractions do not have adequate level of semantics for the compilers to recognize the parallelism.

If the major goal of an abstraction is to provide natural solution description to remove or reduce the accidental costs, then adding adequate annotations should not detract the value of an abstraction too much. Carefully designed annotations shall not involve tedious and detailed low-level programming. The benefit of annotation is that the choice to use it is on the user side. Fail to use it should only reduce the performance and not the correctness. Moreover, the system is free to choose whatever implementation that is deemed suitable. The importance of proper annotation is that it tells the system where and how to pay attention to potential performance optimizations. For example, the parallel loop directives in OpenMP do not in general expose the programmers to the actual parallel implementation of data partitioning and distribution. The system can even choose to ignore the annotation if it thinks that is appropriate. For example, the `register` keyword in the C language is used to hint the compiler for register optimizations. Although it is still a

legitimate keyword, it is now seldom used and the compiler may ignore any of such use due to the advance of register allocation algorithms. Therefore it seems necessary that a successful parallel programming abstraction should incorporate some amount of annotation and control indication from the programmer if it is to be applicable to a wider range of problems.

CHAPTER 3

THE KEY-VALUE-REF MODEL AND THE LOCI FRAMEWORK

Loci is a programming framework under development at Mississippi State University. It is specifically designed for the programming support of data parallel computational field simulations. The current Loci design utilizes a restricted relational model as the basis for its programming model. This relational model is extended in this dissertation to become the more general "key-value-ref" model that allows the modeling and description of a wider range of CFS problems, particularly those involving dynamic components, such as multiphase flow simulation. The details of the Loci framework and its design and implementation can be found in other recent publications [33, 57, 74, 87, 88]. The focus of this chapter is to establish necessary terminologies, notations, and background information for the later chapters.

3.1 Basic Concepts and Terminologies

The fundamental model used in the Loci framework is a variant of the relational model as in the relational database management systems. In this dissertation, it is named as the "key-value-ref" model to make distinctions to the model used in conventional relational database system. The new name chosen in this dissertation also reflects the new developments to the model that are presented in later chapters.

Definition 3.1 The *key-value-ref* model is a variant of the relational model. It is abbreviated as the KVR model. The basic elements in the model are: *key*, *value*, and *ref*.

Definition 3.2 A *key* is an object that can be enumerated in a computation. Only equality test can be performed on keys.

Remark 3.1 In CFS problems, keys usually have geometric meaning, e.g, a discretized space region, and they are also the place where computations occur. Keys are represented by integers in the current KVR model.

Definition 3.3 A *value* is a property associated with a key. On the model level, no operators are defined for values. However in the numerical computation level, they are free to be used in any operations the user defined.

Remark 3.2 Value has a broad meaning. Every attribute in a computation can become a value. Since values have no defined operators in the model, they cannot be compared in the model's language.

Definition 3.4 A *ref* is a special type of value associated with a key. Ref is specifically used to denote the acquaintance of a key to another key. Only equality test can be performed on refs.

Definition 3.5 The basic data structure in the KVR model is *table*. A table is a way of organizing $(key, value)$ and (key, ref) pairs. A KVR table is composed of two columns. The first column records keys, and the second one records either value or ref (in the same table, it cannot be mixed with both types). Given a table $t(key|value)$, its columns are

referenced as $t.k$ and $t.v$. Similarly, $t.k$ and $t.r$ are used to refer to individual column in table $t(key|ref)$.

Remark 3.3 In the current model design and implementation in the Loci framework, a KVR table can only be looked up on the key column. Given a table t and a key k in $\mathrm{dom}(t)$, the look up of the table t with the key k is denoted as $t[k]$.

Proposition 3.1 A key cannot exist on its own, it must be bonded to either a value or a ref. In other words, keys only exist in tables as $(key, value)$ and (key, ref) pairs.

Definition 3.6 A *store* is an alias for a KVR table with "$key|value$" as column names. There is no duplicated keys allowed in a store.

Definition 3.7 A *map* is an alias for a KVR table with "$key|ref$" as column names. There is no duplicated keys allowed in a map.

Definition 3.8 A *multistore* is a store in which duplicated keys are allowed.

Definition 3.9 A *multimap* is a map in which duplicated keys are allowed.

Definition 3.10 The *inverse* of a map or multimap is an operation that projects a new map or multimap with the columns reversed, i.e., the original key column becomes the new ref column, and the original ref column becomes the new key column. The inverse of a map can be a map or multimap. The inverse of a multimap can also be a map or a multimap. The function "$\mathrm{inv}(m)$" denotes the inverse of a map or multimap m.

Definition 3.11 A *param* is a special type of store whose value column has the same contents for every key in the key column.

Remark 3.4 The need for the param type is to facilitate a table data structure implementation that is efficient on memory.

Definition 3.12 The *domain* of a KVR table is defined to be the set of all keys in the key column of the table. The function "$\mathrm{dom}(t)$" is used to denote the domain of a table t.

Definition 3.13 Given a map or multimap, its *image* is defined to be the set of refs that corresponds to a set of keys in its domain. The function "$\mathrm{img}(m, k)$" is used to denote the image of a map m corresponds to the set of keys k in its domain.

Example 3.1 For example, $\mathrm{img}(m, \mathrm{dom}(m))$ gives the image of m for its entire domain.

Definition 3.14 The *pre-image* is defined for map and multimap. Given an image of a map or multimap, the pre-image is the set of keys in the domain that produces the image. The function "$\mathrm{pmg}(m, i)$" is used to denote the pre-image of a map m with an image i.

Example 3.2 For example, $\mathrm{pmg}(m, \mathrm{img}(m, \mathrm{dom}(m)))$ gives $\mathrm{dom}(m)$, the domain of m.

Definition 3.15 A KVR *join* is an operation performed on two KVR tables that results in a merged table. It is a special type of natural join operation as in the relational algebra. Given two tables $t_1(c_1|c_2)$ and $t_2(c_1|c_2)$, the KVR join is denoted as $t_1 \rightarrow t_2$, and it is equivalent to the standard relational algebra notation $\pi_{<t_1.c_1, t_2.c_2>}(t_1 \underset{t_1.c_2=t_2.c_1}{\bowtie} t_2)$.

Proposition 3.2 The KVR join operation produces tables that have either the "$key|value$" or "$key|ref$" columns, i.e., a KVR table. Hence the KVR join is a closed operation.

Proposition 3.3 The definition of KVR join "$t_1(c_1|c_2) \rightarrow t_2(c_1|c_2)$" excludes store and multistore to be in the left side as t_1. This is because the column $t_1.c_2$ is in an equality test and therefore can only be a ref column.

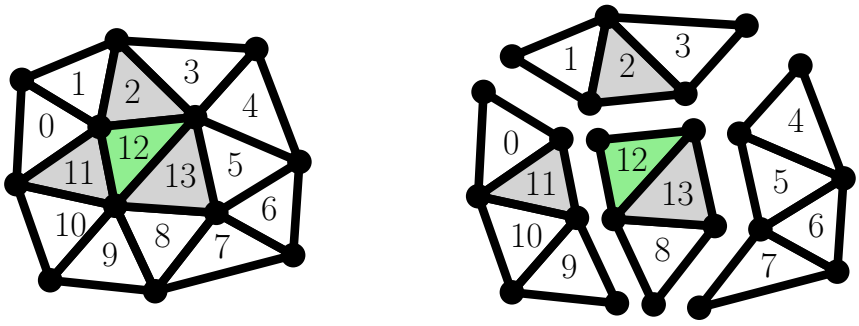

Figure 3.1

An Example 2D Unstructured Mesh

Let us consider an example that illustrates most of the previous definitions. Figure 3.1 shows a two dimensional unstructured mesh. In the view of the KVR model, the cells in the mesh can become keys since they can be enumerated and in a numerical algorithm, cells are often one of the major places where computations are performed. In Figure 3.1, the cell keys are represented by integers. For the purpose of illustration, we can imagine in a computation, the cells are assigned a temperature property with the units of Kelvin. Then such temperature property can become the value associated with the keys representing cells. The temperature cannot itself become keys because it is in general not enumerable.

Suppose an algorithm also uses the neighbor definition for cells. A neighbor of a cell is defined to be all the cells that share one of its edges. For example, in Figure 3.1, cell 12 has its neighbor cells $\{2, 11, 13\}$. This is a commonly used topology in numerical algorithms since this relationship between cells is frequently used to compute weighted summations. Under the KVR model, the neighbor of a cell can become a ref associated with keys representing cells since it defines an acquaintance relationship between cells that essentially lets a cell refer to other cells. These definitions give the "(cell, temperature)" and "(cell, neighbor)" pairs.

Table 3.1

KVR Table Data Structure for the 2D Unstructured Mesh

Temperature (T)	
key	$value$ (Kelvin)
...	...
2	100
11	105
12	106
13	99
...	...

Neighbors (N)	
key	ref
...	...
11	0
12	2
12	11
12	13
...	...

$N \rightarrow T$	
key	$value$ (Kelvin)
...	...
11	106
12	100
12	105
12	99
...	...

Table 3.1 shows the table organization for the definitions just created for the two dimensional unstructured mesh. The table T records the cell and temperature association. It is a store type. Note if it can be guaranteed that the temperature is always the same for every cell, then the table T can become a param type. The table N records the cell and neighbor association. It is a multimap type since the key column contains duplicated keys.

Finally $N \rightarrow T$ produces a new table (a multistore) that combines the data in T and N. The use of this join operation has the similar effect of querying in a relational database. In this particular example, it is used to bring the temperature of all neighbors to all the cells. Suppose the cells in the mesh in Figure 3.1 are the entire keys defined in T and N, then $\mathrm{dom}(T) = \mathrm{dom}(N) = \mathrm{dom}(N \rightarrow T) = \{0\text{–}13\}$, $\mathrm{img}(N, \{12\}) = \{2, 11, 13\}$, $\mathrm{pmg}(N, \{2, 11, 13\}) = \{12\}$. The join $N \rightarrow T$ operation is essentially an indirect access to the table T through table N.

3.2 Programming Interfaces

The Loci framework provides a simple rule system as the main programming interface to use the concepts in the KVR model. The rule system in the Loci framework does not rely on unification [68], which is a technique widely used in logic programming [47]. The KVR model employed in the Loci framework has only two columns in the table. It is not necessary to use unification and this also improves system performance. The Loci rule system resembles the rule definition used in the "Make" [76] compilation tool. However the concept of iteration is supported in the Loci rule system.

Definition 3.16 A *rule* in the Loci framework is a function. The input/output is referred to as the rule signature. The transformation is called the rule body.

Definition 3.17 A *rule signature* is a way to document the input and output variables of a Loci rule. It has the general form $head \leftarrow tail$. The $head$ is referred to as the rule head

and the $tail$ is referred to as the rule tail. They can only contain variables represent KVR tables.

Example 3.3 An example rule signature is: $A \leftarrow T$. Another example: $A \leftarrow (N \rightarrow T)$.

Definition 3.18 A rule body is a general routine that can access the variables defined in the rule signature. They can also define and use local variable that is not KVR tables.

Definition 3.19 A *named variable* is defined as a variable in the rule signature that does not participate any KVR join operation, or a variable that forms a KVR join operation.

Definition 3.20 An *unnamed variable* is defined as the one denotes a table produced by a KVR join in a rule signature.

Example 3.4 For example, in a rule signature $A \leftarrow (N \rightarrow T)$, the table produced by $N \rightarrow T$ is denoted by an unnamed variable. The variables A, N, and T are named variables.

Definition 3.21 A *head unnamed variable* is an unnamed variable appearing in a rule head in the rule signature.

Definition 3.22 The concept *head copy* is defined to be the copy of a head unnamed variable to the last named variable in the KVR join that produces the head unnamed variable. Assume without loss of generality the head unnamed variable is $N \rightarrow A$. Thus to head copy is to copy the table $N \rightarrow A$ to the table A. The copy procedure follows: let $n = \mathrm{dom}(N), a = \mathrm{dom}(A), i = \mathrm{img}(N, n), s = a \bigwedge i, c = \mathrm{pmg}(N, s)$, for each key e in

the key-set c, copy $(N \rightarrow A)[e]$ to $A[N[e]]$. The head copy is a mandatory action after each rule.

Remark 3.5 The requirement for head copy is mandatory because if the output is an unnamed variable, then the result is lost after the rule is completed. In practice, head copy does not result in additional memory copy. It can always be optimized by generating the results directly to the target variable.

Definition 3.23 An *incompatible* head copy is a head copy between tables of different types. It is an error. A *compatible* head copy is a head copy between tables of the same type. Only compatible head copy is allowed. The rule that results in an incompatible head copy is a specification error.

Definition 3.24 The action *pulling* is defined as accessing (reading) the variables documented in a rule tail. Accessing unnamed variable is referred to as *indirect* pulling. Accessing named variable is referred to as *direct* pulling.

Definition 3.25 The action *pushing* is defined as writing the variables documented in a rule head. Writing to named variable is referred to as *direct* pushing. Writing to unnamed variable is referred to as *indirect* pushing.

Proposition 3.4 Indirect pushing is equivalent to direct pushing via a transformation.

Proof: An indirect pushing rule $(M \rightarrow S) \leftarrow X$ is equivalent to the rule $S \leftarrow (M \bowtie X)$. The $\bowtie$ denotes the natural join operation as in relational algebra. Since head copy duplicates $M \rightarrow S$ to S, the two rules achieve the same effect. ■

Remark 3.6 In the proof, the transformed rule $S \leftarrow (M \bowtie X)$ uses the $\bowtie$ operator instead of the KVR join $\rightarrow$ operator in its rule tail. Notice that $M \bowtie X$ merges the table based on the column $M.k$ and $X.k$, i.e., the key column in both tables. This produces a table with three columns $T(key|ref|value)$. The KVR join (as in Definition 3.15) has a specific definition that will only join the ref and the key columns of the two participating tables. Therefore this new form is not expressible using the KVR model.

Proposition 3.5 Indirect pushing is equivalent to indirect pulling via a transformation.

Proof: An indirect pushing rule $(M \rightarrow S) \leftarrow X$ can be transformed to a functionally equivalent indirect pulling rule $S \leftarrow (\mathrm{inv}(M) \rightarrow X)$. Likewise an indirect pulling rule $S \leftarrow (M \rightarrow X)$ can be transformed to an indirect pushing rule $(\mathrm{inv}(M) \rightarrow S) \leftarrow X$. ■

Definition 3.26 A *reduction* is a pushing through an associative operator.

Remark 3.7 A normal pushing can be thought of pushing results using an overwrite operator. For example, in a rule $T \leftarrow S$, when writing data to table T, the old results will be erased. Instead if a reduction on T is used, then its old data will be combined with the new one through the specified associative operator.

Definition 3.27 *Head reduction* is defined to be a head copy between a multistore and a store, or a multimap and a map that uses an associative operator. Assume without loss of generality the head unnamed variable is $N \rightarrow A$. Thus to head reduction is to reduce the table $N \rightarrow A$ to the table A. The reduction procedure follows: let $n = \mathrm{dom}(N), a = \mathrm{dom}(A), i = \mathrm{img}(N, n), s = a \bigwedge i, c = \mathrm{pmg}(N, s)$, let $\bigoplus$ be an associative operator and

$\bigcirc$ be an identify, for each key k in key-set s, set $A[k] = \bigcirc$, for each key e in the key-set c, set $A[N[e]] = (N \rightarrow A)[e] \bigoplus A[N[e]]$. The head reduction is a mandatory action for each rule that has reduction.

Definition 3.28 A rule with reduction uses the head reduction, if necessary, instead of the head copy.

Definition 3.29 The *context* of a rule is defined to be the set of keys where the rule can be applied to.

Remark 3.8 Given a rule and its context, the rule body is applied to each individual key in the context. Thus the rule body can be thought of as a function that takes the variables in the rule tail and a key as inputs. Since the KVR tables can only be looked up based on the key column, the key passed into the rule body is implicitly assumed and is not explicitly expressed in the rule definition.

Remark 3.9 There are indeed many ways to define how the context of a rule can be computed. Because the actual context of a rule is not important to the study presented in this dissertation, it is treated as a black box in most places.

Remark 3.10 The current implementation of the Loci framework defines the rule context as the intersection of all the domains of the variables listed in the rule tail. Given a rule $t \leftarrow s_1, s_2, \ldots, s_n$, then the context of the rule is: $\mathrm{dom}(s_1) \bigwedge \mathrm{dom}(s_s) \bigwedge \ldots \bigwedge \mathrm{dom}(s_n)$. The current Loci framework also supplies the rule context to each rule automatically, freeing the programmer's responsibility to supply a correct rule context.

Codelet 3.1 Example Loci Code for Computing Average Temperature

```
1  // type declaration
2  $type avg_temp store<double> ;
3  $type temperature store<double> ;
4  $type neighbors multiMap ;
5  // rule definition (sinature in the parentheses)
6  $rule pointwise(avg_temp<-(neighbors->temperature)) {
7    int sz = $neighbors.size() ;
8    double sum = 0 ;
9    for(int i=0;i<sz;++i)
10     sum += $neighbors[i]->$temperature ;
11   $avg_temp = sum/sz ;
12 }
```

Codelet 3.1 illustrates the actual Loci syntax to define rules for computing the averaged temperature for a cells from all of its neighbors as the example in Figure 3.1. The Loci programming syntax is based on the C++ language with enhanced syntax used to define Loci variables and rules. In Codelet 3.1 the "`$type`" keyword signals declaration of a Loci variable and its type. The type store is defined as a C++ template. Since store is a "$key|value$" table, the first column is implicitly assumed to the type "`int`" since keys are presently all represented by integers. The parametrized type in the template is the value type. KVR maps do not need template since they are "$key|ref$" tables and hence type "`int`" is assumed for both columns. The "`$rule`" keyword is a directive used to indicate the definition of a rule. The "`pointwise`" keyword indicates the rule is a conventional Loci rule whose rule body is to be applied to the table row by row. The enclosed code within the braces from lines 7–11 is the rule body definition. The "`$`" symbol in the rule body signals the variable following it is a Loci variable. Such an access is implicitly assumed to be a look up into the variable based on the currently enumerated key in

the rule context that is passed in to the rule body. The method "`$neighbors.size()`" in line 7 provides the number of rows associated with a key in a multimap. The use of "`$neighbors[i]->$temperature`" provides a mechanism to access all of the rows associated with a specific key in the unnamed multistore.

Codelet 3.2 Example Loci Code for Accumulating Temperature

```
1  $type accu_temp store<double> ;
2  // initialize to an identity value
3  $rule unit(accu_temp) {
4    $accu_temp = 0 ;
5  }
6  // perform a reduction operation
7  $rule apply( (neighbors->accu_temp)
8               <-temperature)[Loci::Summation] {
9    int sz = $neighbors.size() ;
10   for(int i=0;i<sz;++i)
11     join($neighbors[i]->$accu_temp, $temperature/sz) ;
12 }
```

Codelet 3.2 shows an example where a reduction is performed to compute the temperature contribution a cell can have on all of its neighbors. This is an operation in which each cell adds a fraction of its temperature value to all of its neighbors. The "`unit/apply`" definitions in Loci provide a mechanism to setup a reduction. The "`unit`" definition is used to initialize a variable to a unit value (an identity). The "`apply`" defines an associative operator that combines partial values. In Codelet 3.2 this associative operator is the summation operation. The "`join`" clause provides a way of triggering the associative operator while pushing the results.

3.3 Implementation Strategies

An implementation must solve two core problems for a Loci program to run correctly (not necessary optimally or in high-performance). The first problem to solve is to correctly resolve the rule dependence information so that the program has an execution flow. The second problem is to fill in the critical information in the execution flow so that the program can be ready to run. Examples of information considered critical include the context of rules, parallel synchronization check point, and memory management instruction, etc.

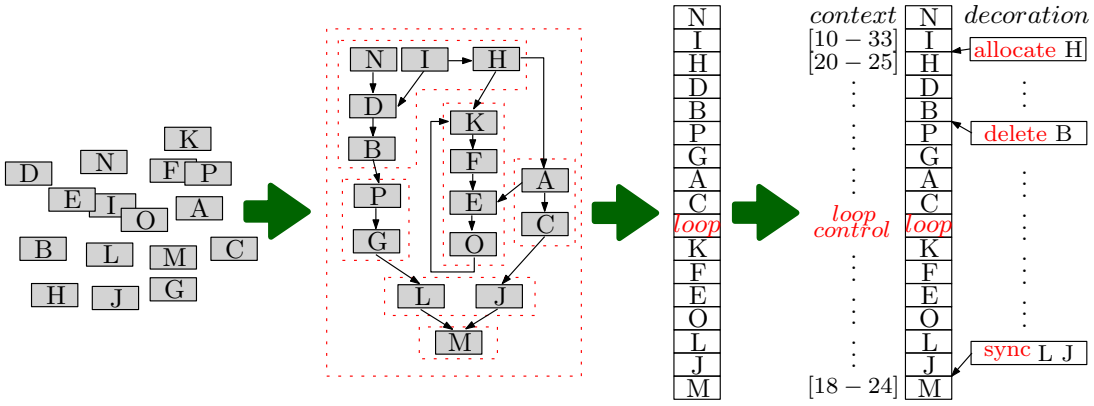

Figure 3.2

Loci Implementation Strategy

Figure 3.2 shows an overview of the major strategies involved in the core implementation of current Loci framework for the KVR model and the rule system. Graph processing and key-set manipulation are the two basic techniques used in the process. Figure 3.2 outlines the four major stages in the current Loci implementation, separated by the large arrows in the figure. The first stage represents the program, i.e., a collection of rules. This

is typically the resulting program a developer supplied to the system. The second stage represents the results obtained from the graph processing analysis. After this stage, all the rules are linked together and form graphs that represent possible functional blocks and execution flow. The graphs obtained at the end of stage two represent many possible flows of execution. Stage three then selects one of the valid execution flow and converts the graph into a list of execution module. The final stage then fills in the needed information based on the analysis of the entire list and any analysis results collected in the previous stages.

3.3.1 Graph Processing and Analysis

A graph is a frequently used model in compiler implementations. In the Loci framework implementation, it is mainly used to model the dependence order for all the rules and the iteration and functional structure for the entire program. Upon scanning all the rules, Loci constructs a dependency graph involving all the rules used by a program. Such a dependency graph is further processed into a hierarchy of simple directed acyclic graphs each represents a functional block (a multilevel graph). Note that not all the nodes and edges in the graph are added at the same time. Some may become part of the multilevel graph at a later time pending further analysis. Graph decoration is a frequently used technique in the Loci framework. It refers to the process of visiting the multilevel graph in a specific order and enhancing the graph by adding, removing, or modifying the existing graph components. For example, the current memory management scheme adopted in the Loci framework is implemented as a series of multilevel graph analysis and decoration.

At the final stage, a graph scheduler will compile the hierarchical graphs into a flat list of execution modules.

3.3.2 Key-set Manipulation

The key-set manipulation is used to resolve all the rule pulling and pushing patterns and also the context for each rule. The manipulation of key-sets is roughly divided into three stages. The first stage begins with the scanning of all the KVR "$\rightarrow$" operators. In a parallel run, such joins will result in tables on a process to be accessed on other remote processes. In this situation, all the unnamed variables in a Loci rule are expanded to include any of the remote data that are not currently on the local process. The second stage renumbers all of the keys on a process to optimize memory allocations. This step is necessary and often indispensable for a parallel run, especially programs that involve unnamed variables. Figure 3.3 shows an example of key renumbering. In the example, the mesh is partitioned into three pieces and each is assigned to a different process. If the original key numbering is retained after the parallel partition, then memory allocation could pose a significant problem. Consider the partition p_0 in Figure 3.3, without renumber, we will need to allocate over $\{0\text{–}10\}$, which wastes the unused space of $\{2\text{–}8\}$. This assumes that the allocation is contiguous. Indeed dynamic data structure such as hash table can be used to avoid the allocation problem. However the run-time performance may degrade significantly. In the next stage, Loci traverses the multilevel graph and performs a series of set operations to build the rule context. This process can also be roughly regarded as solving

a simple Datalog [81] program. Since all the parallel distribution of keys are recorded, the parallel communication schedule can also be generated in this stage.

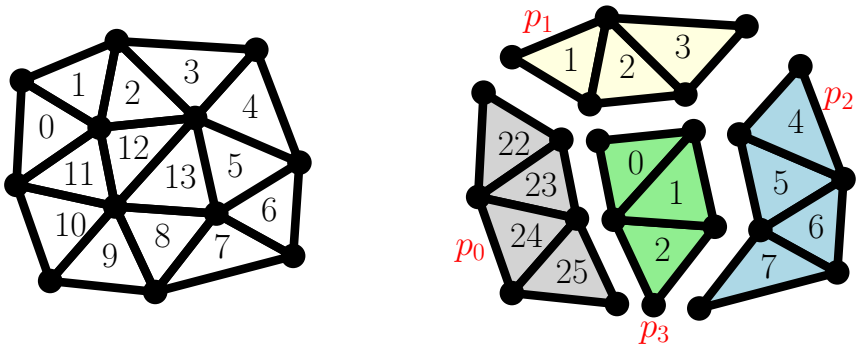

Figure 3.3

Renumbering Distributed Keys

The key-set manipulation presently constitutes the largest overhead in the Loci implementation. Improving its efficiency is still an ongoing research effort. The critical part in the key-set manipulation is however not to improve the performance of the individual set operations. Since the key-set manipulation involves many stages at different time that may be repeated frequently for complicated programs, the most important performance optimization will be to recognize the opportunity to minimize the steps involved in the key-set manipulation and also to aggregate data in a single processing step to take advantage of the set operations. In the current implementation, the key-set manipulation cost is easily amortized as the steps involved tend to be only run once. However improving the performance of this process is critical to the expansion of the application domain of the Loci framework, particularly for dynamic problems.

CHAPTER 4

LAGRANGIAN PARTICLE TRACKING: A CASE STUDY

This chapter provides a detailed study of solving the Lagrangian particle tracking problem on distributed memory computers. The intention of this study is to provide a practical understanding of the utility of the "key-value-ref" model in dynamic CFS problems. The study also serves as a practical case from where observations are made as to what extension and developments are needed for the "key-value-ref" model.

4.1 Problem Setup

The Lagrangian particle tracking model considered here consists of solving ordinary differential equations (ODEs) that describe the Newtonian motion of dispersed particles under the influence of fluid drag forces. The fluid motion is described through the Navier-Stokes equations, which solve for conservation of mass, momentum, and energy of the fluid.

$$\frac{\mathrm{d}x_p}{\mathrm{d}t} = u_p, \quad \frac{\mathrm{d}u_p}{\mathrm{d}t} = D_p(u - u_p) \tag{4.1}$$

The governing equations (Equation 4.1) for the particle movements were derived using the Basset-Boussinesq-Oseen (BBO) assumption that the density of the particle is much larger than that of the fluid and particle size is small compared to turbulence integral length scale, and that the effect of shear on particle motion is negligible.

In Equation 4.1, u_p is the particle velocity, and u is the fluid phase velocity interpolated to the particle location, which is represented as x_p. For the current setting, the body force on the particles such as gravity is neglected for simplicity. The drag force on a solid particle is modeled using a drag-coefficient, C_d: $D_p = \frac{3}{4} C_d \frac{\rho_g}{\rho_p} \frac{|u-u_p|}{d_p}$, where d_p is the particle diameter and C_d is obtained from the nonlinear correlation: $C_d = \frac{24}{\mathrm{Re}_p}(1 + a\mathrm{Re}_p^b)$. $\mathrm{Re}_p = d_p \frac{|u-u_p|}{\mu_g}$ is the particle Reynolds number. The constants $a = 0.15$, $b = 0.687$ are empirically tuned coefficients in the drag correlation.

Since the fluid solver used in this implementation is implicit, it is preferred to choose an implicit Lagrangian equation solver such as Adams-Bashforth backwards differentiation scheme. The one used in this implementation can be expressed in a generic form as:

$$x_p^{n+1} + \sum_{i=1}^{s} \alpha_i x_p^{n+1-i} = \Delta t \beta u_p^{n+1} \tag{4.2}$$

$$u_p^{n+1} + \sum_{i=1}^{s} \alpha_i u_p^{n+1-i} = \Delta t \beta D_p (u^{n+1} - u_p^{n+1}) \tag{4.3}$$

where s is the total number of the multistep scheme, Δt is the time step, and coefficients for the second order scheme ($s = 2$) used in the current implementation are: $\alpha_1 = -\frac{4}{3}$, $\alpha_2 = -\frac{1}{3}$, $\beta = \frac{2}{3}$.

Figure 4.1 shows a snapshot of a simulation of liquid spray in a combustion engine. In the snapshot, a sample of injected droplets are deflected by coaxial swirling flow and are colored and sized by their velocity (the redder and larger a droplet, the greater its velocity). The movement of these droplets and their interaction with the flow in the combustion engine are simulated using the Lagrangian approach. Figure 4.2 shows a cutting plane of

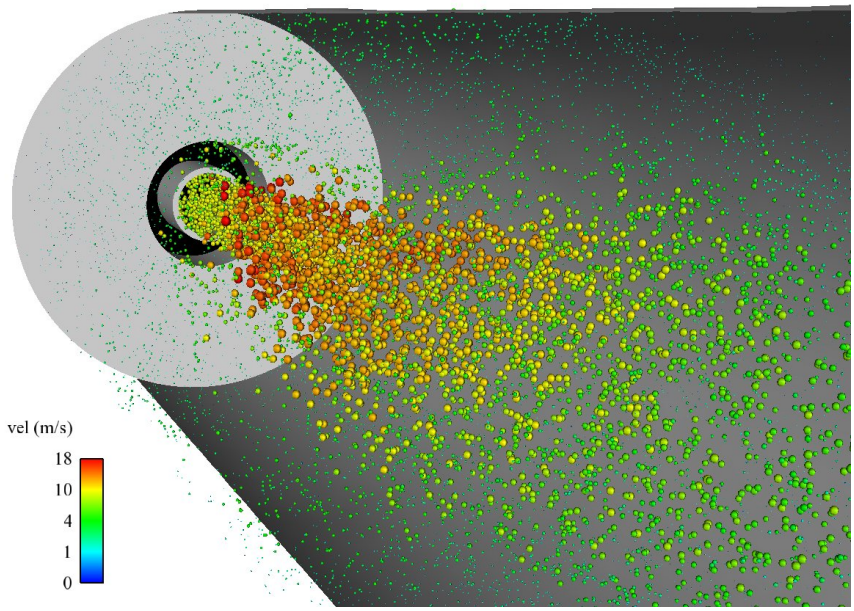

Figure 4.1

Liquid Spray in a Combustion Engine

the simulated combustor model and is rendered with the average particle density on the plane (the redder a region, the denser the particle concentration).

4.2 Available Parallel Formulations

Lagrangian particle tracking is an important example of two-phase disperse flow simulation methodology. It represents a large class of discrete-continuum simulations that involve modeling the time evolution of a large number of discrete elements that communicate with a set of continuum field equations. Some notable uses include from the tracking of electron trajectories in plasma under the forces of electromagnetic fields to tracking droplets coupled to fluid flow. The main objective in Lagrangian particle tracking is to determine the exact physics information of each individual particle based on the particle-

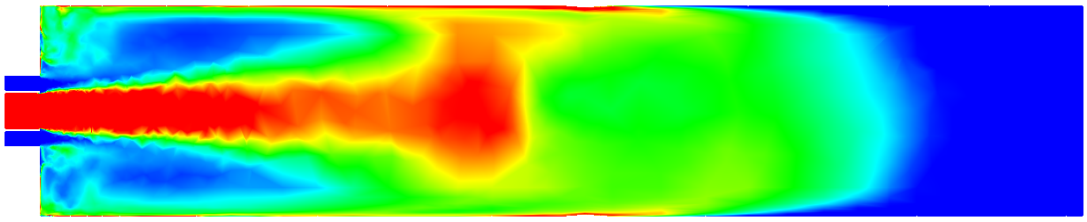

Figure 4.2

Liquid Droplets Density Plot

field interaction. Since such interactions are mostly modeled from interpolated field data, it is therefore required to determine which field cell contains each particle so that the interpolated data can be determined. This usually involves a geometric search procedure on a complex unstructured field mesh. The movement of the particles in the field creates a major difficulty in obtaining an efficient parallel formulation. All the available approaches differ in how to choose the trade-off between an increased parallel computation and an increased parallel communication cost. The formulations can be broadly categorized based on their data decomposition strategies.

4.2.1 The Cell-dec Approach

The cell-dec approach decomposes the field mesh and assigns particles to their containing mesh cells. The process that performs the computation for any given set of mesh cells also performs the computation for particles contained within those cells. Thus the computational load on a process is proportional to the total number of cells and particles assigned to that process. In the cell-dec approach, moving particles in the field mesh amounts to transferring them to new cells that contain the particles. When particles move

into cells owned by other processes, they will be transferred to the corresponding process. One critical aspect of obtaining good performance in the cell-dec approach is obtaining a cell distribution to processes that balances both cell and particle computations. A complication to this approach is caused by the movement, creation, and destruction of particles that typically occur as simulated time advances. As a result, the simulated time evolution of particles can easily defeat any initially optimized data decomposition resulting in severe load balancing problems that can seriously degrade the system performance.

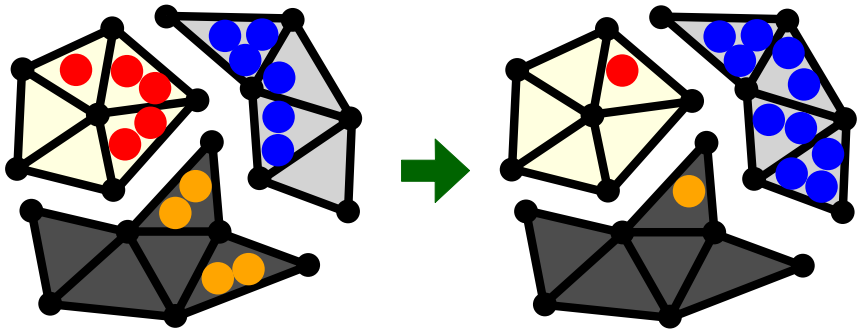

Figure 4.3

Load Balancing Problem in the Cell-dec Approach

For example, Figure 4.3 shows a possible scenario where the particle migration causes a load balancing problem. In the example, the field mesh is decomposed into three parts. Particles are assigned to partitions according to their association with mesh cells. The initial partition is relatively balanced in terms of computational load (assuming that the cost on the cell and particle computations is uniform). At a later time, most particles

migrate to one partition and some exit the mesh. This causes one parallel process to compute most of the particles at that time.

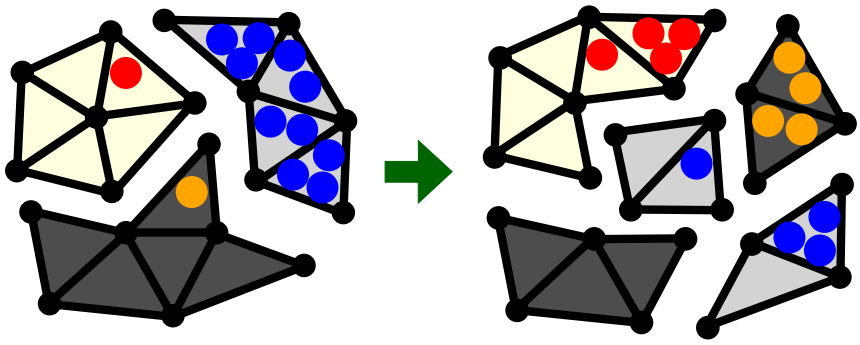

Figure 4.4

Mesh Repartitioning in the Cell-dec Approach

The cell-dec approach has been most commonly used for parallel coupled Lagrangian particle simulation with unstructured field solver. There are plenty recent publications [2, 18, 19, 38, 61] and significant practical applications [17] that employed the cell-dec approach. All of these studies use the the periodic repartitioning strategy to offset the load balancing problem. This repartitioning strategy monitors the computational load on each process. When the load imbalance exceeds a specified threshold, the program is stopped and the mesh is repartitioned taking into account the current particle load. Figure 4.4 shows a possible mesh repartitioning. After the mesh is repartitioned, the program runs on the new decomposition until the next repartitioning step.

There are several approaches to repartition the mesh. Some use a combined weight function that weighs in both cell and particle into the computational load factor [18, 19].

Others [2, 38] use multi-constraint graph partitioning algorithms [45] that try to balance both the cell and particle distributions while also minimizing the interactions among the different partitions.

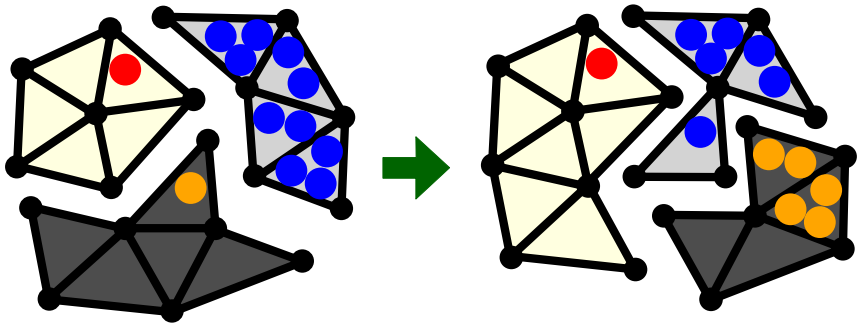

Figure 4.5

Problematic Mesh Repartitioning

The combined weight function approach, although more convenient to compute, can still suffer load imbalance if the weight is not computed carefully. Figure 4.5 illustrates a problematic example. In the figure, the imbalanced mesh is repartitioned using the combined cell and particle number as the weight. It can be seen that the total computational load on each partition is now roughly equal (assuming the cell and particle computational cost is uniform). However since the cell and particle computations exist in different phases, the new partition in Figure 4.5 has severe imbalance in each of the phases involved. Thus the resulting partition will behave poorly despite having a balanced total combined load.

In general, load balancing techniques for multiphase computations need special consideration [84]. To date, multi-constraint partitioning algorithms are generally the best

choice. These algorithms try to balance each individual phase based on multiple weight information. Figure 4.4 is a partition that balances both the cell and particle phases. One drawback of multi-constraint graph partitioning is that it can increase the "edge cut" number. Intuitively this means that the field mesh is decomposed into more disconnected pieces, yielding increased parallel communication cost due to the increased boundaries.

4.2.2 The Dual-dec Approach

The dual-dec approach decomposes both the cell and the particle to parallel processes independently. The process that performs the computation for a given set of cells may not be the one that performs the computation for particles contained within those cells. The dual-dec approach maintains a pointer for each particle to its containing cell (and possibly a list of pointers for each cell to its containing particles) for particle-field interaction. Moving particles in the field merely changes their reference pointers. Therefore the particle movement does not affect the computational load on a process and in general, optimized cell and particle partitions can be predetermined and is immune to particle movement.

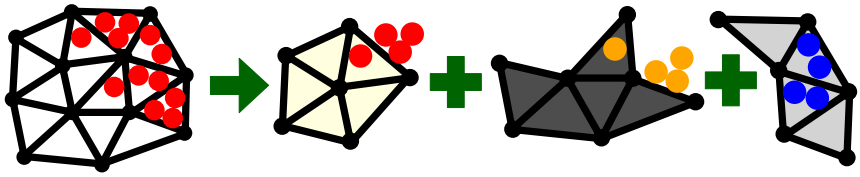

Figure 4.6

Illustration of Partitioning in the Dual-dec Approach

Figure 4.6 shows an example of the dual-dec data decomposition. A drawback of the dual-dec approach can be seen from the figure. Since a particle and its containing cell may be distributed to different processes, the particle-field interaction requires remote data access. In the distributed memory platform, this introduces additional parallel communication overhead. The dual-dec approach is not widely used in engineering applications. There appears to be no documented use of the dual-dec Lagrangian particle simulation with coupled unstructured field solver. The dual-dec approach was used in several cases [14, 16] where the field solver employed a regular Cartesian mesh. In this case, the geometric search required to find the interpolating cell was replaced by an algebraic function that maps between the particle location and cell number, thus simplifying and reducing the cost of parallel communication in particle-field interaction. There has also been investigations to reduce the communication cost in the dual-dec approach by using a space-filling curve distribution of particles [50]. The conceived high cost in parallel communication and software development appear to be the main reason for the limited use of the dual-dec approach in practice.

4.2.3 Other Approaches

There are other effort [24] that employed a particle decomposition with the entire mesh duplicated on all processes. Although such arrangement removes the need to communicate the mesh geometry in the particle location search step, it is not memory scalable. This approach is not feasible in modern particle-field simulations where the field mesh regularly consists of tens to hundreds of million of cells. Others [3] propose a hybrid approach to

improve such memory scalability problem. In this hybrid approach, the field mesh is decomposed into blocks and each block is duplicated on a set of processes. The block parameters can be tuned for different problems to make a trade-off between the memory and communication costs.

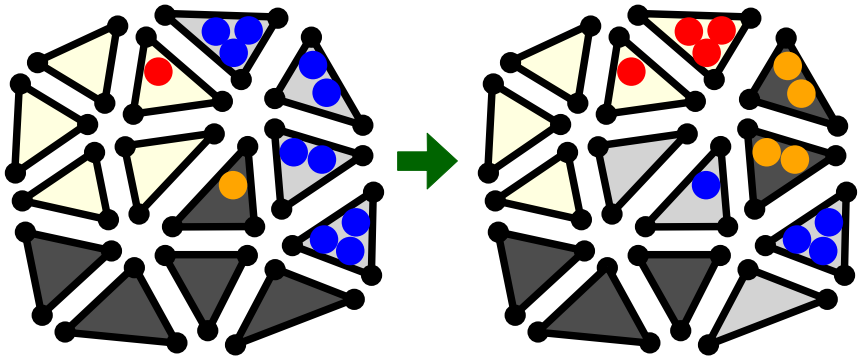

Figure 4.7

Illustration of the Data Migration Strategy

Another alternative approach to developing a particle-field coupled solver would be to implement the solver in a data migration framework such as CHARM++ [42] or DSM-C/MOL [5, 20]. These systems provide mechanisms whereby parallel objects can be migrated among processes to achieve balanced load. In this paradigm, the problem domain is decomposed into multiple medium-grain chunks. Figure 4.7 illustrates such a decomposition. Each process usually holds many of these data chunks. The intelligent run-time system monitors the system performance and in the event of load imbalance, chunks are migrated among the processes to balance workload. This paradigm is an over-decomposition approach that essentially is an advanced version of the cell-dec approach with mesh repar-

titioning. Instead of halting the program and redistributing the mesh geometry completely, this approach dynamically adjusts part of the mesh geometry distribution on the fly and can be considered as an incremental approach. As others [19] have argued that incremental mesh partitioning algorithms can in practice be as expensive as a complete repartitioning, the costs and benefits of the over-decomposition approach is thus not certain. Also since the over-decomposition approach is essentially a variation of the cell-dec approach, it is still prone to many of the problems associated with the cell-dec method.

4.3 Cell-dec and Dual-dec Cost Analysis

The parallel overhead is more tractable in the dual-dec approach than in the cell-dec approach. In the dual-dec approach, the major parallel overhead comes from the parallel communication between particle and the field mesh. In the cell-dec approach, in addition to the load balancing problem, the periodic mesh repartitioning also introduces hidden costs. First the mesh repartitioning in the cell-dec approach may be practically expensive. Repartition the mesh and restarting the program usually indicates that the parallel communication schedule needs to be regenerated. For complex solvers, this cost is usually non-trivial. Second, the mesh repartitioning step in the cell-dec approach usually increases the parallel communication cost experienced by the field solver.

There are other costs in the cell-dec repartitioning technique that are more subtle and harder to quantify such as the effects of the field solver domain decomposition on solution algorithm convergence rates. Often the field solver will employ an implicit time integration scheme that requires solving a large sparse linear system of equations. Typically these

linear systems are solved using preconditioned iterative solution algorithms such as GMRES [71]. Unfortunately, some of the more effective preconditioners (such as incomplete LU (ILU) factorization [70]) do not parallelize well. As a result, it is common practice to apply an approximate blocked preconditioner [25, 75] where the preconditioner is applied independently to the part of the matrix that each process holds. As a result, changing the distribution of the field mesh can reduce the effectiveness of the parallel preconditioner resulting in increased iterations in the linear system solver. Unfortunately the partitions required to balance load in the cell-dec approach tend to degrade the effectiveness of the preconditioner introducing a subtle and hard to quantify cost. Such a problem does not occur in the dual-dec approach as both the field and the particles can choose an optimal partition independently.

Ignoring these hidden costs for the cell-dec approach (they are either hard to model or problem dependent), we can have an estimation for its overhead. Assuming the mesh repartition is effective in detecting the load imbalance, then the redistribution of cells and particles constitutes the major overhead. The all-to-all personalized communication [35] can be used to model such data redistribution. Assuming there are total c cells and p processes, the cell redistribution has a cost of $O(\frac{c}{p})$ if we ignore the communication start-up cost since for large problems, the message size is large enough to become a dominant factor. Assuming there are total n particles, then the particle redistribution has a worst case cost of $O(n)$ when all particles are on a single process. If the load balance detector is effective (i.e., particles are roughly balanced on each process), then the particle redistribution cost can be regarded as $O(\frac{n}{p})$. The communication start-up cost is again ignored in

this case assuming the particle number is large enough. Thus the total cost for the mesh and particle repartitioning is $O(\frac{c}{p} + \frac{n}{p})$ in the cell-dec approach with an effective load imbalance detector.

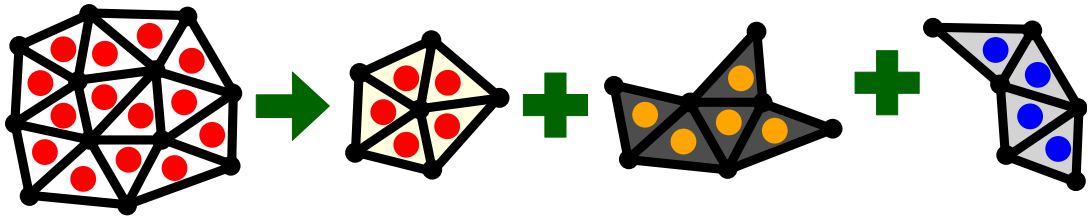

Figure 4.8

An Even Particle Distribution in the Mesh

For the dual-dec approach, the parallel communication between the particle and cell constitutes the major parallel overhead. Assuming there are n particles, p parallel processes, and c mesh cells. The parallel communication overhead is determined by the concentration of particles in the field mesh.[1] There are two special cases. In the first special case, suppose the particles are evenly distributed in the mesh, i.e., every cell has approximately the same number of particles. Figure 4.8 illustrates such a case. This is a trivial case since a particle and its containing cell can be decomposed to the same process. No parallel communication is required in this case for particle-field interaction. Note that the particles will need to be communicated as they move between processes in order to maintain this partition as simulated time progresses. But in this case the communication

[1]Not to be confused with particle distribution on the parallel processes. The particle concentration or distribution in the mesh is determined by the problem's physics characteristics and is not related to parallel computing.

cost will be similar to the communications required by the field solver and can thus be neglected as a separate cost.

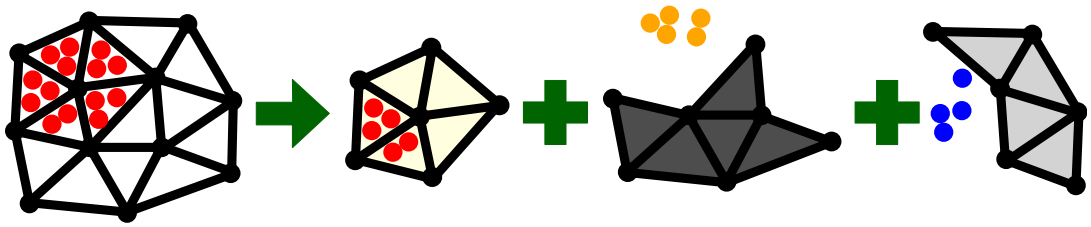

Figure 4.9

A Concentrated Particle Distribution in the Mesh

In the second case, the particles are highly concentrated in a region where their containing cells are all partitioned to the same process, as Figure 4.9 illustrates. Then that single process holds all the cells that contain particles and will be responsible to send all the needed cell information to all other processes for particle-field interaction. This can be performed by using a scatter operation [35] and requires $O(\log p) + O(c_s)$ time on any network with a bisection width of $O(1)$ [35], where c_s is defined as the number of cells that contain particles.

Note this result is dependent on the assumption that all of the particles contained in any given cell are partitioned to the same process. This data locality requirement is critical in minimizing the parallel communication cost. Figure 4.10 shows the effects for spatial and random particle decompositions. In the left case, particles are distributed to processes according to their spatial locality. As a result, most particles contained in the same cell are assigned to the same partition. This means that the mesh partition will only need to

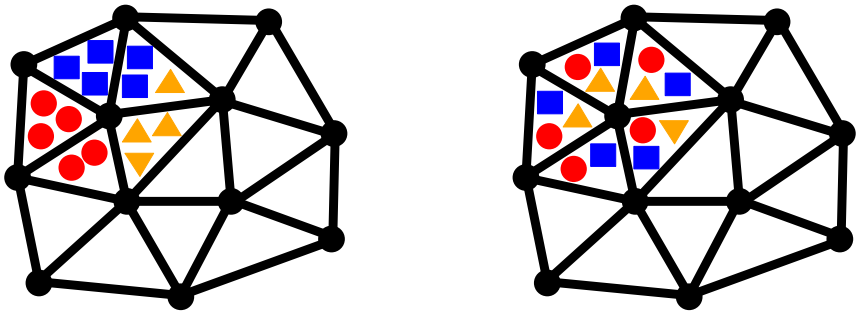

Figure 4.10

The Effects of Spatial Locality in Particle Distribution

send a cell to *one* particle partition. In the right case in Figure 4.10, particles are randomly distributed to processes. As a result, the field mesh partition will have to send a cell to *every* particle partition. This is an $O(1)$ versus $O(p)$ cost difference.

In the general case, the particle concentration in the mesh is between the two special cases outlined previously. The communication required to complete the particle-cell communication is a many-to-many pattern, as illustrated in Figure 4.11. This communication pattern has been studied by several investigations [43, 66]. However none of these studies gave complexity analysis for large messages when network contention is considered. Since many-to-many personalized communication is a subset of the all-to-all personalized communication, its worse case analysis can be based on the all-to-all personalized communication, which is more widely studied on various network architectures. On a network with bisection width of $O(p)$, the all-to-all personalized communication can be performed within $O(s)$ time (with a large message assumption, the start-up cost for the message can be ignored), where s is the total volume of messages sent and received in all steps from a single process. On networks with much lower connectivity such as a ring, the bound is

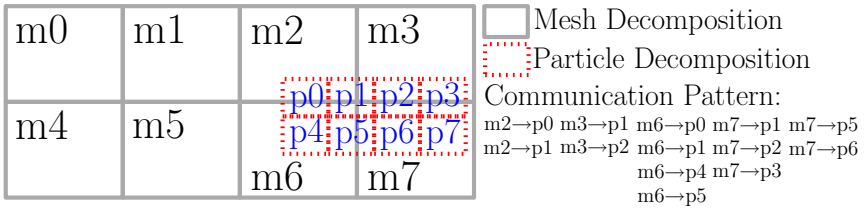

Figure 4.11

Communication Pattern in a General REF-PLPT Example

worse due to network contentions. The optimal time on a ring is $O(ps)$ [48]. However for practical architectures such as switched Ethernet, there exists scheduling algorithms [26] that can map communications in contention free phases. Also on many network architectures, the bisection width will typically grow as the processor number increases.

With these results, the cost of this many-to-many communication on modern high-performance communication systems where the bisection width grows $O(p)$, the communication time can be approximated by $O(\max(c_s, c_r))$, where c_s is now generalized to become the maximum number of cells sent by any mesh partition and c_r is the maximum number of cells received by any particle partition. We can also note that when mesh and particles are distributed evenly to processes, and we assume that all particles contained in a cell are allocated to the same process, then bounds exist on the communication cost given by $c_s \leq \frac{c}{p}$ (we can at most send the cells we have on a process) and $c_r \leq \frac{n}{p}$ (the worse case is that each particle is in exactly one cell).

If we assume cells have a uniform distribution to processes, then the term c_s is bounded by the parallel work of the field solver, while the overhead incurred by c_r cannot exceed the particle's parallel work. Thus under the view of the isoefficiency metric [36], the dual-dec

approach is scalable since the parallel overhead cannot grow asymptotically faster than the parallel work. The effectiveness of the scheme rests on determining if the constant factors associated with this communication cost remain sufficiently small as to not dominate parallel efficiency. This is best evaluated through empirical means.

We can notice that the asymptotic cost of the communications step in the dual-dec approach has similar bounds to the cost of the repartitioning step in the cell-dec approach. In fact, if the cell-dec approach requires a repartition step at a bounded rate, then the asymptotic costs of the dual-dec approach is no worse than that of the cell-dec approach. Thus the two methods can be regarded as having asymptotically the same overhead for highly dynamic particle-field simulation problems.

4.4 Practical Implementation

This section will focus on several higher level strategies in implementing the particle location search and the particle-field interaction in the dual-dec approach. The cell-dec approach relies on the mesh repartition to achieve good performance. Efficiently repartitioning the mesh and regenerating all the communication schedule is related to the actual data structure choice and the problem characteristics.

4.4.1 Particle Location Search

The objective of the particle location search is to find which cell contains the particle, given its position. This involves two decisions. First a predicate is needed to determine whether or not a particle is within a given cell. Second, an algorithm is also required to

examine a sample of cells to decide which one contains the given particle. The predicate needs to be robust and the algorithm needs to quickly filter unpromising cells (reducing the search space).

Algorithm 4.1 Particle In Cell Predicate

Require: particle info. (p), cell geometry ($cell$)
Return: walk (**true**), stay (**false**), target cell in case of walk

```
1: for all f in cell.face do
2:     v1 ← p.pos − f.center
3:     v2 ← f.normal
4:     d ← dot(v1, v2)
5:     if d > 0 then
6:         return (true, neighbor(cell, f)) // the neighbor that shares f with cell
7:     end if
8: end for
9: return (false, ⊥) // all dot products non-positive
```

The projection based predicate is usually a good choice. The left part in Figure 4.12 shows an example. Given a particle's position and the geometry of a cell, we can compute the dot product of the normal of each edge (face in case of 3D) and the vector pointing from the edge center to the particle position. The particle is within the cell if all these dot products are non-positive. This method is usually preferred because it is more robust for irregular geometry than other approaches such as the partial volume method [2]. The algorithm for the projection predicate is shown in Algorithm 4.1. In the algorithm, it returns a pair of data. The first is a flag indicates whether the particle is still in the cell. In the case of not, then the target cell the particle should search next is also returned.

The "walking" algorithm is preferred as the search methodology in unstructured mesh. Since we track particles and know their previous locations, it is much faster to initiate

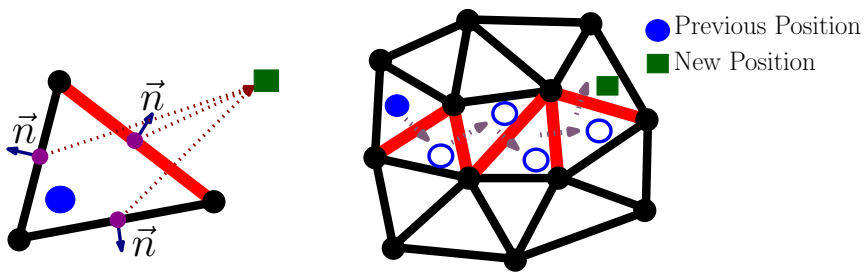

Figure 4.12

Particle Location and the Search Algorithm

Algorithm 4.2 Walking Algorithm for Single Particle

Require: particle info. (p), cell geometry ($p.cell$)
Ensure: particle in the correct cell, or removed (p)

```
1:  loop
2:      (walk, cell) ← in_cell(p, p.cell)
3:      if walk then
4:          p.cell ← cell
5:      else
6:          if cell is boundary then
7:              remove p  // p falls out of boundary
8:          else
9:              return   // located
10:         end if
11:     end if
12: end loop
```

a guided search from the previous position. In typical applications, particles also move slowly in favor of the numerical stability. Thus its new position is usually nearby the previous one. Given a particle's previous containing cell, we can use the in-cell predicate just discussed to determine if the particle is still inside. If not, we then move to the neighbor cell who shares the edge that generates a positive dot product in the predicate test. In this way, we are moving closer to the particle's actual location and will eventually converge to its new position. Figure 4.12 shows an example walk in the right part. All the edges crossed are highlighted in the figure. Such guided search is more effective than a tree based spatial search algorithm if we have the knowledge of the previous position. It is also referred to as the "known vicinity neighbor-to-neighbor" search algorithm [51].

The walking algorithm is shown in Algorithm 4.2. It repeatedly calls the in cell predicate until a true containing cell is reached. In the dual-dec approach, at each step the remote mesh topological geometry needs to be communicated to the local process. The cell-dec approach always has particles and their containing cells on the same process, thus no communication is needed. The communication point for the remote mesh geometry is usually placed before line 2 in Algorithm 4.2. When locating a list of particles, the walking algorithm is best to be performed step by step for all of the particles instead of locating one by one completely. Since particles usually exhibit spatial locality in the mesh, step by step processing can reuse much of the mesh topology in the search process. For both the cell-dec and dual-dec approaches, this saves the frequency of communication and encourages data structure reuse. The algorithm for locating particle list is illustrated in Algorithm 4.3.

Algorithm 4.3 Walking Algorithm for Particle List

Require: particle list info. ($plist$), cell geometry
Ensure: particles in the correct cell, or removed ($plist$)

```
1:  p_walk ← plist  // initially all particles need walk test
2:  while p_walk not empty do
3:     p ← p_walk.front()
4:     (walk, cell) ← in_cell(p, p.cell)
5:     if walk then
6:        p.cell ← cell
7:     else
8:        if cell is boundary then
9:           remove p  // p falls out of boundary
10:       else
11:          remove p from p_walk  // located
12:          append p to p_located  // move to p_located list
13:       end if
14:    end if
15: end while
16: swap(p_walk, plist)
17: if plist is empty then
18:    return
19: else
20:    goto line 1
21: end if
```

4.4.2 Incremental Data Caching

An inherent cost in the dual-dec approach is the parallel communication of remote mesh geometry within the walking algorithm. An "incremental" data caching scheme is an effective method to reduce the size of the data in parallel communication as much as possible. Incremental data caching retains the previously communicated cell geometry such that the data can be reused instead of recommunicated in future searches. Reuse of this cell topology data is typically high since a region of mesh topology tends to be shared by many particle location searches in a number of consecutive time steps. The cache size

can be limited so that a process can store in its cache to a preset value to prevent memory overflow. The communication routine will keep caching data until the limit is reached, after that, data can be replaced using a least recently used policy.

4.4.3 Particle to Process Decomposition

In addition to partitioning the mesh, the dual-dec approach also needs to partition particles. Since particles have a different topology than the field mesh, it is best to partition the particles differently. Since in section 4.3, it is shown that the spatial locality is important in particle to process distribution to minimize parallel communication cost, a partitioning strategy must preserve the coherence of the particle positions. The Orthogonal Recursive Bisection (ORB) method is a good candidate. The ORB method constructs axis-aligned boxes recursively on each dimension and thus particles fall in the same box are spatially close. Methods that utilize space-filling curves could potentially be more scalable, particularly those based on Hilbert indexing [50].

Since particles can be created or destroyed during the simulation, the particle distribution could become imbalanced in the dual-dec approach as the simulation progresses. In addition, since different particles could take on completely different trajectories, the originally imposed partition will eventually become ineffective at balancing load and minimizing communication needs. To address these issues, particles should be repartitioned periodically when these conditions occur. Load imbalance can be detected by computing the maximum and average number of particles per process. Since the maximum number is representative of the actual parallel computation time while the mean is representative

of the ideal parallel computation time, the current efficiency of the particle computation can be estimated by examining these two numbers. Whenever the measured efficiency dropped below a given threshold , repartitioning would be performed. The diffusion of particles in the field on a process is much harder to detect efficiently. One way to avoid the potential cost of such detection is to simply impose a particle to process redistribution at a preset interval.

4.5 Performance Comparison

This section provides a general performance evaluation that compares the cell-dec and the dual-dec approaches in a realistic engineering application of sufficient complexity. Since there is no currently available document on the performance study of the dual-dec approach coupled with complex unstructured field solver, the purpose of such an evaluation is to determine the general competitiveness and effectiveness of the dual-dec approach. However the intention for this evaluation is not to perform an extensive and comprehensive performance study for the dual-dec approach, which is likely to be problem dependent. The purpose of this evaluation is to determine is there any fundamental reason that prevent the dual-dec approach from being used in practice? Section 4.3 shows for unsteady state particle-field simulations, the cell-dec and the dual-dec approaches indeed have the same asymptotic overhead. This section serves to answer whether or not in practice this result will hold.

The application used in the evaluation in this section is a combustion engine spray simulation. A snapshot of such a simulation is already shown in Figure 4.1. In the sim-

ulation, particles are injected into the combustor chamber through the center of a coaxial injector. Swirling flow is injected in the outer coflow region of the injector. This swirling flow, through particle-fluid coupling, causes the particles to flow in a complex pattern where they become slung out towards the combustor walls and entrained in recirculating flows. The low speed fluid flow is simulated using a preconditioned low-Mach formulation Navier-Stokes solver called Chem code (developed using the Loci framework) [55, 56]. Both of the cell-dec and the dual-dec methods for Lagrangian particle tracking are manually implemented and coupled with the Chem code. The baseline physics model for the particle is the one outlined in section 4.1. The geometric search algorithm used is outlined in Algorithm 4.3.

In order for the evaluation to be realistic and representative, the linear system solver in the Chem code is set to run a fixed number of iterations regardless of the residual error reduction. Thus, changes in mesh decomposition do not cause a change in performance, but instead cause differences in the solution error. Therefore the measurements do not include the subtle effect of partition on solver algorithm performance. Since these effects will be highly problem dependent, including them in this measurement would reduce the general applicability of the results.

The implementation of the cell-dec method follows the mainstream strategy [17, 38]. The PARMETIS package [46] is used to perform a multi-constraint graph partitioning upon the load becomes imbalanced. The repartitioning of the code is implemented as a restart as in other major applications [17]. During a restart point, a set of solution, mesh, and particle distribution files are written. The program can be stopped at anytime (e.g., when load is

imbalanced) and restarted from the these restart files. Upon restart, a new balanced partition will be generated. The implementation of the dual-dec method follows the discussion in section 4.4 and uses the incremental caching and the ORB partitioning strategy.

In the performance measurement setup, I have timed 100 Chem iterations for both the cell-dec and dual-dec codes. Both of them started from the same initial conditions that were generated by running the simulation until the particle number is sufficiently large and particles are sufficiently developed in the combustor chamber. The reason for this setup is to avoid the need for restart in the cell-dec code since the cost associated with the restart depends on the real application and particular implementation strategy, and it is hard to quantify the cost accurately. In reality, when a simulation has large number of particles sufficiently developed, it will not be restarted frequently to avoid the high cost (even if the restart is implemented by automatic run-time decision without I/O cost). Such a setup also avoids the need for particle redistribution in the dual-dec code. Therefore this setup measures the best potential for both codes for the same problem. The result is more representative to evaluate the essential cost in both codes.[2] In addition, the timing

[2]The reasoning of this claim follows: Suppose the cost (time) of cell-dec code is $A + B$, in which A is the cost of restart and B is the rest of the cost. While the cost of the dual-dec code is $C + D$ total, where C represents the particle redistribution cost and D represents the rest of the cost. In the cell-dec approach, the A cost is a variable factor, it is implementation dependent. Moreover the choice of restart frequency will impact A and B either. Too frequent restart will increase A and decrease B, while too sparse restart will decrease A and increase B. It is hard to know the optimal choice of frequency of restart that will minimize $A + B$. For the dual-dec code, the C part in the cost is a variable factor since the decision to perform the redistribution is arbitrary. By choosing a problem with large particle number and a short run, we can eliminate the A and C costs, and therefore we can measure the best potential for both codes, which is a more fair and representative comparison. In other words, this setup measures whether the unavoidable communication cost in dual-dec code is worth to trade for the unavoidable imbalanced computational cost in the cell-dec code in this problem. Note the unavoidable communication cost in the dual-dec code and the unavoidable imbalanced computational cost in the cell-dec code is problem dependent. However we cannot be problem independent. Such a setup measures a fair comparison for this problem and can give insights into such a trade-off.

measurement I have gathered is the wall time measurement that includes virtually everything, e.g., the fluid, particle, and the initial I/O operation[3]. Note we cannot measure and compare the individual components in both codes if the goal is to evaluate the overall performance characteristics and the trade-off. While the unavoidable communication cost in the dual-dec code can probably be singled out, the unavoidable imbalanced computational cost in the cell-dec code is much harder to separate. Moreover the wall time is what is important in practice. Ideally we would like to measure and compare the optimal run-time for both the cell-dec and the dual-dec code for this problem. However we do not know how to configure the optimal run-time. While one possible way is to explore a large set of experiments with different configurations, the setup used here directly measures the unavoidable costs in both codes and is a reasonable approximation that dramatically reduces the experimental exploration space. This approach is also more aligned with practice because it is not feasible to try every configuration in a production run to get good performance. People will most likely estimate the variable parameters and run the program. Moreover the purpose of the experiments is to find a case that supports the usefulness of the dual-dec approach while trying to be as objective as possible.

I have chosen to use nine mesh and particle combinations. The mesh is progressively refined to have roughly two, eight, and fifteen million cells. While the particle number is increased by varying the injection rate to produce roughly one, six, and fourteen million particles in the simulation. In this way, the spatial distributions of particles follow roughly

[3]Note as a complete run, the initial I/O is required to correctly read in the configuration. This cost is included in both codes and should be the same since the implementation used is the same. Thus its inclusion does not reduce the representativeness of the measurement.

the same pattern. Two computing platforms are used in the measurements. The first is a large-scale diskless Linux cluster with SUSE Linux Enterprise Server 9. The cluster is composed of 512 Sun Microsystems SunFire X2200 M2 servers, each with two dual-core AMD Opteron 2218 processors (2.6GHz) and 8GB of memory. The system is interconnected using Gigabit Ethernet. Up to 32 nodes (128 processors) are used on the system for the measurements. The other platform is an SGI Origin 3900 with 480 700MHz MIPS R16000 processors and total 480GB of memory, running IRIX 6.5. Up to 384 processors are used for the measurements. The cluster results are included to show the code performance on a popular modern architecture with a modest communication infrastructure, while the SGI Origin results are included to measure the scalability of the codes on an architecture with a favorable computation to communication balance. The SGI architecture results show the potential for the codes on parallel architecture with a high-speed and low latency network.

Table 4.1

Particle Code Linux Cluster Timing (Mesh size = 2M)

particle size	1.1M		5.7M		13.3M	
process	cell[a]	dual[b]	cell	dual	cell	dual
8	1863[c]	1682	2247	2095	2848	2755
16	978	911	1166	1174	1588	1559
32	574	544	705	689	918	885
64	341	370	427	505	580	592

[a]Denotes the cell-dec code.

[b]Denotes the dual-dec code.

[c]All results unit is second.

Table 4.2

Particle Code Linux Cluster Timing (Mesh size = 8M)

particle size	1.2M		6.0M		14.5M	
process	cell	dual	cell	dual	cell	dual
32	1915	1726	2098	2033	2528	2460
64	1086	1013	1196	1242	1618	1495
128	657	642	735	805	1038	1068

Table 4.3

Particle Code Linux Cluster Timing (Mesh size = 15M)

particle size	1.2M		6.1M		14.6M	
process	cell	dual	cell	dual	cell	dual
64	1876	1759	2046	1936	2451	2180
128	1110	1048	1175	1215	1383	1522

Tables (4.1 – 4.3) show the timing results on the Linux cluster. Note not every combination is run under the same set of processors. This is due to the memory constraint on the Linux cluster node. From these timing results, we can see that for some cases, the cell-dec code outperforms the dual-dec code, while in some cases, it performs worse. In general, the timing difference is not significant. This suggests that the two codes perform approximately the same for these problem configurations on the Linux cluster. We can then conclude that for this problem case, the unavoidable parallel communication cost in the dual-dec code is roughly equal to the unavoidable load imbalance cost in the cell-dec

code. This is a good indication that the dual-dec code is actually very usable in a practical production run. In fact, if the restart cost in the cell-dec code cannot balance the particle redistribution cost in the dual-dec code, then the dual-dec code can be superior in this case.

Table 4.4

Particle Code SGI Origin Timing (Mesh size = 2M)

particle size	1.1M		5.7M		13.3M	
process	cell	dual	cell	dual	cell	dual
8	6046	5228	7174	6698	9355	8643
16	3182	2495	3684	3443	4950	4713
32	1590	1297	1955	1731	2791	2476
64	804	671	1141	1053	1685	1443

Tables (4.4 – 4.6) show the timing results on the SGI Origin machine. Although there is no memory constraint on this platform (because it is an SMP machine), the same problem-processor combination is measured in order to be consistent with the Linux results. However since I am able to use more processors on the SGI machine, I also measured the largest mesh configuration on 256 and 384 processors. The SGI timing results suggest that the dual-dec code is outperforming the cell-dec code for all the measured entries in the tables. This is a reliable sign that indicates the performance of the dual-dec code is generally superior than the cell-dec code on the SGI machine for this problem. Based on the Linux cluster results, this is to be expected since the SGI machine has a faster network interconnection, which would reduce the parallel communication cost in the dual-dec code. This shows that on a high-performance network infrastructure, the communication cost in the

Table 4.5

Particle Code SGI Origin Timing (Mesh size = 8M)

particle size	1.2M		6.0M		14.5M	
process	cell	dual	cell	dual	cell	dual
32	6297	5113	7141	6082	7910	6808
64	3630	2446	4122	2985	4713	3829
128	1921	1439	2254	1700	2668	2094

Table 4.6

Particle Code SGI Origin Timing (Mesh size = 15M)

particle size	1.2M		6.1M		14.6M	
process	cell	dual	cell	dual	cell	dual
64	6655	4550	6704	5100	7055	6121
128	3728	2505	3763	2750	4065	3191
256	1969	1455	2110	1700	2745	1949
384	1506	1319	1786	1409	2078	1774

dual-dec code can possibly become smaller than the tolerable computational imbalance cost in the cell-dec code. Though the testing results only support the particular combustor problem case, it at least provides an example where the dual-dec code can be useful and can actually be tuned to outperform the cell-dec code.

Table 4.7

Final Particle Number Distribution (Mesh size = 2M)

particle size		1.1M		5.7M		13.3M	
process		cell	dual	cell	dual	cell	dual
8	mean	145519	145566	726703	726952	1697549	1697878
	max	202322	146848	985702	734762	2150672	1715189
16	mean	72788	72784	363393	363389	848991	848904
	max	137621	73401	496510	367238	1423203	857581
32	mean	36355	36374	181658	181731	424434	424417
	max	70584	36845	373084	183636	843775	428685
64	mean	18202	18199	90880	90847	212174	212192
	max	50779	18698	176003	91971	604504	214621

As a mean to understand the problem characteristics, I also counted the particle distribution at the end of the 100th Chem code iteration (the last iteration in the measurements) for all the combinations in the previous measurements. The result is presented in Tables (4.7 – 4.9). The particle numbers on each process within the dual-dec code is to be expected. The mean and max value do not in general deviate too much and the difference[4] is so small that they can be ignored. The final particle distribution in the cell-dec code is

[4]The only dual-dec case that did a run-time particle redistribution is the 1.2M particle with 15M mesh case on 384 processors on the SGI machine. All other cases did not perform any automatic run-time particle distribution. Thus the numbers presented are a true indication of the load balancing quality for the dual-dec code.

Table 4.8

Final Particle Number Distribution (Mesh size = 8M)

particle size		1.2M		6.0M		14.5M	
process		cell	dual	cell	dual	cell	dual
32	mean	37747	37749	191858	191883	463321	463313
	max	94171	38252	432992	193310	1734438	466119
64	mean	18885	18884	95911	95937	231658	231645
	max	65799	19336	430177	96832	1638682	233191
128	mean	9433	9435	47968	47965	115829	115824
	max	66284	9724	387081	48469	1351579	116808

Table 4.9

Final Particle Number Distribution (Mesh size = 15M)

particle size		1.2M		6.1M		14.6M	
process		cell	dual	cell	dual	cell	dual
64	mean	19015	19013	97673	97681	233636	233707
	max	65047	19416	449239	98360	1591527	234896
128	mean	9509	9504	48830	48845	116851	116852
	max	33373	9777	386858	49222	1012322	117655
256	mean	4754	4752	24418	24418	58416	58426
	max	20246	5140	539646	24861	659531	59046
384	mean	3169	3168	16280	16283	38944	38947
	max	16593	3275	441152	16420	553083	39487

however much worse. For some cases, the max value is over ten times larger than the mean value. Recall that the mean value represents the ideal particle load, and the max value is an indication of the imbalance factor. This showed the particles are severely imbalanced in the end for many of the tested cases in the cell-dec code.

This is an indication of the dynamic nature of the problem used in the measurement. This may also suggest that more sophisticated partitioning algorithms are needed for the load balancing restart in the cell-dec code. The multi-constraint partitioning algorithms (the PARMETIS package) used in the cell-dec code only takes into account the static information at the partitioning time. Unlike the single-constraint graph partitioning algorithm, a multi-constraint graph partitioning algorithm usually produces partition on each process that has many disjoint pieces (more edge cuts). Such a result is to be expected due to the multiple constraints imposed. However for the particle problem, this may produce partitions that can cause large number of particles to quickly escape some processes due to the irregular mesh shape and connectivity (e.g., there may be many holes and disconnections in the local mesh geometry, effectively increasing the mesh boundary significantly on each process). This can quickly lead to a new imbalanced load that defeats the optimal partition produced not long ago. Instead if the partitioning algorithm can take into account some of the particle trajectory information and can compensate particle movement by adjusting the partitioning accordingly, then the load imbalance may develop more slowly. In other words, we probably need to also treat the particle movement information as an additional constraint in the mesh partitioning process. However incorporating such dynamic informa-

tion may be hard. For example, finding a proper representation for such "velocity weight" in an algorithm can be a challenge.

Table 4.10

Linux Cluster Timing (Mesh size = 8M, 2nd Chem Iteration Only)

particle size	1.2M		6.0M		14.5M	
process	cell	dual	cell	dual	cell	dual
32	19.80	17.72	20.62	23.17	22.42	24.00
64	11.23	9.79	11.97	11.84	13.40	13.92
128	6.17	6.25	6.96	7.43	7.50	8.51

One additional measurement is performed to provide an example for the potential performance of a nearly balanced cell-dec code versus the dual-dec code. Table 4.10 shows a measurement on the 8M mesh with all the particle combinations where the timing is only measured for the second Chem iteration (which includes the fluid solver time with the particle code time). Table 4.11 shows the corresponding mean and max particle distribution for the problem. Since this is at the very beginning of the simulation run, even the cell-dec code should have a nearly balanced particle load as suggested by Table 4.11. The timing results in Table 4.10 however suggest that both of the codes perform at the same level. In some of the cases, the dual-dec code is even faster than the cell-dec code. Such a performance behavior may be caused by several factors. For instance, since the cell-dec code uses the multi-constraint mesh partitioning that increases the partition edge cut, the fluid solver may take different time in each of the code. Identifying and analyzing the performance characteristics of individual component and their interaction is beyond the goal of

Table 4.11

Second Iteration Particle Distribution (Mesh size = 8M)

particle size		1.2M		6.0M		14.5M	
process		cell	dual	cell	dual	cell	dual
32	mean	36913	36911	187956	187956	453806	453789
	max	39259	36926	200779	187985	478473	453846
64	mean	18456	18455	93980	93979	226895	226899
	max	22876	18463	107132	93996	254464	226928
128	mean	9227	9227	46988	46990	113453	113451
	max	11661	9234	60023	46999	140240	113472

this dissertation. However these two tables show that the trade-off between the two codes can be complicated.

Finally Table 4.12 shows a rough estimation of the restart cost in the cell-dec code and the run-time particle distribution cost in the dual-dec code for the 8M mesh configuration. The restart cost estimation includes the mesh and particle data structure reading I/O cost with the fluid solver start-up cost (the time to generate a Loci schedule since the fluid solver is implemented using the Loci framework). However such an estimation is highly implementation dependent. Also change of strategies such as avoiding the use of I/O and Loci framework rescheduling by using other approaches such as the mobile objects [20] can vary the restart cost, possibly reducing it by a large margin. Compared to the program execution time in Table 4.2, the cell-dec code restart cost is quite high. It is clear that for this configuration, we cannot afford to increase the restart frequency. The particle distribution time in the dual-dec code, while not negligible, is however quite small.

Table 4.12

Estimated Restart and Particle Dist. Cost on Cluster (Mesh size = 8M)

particle size	1.2M		6.0M		14.5M	
process	cell	dual	cell	dual	cell	dual
32	236.5	0.48	247.5	4.31	298.9	11.24
64	154.6	0.50	166.6	2.35	190.6	6.60
128	195.0	0.63	219.8	2.03	199.7	4.43

4.6 Summary

This chapter presented a detailed study of the two major parallel formulation approaches, the cell-dec and the dual-dec approaches, for Lagrangian particle tracking simulation coupled with an unstructured field solver. The cost analysis suggests that for unsteady state particle-field simulation problems, the theoretical parallel overhead within the cell-dec and the dual-dec code is on the same asymptotic order. A practical performance comparison based on a realistic engineering application confirms this result. The performance benchmark also suggests the dual-dec approach could be more effective than the cell-dec approach on a high-performance network infrastructure. This is a direct suggestion for the utility of the dual-dec approach in practical engineering application.

CHAPTER 5

THE KEY-SPACE CONCEPT

5.1 Reflection on the Key-value-ref Model

Given the case study on the Lagrangian particle-field simulation, what can be learned as the necessary components in its software development? And what is needed in the "key-value-ref" model to support the development? From a designer's point of view, the essential questions would be: 1) What solution approach to use? 2) How are the particle and field equations coupled? 3) How is the software structured that can support a reasonable implementation and flexible future extension and maintenance?

The case study suggests that the cell-dec and dual-dec approaches are both effective. The manual formulation of these two approaches is largely driven by the need to achieve good parallel performance and to make a trade-off between the parallel overheads. In the KVR model, however, these two approaches are more naturally formulated. Since both particles and cells can be enumerated, then they both can be keys, or one of them be keys and the other one be the values. The cell-dec approach corresponds to a design where the cell is key and particle is the value associated with the cell. The dual-dec approach corresponds to a design where both the cell and particle are keys. Particle-field interaction easily follows in the design. In the KVR cell-dec design, since particle is value associated

with cell, who is a key, then cell and particle will be grouped in the same table. Interaction in this case is therefore trivial. In the KVR dual-dec design, since the normal method keys can interact is through a ref components, therefore we can establish a "$key|ref$" table for either the particle or the cell, or both. A question that follows this design is: Is there any difference in choosing a particle to cell ref, or a cell to particle ref, or both?

The basic KVR program structure is through rule composition, which provides a highly flexible software architecture. This allows the program behavior to be extended, modified, or customized all without the need to change the existing software base. The case study also shows that the critical component in the cell-dec design is the mesh repartitioning and the critical component in the dual-dec design is to determine an appropriate particle-to-process distribution. Both components rely on the parallel computing concept, which the current KVR model lacks since it is an implicit parallel model. While an implicit parallel model is often desirable, experience shows (as the review in chapter 2) that it is often also ineffective. However we do not want to extend a model that has overly complex parallel concepts that could complicate the design decisions. Since the particle-field simulation case study suggests that data distribution and balancing is the key to achieve good parallel performance, it is helpful to incorporate this concept into the KVR model. Since the mesh repartitioning decision in the cell-dec approach and the particle-to-process distribution in the dual-dec approach all relate to the concept of key distribution and grouping on parallel processes, it suggests that the ability to represent key group is critical in the KVR model.

This chapter develops the "key-space" concept in the KVR model and show that it addresses these critical software design and structuring problems. Along the way, design

questions such as the ref design in the dual-dec representation in the KVR model outlined in the previous discussion are also clarified. It is shown that indeed different ref designs do make a difference and that the key-space concept is useful in guiding a designer to reach a good decision.

5.2 The Key-space Concept

The "key-space" is the major development in this dissertation for the KVR model. It is a mechanism for describing the grouping of keys in the KVR model, and this allows different groups of keys to be managed differently. This is indeed a concept that only adds one additional attribute to each key in the KVR model, and yet from the lessons learned in the case study on the Lagrangian particle-field simulation, this simple categorization of keys is all that is needed to support the modeling of a dynamic CFS simulation and to distinguish the strategies in different solution formulations. Under the key-space concept, the KVR model presented in chapter 3 (the original KVR model) can be regarded as a single anonymous key-space with no operators and functions defined on the key-space level. Therefore the original KVR model is a special case of the KVR model with the key-space concept. This implies that all previous software modeled and implemented using the original KVR model and its programming interface are still valid and compatible with this key-space concept. Since key-space is a description of key groups, two additional questions follow: 1) How is a key-space formulated? 2) Do key-spaces interact? If so, how do they interact and what such interactions imply to program structure and development?

Definition 5.1 A *key-space* is a set of keys grouped together. The basic elements in a key-space are: the topology, the dynamism, and the tunnel.

Definition 5.2 A *key* in the KVR model is an object that can be enumerated. A key must belong to exactly one key-space. Only equality and key-space membership tests are permitted on a key. The predicate "$\mathrm{mem}(k, S)$" performs the membership test on a key k and a key-space S.

Remark 5.1 The definitions of key-space and key did not specify how a key is assigned to a key-space. This is left to be determined by the designer. On the model level, an isolated key is an object with no associated information. This is one reason why an isolated key can only participate the equality and key-space membership tests. Additional information is needed for key-space assignment of a key.

Definition 5.3 The *lifetime* of a key-space refers to the period of time between the key-space's creation and destruction.

Remark 5.2 The lifetime of a key-space in the current implementation lasts throughout the entire application.

Definition 5.4 Given a set of keys s and a key-space K, the intersection of s and K is defined to be the set of keys in s that are members of K. The function "$\mathrm{ist}(s, K)$" denotes the intersection of s and K.

Definition 5.5 The *k-domain* is defined for a table with respect to a key-space. It is the set of keys in the domain of the table that are members of the key-space. The function "$\mathrm{kdom}(t, K)$" is used to denote the k-domain for a table t with a key-space K.

Definition 5.6 The *k-image* is defined for a map or a multimap with respect to a key-space. The function "$\mathrm{kimg}(m, d, K)$" denotes the k-image of a map or a multimap m on a set of keys d with respect to the key-space K. The k-image can be computed by: $\mathrm{kimg}(m, d, K) = \mathrm{img}(m, \mathrm{ist}(d, K))$.

Definition 5.7 Given a map or multimap m and a key-space K, define the key-sets $e = \mathrm{img}(m, \mathrm{kdom}(m, K))$, $i = \mathrm{ist}(e, K)$, $o = e - i$. The key-set i is named the *internal* image of m with respect to key-space K, and the key-set o is named the *external* image of m with respect to key-space K. The function "$\mathrm{iimg}(m, K)$" is an alias for i, and the function "$\mathrm{eimg}(m, K)$" is an alias for o.

5.2.1 Key-space Topology

The keys in a key-space can be further partitioned into subgroups, and such partition is determined by the topology of the key-space. The major purpose for partitioning the keys in a key-space into subgroups is to have the ability to distribute a set of keys in the key-space onto a set of parallel processes.

Definition 5.8 The *topology* of a key-space consists of a function and a set of tables. This function is called *partition* function. The set of tables is named *critical structures*. The partition function takes an integer n and the critical structures as inputs and maps each

key in the k-domain of each critical structure with respect to the key-space onto an integer within the range $[0, n-1]$.

Remark 5.3 Normally the integer parameter n supplied to the partition function represents the total parallel process number. Thus the partition function maps the key-space onto a given set of parallel processes. However the partition function can be used for other purposes as well.

Proposition 5.1 The key-space topology is not a mandatory specification. A key-space can have no topology at all.

Proposition 5.2 For a key-space with a topology definition, the partition function will be invoked if there is a change in any of the critical structures.

For example, in the cell-dec formulation of the Lagrangian particle tracking, all the cells can be grouped into a key-space. This key-space can have a topology that consists of a multi-constraint graph partitioning function and a set of maps that define the cell connectivity and a set of stores that define the cell computational weight as the critical structure. The movement of particles will cause the weight stores to change. According to Proposition 5.2, this change will cause the key-space topology to be reevaluated. In the dual-dec formulation of the Lagrangian particle tracking problem, all the particles can be grouped into a key-space. Such a key-space can have a topology that consists of an ORB partition function and a store that defines the particle position as the critical structure. The movement of the particles will cause the position store to change, which in turn causes the topology to be reevaluated.

The notion of key-space topology exposes the parallelism and requires a partition function. A quality implementation may not be trivial. A strategy to simplify the effort of defining key-space topology is to create a series (or hierarchical) abstract key-space classes (in the object-oriented design methodology term). Each of the abstract key-space class contains a complete definition of a topology type. A key-space inherits such abstract key-space automatically obtains the partition function definition. All the derived key-space has to do is to define the critical structures. Although it is not possible to enumerate all topologies that can occur in real applications and wrap them in predefined abstract key-spaces, the common geometric and numerical constructs should cover most of the cases. This situation can be compared to the LaTeX [49] typesetting system. LaTeX is widely regarded as a superior document preparation tool for technical writing. However it is relatively hard to design new document format and layout from the scratch. Most of the documents designed in LaTeX will use one of its predefined layouts and only need to have minor adjustments.

5.2.2 Key-space Dynamism

Using the key-space concept, the cell-dec approach can be designed in the KVR model by grouping the cells in a key-space. Similarly the dual-dec approach can be designed in the KVR model by grouping all the particles in a key-space and all cells in another key-space. However we can note that there is a difference in the cell key-space in the cell-dec design and the particle key-space in the dual-dec design. The cell-dec cell key-space remains unchanged throughout the entire program execution. While the dual-dec particle

key-space can have keys (i.e., particles) created and destroyed at run-time. The concept of key-space dynamism is designed to make distinctions between these phenomena.

Definition 5.9 The key-space *dynamism* is a tag for every key-space. The dynamism tag for a key-space is either *dynamic*, or *static*. A key-space with a dynamic tag is called dynamic key-space. A key-space with a static tag is called static key-space.

Definition 5.10 A static key-space K has the following properties:

1. For each table t, $\mathrm{kdom}(t, K)$ is fixed during K's lifetime.
2. For each map or multimap m in a join sequence: $t_1 \rightarrow \ldots \rightarrow t_i \rightarrow m \rightarrow p_1 \rightarrow \ldots p_j$, $i \geq 0, j > 0$, for each key k in $\mathrm{kdom}(m, K)$, $m[k]$ is fixed during K's lifetime.

Definition 5.11 If a key-space is not static, then it is dynamic.

Remark 5.4 The dynamism of a key-space is a run-time property. Thus a dynamic key-space may satisfy Definition 5.10 for certain time period during its lifetime.

Definition 5.12 If a dynamic key-space satisfies Definition 5.10 for a certain period of time during its lifetime, then this period of time is referred to as the *stasis time*.

Proposition 5.3 A static key-space cannot be determined by static program analysis unless with a specifically designed programming interface.

Remark 5.5 A properly designed programming interface can constrain the freedom of programming such that Definition 5.10 can be recognized to be true for static key-spaces. Later sections will show such a design.

Proposition 5.4 If a key-space cannot be identified in the static program analysis to be a static key-space, then it is treated as a dynamic key-space.

Remark 5.6 It is usually much better to confirm the stasis of a key-space through the static program analysis. Although there can be different implementation strategies for a dynamic key-space, the overhead involved is usually much larger than a static key-space implementation.

Remark 5.7 The current implementation also allows the dynamism of a key-space to be specified at its definition time to simplify some of the analysis process.

The dynamism of a key-space is important to its implementation. In general, a static key-space is more easily implemented. Due to its properties, a global analysis and optimization can be performed by static analysis. The Loci framework discussed in chapter 3 can be regarded as an implementation for a static key-space with no topology. A dynamic key-space, on the other hand, is more difficult to have an efficient implementation due to the freedom allowed regarding the lifetime of keys and the table structure.

5.2.3 Key-space Tunnel

If the KVR model defines multiple key-spaces within an application, then it is possible that these key-spaces will interact. For example, a dual-dec formulation within the KVR model is to construct two key-spaces, one for the cells and one for the particles. Since the particle and cell interact, there is a need to allow interactions between the key-spaces.

Definition 5.13 Key-space *interaction* is defined to be the reference of keys from one key-space to another key-space. Specifically, given two key-spaces K_1, K_2, and a map m, if either $\mathrm{ist}(\mathrm{eimg}(m, K_1), K_2)$ or $\mathrm{ist}(\mathrm{eimg}(m, K_2), K_1)$ is not empty, then we say key-space K_1 and K_2 interact. If $\mathrm{ist}(\mathrm{eimg}(m, K_1), K_2)$ is not empty, then we say K_1 reacts to K_2, denoted as $K_1 \leftrightharpoons K_2$. If $\mathrm{ist}(\mathrm{eimg}(m, K_2), K_1)$ is not empty, then we say K_2 reacts to K_1. If $K_1 \leftrightharpoons K_2$ and $K_2 \leftrightharpoons K_1$, then it is denoted as $K_1 \Leftrightarrow K_2$.

The key-space interaction has profound implication to the performance of an application. Recall at the opening introduction of this chapter, a question is posited regarding the design choice of particle-field interaction in the dual-dec approach designed using the KVR model. I will repeat the question here: In the KVR dual-dec design, to communicate data between the particles and cells, we can create different kinds of maps that link the particle and cells together. Will there be a difference in any of these designs? If so, which one is best? This question is studied in detail here.

In the KVR dual-dec design, we have two key-spaces, the cell key-space (named C) containing all the cells, and the particle key-space (named P), which contains all the particles. The particle key-space clearly has to be dynamic because particles (keys) are created and destroyed at run-time. According to Proposition 3.1, tables containing particle keys then must be resized at run-time, thus the particle key-space violates Definition 5.10 and is a dynamic key-space. The dynamism of the cell key-space is however unclear at this moment and depends on the actual design. Suppose that in the particle-field coupling, not only the particles read cell information, but also the cells need particle information for the numerical solution of the field equations. A natural and convenient design is to

use two maps, one (named m_1) uses the particle as the domain and cell as the image, the other one (named m_2) uses the cell as the domain and the particle as the image. The map m_1 is used by the particles for the pulling (through $m_1 \rightarrow t$) of cell information, and the map m_2 is used by the cells to retrieve the particle information (through $m_2 \rightarrow s$). Then in this design, the cell key-space C becomes dynamic according to Definition 5.10. Since $\mathrm{kimg}(m_2, \mathrm{kdom}(m_2, C), C) = \mathrm{kdom}(m_1, P)$, and $\mathrm{kdom}(m_1, P)$ is essentially the set of keys representing all particles, which is not fixed at run-time. In this case, the two key-spaces have a relation: $C \Leftrightarrow P$.

Table 5.1

Statistics of a Disk-particle Simulation

disk-particle simulation, 5000 iterations total on 3 processors			
exec time	sched time	critical sched	avg search step
7138.6(s)	7.4(s)	1.4(s)	3
	1st iter	2500th iter	5000th iter
time	0.3(s)	1.7(s)	2.1(s)
particle	131	351270	452701

In such a design, both key-spaces are dynamic. As discussed, dynamic key-space is difficult to implement. The most straightforward way to implement dynamic key-space is to rerun the key-set manipulation (see section 3.3.2) for the entire key-space whenever the k-image of a map changes. Table 5.1 shows some statistical information regarding a particle-disk simulation problem. This is a manually implemented dual-dec program with the fluid solver implemented using Loci. The "sched time" shown in the table is

the total time for Loci to assemble and start to run the fluid solver. It is trivial compared to the total execution time (which includes both the fluid and particle execution time). The "critical sched" time refers to the Loci scheduling steps that would be repeated at run-time to reproduce the fluid solver scheduling. In Table 5.1, the timing measurement in the second row shows the execution time for a single simulation step with the fluid and particle solvers time combined. It can be seen that this time is comparable to the essential scheduling time. Moreover the particle geometry search averages three steps per simulation step. In other words, if the cell key-space is dynamic, we will have to repeat the essential scheduling three times per simulation step. With the measured essential scheduling cost (as in the "critical sched" entry in Table 5.1), the total overhead will be prohibitive.

An alternative design is to use only one map (named m) with the particle as domain and the cell as image. The retrieving of particle information by the cell can be designed through the use of reduction to any cell table t joined with the map m ($m \rightarrow t$). This design achieves the same effects as the first one. However the cell key-space becomes static in this design since $\mathrm{kdom}(m, C)$ is empty and thus C satisfies Definition 5.10 (assuming C does not have other violations of Definition 5.10). This would allow the cell key-space to be implemented significantly more efficient. In this case, the two key-spaces have a relation: $P \Leftarrow\!\!\shortmid C$. As a summary, the first design uses two maps, and both key-spaces performs a pulling action to retrieve data from the other one. This results in both key-spaces to be dynamic. While in the second design, only one map is used. The interaction between the cell and particle key-space is entirely performed by the particle key-space via

both the pulling and pushing actions. In this design, the particle key-space is dynamic and the cell key-space is static. Intuitively a pulling action will transfer the dynamism of the key-space being pulled into the key-space that initiated the action. While a pushing action does not transfer the dynamism of the key-space that initiated the action into the receiving key-space.

Proposition 5.5 Given two key-spaces K_1 and K_2, suppose K_1 is dynamic and $K_2 \leftleftarrows K_1$, then K_2 has to be treated as dynamic in the program implementation. This says that the dynamism of a key-space can be affected by other key-spaces that it reacts to.

Proof: Since $K_2 \leftleftarrows K_1$, then by Definition 5.13, there exists a map m, its k-domain in K_2, $\mathrm{kdom}(m, K_2)$, and its external image in K_1, $\mathrm{ist}(\mathrm{eimg}(m, K_2), K_1)$, are both not empty. Since K_1 is dynamic, then either the set $\mathrm{ist}(\mathrm{eimg}(m, K_2), K_1)$, or the correspondence between $\mathrm{kdom}(m, K_2)$ and $\mathrm{ist}(\mathrm{eimg}(m, K_2), K_1)$ *may* change in the program. Therefore a static program analysis cannot confirm the stasis of K_2. According to Proposition 5.4, K_2 is effectively a dynamic key-space in the program. ■

Proposition 5.6 Given two key-spaces K_1 and K_2, suppose $K_2 \leftleftarrows K_1$, then the dynamism of K_1 is not affected by K_2.

Proof: Since $K_2 \leftleftarrows K_1$, then by Definition 5.13, there exists a map m, its k-domain in K_2, $\mathrm{kdom}(m, K_2)$, and its external image in K_1, $\mathrm{ist}(\mathrm{eimg}(m, K_2), K_1)$, are both not empty. Suppose K_1 is dynamic, then it is obvious that it will remain dynamic under this case. Now suppose K_1 is static, then by Definition 5.10, $\mathrm{kdom}(m, K_1)$ is either fixed or empty, and for a key k in $\mathrm{kdom}(m, K_1)$, $m[k]$ will remain the same at run-time. By

Definition 5.2, $\mathrm{kdom}(m, K_2) \bigwedge \mathrm{kdom}(m, K_1)$ is empty. Thus $m[k]$ will remain static at run-time and K_1 remains static. ■

These analyses suggest that we should encourage the pushing action from an undoubted dynamic key-space to other key-spaces in order to reduce its coupling and to limit the dynamic effects. There is another important reason why we should favor the pushing design choice. Consider in the KVR dual-dec approach. If we use the pulling design by creating a map m with cell as domain and particle as image, then this opens the possibility to revert the dual-dec design to the cell-dec design. Since in the pulling method, the operation is initiated by the cell key-space, it can request any computation through the map m, then the same load balancing problem plagued the cell-dec method will return because of the imbalanced parallel distribution of $\mathrm{img}(m, \mathrm{dom}(m))$. Although there is no definitive conclusion of the superiority of the cell-dec and dual-dec approaches, yet such pulling design inherits the disadvantages of both of the approaches (there is high communication cost and imbalanced computation), which is surely an unwise choice. In the pushing design, since the computation is always initiated by the key-space that produced the table in the rule tail, the computation is therefore always balanced (since each key-space should always be balanced), we only pay the communication cost, which is not avoidable in the dual-dec design.

Proposition 5.7 Given two key-spaces K_1 and K_2. The effect of $K_1 \Leftrightarrow K_2$ can be achieved by $K_1 \Leftarrow\!\!\triangleright K_2$. This is called the *push design*, denoted as $K_1 \Leftrightarrow\!\!\triangleright K_2$.

Proof: Given two key-spaces K_1 and K_2. Since $K_1 \Leftrightarrow K_2$, then by Definition 5.13, there exists a map m, its k-domain in K_2 and external image in K_1 are both not empty. Let $d = \mathrm{pmg}(m, \mathrm{ist}(\mathrm{eimg}(m, K_2), K1))$. Create a new map m_n and copy the portion of d in m to m_n. Remove the domain d from m resulting in m_o. Without loss of generality, for any rule: $t \leftarrow (m \rightarrow s)$, rewrite it as: $t \leftarrow (m_o \rightarrow s)$ and $t \leftarrow (m_n \rightarrow s)$. According to Proposition 3.5, $t \leftarrow (m_n \rightarrow s)$ can be transformed to: $(\mathrm{inv}(m_n) \rightarrow t) \leftarrow s$. Since both $\mathrm{ist}(\mathrm{eimg}(m_o, K_2), K_1)$ and $\mathrm{ist}(\mathrm{eimg}(\mathrm{inv}(m_n), K_2), K_1)$ are empty, then K_2 does not react to K_1, thus $K_1 \nLeftrightarrow K_2$. ■

Definition 5.14 A key-space *tunnel* is a map or multimap defined with respect to a key-space. Given a key-space K and a map or multimap t, if the external image of t with respect to K, $\mathrm{eimg}(t, K)$ is not empty, and the $\mathrm{kdom}(t, K)$ is also not empty, then t is referred to as the tunnel with respect to K. The KVR join from a key-space tunnel to other tables is denoted by a special notation "$\rightsquigarrow$." Suppose given another key-space P, if $\mathrm{ist}(\mathrm{eimg}(t, K), P)$ is not empty, then the tunnel t is said to be from K to P, denoted as $t\langle K, P\rangle$. K is called the source key-space, while P is called the target key-space.

Proposition 5.8 When a key-space is defined, if it is intended to have tunnel, then the tunnel and its target key-space must be specified.

Remark 5.8 Proposition 5.8 is required as part of the programming interface to enable the recognition of static key-space in the program analysis (see section 5.3.3).

Example 5.1 For example, suppose t is a key-space tunnel for K, and s is a table, the $t \rightsquigarrow s$ denotes the KVR join between t and s. Note, an inverse join with the tunnel on the right side is a conventional join and is denoted as the normal form $s \rightarrow t$.

Proposition 5.9 Given a key-space K, suppose t is one of its tunnels. Also suppose there is a map or multimap m, and let $d = \mathrm{kdom}(m, K), i = \mathrm{kimg}(m, d, K)$, if d is not empty and $i \bigwedge \mathrm{kdom}(t, K)$ is not empty, then $t \rightarrow m$ is also a tunnel with respect to K.

Proposition 5.10 In a chain of KVR join, $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n$, there can be only one $\rightsquigarrow$ inside. In other words, there can be only one key-space tunnel in a chain of KVR joins.

Remark 5.9 Such a requirement is to prevent keys from more than two key-spaces being joined together. In other words, key-spaces can never indirectly interact. This requirement is imposed to make the implementation feasible.

Remark 5.10 When formulating a design in the KVR model, it is wise to formulate undoubted dynamic key-space first and then create tunnels in such dynamic key-space to other key-spaces when key-space interaction is necessary. In short, consider the push design in Proposition 5.7.

5.2.4 An Example Key-space Definition

The key-space topology, dynamism, and tunnel constitute the major components of a key-space. When defining a key-space, these are the basic properties that need to be specified. In the current KVR model implementation, only one abstract key-space class based on the ORB topology is provided. More are planned in the future, for example,

abstract key-spaces based on space-filling curves and other spatial geometric structures as the topologies.

Codelet 5.1 Particle Key-space Definition

```
1  $type p2c dMap ; $type ppos dstore<Vec3d> ;
2  $type orbsp, orbpr param<int> ;
3  $type orbth param<float> ;
4  $rule default(orbsp,orbth,orbpr) {
5    $orbsp=0 ; $orbth=1.1 ; $orbpr=250 ;}
6  $keyspace OrbKeySpace(p2c,ppos,orbsp,orbth,orbpr),
7    property(name("ParticleSpace") | dynamism(DYNAMIC) |
8             critical("ppos") | tunnel("p2c","CellSpace")) {
9    if($orbsp >= $orbpr) // force distribution
10     $orbsp=0, key_ptn=orb_partition($ppos), return true ;
11   int maxp = mxdoms($ppos) ; int meanp = mndoms($ppos) ;
12   if(maxp > meanp*$orbth) // particle not balanced
13     $orbsp=0, key_ptn=orb_partition($ppos), return true ;
14   ++$orbsp, return false ; }
```

Codelet 5.1 presents the definition of a key-space that inherits the abstract ORB key-space. Note unlike the previous codes, the definition syntax starts from line 6 has not been implemented yet. The lower level Loci framework programming interface actually provides a set of C++ APIs for application programming. However the syntax of these APIs can be quite verbose, making them cumbersome to typeset and unsuitable for presentation purpose. We have created a streamlined programming syntax as shown in the previous code examples. A separate Loci preprocessor is used to translate such code into the actual Loci C++ interface. Because the work presented here is quite new, I have not yet incorporated the new syntax into the Loci preprocessor. However the presented code here

has no fundamental difference to the actual implementation. It was written in this form to enhance the visual appearance.

The definition of the key-space starts from line 6 in Codelet 5.1. The "`OrbKeySpace`" keyword indicates that the definition is inherited from the predefined ORB key-space definition. The variables enclosed in the following pair of parentheses indicate the KVR tables that will be accessed by the partition function in the key-space. The "`p2c`" is the key-space tunnel. Note its type declaration says it is a "`dMap`." All the "d*type*"s in Loci represent dynamic data structures that are either hash- or tree-based. The reason for this is discussed in section 7.1. The "`ppos`" is the particle position store and is used as the critical structure. Note in the dual-dec design, because particles are themselves keys, they can therefore have their own tables such as the `p2c` and `ppos`. The `ppos` is also used to deduce the particle load on each process.

The definition of this particle key-space is directly aligned with the discussion in section 4.4.3. Recall from the discussion, the particles to process partition can are determined by two factors, the first being the particle load and the second being reaching a preset interval. The variables "`orbsp`", "`orbth`", and "`orbpr`" serve as the reference parameters. The `orbsp` records how many times the key-space critical structure has changed (this is supplied automatically by the system), `orbth` is the load balancing threshold discussed in section 4.4.3, and `orbpr` is the preset interval number for forced key-space distribution. Note in line 8 the definition of the key-space tunnel takes another key-space name. This is required as in Proposition 5.8. The source key-space of the tunnel is implicitly assumed to be the particle key-space that is being defined. The definition of the topology partition

function starts from line 9 in Codelet 5.1. It is actually just a wrapper of the topology partition function provided in the abstract ORB base key-space definition. Line 9 tests if the distribution needs to be enforced. The method `orb_partition` is the partition function provided by the base ORB key-space. Line 12 tests if the particle load is imbalanced and distributes the key-space if the criterion is met. The "`mxdoms`" and "`mndoms`" in line 11 are two methods provided in the base ORB key-space definition that return the global maximum and mean number of the keys in the domain of the supplied store or map in the argument. The rule in line 5 is the rule used in Loci to initialize default value for tables.

Such a definition of the particle key-space can also be thought of a form of annotation for the Loci programming interface since without this critical piece of information, Loci is not able to perform parallel composition and scheduling for the dual-dec program. Granted, such an annotation exposes underlying parallelism. However this solution is perhaps the best we can use if we want the KVR model to be applicable as a generic tool. If we were to create a model *only* for the proposed dual-dec approach for the particular particle-fluid simulation example discussed in this dissertation, then the entire definition in Codelet 5.1 becomes the domain-specific semantic information that can be assumed by the model itself and the user can thus be freed to be aware of such details. The usefulness of such model is then confined to the very narrowly defined problems.

5.3 The Key-space Programming Interface

This section presents the programming interfaces that allow the use of the dynamic key-space concept to develop programs. These new programming interfaces are compatible with the ones presented in chapter 3 and can be thought of as a superset.

5.3.1 Key Creation and Destruction

The addition of dynamic key-space calls for additional support for key-set management functionality in the programming interface since keys in a dynamic key-space may be created or destroyed. When designing new interface that supports key creation and destruction, there are several considerations. First, key creation and destruction is best to be addressed by a different type of rule that is solely defined for such purpose. Since if key creation and destruction is mixed with computations, then the precomputed KVR joins in the rule signature can possibly be invalidated. In addition, arbitrary key creation and destruction can also possibly lead to ill-defined rule context since if the context is allowed to be modified during the rule execution, then the rule execution may not terminate. Having dedicated rules for key creation and destruction forces the key management activity to be separated from the computation.

A natural question for this choice is whether such a design (using separate key creation and destruction interface) will limit the design expressiveness. For example, in the particle-field simulation problem studied before, particles may be removed when they fall outside of the mesh boundary. This does not present a problem for a code that uses the cell-dec design, since in the code particles are values and are therefore easily removed. In

the dual-dec design there will be a similar computation that locates particles in the mesh. If we do not allow the destruction of keys, then does this present a design problem? The solution is to use tagging, i.e., the use of an additional store to tag whether a key during an arbitrary rule is to be destroyed. And such tag information is then processed later. The code in section 6.2 will present an example. This may seem to be an awkward solution and is less flexible than the on the fly erase design. However the resulting benefit is worth the cost. And this also shows that the design expressiveness is not limited.

Codelet 5.2 Key Creation and Destruction Rules

```
1  $type source blackbox<SeqContainer> ;
2  // target_list can be arbitrary Loci variables
3  $rule insertion(target_list<-source) {
4    // Loci automatically provides the key and value,ref pair
5    $target1 = $source.value1 ; $target2 = $source.ref ;
6    $target3 = $source.value2 ; // ......
7  }
8  $type target param<bool> ;
9  // source_list can be arbitrary Loci variables
10 $rule destroy(target<-source_list) {
11   bool cond = (true|false) ; // result of sources evaluation
12   if(cond) destroy_key($k) ;
13 }
```

Two new Loci rule types are created for the key creation and destruction. Codelet 5.2 illustrates the general syntax and several requirements. The rule "`insertion`" is responsible for creating keys. It can only have one source variable and its type must be the one indicated in line 1. The type "`blackbox`" is a special Loci type that is designed for any external datatypes. In other words, blackbox does not correspond to the usual KVR tables. It can

contain any type and Loci does not in general manage the objects stored inside. In this case the source must be a blackbox that contains the type "`SeqContainer`." This is a generic type (a C++ template) that is a model of sequential container. It can be any container type as long as it provides an iterator interface and the "`SeqContainer::value_type`" type declaration. Valid examples include the "`std::vector`" and "`std::list`," etc.

The reason for such an arrangement is because keys do not exist independently, they must be bound to value or refs. Therefore when a key is created, it must be bound to all the intended values and refs. Otherwise mismatch or other errors can occur (such as loose keys that do not bind to anything). Thus the insertion rule only takes one source variable that has all the intended values and refs packed inside. The insertion rule automatically provides an iteration over the source variable and supplies each entry with a matching key that is newly generated. All users have to do is to decide how to assign such key-value or key-ref pairs to the target storage, and also to determine which key-space the new key is to be assigned to.

The destroy rule shown in line 10 in Codelet 5.2 is responsible for key destruction. Its requirement is just the opposite to the insertion rule. It does not restrict the source variables, however it can only have one target variable in the type "`param<bool>`." This target variable is actually not used inside the rule. It is provided as an evidence to indicate that the rule is completed. The purpose is to provide necessary dependence information should there be a need. The general pattern of the destroy rule is to evaluate the rule tail. Since we are just dealing with a single key in the rule's definition (recall Loci automatically builds the context for the rules, users just define transformation for a single key where

such definition will be applied to all applicable keys by Loci), we can just tell the system whether to destroy it or not based on the evaluation result. The "`destroy_key`" method provides a mechanism to signal Loci framework the destruction of the key. The "`$k`" inside is just a placeholder for the key being evaluated. Section 6.2 will provide concrete examples of both rules.

5.3.2 The Key-table Placement and Rule Annotation

The key-space concept defined so far does not restrict a table domain to contain keys from different key-spaces. In fact, this is why the k-domain is defined. For example in the KVR dual-dec design, if a store is defined to hold key and position pair, then one can put arbitrary keys in the store, such as storing the position of the defining nodes for cells and the position of particles together in such a store. This is indeed a simple and elegant design on the model level, for it provides a unified view to the key partition and lets one to do programming almost oblivious of the existence of key-space.

Although this is a desirable property, it puts too much burden to the model implementation. It is hard if not impossible to manage mixed keys in a table domain, and efficiency is certainly also a major concern. Consider an example, suppose we have a KVR dual-dec design with a cell key-space C, and a particle key-space P, and a table t. Let $d = \mathrm{kdom}(t, C), s = \mathrm{kdom}(t, P)$. If both d and s are not empty, then t has its domain mixed with keys from both key-spaces. In this case, the implementation of t may incur a large memory or performance penalty. Let $g = \min(s) - \max(d)$,[1] assuming that

[1] Assume the function "$\min$" returns the minimum element in a set, and the function "$\max$" returns the maximum element in a set.

$\min(s) > \max(d)$. If t is implemented using an array, then $O(g)$ amount of space will be wasted. In practice, this amount is often significant, and there is no easy method to minimize the number g. If t is implemented using a dynamic data structure, then at least the performance of access to the set d within t is compromised, for since the cell key-space is static, then all its tables should be implemented with the fastest possible data structure. Relying on static analysis to automatically split t is not possible since its domain could change at run-time.

Proposition 5.11 Given a table t, and a key-space K in the KVR model, if $\mathrm{kdom}(t, K)$ is not empty, then it is required that $\mathrm{kdom}(t, K) = \mathrm{dom}(t)$. This is called the key-table placement restriction. It effectively assigns each table to an owning key-space.

Remark 5.11 Another reason for imposing the key-table placement restriction is to have a programming interface that allows the program analysis procedure to recognize a static key-space (refer to Proposition 5.3 and section 5.3.3).

Corollary 5.1 Given a rule $t \leftarrow s_1, s_2, \ldots, s_n$ and a key-space K, let c denotes the context of the rule. If $c = \mathrm{dom}(s_1) \bigwedge \mathrm{dom}(s_s) \bigwedge \ldots \bigwedge \mathrm{dom}(s_n)$, and $\mathrm{ist}(c, K)$ is not empty, then $\mathrm{ist}(c, K) = c$.

Remark 5.12 The current rule context in the Loci framework is defined as in the Corollary 5.1, thus the result would apply to all Loci rules.

Corollary 5.1 suggests that under the key-table placement requirement, the effects of the key-spaces concept can be viewed as to partition the rules into different groups and

process them differently (since the rule context now equals the rule itself). Such a view leads to another annotation in the programming interface, i.e., a rule must be properly tagged with the key-space that it belongs to. Codelet 5.3 shows the general form of such annotation. In order to save some typing, I also propose a concise form of key-space tagging for rules, shown in Codelet 5.3 from line 5. These preprocessor directives are not yet implemented, but will be used to illustrate the programming.

Codelet 5.3 Example Rule Annotation

```
1 $rule pointwise(source<-target),
2   keyspace_tag("ParticleSpace"), keyspace_dist() {
3   // define computation
4 }
5 $keyspace_tag("ParticleSpace") {
6   $rule pointwise(target1<-source1) { /* ...... */ }
7   $rule apply(target2<-source2) { /* ...... */ }
8   // ......
9 }
```

In reality, such key-space annotation in a rule can be largely avoided since a careful analysis by the system can deduce a rule's key-space ownership based on the initial key assignment in all the tables and the rule dependence information. Such annotation is imposed nevertheless to be more convenient and as an aid to the current implementation. There is another rule annotation that is allowed. It is shown in line 2 in Codelet 5.3. It instructs the system to consider a key-space topology evaluation after the rule. This annotation is added to give extra controls to the programmer. Normally one does not have to interfere the key-space topology evaluation activity. Once the key-space declares its topology parameters,

the system will monitor them and perform appropriate evaluation accordingly. Such manual annotation allows additional flexibility should the programmer consider it necessary to evaluate the key-space topology regardless of the status of its critical structures.

5.3.3 Recognizing Static Key-space From Rule Analysis

This section will show how to recognize a static key-space from a program analysis based on the current key-space programming interface design. Although the current interface design allows the specification of key-space dynamism at its definition time (an example is given in Codelet 5.1), the procedure outlined in this section can be used in an implementation to determine the key-space dynamism automatically, or be used to check the specification.

Proposition 5.12 By Proposition 5.11, each table's domain must belong to exactly one key-space. Suppose table $\mathrm{dom}(t)$ is within a key-space K, then we say K owns t.

Proposition 5.13 Given a table t and its owning key-space K, if t is not K's tunnel, then K owns $t \rightarrow s_1 \rightarrow \ldots \rightarrow s_n (n \geq 1)$, $\mathrm{inv}(t)$ and $s_1 \ldots s_n$. If t is K's tunnel, then by Proposition 5.8, t has its source and target key-spaces specified. K is surely t's source key-space. Assume P is t's target key-space. Then K owns $t \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$, and P owns $\mathrm{inv}(t)$ and $s_1 \ldots s_n$.

Proposition 5.14 Given a rule: $t \leftarrow s_1, s_2, \ldots, s_n$, if key-space K owns $s_1, s_2, \ldots, s_n$, then K also owns t. Otherwise if $s_1, s_2, \ldots, s_n$ are owned by more than one key-spaces, then t_1 is not owned by any key-space.

Proposition 5.15 Given a rule: $(m \rightarrow t) \leftarrow s_1, s_2, \ldots, s_n$, if key-space K owns $s_1 \ldots s_n$ and m, then the ownership of the rule's target variable t can be determined according to Proposition 5.13. If $s_1 \ldots s_n$ and m are owned by more than one key-spaces, then t is not owned by any key-space.

Proposition 5.16 Given a key-space K, checking through all rules and if all maps or multimaps owned by K appearing in a join sequence: $t_1 \rightarrow \ldots \rightarrow t_i \rightarrow m \rightarrow p_1 \rightarrow \ldots p_j$, $i \geq 0, j > 0$ do not appear in any rule head, and there is no "`insertion`" and "`destroy`" rules tagged as in K, then K can be determined as a static key-space.

Remark 5.13 All the above procedures are based on the KVR model properties and the programming interface design described in Proposition 5.8, Proposition 5.10, Proposition 5.11, Corollary 5.1, and the key creation and destruction rules in section 5.3.1.

CHAPTER 6

PROGRAMMING PARTICLE TRACKING USING THE KVR MODEL

It is useful to see how the cell-dec and dual-dec approaches are actually implemented using the KVR model and its rule programming interface. This section presents both designs in the Loci framework with the newly developed key-space concept. The cell-dec implementation does not support the automatic mesh repartitioning (manual mesh repartitioning is however supported) since the present implementation of the KVR model does not yet support key-space topology for a static key-space. The dual-dec code makes full use of the concepts and programming interfaces developed in chapter 5. These code examples serve as small documents on how the programming interfaces are used in developing real problem solutions. They also serve to illustrate the actual simplicity of the programming interface and that it has largely removed the complexities in the parallel programming process. Since the cell-dec code implementation only uses a subset of the model and programming interface that is equivalent to the original Loci framework, the comparison of the two codes can also show that the newly designed programming interface keeps the original style and feature, and that it is really a superset of the original one and is in fact compatible with the original Loci framework. Any valid previous Loci program is still a valid program in the new model and programming interface. In order to simplify the code presentation, only the particle location search is presented here. Though the nu-

merical computation on particle is not presented, it has a similar structure and does not differ too much from the particle geometric search implementation. In the end, these two implementations are compared.

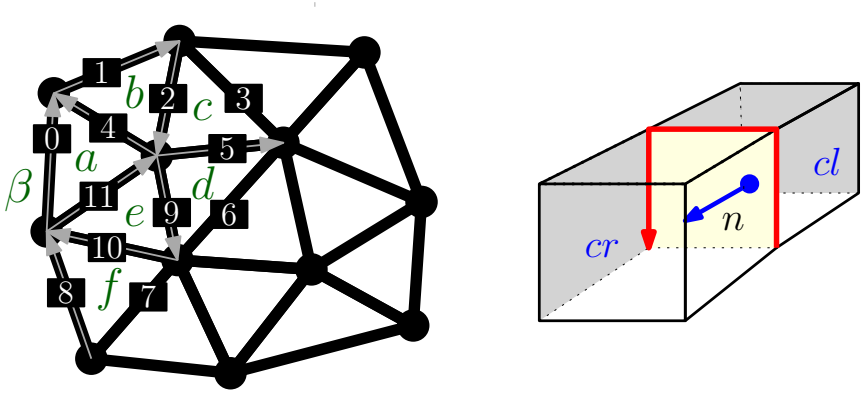

Figure 6.1

Mesh Data Organization

Before describing the actual implementation, it is useful to first introduce how the mesh data structures are organized in practice. In reality, the mesh data structure are not always organized based on cell. A useful mesh format is to use a face based data structure. This approach directly aligns the data organization with many common algorithms used in numerical simulations. In such a mesh organization, the basic mesh topology is stored in five Loci maps. These are the "`cl`", "`cr`", "`upper`", "`lower`", and "`boundary`" maps. Figure 6.1 shows a three dimensional view of the `cl` and `cr` data structure in the right part. In the mesh format, all the face (edge in 2D) nodes are arranged by a specific ordering, that is, when using the right-hand rule, the normal of the face would point away from the

left cell and into the right cell. In addition, all boundary faces have a normal pointing out of the boundary, which means that the left cell is the one on the boundary and the right cell does not exist (because it falls outside the boundary). The left cell is stored in `cl` and the right cell in `cr`. When we inverse `cl`, we get a multimap named `upper`. The same procedure applies to multimap `lower` and it comes from the inversion of the map `cr`. The specific reason for such an arrangement of data structures is to associate the matrix non-zero structure occurred in many numerical algorithms directly with the mesh data structure. The multimap `boundary` is obtained from the mapping of the cell to mesh boundaries. Table 6.1 provides an example of such data structures based on the left part in Figure 6.1. Although the mesh is two dimensional, the organization would still apply.

Table 6.1

Mesh Data Structures Setup

`cl`		`cr`		`upper`		`lower`		`boundary`	
k	r	k	r	k	r	k	r	k	r
0	a	2	c	a	0	a	4	a	β
1	b	4	a	b	1	a	11	b	β
2	b	5	c	b	2	c	2	f	β
4	b	9	d	b	4	c	5	$\vdots$	$\vdots$
5	d	10	f	e	9	d	9	$\vdots$	$\vdots$
8	f	11	a	e	10	f	10	$\vdots$	$\vdots$
9	e	$\vdots$	$\vdots$	e	11	$\vdots$	$\vdots$	$\vdots$	$\vdots$
10	e	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
11	e	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Given such a mesh topology, we can conveniently retrieve any structural information. For example, given a cell whose key is k, we can look into all of the entries in `upper[k]` and `lower[k]` to get all of its interior enclosing faces (edges in 2D). While `boundary[k]` will give us whether the cell is located on the boundary, and how many of them (usually one, unless in a corner). Such information can be readily verified from Table 6.1. Also since `cl` and `upper` are inversion to each other, and so is true for `cr` and `lower`, then given a cell we can find all of its neighbors through `lower->cl` and `upper->cr`. This also illustrates a use of composition. Instead of creating a special neighbors map, we can compose existing ones to achieve the purpose, and it is a better solution approach. Note, the key assignments in Table 6.1 for all the cells are alphabetical. This is mainly for clarity and illustration purpose. In reality, we would assign all of them as integers.

6.1 A Cell-dec Implementation

Given the mesh data structure organization, we are ready to declare all the major datatypes used in the implementation of the cell-dec approach. Codelet 6.1 defines a utility structure, the three dimensional vector type with two of its operations, and a particle type, which in this case only contains its position and velocity data. The "`cross`" and "`fmn`" fields in the particle structure will be used later and shall be clear from the source code presented later. Codelet 6.2 contains all the datatypes pertain to Loci in the cell-dec program.

Figure 6.2 illustrates the structure of the cell table in the cell-dec program. Since a cell can contain multiple particles at a given time, all its particles are linked together as

Codelet 6.1 Utilities Definition

```
struct Particle {
  Vec3d pos,vel ;
  int cross ;
  char fmn ;
} ;
inline double
dot(const Vec3d& v1, const Vec3d& v2) {
  return v1.x*v2.x + v1.y*v2.y + v1.z*v2.z ;
}
inline Vec3d
operator-(const Vec3d&v1, const Vec3d& v2) {
  return Vec3d(v1.x-v2.x, v1.y-v2.y, v1.z-v2.z) ;
}
```

```
struct Vec3d {
  double x,y,z ;
  Vec3d(double xx=0,
        double yy=0,double zz=0)
    :x(xx),y(yy),z(zz) {}
} ;
```

Codelet 6.2 Major Cell-dec Type Declarations

```
$type cl, cr Map ;
$type upper, lower, boundary multiMap ;
// cc = cell center, fc = face center, fn = face normal
$type cc, fc, fn store<Vec3d> ;
// these are the major stores for particle values
$type ps, pw, pl, pc store<std::list<Particle> > ;
```

a list. Therefore the declaration of the stores that contain particles in Codelet 6.2 has the "`std::list`" type in the template. Codelet 6.2 declares four major particle stores. The need of multiple such stores will be clear from the tracking code presented later. In Codelet 6.2 we are only declaring the effective store names, it is by no means that there will be four distinct copies of tables created. It is up to the Loci framework and the later development to determine the numbers of real physical data copies. This shows that we are programming at a conceptual level.

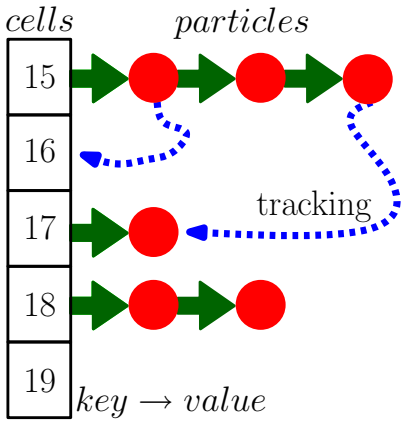

Figure 6.2

Cell-dec Cell-particle Structure Organization

Given these definitions, we are ready to implement the particle walking algorithm. Codelet 6.3 defines an auxiliary function that tests if a particle should cross a face. The flip test in line 5 is necessary to ensure that the face normal vector points outward and away from the cell center. Refer to Figure 4.12 for the visualization. This implementation

Codelet 6.3 Particle Face Test

```
1 bool // auxiliary function tests a particle and a face
2 cross_face(const Vec3d& pos, const Vec3d& cc,
3            const Vec3d& fc, const Vec3d& fn) {
4   Vec3d v1 = pos - fc ; Vec3d aux = cc - fc ;
5   double flip = (dot(fn, aux)<0) ? 1.0 : -1.0 ;
6   if(dot(v1, flip*fn) > 0) return true ;
7   return false ;
8 }
```

is not a direct correspondent to Algorithm 4.1 because the data structure organizations are different.

Codelet 6.4 presents the source code for the complete particle in cell test. This code corresponds to the **while** loop (one location step) in Algorithm 4.3. The intention of the rule in Codelet 6.4 is to split the source particle store "`ps`" into "`pl`" (located particles) and "`pw`" (still locating particles). The "`{n}`" structure attached to these names indicate that they are in an iteration named "n" and the rule is defining computation for this iteration. The "`inplace`" usage in line 3 is an indication to rename `ps` to `pw` at the end of the rule. This is done for performance reasons. The normal behavior without the `inplace` clause is to copy `ps` to `pw`, but renaming it avoids unnecessary copy overhead. This is roughly the only annotation used in the present Loci framework to indicate where performance optimization might be necessary. As discussed in the chapter 2, such control indications from programmers are sometimes needed.

Also notice that the particle face testing order used in Codelet 6.4 is important. The code first tests the `upper` and `lower` faces, the `boundary` faces are tested in the last step. The reason for such a testing order is that the particle face crossing test may suggest

Codelet 6.4 Cell-dec Particle-cell Location

```
$rule pointwise(pl{n},pw{n}<-ps{n},cc,
                (upper,lower,boundary)->(fn,fc)),
  inplace(pw{n}|ps{n}) { // rename ps to pw
  std::list<Particle>::iterator li2, li = $ps{n}.begin() ;
  while(li != $ps{n}.end()) {
    for(int i=0;i<$upper.size();++i) {
      if(cross_face(li->pos,$cc,$upper[i]->$fc,$upper[i]->$fn))
        li->cross=i, li->fmn='u', ++li, continue ; }
    for(int i=0;i<$lower.size();++i) {
      if(cross_face(li->pos,$cc,$lower[i]->$fc,$lower[i]->$fn))
        li->cross=i, li->fmn='l', ++li, continue ; }
    for(int i=0;i<$boundary.size();++i) {
      if(cross_face(li->pos, $cc, $boundary[i]->$fc,
                    $boundary[i]->$fn)) // remove particle
        li2=li, $ps{n}.erase(li), li=++li2, continue ; }
    li2=li ;// particle located
    $pl{n}.splice($pl{n}.end(), $ps{n}, li) ; li=++li2 ; }}
```

multiple faces to cross. If there is an interior face that can be crossed, we need to cross that first since crossing a boundary face means the particle falls outside and will be removed. If a particle moves, we record the crossing face information in the `cross` and `fmn` data fields for later use.

Codelet 6.5 shows the code for particle transfer. It computes the store "pc," which contains the particles that are still searching their containing cells, but have been sent to the correct cells for the next search step. The rule tail of the `apply` rule contains `pw`, which implies a dependence relation to the code defined in Codelet 6.4 (since `pw` is generated by the rule in Codelet 6.4). Since the store `pw` contains particles that fall outside of their current containing cell and are still in the previous cells, transferring them to the neighbor cells will give `pc` for the next search step. A reduction operation is used to define such

Codelet 6.5 Cell-dec Particle-cell Transfer

```
1  $rule unit(pc{n}) { $pc{n}.clear() ;} // identity list value
2  // a list merge operator
3  template<class T> struct ListMerger { using namespace std ;
4    void operator()(list<Particle>& r, const Particle& s) {
5      r.push_back(s) ;}} ;
6  // the real reduction rule to transfer particles
7  $rule apply(lower->cl->pc{n}, upper->cr->pc{n}
8              <-pw{n}) [ListMerger] {
9    std::list<Particle>::iterator li2, li = $pw{n}.begin() ;
10   while(li != $pw.end()) {
11     if(li->fmn == 'u') { // look-up previously stored
12       join($upper[li->cross]->$cl->$pc{n}, *li) ; ++li ;
13     } else if(li->fmn == 'l') { // map id and cross id
14       join($lower[li->cross]->$cr->$pc{n}, *li) ; ++li ;
15     } else throw Error("particle fmn error!") ; }}
```

transfer (i.e., each cell contributes particles to their neighbors). As discussed previously in this section, the composition of the maps `lower->cl` and `upper->cr` can be used to reach all neighbors of a given cell. An associative operator for list merging is also defined in the code.

Codelet 6.6 contains implementation on the location iteration setups. The "`{n=0}`" notation attached to each variable is a mean to cause initialization of the corresponding store at the beginning of iteration `n`. The "`{n+1}`" notation attached to each variable is a way to define how to advance such an iteration. This is an inductive definition of an iteration and is the major expression of loop in the Loci programming interface. The iteration terminate condition is defined in Codelet 6.7. If there is no particles contained in the store `pw` (particles moving) at each step, then we can stop the location process. The "`conditional`" clause in line 13 signals the rule's effectiveness is based on the condition

Codelet 6.6 Cell-dec Location Iteration Setup

```
// initialize iteration {n}, plocate is the initial particles
$type plocate store<std::list<Particle> > ;
$rule pointwise(ps{n=0}<-plocate),inplace(ps{n=0}|plocate) {}
// pll is the accumulated particles located in {n}
$type pll store<std::list<Particle> > ;
$rule pointwise(pll{n=0}<-plocate) { $pll{n=0}.clear() ;}
// define the advance {n+1} <- {n} action
// communicated particles continue to search
$rule pointwise(ps{n+1}<-pc{n}),inplace(ps{n+1}|pc{n}) {}
// located particles in {n} get accumulated
$rule pointwise(pll{n+1}<-pll{n},pl{n}),
  inplace(pll{n+1}|pll{n}) { $pll{n+1}.
    insert($pll{n+1}.end(),$pl{n}.begin(),$pl{n}.end()) ; }
```

Codelet 6.7 Cell-dec Location Iteration Terminate

```
// determine if all particles are located
$type nlp param<int> ;
$rule unit(nlp{n}) { $nlp{n} = 0 ; }
$rule apply(nlp{n}<-pw{n}) [Loci::Summation] {
  join($nlp{n}, $pw{n}.size()) ; }
// define the condition to terminate iteration {n}
$type finished param<bool> ;
$rule singleton(finished{n}<-nlp{n}) {
  $finished{n} = (0 == $nlp{n}) ; }
// define what to do on termination of location loop {n}
// all accumulated located particles get renamed
$type plocated store<std::list<Particle> > ;
$rule pointwise(plocated<-pll{n}),conditional(finished{n}),
  inplace(plocated | pll{n}) {}
```

indicated. Another reduction rule is used to compute the number of particles that are not yet located. The `apply` rule in Codelet 6.7 pushes one for each particle into the counter "`nlp`" through the summation operation.

Codelets (6.1 – 6.7) constitute the major implementation of the particle location search in the cell-dec approach using the Loci framework. These are not pseudocode however, they are real legitimate Loci source codes and can be readily compiled and run if the missing pieces are supplied. It can be seen that it is very succinct to write parallel programs in Loci utilizing the KVR model and its rule programming interface. These codes are completely oblivious to the parallelism. And they are also oblivious to the rule context. Each of the rules is defined with a single cell in mind. The final execution context of each rule is calculated by Loci. In addition, the essential dependence of the composition is documented through the rule signatures, there is no need to supply a driver function to manually combine them together.

6.2 A Dual-dec Implementation

Codelets (6.8 – 6.13) present the program source code for the particle location search in the dual-dec approach. This new implementation achieves the same functionality as those codes presented in Codelets (6.1 – 6.7), but makes use of the new key-space concept and programming interfaces. The particle key-space definition has already been discussed in Codelet 5.1. This section will present the basic algorithms and program organization. They are mainly the same as the cell-dec code presented early, the major difference is the code in this section treats the particle as keys instead of values.

Codelet 6.8 Main Dual-dec Type Declarations

```
1 struct RefParticle { // this types is used in
2   Vec3d pos, vel ;   // key creation
3   Key cell ; } ;
4 $type cl, cr dMap ; // notice all are now "dContainers"
5 $type upper, lower, boundary dmultiMap ;
6 $type cc, fc, fn dstore<Vec3d> ; $type pdy param<bool> ;
7 $type ppos dstore<Vec3d> ; $type p2c dMap ;
8 $type plr, prr dstore<bool> ; // used to help key destruction
```

Codelet 6.8 similarly declares the major datatypes used in the new formulation. The new particle structure defined from line 1 is not the structure used in the main computation. It is instead used as an auxiliary datatype in particle creations. In the dual-dec method, since particle themselves are keys, there is no need to define a special datatype for particle. Another note is that in Codelet 6.8, all of the stores and maps now become "d*type*"s. As mentioned before, this is required for any table defined in a dynamic key-space. The rationale behind such a choice is outline in section 7.1.

Proposition 6.1 The table type declaration in a dynamic key-space follows these rules:

1. Any store or map directly owned by a dynamic key-space should be declared as the corresponding "d*type*."
2. Any store or map accessed through the key-space tunnel is also declared as its corresponding "d*type*" (no matter which key-space it belongs to).

Example 6.1 For example, the map "p2c" belongs to the particle key-space, which is dynamic. Therefore its type must be dMap. Suppose we intend to access a cell key-space (assuming static) store named "temp" through the particle key-space tunnel p2c->temp, then temp needs to be declared as "dstore" in any particle key-space rules despite that temp is declared as "store" in the cell key-space.

Codelet 6.9 Dual-dec Particle-cell Location

```
$keyspace("ParticleSpace") { // key-space tag
  $rule pointwise(plr{n+1},prr{n+1},p2c{n+1}<-p2c{n},ppos{n},
/*reference to*/  p2c{n}->(upper,lower,boundary)->(fn,fc),
/* cell space */  p2c{n}->lower->cl,p2c{n}->upper->cr,
/* via tunnel */  p2c{n}->cc,plr{n},prr{n}),
    inplace(p2c{n+1}|p2c{n}), // rename results for efficiency
    inplace(plr{n+1}|plr{n}),inplace(prr{n+1}|prr{n}) {
    if($plr{n}) return ; // located particle, just quit
    for(int i=0;i<$p2c{n}->$upper.size();++i) {
      if(cross_face($ppos{n},$p2c{n}->$cc,
                    $p2c{n}->$upper[i]->$fc,
                    $p2c{n}->$upper[i]->$fn))
        $p2c{n+1}=$p2c{n}->$upper[i]->$cr, return ; }
    for(int i=0;i<$p2c{n}->$lower.size();++i) {
      if(cross_face($ppos{n},$p2c{n}->$cc,
                    $p2c{n}->$lower[i]->$fc,
                    $p2c{n}->$lower[i]->$fn))
        $p2c{n+1}=$p2c{n}->$lower[i]->$cl, return ; }
    for(int i=0;i<$p2c{n}->$boundary.size();++i) {
      if(cross_face($ppos{n},$p2c{n}->$cc,
                    $p2c{n}->$boundary[i]->$fc,
                    $p2c{n}->$boundary[i]->$fn))
        $plr{n+1}=true, $prr{n+1}=true, return ; }
    $plr{n+1} = true ; }}
```

Codelet 6.9 is the main implementation for the particle location search. It reuses the utility function "cross_face" defined in Codelet 6.3. Unlike the cell-dec implementation, which uses two steps (the locate step (Codelet 6.4) and the transfer step (Codelet 6.5)), this dual-dec implementation only uses a single step to achieve the particle location. This is due to the fact that particles are being treated as keys, therefore there is no need to transfer them. This single rule definition in Codelet 6.9 also serves as the particle location iteration advancing step definition. Basically it calculates a new particle to cell map from the old one based on the faces particles crossed in a single search step. In addition, it also generates two auxiliary stores "plr" and "prr" to represent the particle location results and particle removal results respectively. These two stores act as tags to mark the single step search results. The use of these two flags is resulted from the requirement that keys be created and deleted in a dedicated rule. See the discussions from section 5.3.1 for details.

Codelet 6.10 Dual-dec Particle Destruction

```
1 $keyspace("ParticleSpace") {
2   $rule destroy(pdy<-prr) {
3     if($prr) destroy_key($k) ;
4   }}
```

Codelet 6.10 shows the particle key removal code. It is a special destroy rule discussed in section 5.3.1. It takes the source from the prr store, which records the particle deletion information. Its target is a special flag variable in the type of param<bool> used to indicate the completion of this rule. Its use will be clear in the later part of the code. Also notice that all the variables in the rule do not have the iteration identifier {n} attached. Such a

Codelet 6.11 Dual-dec Particle Creation

```
1  $keyspace("ParticleSpace") {
2    $type pinject store<RefParticle> ; // particles injected
3    $type pstream blackbox<std::vector<RefParticle> > ;
4    $rule blackbox(pstream<-pinject) {
5      $pstream.insert($pstream.end(),
6                      $pinject.begin(), $pinject.end()) ; }
7    $type p2c_new dMap ;
8    $type ppos_new dstore<Vec3d> ;
9    $rule insertion(p2c_new,ppos_new<-pstream) {
10     $p2c_new = $pinject.cell ;
11     $ppos_new = $pinject.pos ; }}
```

rule is named "stationary time rule" in the Loci framework. It can be scheduled to any iteration that requires the result. Such rules are also part of the Loci framework's strategy to encourage code reuse.

Codelet 6.11 presents the particle creation. It is achieved by using the special insertion rule type. As a requirement, there can be only one source for the insertion rule and its type be that of any `blackbox<SeqContainer>`. In this case it is an `std::vector`. The rule in line 4 is used to collect the newly created particles into the required data format. In this case it is needed because the particles are created as values (similar to that in the cell-dec approach) by another boundary condition code. It is also possible to create particles directly to bypass the data format conversion step. The rule type `blackbox` in line 4 is a special rule that generates a `blackbox` type that is required for the insertion rule.

Codelet 6.12 contains the code for the termination condition of the particle search iteration. It is essentially the same as the source code in Codelet 6.7. The code computes the number of particles that are not located and terminates the iteration when all of the

Codelet 6.12 Dual-dec Particle Location Loop Exit Condition

```
1 $keyspace("ParticleSpace") {
2   $type nlp param<int> ;
3   $rule unit(nlp<-pdy) { $nlp = 0 ; }
4   $rule apply(nlp<-plr) [Loci::Summation] {
5     if(!$plr) join($nlp{n}, 1) ; }
6   $type finished param<bool> ;
7   $rule singleton(finished{n}<-nlp{n}) {
8     $finished{n} = (0 == $nlp{n}) ; }}
```

Codelet 6.13 Dual-dec Particle Location Loop Setup

```
1 $keyspace("ParticleSpace") {
2   $type p2c_locate dMap ;
3   $rule pointwise(plr{n=0},prr{n=0},p2c{n=0}<-p2c_locate),
4     inplace(p2c{n=0}|p2c_locate) {$plr{n=0}=$prr{n=0}=false ;}
5   $type p2c_located dMap ;
6   $rule pointwise(p2c_located<-p2c{n}),
7     conditional(finished{n}),inplace(p2c_located|p2c{n}) {} }
```

particles are located. The only difference is in line 3. Notice the particle deletion rule flag "pdy" is used to ensure the removal of particles before counting. However this is only a design choice since the removed particles do not affect the counting, it can be placed in other places in the code. Codelet 6.12 shows one of its usages. Codelet 6.13 shows the rest of the setup for the particle location search iteration. They are presented for the sake of completeness for the code.

It can be seen from Codelets (6.8 – 6.13), programming in the new key-space interface for the dual-dec design is not much different than programming the cell-dec approach in Codelets (6.1 – 6.7). Both source codes have similar size and use the similar structure. The essential difference from the two codes only comes from the different treatment of particle

removal. The definition of the particle key-space (shown in Codelet 5.1) is probably the most significant effort involved in the new programming interface.

In terms of software structure, the dual-dec code presented in this section do have some advantages over the cell-dec code. In the cell-dec code, since particle is treated as value, its entire definition is enclosed in a structure. This makes the source code evolution and maintenance harder. For example the particle type declared in Codelet 6.1 only contains minimum information about a particle. As the physics model become more complex, more data fields will be expanded in the particle type declaration. These data fields will end up being hard-wired into the source code dealing with physics. There is no easy way to separate the various parts from each other. As a result, upgrading the particle model requires one to manually search through the corresponding source code and perform necessary changes, which is inconvenient and error-prone. In the dual-dec code, the code structure is more flexible. Since particles are keys and can own arbitrary values and refs, one can always define new data associated with particles as the physics models evolve and let the Loci framework compose the program.

CHAPTER 7

A DYNAMIC KEY-SPACE IMPLEMENTATION

7.1 Implementation Strategies

This section will focus on the implementation strategies for a dynamic key-space and its interaction with other key-spaces. The implementation of a static key-space without the key-space topology support is already covered by the current Loci framework. Recall from section 3.3, there are two fundamental steps in the current Loci implementation strategy: the graph processing and the key-sets manipulation. The graph processing step is used to resolve rule dependence relations and to break the entire program into functional blocks. The key-sets computation is used to resolve rule context information and to generate the parallel communication subroutines.

Recall that the key-space concept developed in chapter 5 can be viewed as a layer added on top of the current KVR model (from Corollary 5.1). Its functional purpose in the system can be viewed as further partition of the rules into individual key-space and then to be processed according to the key-space definition. Thus the basic strategies illustrated in section 3.3 and in Figure 3.2 remain the same, and all of them are compatible with the new key-space concept. For a dynamic key-space implementation, there are two new additions to the steps outlined in Figure 3.2: 1) In the graph processing stage, in addition to partition

the rules into a DAG hierarchy, these DAGs are further partitioned into blocks of rules that are in the same key-space. 2) The last step in Figure 3.2 is repeated at run-time and that the analysis become per rule (a single block in the list in Figure 3.2) based instead of based on the entire list as in the current Loci implementation.

Recall from section 5.3.2, since all the rules in a dynamic key-space are annotated, it is then easy to group them into a separate block in the DAG hierarchy. This step is only performed once at the beginning of the scheduling step and the cost is generally negligible. The second step outlined presents a major challenge since it is invoked repeatedly at run-time and its cost is in general non-trivial. Its necessity can be illustrated via a visualization of the KVR model.

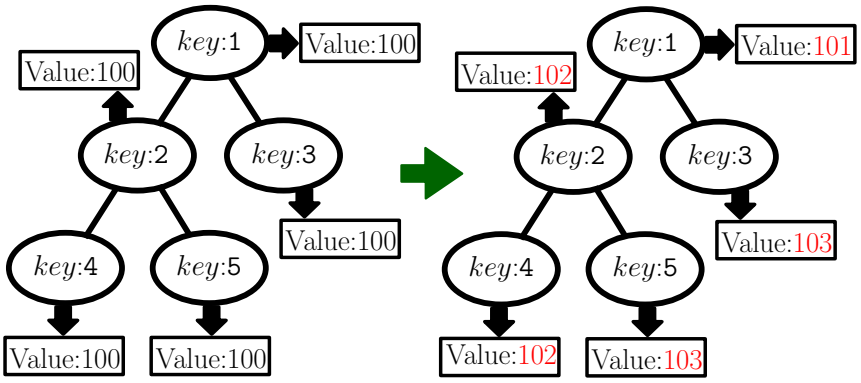

Figure 7.1

Static Key-space View

Figure 7.1 presents a view of a static key-space mapped onto the KVR model. The nodes in the figure denote "key," the edges represent the "ref" relation, and the rectangle tag on each node is the "value" associated with "keys." Essentially in such a static key-

space, the "key" and "ref" parts of the model are fixed, only the "value" part of the model is evolving at run-time. If we view the "key" and "ref" part of the model as a tree, then the connectivity of such a tree never changes through out the entire application. This property allows more opportunity to optimize the key-space. Because we can "see through" the entire tree structure, the tree can then be processed in an entirety, for example, we can traverse it from top to the bottom in one process, and from the bottom to the top in another process. Such traversals allow the collection of various information to have a global analysis. In a static key-space, this is not only possible but also worth the cost as such performance optimization can have a lasting effect throughout the entire program execution and can easily amortize the global analysis cost.

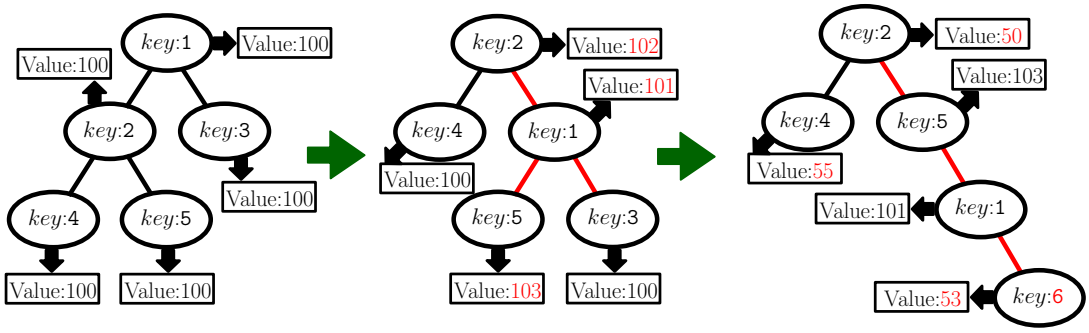

Figure 7.2

Dynamic Key-space View

Figure 7.2 presents a view of a dynamic key-space mapped onto the KVR model. In this situation, each part of the KVR model is possible to change. This effectively made the tree structure volatile: the nodes in the tree can disappear and reemerge, the edges that

connect individual nodes can be reconfigured. Many of such dynamism depends on the run-time computation results and is not possible to predict. Thus we lose the capability to see through the entire tree structure and in turn, the processing of such a tree can only be per node based.[1] This amounts to the per rule processing strategy used in the dynamic key-space implementation discussed here.

Algorithm 7.1 Single Rule Compilation in a Dynamic Key-space

Require: a Loci rule and its key-space info.
Ensure: proper execution of the rule
1: pre-processing
2: data expansion for joins
3: rule context evaluation
4: $rule\ execution$
5: post-processing

Algorithm 7.1 illustrates the basic steps involved to process a rule in a dynamic key-space. Line 4 in the algorithm represents the rule body defined with each rule. The pre-processing step sets up necessary data structure, while the post-processing step performs clean ups that may become needed. In some cases, the results may need to be reduced to all processes. This is also performed in the post-processing step. The discussion provided here will focus on the two major steps shown in lines 2 and 3. The expansion phase in

[1]It is possible to recognize sub-trees that can be processed as a whole. Aside from the complexity of such analysis, its real benefit is unknown as the complexity of run-time sub-tree processing may outweigh its optimization capability. Evaluating such strategies will become a future research work. However the present strategy can already be regarded as sub-tree processing as the static and dynamic key-spaces are partitioned into different sub-trees and are processed differently. This can be regarded as the major purpose of the key-space concept: without the key-space abstraction, a single dynamic tree component will force the entire tree to be processed sub-optimally. Another way to look at optimization is to recognize the stasis time (Definition 5.12) of a dynamic key-space, i.e., the time period in which the tree connectivity is fixed. Then during such a time period, the tree can be processed as a whole to enable more aggressive optimization. However the same problem as in the sub-tree processing can occur, i.e., to determine whether the stasis time is long enough to warrant the potentially expensive optimization steps.

line 2 analyzes a rule's signature and resolves all the join ("→") operators that invoke parallel data movement. In a static key-space, this parallel communication schedule would be generated just once and can be optimized globally within the entire application. The context evaluation step in line 3 finds all the keys in the key-space that can be applied to a rule. Again in the static key-space, this step is only evaluated once and is globally optimized within the entire application. The basic algorithms and details for these steps remain the same as those used in the static key-space (described in detail in the past publications [33, 54]) and are hence treated as black boxes here. The focus of this discussion will be on a infrastructure that can adequately reduce the run-time cost associated with these essential steps as in a dynamic key-space global optimization is no longer available and these steps are evaluated multiple times at the run-time.

7.1.1 The Dynamic Data Expansion Infrastructure

The dynamic data expansion infrastructure is designed to implement the data expansion phase defined in line 2 in Algorithm 7.1. Its purpose is to reduce the frequency of parallel data communication to the minimal while ensuring the correctness of the program. The general strategy outlined in Algorithm 7.1 implies that the data expansion phase will be performed every time a dynamic key-space rule is scheduled. This is however not always necessary if we allow communicated data to be cached properly.

Consider an example shown in Figure 7.3. This example shows the dependence relation between two rules and their placement in the entire program. Rule 1 generates y that is consumed by rule 2, and they are also in different iteration hierarchies. Suppose rule 2

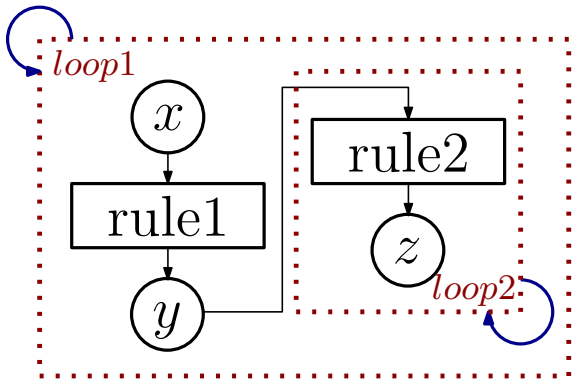

Figure 7.3

An Example Rule Dependency for Parallel Data Expansion

is in a dynamic key-space and its use of y is from a join operation. Thus y needs to be communicated properly to include all accessible data from all remote parallel processes before rule 2 can proceed. According to Algorithm 7.1, such parallel data retrieval occurs whenever rule 2 is encountered. Since rule 2 is inside loop 2, the iteration bounds will determine the number of times to invoke the parallel data communication on y. However we can observe that since y is generated by rule 1 in loop 1, its value is fixed throughout the lifetime of loop 2. Therefore the y value communicated from all processes is always valid within loop 2.

This observation suggests we can cache all parallel communicated data for dynamic key-space rules. The subsequent communication point (as line 2 in Algorithm 7.1) only needs to check whether the data cache is valid or not and only initiates the communication whenever the data cache is not valid. This also suggests we need a mechanism to invalidate the parallel data cache. The need for cache invalidation is obvious. Consider the example

in Figure 7.3 again, if loop 1 is rerun, then y would be recomputed and thus needs to be communicated across all the parallel processes again for rule 2.

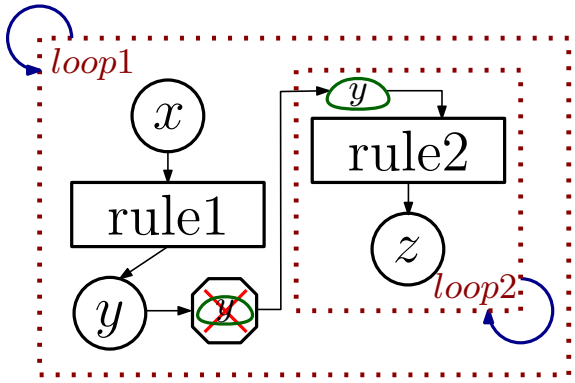

Figure 7.4

Parallel Data Expansion Cache and Invalidation

Figure 7.4 shows the general strategy used to cache parallel data expansion and its invalidation. We can insert a cache invalidator as soon as new data is generated. We perform a scan of all the dynamic key-space rules' signature to identify all the variables that need to have corresponding data cache invalidator. For example, we can identify all the join operations used in a rule by searching $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n$ in all the rules' signature. In this case, all variables from t_2 to t_n will need to have data cache invalidator for their parallel expansion. Subsequently the graph structure and the scheduling list are modified to accommodate the new data cache invalidator.

Such a data cache strategy also handles dynamic join pattern and key-space distribution automatically. For example in Figure 7.4, if rule 2's signature is $z^{n+1} \leftarrow (z^n \rightarrow y)$, then such a join pattern becomes dynamic since z is evolving each time the rule is computed.

Thus the expansion of y also becomes dynamic each time the rule is run. This case can be treated as an automatic invalidation of the y cache used by rule 2. This has a nice property that it only invalidates the necessary portion of y cache. For example, suppose $z^n \rightarrow y$ yields a key-set of $\{0\text{–}9\}$ and $z^{n+1} \rightarrow y$ yields a key-set of $\{5\text{–}15\}$. Then only the set $\{10\text{–}15\}$ is needed to be expanded since the portion $\{5\text{–}9\}$ is already expanded by the previous expansions. A key-space distribution has the same effect of creating a dynamic join pattern. Since the key/process ownership is shuffled during a key-space distribution, the join operation would yield different key-set. This is handled in the data cache in the same way that only the missing parts are further expanded.

The data expansion cache and invalidation strategy also allows the implementation of various options. For example, the invalidator appended after a variable can selectively invalidate its expansion cache. Still consider the example in Figure 7.4, if it is determined that a portion of y is indeed the same in different iterations of loop 1 (e.g., the key-set $\{2\text{–}4\}$ can be determined to be the same as in the previous iteration), then such a portion can be excluded from the invalidation. The incremental data caching strategy discussed in section 4.4.2 can also be implemented within this infrastructure. Reusing the previous example, suppose $z^1 \rightarrow y$ yields a key-set of $\{0\text{–}9\}$, $z^2 \rightarrow y$ yields a key-set of $\{5\text{–}15\}$, and $z^3 \rightarrow y$ yields a key-set of $\{1,3\text{–}8,12\text{–}16\}$. The incremental caching will not purge any of the old data expansion from the cache as long as the cache limit is not reached. This allows the $z^3 \rightarrow y$ expansion to reuse the data in key-set $\{1,3,4\}$ without the need to re-expand them. Note this works automatically with the cache invalidator. In the example,

as long as y is valid, its expansion can always be cached. Once the y expansion cache is invalidated, the incremental caching will automatically start off again.

7.1.2 Dynamic Rule Context Evaluation

The rule context evaluation outlined in line 3 in Algorithm 7.1 follows the same pattern as the dynamic data expansion problem in the previous section. For a rule whose inputs do not change, we do not need to evaluate its context again. Consider the example shown in Figure 7.3, if rule 2 has a signature $z \leftarrow y$, then its context stays the same during the lifetime of loop 2 since it depends on the domain of y, which stays fixed in loop 2. Therefore the same caching and invalidation strategy used in the dynamic data expansion infrastructure can be reused in the rule context evaluation.

Each dynamic key-space rule would cache its execution context until it is invalidated. The invalidation setup follows the approach outlined in the previous section except for one difference. In the dynamic rule context evaluation, there is only one cache for the entire rule execution context. Any of the invalidation would invalidate the entire rule context cache. This decision is used to conserve memory consumption for rule context cache. Consider an example dynamic key-space rule with the signature: $t \leftarrow s_1, (s_2 \rightarrow s_3 \rightarrow s_4), s_5, (s_6 \rightarrow s_7)$. If only s_7 invalidates its context, then the most efficient way to reevaluate the rule context is to reevaluate the portion $s_6 \rightarrow s_7$ and then merge the result with the other portions. However this would also mean to store individual context information for each individual variable in the rule signature and each portion of the rule signature (e.g., $s_2 \rightarrow s_3$, $s_3 \rightarrow s_4$ etc.). Storing all these contexts as key-sets will consume a large

portion of memory in practice. Thus there is only one context cache for any dynamic key-space rule and any of its inputs context invalidation will cause the entire rule context to be reevaluated.

7.1.3 Data Structure Choice

There are two data structure design choices in the dynamic key-space implementation. The first one is whether to use a split storage or a unified storage when expanding data. Consider an example where the domain distribution of a variable x on three processes is: p_0: {0–5}, p_1: {15–18}, and p_2: {22–29}. Suppose in a parallel computation, the requested x domain on p_0 is {1–3,16,25–28} (denoted as x^r). Then for p_0, the remote domain that needs expansion is {16,25–28}, while the domain {1–3} is local and does not need to be communicated in parallel. The data expansion routine, however, has two choices. One way is to create an expanded variable x^e with the remote domain {16,25–28}. Therefore the on p_0: $x^r = x \cup x^e$. In this case, subsequent access to x^r is split to x and x^e. For example, given a key 2, the assess should be made to x, while the access for key 20 should be made to x^e. There is a cost associated in determining how to perform the split access. The other way is to create x^e with the remote domain {16,25–28} communicated from other processes and the local domain {1–3} copied to x^e as well. In this case, $x^r = x^e$ and there is no split cost involved in data access. However there is an additional copy overhead incurred when merging the local domain into x^e.

The current implementation of the data expansion phase favors the second choice, i.e., using a unified data storage to avoid the split cost. The cost trade-off within the two choices

are largely problem dependent. However an added benefit of the unified expansion storage choice is that the programming logic is also simplified. Table 7.1 shows one particular measurement to compare the cost trade-offs between the split and unified data storage choice. The numbers in the "copy," "total," and "split tests" columns are summed from all the participating processes. The last two column show the timing results from the slowest process (to show the overall practical effects). The "copy" column shows the total number of copies performed in the unified storage method. The "split test" column shows estimated total number of range tests (i.e., determine if an integer falls in an interval $[b, e]$) performed in the split storage method in order to determine the split. The "total" column is the total number of requested data items. Therefore the number in the "total" column minus the number in the "copy" column is the data items that are communicated from remote processes. The timing results in the last two columns include the copy, parallel communication, and computation (with data access, split or not) time.

Table 7.1

Split vs Unified Data Storage Benchmark

data item size: 176 bytes						
process	copy	total	copy ratio	split tests	split (s)	unify (s)
32	217054	16009987	1.36%	365838405	50.3	46.8
64	182168	16196507	1.12%	295625345	35.3	32.2
128	82326	16496829	0.50%	99917370	21.7	22.0

The results in this particular case favors the unified storage approach. The copy ratio in Table 7.1 indicates that only a small amount of data items are actually locally copied,

the majority of the data items are communicated. This is not surprising as the test data came from a simulation with a dual-dec implementation of the Lagrangian particle tracking problem. Such an effect is caused by the independent distribution of particles and mesh cells, which causes most particles to refer to cells that are not local. Since problem properties differ widely, the best strategy is to use an adaptive learning algorithm to select the appropriate strategy for a specific problem setup based on run-time profiling results or a guided learning process. This highlights an important future research area: the use of run-time profiling based optimization. Some AI techniques may also be useful to aid decision making in these situations.

Another choice of data structure relates to the use of static array based or dynamic hash based container. In a dynamic key-space, keys can frequently be generated, destroyed, and distributed. Therefore the containers that store key-value or key-ref pairs need to accommodate such changes. Static array based container offers the fastest access time, but is inflexible to insert and erase. Also to use array based container, we need to remap keys in a compact form as discussed in section 3.3.2 lest fatal memory allocation problems. A hash based container offers freedom in all these areas, but it is slow to access. The current dynamic key-space implementation favors the use of hash based container. The reason is that for a dynamic key-space, it is usually anticipated that the key creation and destruction will be frequent. Thus the additional key management overhead for using an array based container can easily outweigh the access speed benefit.

Table 7.2 assesses the data structure access performance. The measurements compare timing results from accessing an array and a trie data structure. The trie structure is used

Table 7.2

Array vs Trie Data Structure Access Benchmark

access num	array (s)	trie (s)
327797400	247.0	261.1
6555937500	474.5	518.0
13111864500	944.9	1025.4

in the current dynamic key-space implementation as a replacement to hash table. Since the keys in present abstraction is integers, the entire permutation of the available keys are encoded in a three-level trie structure. Thus accessing associated data for a key in such a data structure involves three bit processing and three pointer dereferencing. Ideally it should be about three times slower than accessing an array based container. However in reality, we never access a container alone. Often accessing containers is mixed with interleaved computation on the accessed data. Thus in practice, the effects of choosing a dynamic hash based container shall be measured in a real setting. The timing results in Table 7.2 comes from a numerical integration routine within the Lagrangian particle code. It represents the typical use of data access and computation. The results show that the trie performance is within 10% of the array. This is indeed an acceptable number. Consider the time that it saves in all the key management routine, this is actually superior than the array based approach. Although individual problems differ, this example did suggest that we reconsider the data structure choices in certain types of numerical computations where dynamic structural change is frequently anticipated.

7.1.4 Key Management

The key management in a dynamic key-space deals with key generation, destruction, and distribution. For the purpose of key generation and destruction, a key manager is used. Initially the available key-space is partitioned evenly for all the processes. For example, if we use n-bit integer to represent keys and have p processes, then each process will get $\frac{2^n}{p}$ unique partition of the n-bit integer as the local key-space. The key manager on each process then keeps a set of available keys for generation purpose. Any destroyed keys will be recycled. Such a design has a potential drawback. Depending on problems, the rate of key generation may not be the same on each process. Then some processes may exhaust their available keys quicker than others. Dynamic balancing of keys among processes is a solution. However this involves costly parallel operations. A better and simpler solution is to keep this design and increase the available integer bits. For example, a 64-bit integer is practically enough to handle any problem in the foreseeable future. When partitioning a 64-bit integer in the key manager, we do not need to use the full 64 bits on each process. A 32-bit local key-space is enough for distributed memory machines since the addressable memory by a process is typically within 32 bits. Thus in such an environment, we can never have a process running out of keys locally. The key manager on each process would only need to record a 64-bit offset in the global space. All local operations can be done within the 32-bit space. This not only saves memory storage, but also increases processing speed.

Normally when generating or destroying keys, the key manager needs to perform a parallel synchronization operation to gather all the key distribution information. For ex-

ample, suppose on a three-process system, the initial key distribution follows: p_0: $\{0$–$9\}$, p_1: $\{100$–$105\}$, and p_2: $\{200$–$207\}$. Such a distribution needs to be kept synchronized on all the key managers since the distribution information is needed when resolving all the join patterns in a rule. Suppose p_0 destroys $\{0,3,4\}$ and p_1 generates $\{106\}$, then each process will need to gather the new key distribution: p_0: $\{1,2,5$–$9\}$, p_1:$\{100$–$106\}$, and p_2: $\{200$–$207\}$. This parallel communication step is costly if performed frequently. However for certain dynamic key-space, an optimization can remove this gathering requirement.

Proposition 7.1 Given a dynamic key-space K, for all the rules in K, if every join sequence follow the pattern: $t_1 \rightsquigarrow\ t_2 \rightarrow t_3 \rightarrow \ldots \rightarrow t_n$, then the key/process distribution information in K does not need to be maintained.

Proof: Given a key-space K, for any of a join sequence inside a rule in K: $t_1 \rightsquigarrow\ t_2 \rightarrow t_3 \rightarrow \ldots \rightarrow t_n$, let t denote the resulting table. Since the key-space tunnel is the first variable appearing in the join, then according to Definition 3.15 and Proposition 5.10, the $\mathrm{iimg}(t, K)$ is empty. Thus all the keys within K are not affected. ■

Remark 7.1 The particle codes presented in Codelets (6.8 – 6.13) satisfy this condition. And indeed such optimization is used to schedule those codes.

The run-time key-space topology evaluation is handled by proper graph processing. Recall from section 5.2.1, if a key-space has associated critical structure definition, then its topology is subject to be reevaluated at run-time. Consider the early example shown in Figure 7.3, if variable z is declared as the critical structure for the key-space that contains rule 2, then a key-space topology check point will be appended after z. This is easy to

justify: whenever a critical structure is recomputed, then the topology of the key-space may changes, therefore we need to reevaluate it again.

7.2 Performance Evaluation

To evaluate the practical performance of the dynamic key-space implementation and its interactions with the rest of the current implementation of the Loci framework, I continue to use the Lagrangian particle tracking example discussed throughout this entire dissertation. The dual-dec implementation is used to benchmark the complete dynamic key-space implementation discussed in this chapter as it exercises most of the new programming features created in chapter 5. The manual dual-dec code discussed in chapter 4 is used as the basis for performance comparison. It is not only a realistic engineering application of sufficient complexity, but also is a highly optimized parallel implementation.

7.2.1 Development Overview

Part of the manual dual-dec code is written utilizing the original Loci framework, particularly those parts that interact with the fluid field solver. All particle related codes are developed manually using the MPI library with the C++ programming language. The manual particle code has evolved for about one and half years. During this period of time, it is rewrote several times and is extensively polished and optimized as we further our understandings of the performance related issues, and the physics models used also gradually matured. The core of the manual dual-dec particle code currently consists of approximately 13861 lines of Loci and C++ code.

A functionally equivalent core dual-dec Lagrangian particle tracking code is also developed using the new KVR model and the programming interface discussed in chapter 5. (e.g., Codelets (6.8 – 6.13) presented in chapter 6 show a naïve particle location search implementation in the new programming abstraction). This new code (terms as the KVR particle code hereafter) uses the same physics model and algorithms as in the manual code. The manual and the KVR codes are verified to compute the same results for the same problem configuration. The new KVR particle code was developed in roughly three-week's time. Although much of the related design experience was shared with and borrowed from the manual particle code. The new KVR particle code currently consists of approximately 6899 lines of Loci and C++ code, which is about half of the manual code length. As a comparison, the manual particle code devotes approximately 7616 lines of code (mostly programming in MPI and data structure management) in managing the parallel structure and communication used in the particle walking algorithm and the particle-fluid coupling model. All of these efforts are not necessary in the new KVR particle code. Only a 122 lines of particle key-space definition is created, and the rest of the code is mostly devoted to the geometric and numerical algorithms and is also oblivious of parallellism. Thus much of the accidental cost in the development is removed in the KVR programming interfaces. In addition, the new KVR particle code is more composable than the manual particle code. For example, extending the physics model only requires the addition of new rules, instead of going through the entire code base and modify each function and data structure manually as is needed in the manual particle code.

7.2.2 Run-time Performance Benchmark

To assess the run-time performance of the new KVR particle code, the same combustor spray simulation as presented in section 4.5 is used. Two simulation scenarios are created: the first one starts with no particles in the combustor, but large numbers of particles are injected at each time step. The particles are only injected on a few parallel processes, thus such a setting will cause significant imbalance of particle number at each time step among all the participating processes, which in turn will trigger the key-space redistribution at each of its topology check points. This setting also causes large amount of keys being created at each time step, which also leads to intensive insertion operations in the trie data structure used as the fundamental container in the dynamic key-space. Therefore this test case is intended to expose the cost of rapid and frequent key creation and distribution. The second test case starts from roughly twelve million particles in the combustor model and no particle is injected during the simulated time steps. Thus the particle number will maintain perfect balance during the simulation and no key-space distribution will be triggered. This test case emphasizes the cost of dynamic control components (such as the dynamic data expansion management and the dynamic context evaluation, etc.) in the implementation of the dynamic key-space. The huge number of particles in this testing case will stress these components in the implementation.

Three combustor meshes consisting of approximately two, eight, and fifteen million cells are used. The same setting is run for both of the manual particle code and the newly developed KVR particle program. For the second test case, the I/O cost in reading in the existing restart particles are not included in the measurement, nor is the Loci scheduling

cost for the static fluid field solver measured in all of cases. A total of 100 simulated steps are timed for both codes on a diskless Linux cluster with SUSE Linux Enterprise Server 9. The cluster is composed of 512 Sun Microsystems SunFire X2200 M2 servers, each with two dual-core AMD Opteron 2218 processors (2.6GHz) and 8GB of memory. The system is interconnected using Gigabit Ethernet. Up to 32 nodes (128 processors) are used on the system for the measurements.

Table 7.3

Performance Under Rapid Particle Injection (2M Mesh)

init particle number = 0					
code	8p[a]	16p	32p	64p	128p
manual[b]	1785.4[c]	973.5	538.5	355.3	234.0
kvr[d]	1816.0	1008.6	579.5	401.0	343.6
diff (%)[e]	+1.7	+3.6	+7.6	+12.9	+46.8

[a]Parallel process number.

[b]Denotes the manual dual-dec Lagrangian particle tracking code.

[c]Result unit is second.

[d]Denotes the KVR dual-dec code.

[e]Denotes the timing difference of the KVR code relative to the manual code.

Tables (7.3 – 7.5) show the timing results for the rapid particle injection testing case. The new KVR particle code is slower than the manual particle code. This is usually to be expected as layers of abstraction usually add costs. The timing results suggest in most cases, the difference is not significant. There are some indication that under large number of processes, the KVR code produces more overhead (as percentage in the overall time)

Table 7.4

Performance Under Rapid Particle Injection (8M Mesh)

init particle number = 0			
code	32p	64p	128p
manual	1759.7	1018.5	643.8
kvr	1861.6	1099.3	768.4
diff (%)	+5.8	+7.9	+19.4

Table 7.5

Performance Under Rapid Particle Injection (15M Mesh)

init particle number = 0		
code	64p	128p
manual	1643.0	952.6
kvr	1739.4	1098.7
diff (%)	+5.9	+15.3

Table 7.6

Performance Under Stable Large Particle Number (2M Mesh)

particle number = 12M					
code	8p	16p	32p	64p	128p
manual	3429.5	1769.1	935.8	513.3	302.5
kvr	3874.3	2017.9	1095.0	623.1	394.2
diff (%)	+13.0	+14.1	+17.0	+21.4	+30.3

Table 7.7

Performance Under Stable Large Particle Number (8M Mesh)

particle number = 12M			
code	32p	64p	128p
manual	2657.0	1409.1	838.7
kvr	3109.2	1694.1	1044.8
diff (%)	+17.0	+20.2	+24.6

as in smaller number of processes. Following paragraphs will discuss this issue further. Tables (7.6 – 7.8) show the timing results for the stable large particle number testing case with no particle injection. These tests more accurately reflect the dynamic control components and key-set manipulation costs in the dynamic key-space implementation. The KVR particle code has in general more overhead under this testing case than in the first injection testing scenario. This in part is due to that the fluid solver takes more percentage of timing in the injection testing case, thus suppressing the particle component cost somewhat. On the other hand, this also suggests that the overhead of the dynamic key-space can grow larger with the number of particles.

Tables (7.3 – 7.8) gave the overall performance measurements. The results reflect the combined effects of the all the participating components in the program. The interactions of these different parts can be rather complicated and can cause unpredictable performance effects. In order to understand more clearly the fundamental cost in the present dynamic key-space implementation, we can identify the major steps that lead to overhead and measure their cost individually. Given that the KVR code is synthesized according to the basic strategies outlined in Algorithm 7.1 and the rest of the implementation sections, we can

Table 7.8

Performance Under Stable Large Particle Number (15M Mesh)

particle number = 12M		
code	64p	128p
manual	2221.8	1370.6
kvr	2607.1	1635.6
diff (%)	+17.3	+19.3

Table 7.9

Overhead of Dynamic Key-space (2M Mesh, Rapid Particle Injection)

mode	8p	16p	32p	64p	128p
theory[a]	6.7	5.5	5.4	7.0	12.2
actual[b]	30.6	35.1	41	45.7	109.6

[a]Denotes measurements of cost of the essential operations that are not in the manual code.

[b]Denotes the actual cost difference between the manual and KVR codes shown in Tables (7.3 – 7.8).

Table 7.10

Overhead of Dynamic Key-space (8M Mesh, Rapid Particle Injection)

code	32p	64p	128p
theory	5.3	6.8	13.0
actual	101.9	80.8	124.6

Table 7.11

Overhead of Dynamic Key-space (15M Mesh, Rapid Particle Injection)

code	64p	128p
theory	6.9	11.8
actual	96.4	146.1

Table 7.12

Overhead of Dynamic Key-space (2M Mesh, Stable Large Particle Number)

mode	8p	16p	32p	64p	128p
theory	256.9	138.4	66.7	33.5	17.7
actual	444.8	248.8	159.2	109.8	91.7

Table 7.13

Overhead of Dynamic Key-space (8M Mesh, Stable Large Particle Number)

code	32p	64p	128p
theory	114.8	51.2	33.6
actual	452.2	285.0	206.1

Table 7.14

Overhead of Dynamic Key-space (15M Mesh, Stable Large Particle Number)

code	64p	128p
theory	75.7	57.3
actual	385.3	265.0

indeed conclude that the steps involve key-set manipulation are missing from the manual code. They are mainly the steps that determine the context of each rule (such as that in line 3), and the steps to compute the key distribution that precedes the key-space topology evaluation. The manual code does not use the concept of "key" for the particles, hence it does not need to determine these information at the run-time. In this regard, it can be considered that the major overhead of the KVR model is its use of the key concept. In addition, since the current dynamic key-space uses the trie data structure, its run-time performance loss from data access to the tries is also an overhead that is not presented in the manual code.

Tables (7.9 – 7.14) show the measurements of such three individual components in the KVR code. The "theory" row in the tables represent the sum of the cost in the rule context evaluation step, key distribution step, and the performance loss from the trie data structures in the KVR particle code. This cost represent the currently unavoidable overhead in the dynamic key-space implementation. The "actual" row in Tables (7.9 – 7.14) is the actual timing difference between the manual and the KVR particle code taken from the previous results in Tables (7.3 – 7.8). These results show that the theoretically unavoidable costs in the KVR code is actually quite small. There is a large gap between the cost of these unavoidable components and the actually measured cost difference. We can notice that such a gap increases significantly in the injection cases (Tables (7.9 – 7.11)) as the process number increases, particularly under the 128-process cases. This indicate that there must be other sources of overhead. Either there are numerous of such overhead sources, or that a few significant overhead are overlooked.

Table 7.15

Actual Data Distribution Cost (2M Mesh, Rapid Particle Injection)

manual: 1 dist. kvr: 24 dist.					
code	8p	16p	32p	64p	128p
manual	7.5	6.9	6.9	8.3	11.0
kvr	11.5	14.0	15.9	20.8	70.3

Table 7.16

Actual Data Distribution Cost (8M Mesh, Rapid Particle Injection)

manual: 1 dist. kvr: 24 dist.			
code	32p	64p	128p
manual	7.3	8.1	11.1
kvr	16.7	20.5	73.8

Table 7.17

Actual Data Distribution Cost (15M Mesh, Rapid Particle Injection)

manual: 1 dist. kvr: 24 dist.		
code	64p	128p
manual	8.1	11.7
kvr	20.1	72.8

Tables (7.15 – 7.17) are measures for the actual cost of data redistribution in the manual and KVR code for the injection testing case. In the manual code, since all particles and their related data are stored in a list, there is only one data structure to redistribute. In the KVR code, the design emphasizes the program composition and in turn, all particle related data are scattered in different data structures. In this case, there are 24 data structures for the particle related information in the KVR code. The total data volume in both codes should be equal in theory, although in reality the KVR code could have a larger data volume due to the use of several additional copies and control variables. When redistributing data, the KVR code does not combine all these 24 structures, instead, they are communicated one by one. Combining them together has the effects of reducing the parallel communication start-up costs. However careful analysis is needed in order to combine structures with different domains. It is a future research work to determine whether an effective analysis can be found and its cost trade-off. With this setting, the KVR code performs significantly more communication steps in the data redistribution than the manual code. In the present dynamic key-space implementation, distribution of one data structure is guaranteed to incur two MPI collective calls. The following communication steps use point to point MPI calls and the actual number thus varies. With a rough estimation, the KVR code should have an order of magnitude more MPI calls in the data distribution step than that in the manual code.

The results in Tables (7.15 – 7.17) show that on process number up to 64, the actual cost difference between the two code is not significant. The amount of slowdown of the KVR code data distribution is expected due to its larger data volume and an order of mag-

nitude more MPI calls. However on 128 process, the gap becomes much larger. The exact reason is not yet determined. The current best explanation is that this is affected by the particular hardware setup. On the Linux cluster that the experiments were performed, 128 processors are grouped in a single rack. Faster network interconnections are used within a rack and lower performance interconnections are used between the racks. Therefore if communicating processes are in different racks, then the performance may be much worse than if they were in the same rack. The exact process-rack mapping is not possible to determine in the current system. However the cluster node allocation can be obtained and indeed the 128-processor request was allocated from processors on different racks. Since a full 128-process run will use all the available processors, then at least the 128-process testing cases involve inter-rack communication. Since the KVR code has much more MPI calls, it may become more susceptible to inter-rack communication performance drop. This could be one explanation for the large performance gap observed in the 128-process run. However, it is not clear if inter-rack communication is involved in other smaller process number cases. Also the amount of inter-rack communication may affect the final performance. Thus even if the 64-process run involves small amount of inter-rack process communication, the performance degradation may not be severe enough to be noticeable.

Table 7.18

Fluid Solver Comm. Timing (2M Mesh, Rapid Particle Injection)

code	8p	16p	32p	64p	128p
manual	143.5	110.5	80.4	78.9	66.3
kvr	116.6	161.2	115.2	122.8	96.1

Table 7.19

Fluid Solver Comm. Timing (8M Mesh, Rapid Particle Injection)

code	32p	64p	128p
manual	269.2	232.0	138.5
kvr	321.6	310.1	191.7

Table 7.20

Fluid Solver Comm. Timing (15M Mesh, Rapid Particle Injection)

code	64p	128p
manual	168.2	249.9
kvr	179.2	299.0

There is another suspected major source of overhead in the KVR code. It is observed that the MPI calls in the KVR code are much less synchronized than those in the manual code. This may be caused by an imbalance at one point in the code, and it is possible that this can lead to a chain reaction that causes multiple synchronization problems in the MPI calls. The overall effects of such asynchronism is that the wall time is slowed down since it is determined by the slowest process. The extent of this problem is hard to measure exactly. However there are evidence that it is responsible for a large portion of the theory and actual performance gap shown in Tables (7.9 – 7.14). Tables (7.18 – 7.23) show the measurements of the parallel communication time in the fluid solver for both the manual and the KVR codes. The fluid flow solver used in these two codes is the

Table 7.21

Fluid Solver Comm. Timing (2M Mesh, Stable Large Particle Number)

code	8p	16p	32p	64p	128p
manual	66.6	107.9	77.4	60.3	59.0
kvr	76.2	120.3	89.1	83.7	86.0

Table 7.22

Fluid Solver Comm. Timing (8M Mesh, Stable Large Particle Number)

code	32p	64p	128p
manual	279.5	209.9	131.5
kvr	288.6	249.5	171.9

Table 7.23

Fluid Solver Comm. Timing (15M Mesh, Stable Large Particle Number)

code	64p	128p
manual	188.4	261.7
kvr	221.9	284.2

same and ideally the timing results would be identical. However the actual measurements suggest that the fluid solver spent more time on parallel communication in the KVR code than in the manual code. This is a good indication that the communication routines in the KVR code are less synchronized. The only difference in the fluid flow solver in the manual and KVR particle codes is the scheduling order of rules. Since the particle part in the manual and the KVR codes are implemented differently, the overall scheduling order in the two codes are different. This can contribute to the increased cost in the fluid flow solver communication routines in the KVR code. There is also evidence that the particle communication routines in the KVR code have similar synchronization problem. This appears to be practically a major performance overhead in the KVR code. The consistent slowdown in the fluid flow solver communication time within the KVR code also suggests that the particle components in the KVR code are less balanced than the ones in the manual code. Although the reverse case is also possible, i.e., the fluid flow solver is causing the synchronization problem. Because the manual code has only one central particle list data structure, it is less susceptible to this effects. Identifying the exact cause of this problem can be hard. However its practical effects will be important. A careful research work is needed to identify the cause of the problem.

The performance analysis suggest that the present dynamic key-space implementation typically incur a 20% or less performance penalty compared to a highly tuned manual code. Whether this is an acceptable threshold depends on the individual problem and the practical needs. There are however several ways to further improve the dynamic key-space implementation. The measurements show that the essential unavoidable costs in

the dynamic key-space implementation are modest. The largest of the unavoidable costs is perhaps the dynamic rule context evaluation. It is an essential part in the KVR model and is not avoidable. Its cost is proportional to the number of keys involved. The manual particle code does not have this cost since the context in the code is implicitly assumed to be all existing particles. However for programs involve multiple components, the effective computational bounds for each function may also need to computed in a manual code. Therefore a manual program may also have such cost involved. Although in practice, a manual program may choose to use inexact bounds in order to simplify the logic and the amount of computations involved. If we allow annotation in the dynamic key-space to indicate the proper context for each rule, then this cost can be avoided. However in doing so, the present Loci framework loses one of its biggest advantages. In fact, the name of Loci is a reference to the capability of discovering the context for each rule. However adding this annotation may not be a bad idea. First, the choice of using annotation is at the user discretion. They can choose to not use such context annotation, then the system will behave the same (and incur certain costs at run-time). However when in the case the users can predict the rule context (as in the particle example), then they can simply annotate and avoid the run-time cost. Moreover, one can choose to annotate certain rules with context while leaving others to the system to determine. Also even if a rule is annotated with a context, the system may ignore it and opt for determining the context on its own. This can in effect to be used to perform a "bounds check" similar to array access implemented in some languages. If the system discovers that the automatically determined context does

not match the annotation, it can then issue proper warning. This is in fact a promising direction and is worth to be considered in the future work.

Another possible improvement is to find alternatives for the trie data structure. The current performance measurement suggests that the overhead incurred by the trie structure is small. This is however due to the mixture of computation with data access. In a program with heavy memory access, the trie structure may become a major performance bottleneck, although the chance for such a situation is small in numerical programs. One possible alternative is to use arrays to replace the trie. Then there is a need to perform renumbering on the keys accessible on each process. This situation is addressed in section 7.1.3 to be undesirable for highly dynamic key-space as the management overhead easily exceeds the data structure performance loss. Another alternative is to design smart containers for parallel data access use. It is hard to predict what the final specification for such a design would be. The goal is to design parallel container capable of managing sparse index with low space and access cost. The other direction to consider is to design containers with multiple views. For example, the manual particle code uses a linked list as the central data structure. The advantage is that there is very little memory copies involved in the manual code. In contrast, the KVR particle code has more data copies involved due to the data structure choice. This also contributes to the performance loss. The linked list lacks random access capability. However in the manual code, such random access capability is not needed, sequential access is enough to process all the particles in a list. One way to reduce the trie structure access cost is to also design a sequential access mode for it. The difficulty is to determine an access plan if the given index set is not contiguous. Another

way is to perform a global analysis to the program and convert any of the trie data structure to a list when random access is not required. Whether these strategies are effective needs to be evaluated practically.

The measurements also suggest that some of the "accidental" events actually dominate most of the current performance loss, such as the present data distribution routine is susceptible to low performance network and that the parallel process is less synchronized in the KVR code resulting in an increase in the turnaround time. Future research will need to address these questions.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

The study presented in this dissertation seeks to improve the development process for computational field simulation software, with a focus on modeling the essential components in CFS problems with a dynamic nature, such as those often exhibited by many multiphase flow simulation problems. The "key-value-ref" model is examined in detail and the "key-space" concept and its associated programming interface is developed as a new addition to the model. It is shown in this dissertation that the critical structures that distinguish the various CFS program formulation approaches can be modeled by the "key-value-ref" model with the key-space concept.

The benefit of such a model is two-fold. First, the newly developed programming interface clearly removes much of the accidental complexity in developing portable parallel CFS programs on modern distributed memory platform. Much of the resource management burden is removed as is clearly shown by the example source codes. More importantly, the model developed in this dissertation addresses several questions in the initial software design process where the most important decisions are often made. In essence, the model advocates a view where an object can choose to either initiate a computation or receive a computation, and that the computations are grouped by similar objects that initiate computations. Different object roles and grouping strategies result in different design

approaches that naturally correspond to the various parallel formulations of CFS problems that are widely considered standard. This model view also recognizes the essential performance cost trade-off in different design approaches by categorizing different groupings as static or dynamic, and it also proposes guidelines (e.g., as in Proposition 5.7) to craft a design that is likely to be able to yield reasonable performance. However one must note that the model itself does not solve problems, nor does it magically remove any fundamental performance cost in the resulting parallel program. The fundamental performance cost is, after all, fundamental. The model only exposes the trade-off in a more sensible way and offers suggestions to limit their scope so that fundamental decisions in the software are more easily made. The ultimate solution is certainly to be formulated by a designer.

In a short summary, there are several major contributions that this dissertation makes to the current body of knowledge regarding parallel high-performance computing research and portable parallel software development in general. First, it provides a comprehensive investigation of two competing parallelization strategies for the parallel Lagrangian dispersed multiphase flow problem. This investigation shows that a little used methodology has great potential to deliver reliable and automatic parallel performance, particularly on parallel computing hardware with low-latency and high-bandwidth networks. The second contribution is the design and implementation of the key-space concept in the key-value-ref model. It is shown that this new model allows for a compact representation of the parallel Lagrangian dispersed multiphase flow problem and is able to reveal the critical factors in the design of these classes of problems. It is also shown that the resulting programming system provides a significant reduction in the complexity of the implementation of these

algorithms and facilitates significant automation in the development of parallel software for this important class of problems.

The performance study of the current model implementation shows that there is a marginal cost (around 20% for larger problem, and less for smaller problem) compared to a highly optimized program designed in manual. The performance evaluation also suggests that the fundamental cost in the model under the current understanding is much smaller. Future work needs to investigate the specific cause of these non-essential performance drops and find solutions for improvements. However we should also note that the manual implementation used in the performance comparison is a much more specialized design that sacrifices the design flexibility in order to obtain performance. Much of the cost in the current model implementation comes from the cost associated with its great composability and guaranteed consistency. If a manually designed program were also to address these issues, then it is likely to incur similar costs. However one can easily choose the desirable levels of design within the current model and its programming interface. For example, should the absolute performance outweigh the software engineering benefits, then the model's cost can be greatly reduced by using appropriate annotations and a sophisticated design. Thus in this regard, the model and its implementation can be thought of as providing a large design space at a relatively low cost.

Regarding the model itself, perhaps the most needed follow-up research work is to investigate whether or not the key-space concept is able to provide essential insights to other dynamic CFS problems such as simulations with adaptive mesh refinement. With the key-space concept and the model in the current form, such applications can certainly

be described. A possible solution is to define a single dynamic key-space whose topology is linked to the refinement process. Then after the mesh if refined, the key-space topology will be automatically reevaluated and the entire program is rescheduled. However such a formulation can possibly incur large overhead. The essence of the current model is to encourage a design where many key-spaces interact. This design approach is more promising to limit the fundamental performance cost within a small region in a program, thus help improving the entire program performance. New semantics in the model such as key-space split, merge, and the conversion between dynamic and static key-spaces may need to be explored to support more design choices and also to provide further insights. With these developments, it is possible to evolve the model into a fundamental way of describing numerical computation similar to the role of the object-oriented model in describing common computational problems.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd edition, SIAM, 1999.

[2] S. V. Apte, K. Mahesh, P. Moin, and J. C. Oefelein, "Large-eddy Simulation of Swirling Particle-laden Flows in a Coaxial-jet Combustor," *International Journal of Multiphase Flow*, vol. 29, no. 8, 2003, pp. 1311–1331.

[3] N. G. Azari and S.-Y. Lee, "A New Approach to Task Decomposition for Parallel Particle-in-cell Simulation," *International Journal of Modelling and Simulation*, vol. 18, no. 3, 1998, pp. 239–250.

[4] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc Users Manual*, Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[5] K. Barker, N. Chrisochoides, J. Dobbelaere, D. Nave, and K. Pingali, "Data Movement and Control Substrate for Parallel Adaptive Applications," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 2, 2002, pp. 77–101.

[6] G. E. Blelloch, "Programming Parallel Algorithms," *Communications of the ACM*, vol. 39, no. 3, March 1996, pp. 85–97.

[7] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a Portable Nested Data-Parallel Language," *Journal of Parallel and Distributed Computing*, vol. 21, no. 1, April 1994, pp. 4–14.

[8] R. D. Blumofe, C. F. Joerg, B. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*, Santa Barbara, California, July 1995, pp. 207–216.

[9] C. E. Brennen, *Fundamentals of Multiphase Flow*, Cambridge University Press, 2005.

[10] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, anniversary edition, chapter 16, Addison-Wesley, 1995.

[11] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, anniversary edition, chapter 17, Addison-Wesley, 1995.

[12] D. L. Brown, G. S. Chesshire, W. D. Henshaw, and D. J. Quinlan, "OVERTURE: An Object-oriented Software System for Solving Partial Differential Equations in Serial and Parallel Environments," *Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, Minnesota, March 1997.

[13] D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.

[14] W. Camp, S. Plimption, B. Hendricson, and R. Leland, "Massively Parallel Methods for Engineering and Science Problems," *Communications of the ACM*, vol. 37, no. 4, April 1994, pp. 31–41.

[15] D. Cann, "Retire FORTRAN? A Debate Rekindled," *Communications of the ACM*, vol. 35, no. 8, August 1992, pp. 81–89.

[16] E. A. Carmona and L. J. Chandler, "On Parallel PIC Versatility and the Structure of Parallel PIC Approachs," *Concurrency: Practice and Experience*, vol. 9, no. 12, December 1997, pp. 1377–1405.

[17] Center for Integrated Turbulence Simulations, Stanford University, "LES Group Unstructured LES Code Applications," `http://www.stanford.edu/group/cits/research/combustor/applications.html`, (current 8 Feb. 2009).

[18] J.-R. C. Cheng, M. T. Jones, and P. E. Plassmann, "A Portable Software Architecture for Mesh-independent Particle Tracking Algorithms," *Parallel Algorithms and Applications*, vol. 19, no. 2–3, 2004, pp. 145–161.

[19] J.-R. C. Cheng and P. E. Plassmann, "A Parallel Particle Tracking Framework for Applications in Scientific Computing," *The Journal of Supercomputing*, vol. 28, 2004, pp. 149–164.

[20] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel, "Mobile Object Layer: A Runtime Substrate for Parallel Adaptive and Irregular Computations," *Avances in Engineering Software*, vol. 31, no. 8-9, August 2000, pp. 621–637.

[21] C. T. Crowe, ed., *Multiphase Flow Handbook*, CRC Press, 2005.

[22] D. Dailey and C. E. Leiserson, "Using Cilk to Write Multiprocessor Chess Programs," *The 9th International Conference on Advances in Computer Chess*, University of Paderborn, Germany, June 1999.

[23] C. J. Date, *An Introduction to Database Systems*, 8th edition, Addison Wesley, 2003.

[24] B. Di Martino, S. Briguglio, G. Vlad, and P. Sguazzero, "Parallel PIC Plasma Simulation Through Particle Decomposition Techniques," *Parallel Computing*, vol. 27, 2001, pp. 295–314.

[25] I. S. Duff and H. A. van der Vorst, "Developments and Trends in the Parallel Solution of Linear Systems," *Parallel Computing*, vol. 25, no. 13-14, December 1999, pp. 1931–1970.

[26] A. Faraj, X. Yuan, and P. Patarasuk, "A Message Scheduling Scheme for All-to-all Personalized Communication on Ethernet Switched Clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 2, February 2007, pp. 264–276.

[27] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A Report on the SISAL Language Project," *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, December 1990, pp. 349–366.

[28] I. Foster, "The Grid: A New Infrastructure for 21st Century Science," *Physics Today*, February 2002.

[29] S. D. Fraser, F. P. Brooks, Jr., M. Fowler, R. Lopez, A. Namioka, L. Northrop, D. L. Parnas, and D. Thomas, ""No Silver Bullet" Reloaded: Retrospective on "Essence and Accidents of Software Engineering"," *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, Montreal, Quebec, Canada, 2007, pp. 1026–1030, ACM, Panel Session.

[30] M. Frigo, "A Fast Fourier Transform Compiler," *Proceedings of ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, May 1999, pp. 169–180.

[31] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, 2005, pp. 216–231, Special Issue on Program Generation, Optimization, and Platform Adaptation.

[32] D. Gay and A. Aiken, "Memory Management with Explicit Regions," *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, 1998, pp. 313–323.

[33] T. George, *A Distributed Memory Implementation of Loci*, master's thesis, Mississippi State University, Mississippi State, Mississippi, December 2001.

[34] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf, "The Cactus Framework and Toolkit: Design and Applications," *Vector and Parallel Processing (VECPAR'2002) 5th International Conference*, 2003.

[35] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd edition, Addison-Wesley, 2003.

[36] A. Grama, A. Gupta, and V. Kumar, "Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures," *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*, vol. 1, no. 3, August 1993, pp. 12–21.

[37] M. W. Hall, J.-A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, vol. 29, no. 12, 1996, pp. 84–89.

[38] F. Ham, S. Apte, G. Iaccarino, X. Wu, M. Herrmann, G. Constantinescu, K. Mahesh, and P. Moin, *Unstructured LES of Reacting Multiphase Flows in Realistic Gas Turbine Combustors*, Annual research briefs, Center for Turbulence Research, Stanford Univ./NASA Ames, Stanford, California, 2003.

[39] W. D. Henshaw, "OVERTURE: An Object-Oriented Framework for Overlapping Grid Applications," *AIAA Conference on Applied Aerodynamics*, 2002.

[40] High Performance FORTRAN Forum, *High Performance FORTRAN Language Specification, Version 1.0*, Tech. Rep. CRPC-TR92225, Rice University, 1993.

[41] C. F. Joerg, *The Cilk System for Parallel Multithreaded Computing*, doctoral dissertation, Massachusetts Institute of Technology, January 1996.

[42] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, September 1993, pp. 91–108.

[43] L. V. Kale, S. Kumar, and K. Varadarajan, "A Framework for Collective Personalized Communication," *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[44] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. J. Humphrey, J. Reynders, S. Smith, and T. Williams, "Array Design and Expression Evaluation in POOMA II," *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Santa Fe, New Mexico, 1998, pp. 231–238, Springer-Verlag.

[45] G. Karypis and V. Kumar, "Multilevel Algorithms for Multi-constraint Graph Partitioning," *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, San Jose, CA, 1998, pp. 1–13, IEEE Computer Society.

[46] Karypis Lab, "The ParMETIS Software Package," `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`, (current 8 Feb. 2009).

[47] R. Kowalski, "Predicate Logic as a Programming Language," *Proceedings of IFIP Congress*, J. Rosenfeld, ed., North-Holland, Amsterdam, 1974, pp. 569–574.

[48] C. C. Lam, C.-H. Huang, and P. Sadayappan, "Optimal Algorithms for All-to-all Personalized Communication on Rings and Two Dimensional Tori," *Journal of Parallel and Distributed Computing*, vol. 43, no. 1, 1997, pp. 3–13.

[49] L. Lamport, *LaTeX: A Document Preparation System*, 2nd edition, Addison-Wesley, 1994.

[50] W.-K. Liao, C.-W. Ou, and S. Ranka, "Dynamic Alignment and Distribution of Irregularly Coupled Data Arrays for Scalable Parallelization of Particle-in-cell Problems," *Proceedings of the 10th International Parallel Processing Symposium*. 1996, pp. 57–61, IEEE Computer Society.

[51] R. Löhner, "Robust, Vectorized Search Algorithms for Interpolation on Unstructured Grids," *Journal of Computational Physics*, vol. 118, no. 2, May 1995, pp. 380–387.

[52] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí, "Parallel Functional Programming in Eden," *Journal of Functional Programming, Special Issue on Functional Approaches to High-Performance Parallel Programming*, vol. 15, no. 3, 2005, pp. 431–475.

[53] E. A. Luke, "Loci: A Deductive Framework for Graph-Based Algorithms," *Third International Symposium on Computing in Object-Oriented Parallel Environments*, S. Matsuoka, R. Oldehoeft, and M. Tholburn, eds. December 1999, number 1732 in Lecture Notes in Computer Science, pp. 142–153, Springer-Verlag.

[54] E. A. Luke, *A Rule-Based Specification System for Computational Fluid Dynamics*, doctoral dissertation, Mississippi State University, Mississippi State, Mississippi, December 1999.

[55] E. A. Luke, "On Robust and Accurate Arbitrary Polytope CFD Solvers," *Proceedings of the* 18^{th} *AIAA Computational Fluid Dynamics Conference*, Miami,FL, June 2007, AIAA, AIAA Paper #2007-3956.

[56] E. A. Luke and P. Cinnella, "Numerical Simulations of Mixtures of Fluids Using Upwind Algorithms," *Computers and Fluids*, vol. 36, December 2007, pp. 1547–1566.

[57] E. A. Luke and T. George, "Loci: A Rule-Based Framework for Parallel Multi-Disciplinary Simulation Synthesis," *Journal of Functional Programming, Special Issue on Functional Approaches to High-Performance Parallel Programming*, vol. 15, no. 03, 2005, pp. 477–502, Cambridge University Press.

[58] N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar, "Next-generation Generic Programming and its Application to Sparse Matrix Computations," *International Conference on Supercomputing*, 2000, pp. 88–99.

[59] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," `http://www.mpi-forum.org/`, (current 8 Feb. 2009).

[60] J. C. Mitchell, *Concepts in Programming Languages*, Cambridge University Press, 2003.

[61] P. Moin and G. Iaccarino, "Complex Effects in Large Eddy Simulations," *Complex Effects in Large Eddy Simulations*, vol. 56 of *Lecture Notes in Computational Science and Engineering*, Springer Berlin heidelberg, 2007, pp. 1–14.

[62] OpenMP Architecture Review Board, "The OpenMP Application Program Interface," `http://www.openmp.org`, (current 8 Feb. 2009).

[63] S. Peyton Jones, ed., *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, 2003.

[64] R. F. Pointon, P. W. Trinder, and H.-W. Loidl, "The Design and Implementation of Glasgow Distributed Haskell," *International Workshop on the Implementation of Functional Languages*. 2000, number 2011 in Lecture Notes in Computer Science, pp. 54–70, Springer-Verlag.

[65] K. H. Randall, *Cilk: Efficient Multithreaded Computing*, doctoral dissertation, Massachusetts Institute of Technology, June 1998.

[66] S. Ranka, R. V. Shankar, and K. A. Alsabti, "Many-to-many Personalized Communication with Bounded Traffic," *Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, 1995, pp. 20–27.

[67] L. Rauchwerger and D. A. Padua, "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization," *Conference on Programming Languages Design and Implementation*, La Jolla, CA, June 1995.

[68] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of ACM*, vol. 12, 1965, pp. 23–41.

[69] S. Rus and L. Rauchwerger, *Hybrid Dependence Analysis for Automatic Parallelization*, Tech. Rep. TR05-013, Texas A&M University, November 2005.

[70] Y. Saad, "ILUM: A Multi-elimination ILU Preconditioner for General Sparce Matrices," *SIAM Journal on Scientific Computing*, vol. 17, no. 4, 1996, pp. 830–847.

[71] Y. Saad and M. H. Schultz, "GMRES: A Generalized Minimum Residual Algorithm for Solving Nonsymmetric Linear Systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, July 1986, pp. 856–869.

[72] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-time Parallelization and Scheduling of Loops," *IEEE Transaction on Computers*, vol. 40, no. 5, May 1991, pp. 603–612.

[73] J. G. Siek and A. Lumsdaine, "The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra," *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Santa Fe, New Mexico, 1998, pp. 59–70, Springer-Verlag.

[74] K. Soni, *Work Replication: A Communication Optimization for Loci*, master's thesis, Mississippi State University, Mississippi State, Mississippi, December 2005.

[75] M. Sosonkina, Y. Saad, and X. Cai, "Using the Parallel Algebraic Recursive Multilevel Solver in Modern Physical Applications," *Future Generation Computer Systems*, vol. 20, no. 3, April 2004, pp. 489–500.

[76] R. M. Stallman and R. McGrath, *GNU Make: A Program for Directed Compilation*, Free Software Foundation, 2002.

[77] A. Stepanov and M. Lee, *The Standard Template Library*, Tech. Rep. 95-11(R.1), HP Laboratories, November 1995.

[78] J. F. Thompson, B. K. Soni, and N. P. Weatherill, eds., *Handbook of Grid Generation*, CRC Press, 1998.

[79] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones, "Algorithm + Strategy = Parallelism," *Journal of Functional Programming*, vol. 8, no. 1, January 1998, pp. 23–60.

[80] P. W. Trinder, H.-W. Loidl, and R. F. Pointon, "Parallel and Distributed Haskells," *Journal of Functional Programming*, vol. 12, no. 4 & 5, July 2002, pp. 469–510.

[81] J. D. Ullman, *Principles of Database and Knowledge-base Systems, Volume I: Classical Database Systems*, Computer Science Press, 1988.

[82] T. L. Veldhuizen, "Blitz++: The Library That Thinks It Is a Compiler," *Modern Software Tools for Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Springer-Verlag, 1997.

[83] T. L. Veldhuizen and D. Gannon, "Active Libraries: Rethinking the Roles of Compilers and Libraries," *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing (OO'98)*. 1999, SIAM Press.

[84] J. Watts, M. Rieffel, and S. Taylor, "A Load Balancing Technique for Multiphase Computations," *Proceedings of High Performance Computing '97*, 1997, pp. 15–20.

[85] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," *IEEE/ACM Conference on Supercomputing 1998*, 1998.

[86] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A High-performance Java Dialect," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, September-November 1998, pp. 825–836.

[87] Y. Zhang and E. A. Luke, "Dynamic Memory Management in the Loci Framework," *Scalable Computing: Practice and Experience*, vol. 7, no. 3, 2006, pp. 27–38.

[88] Y. Zhang and E. A. Luke, "Concurrent Composition using Loci," *IEEE/AIP Computing in Science and Engineering*, vol. 11, no. 3, May/June 2009, pp. 27–35.