Roxana Diaconescu

Object Based Concurrency for Data Parallel Applications: Programmability and Effectiveness

Department of Computer and Information Science
Norwegian University of Science and Technology
N-7491 Trondheim, Norway

URN:NBN:no-3329

NTNU Trondheim
Norges teknisk-naturvitenskapelige universitet
Doktor ingeniøravhandling 2002:83
ISBN 82-471-5483-8
ISSN 0809-103x
Institutt for datateknikk og informasjonsvitenskap
IDI-rapport 2002:09
ISSN 0802-6394

URN:NBN:no-3329

Abstract

Increased programmability for concurrent applications in distributed systems requires automatic support for some of the concurrent computing aspects. These are: the decomposition of a program into parallel threads, the mapping of threads to processors, the communication between threads, and synchronization among threads. Thus, a highly usable programming environment for data parallel applications strives to conceal data decomposition, data mapping, data communication, and data access synchronization.

This work investigates the problem of programmability and effectiveness for scientific, data parallel applications with irregular data layout. The complicating factor for such applications is the recursive, or indirection data structure representation. That is, an efficient parallel execution requires a data distribution and mapping that ensure data locality. However, the recursive and indirect representations yield poor physical data locality. We examine the techniques for efficient, load-balanced data partitioning and mapping for irregular data layouts. Moreover, in the presence of non-trivial parallelism and data dependences, a general data partitioning procedure complicates arbitrary locating distributed data across address spaces. We formulate the general data partitioning and mapping problems and show how a general data layout can be used to access data across address spaces in a location transparent manner.

Traditional data parallel models promote instruction level, or loop-level parallelism. Compiler transformations and optimizations for discovering and / or increasing parallelism for Fortran programs apply to regular applications. However, many data intensive applications are irregular (sparse matrix problems, applications that use general meshes, etc.). Discovering and exploiting fine-grain parallelism for applications that use indirection structures (e.g. indirection arrays, pointers) is very hard, or even impossible.

The work in this thesis explores a concurrent programming model that enables coarse-grain parallelism in a highly usable, efficient manner. Hence, it explores the issues of implicit parallelism in the context of objects as a means for encapsulating distributed data. The computation model results in a trivial SPMD (Single Program Multiple Data), where the non-trivial parallelism aspects are solved automatically.

This thesis makes the following contributions:

- It formulates the general data partitioning and mapping problems for data parallel applications. Based on these formulations, it describes an efficient distributed data consistency algorithm.

- It describes a data parallel object model suitable for regular and irregular data parallel applications. Moreover, it describes an original technique to map data to processors such as to preserve locality. It also presents an inter-object consistency scheme that tries to minimize communication.

- It brings evidence on the efficiency of the data partitioning and consistency schemes. It describes a prototype implementation of a system supporting implicit data parallelism through distributed objects. Finally, it presents results showing that the approach is scalable on various architectures (e.g. Linux clusters, SGI Origin 3800).

URN:NBN:no-3329

Table of Contents

List of Figures

List of Tables

Preface

This dissertation is submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree *Doktor Ingeniør*. This work has been completed at the Department of Computer and Information Science, NTNU, Trondheim. Parts of the work were completed during a research stay with the Stanford Suif Compiler Group, at the Computer Science Department, Stanford University, California, USA.

Acknowledgments

This work embodies support and contributions from many people. First, there are the committee members, whom I thank for the valuable comments helping to improve the quality of the thesis. I would like to thank Reidar Conradi for being my primary supervisor and for giving me the complete freedom in conducting this research. I would like to thank my second supervisor, Einar Rønquist for the valuable support and feedback related to the numerical analysis aspects of this work.

A special thank to Professor Monica Lam for inviting me to visit the Suif Compiler Group at Stanford and for her dedication in supervising and guiding me during a critical period of my research. I would like to thank all the members of the Suif group and the support staff at Stanford for making my stay unforgettable.

Many thanks to all the past and current members of the Software Engineering Group at NTNU for providing me with a nice and friendly research environment. Especially, I would like to thank Alf Inge Wang, Monica Divitini, Torgeir Dingsoyr, Ekaterina Prasolova-Forland, Tor Stålhane, Carl-Fredrik Sorensen and Maria Letizia Jaccheri. Special thanks to my colleague and friend Elisabeth Bayegan for sharing the four years of growing together. Thanks to my old friend and colleague Zoran Constantinescu for the valuable and endless discussions throughout the years.

As a part of the Computational Science and Engineering project at NTNU, I would like to thank all the members for creating a frame for collaborating and working with other departments at NTNU. Especially I would like to thank Syvert P. Nørsett, Brynjulf Owren, Anne Kværnø, Trond Kvamsdal, Richard E. Blake, Elena Celledoni, Hallgeir Melbø, Jens Holmen, Bård Skaflestad and many others.

Many thanks go to friends from the department for the social aspects of my graduate student life. Among these are Pauline Haddow, Marco Torchiano, Diego Federici, Daniel Moody, Christian Mönch, Amund Tveit, and many others.

Last, but not least, I owe the professors from my undergraduate school, Computer

Science Department of the Politehnica University of Bucharest (PUB), a deep thank for my formation.

A warm and hearty thank to my family for their precious support and encouragement all these years. Friends from Romania and all over the world have inspired me and contributed with their care throughout the years.

Roxana Diaconescu
August 9, 2002

Part I

Setting

Chapter 1

Introduction

1.1 Introduction

Concurrent programming initially evolved around operating systems design and implementation. However, for more than two decades this topic has found interest in other application areas such as distributed problem solving and planning in Artificial Intelligence (AI), real-time, embedded control systems, and large-scale, high-performance computations, just to name a few. In the past decade scientific computing has emerged as a field that uses concurrent computing as a means to decrease the execution time for applications in science or grand-challenge problems (e.g. avionics, computational biology, climate modeling, etc.) that would otherwise require extremely long computation time. The complexity of such problems solved by using parallel computing is ever-increasing, and the applications become larger and harder to manage.

While it is important to consider the low-level details of scientific concurrent computing to account for efficient execution, abstracting away from the machine architecture ensures conceptual generality. The concern for delivering high performance for these applications should interplay with the concern for a software development methodology to ensure not only efficient, but also robust, extendable and reusable concurrent programs.

Research into programmability of concurrent scientific applications has revolved around parallelizing compilers for Fortran languages for both shared-memory and distributed-memory architectures. Research into effectiveness of these applications has revolved around Fortran libraries tuned to exploit the underlying machine architecture. Although much of the insight into systematically exploiting parallelism is awarded to such “bottom-up” approaches, this is an extremely complex and narrow path and may lead to problems due to its tight relation to the language and architectural model. The Fortran language has an uncertain future since fewer and fewer professionals are trained to use it. Parallelizing compilers are thus very useful for transforming legacy applications without relying on human resources.

On the other hand, a “top-down” approach that considers the domain-specific abstractions and captures the infrastructure building blocks common across applications allows for more flexibility in adopting modern techniques for software development.

In the last decade, research on applying object-based[1] techniques to scientific concurrent applications has revealed new challenges. That is, it is hard to prove the effectiveness of the resulting end-user applications. On the other hand, many of these high-level solutions adopt the traditional Fortran approach towards exploiting fine-grain parallelism.

We are interested in shifting the approach towards the non-traditional, manual parallelized applications and study their characteristics. Thus, we explore the applicability of object-based techniques in the context of coarse-grain parallelism. We are interested in answering questions such as:

- What is the main source of parallelism in the scientific applications?
- What is the main dimension for decomposing a problem, and what is the granularity of such a decomposition to exploit the parallelism inherent in the applications?
- What are the high-level programming concepts that increase programmability, while guiding the system software to effectively exploit parallelism in scientific applications?

Traditional language support for concurrency expresses functional decomposition of a program along activities or control. Such decomposition is also called task parallelism. Data parallelism means that the same activity, or task, can be applied concurrently to different instances of data. Decomposing a data parallel program along activities, or control, results in fine-grain concurrency. Decomposing a data parallel program along data, results in coarse-grain parallelism. Shared-memory concurrency is suitable to express fine-grain and task parallelism. Distributed-memory concurrency is suitable to express coarse-grain and data parallelism.

This thesis explores the techniques for increasing the programmability and effectiveness for data parallel applications in distributed memory. The context of this work is that of object-based concurrency and data parallelism in the field of scientific, high performance computing. The work focuses on the development of a high-level programming environment to support the application writer in designing and implementing distributed-memory data parallel programs.

1.2 Objectives

Scientific computing is a non-traditional application domain for computer science, especially for software engineering. While its social dimension through the technological impact is indisputable, it is considered a very narrow niche compared with domains which impact every day life, such as information systems, business applications, etc.

[1]We use the term object-based to refer to both object-oriented approaches and approaches that have the notion of encapsulation and consequently, the notion of object, but do not meet all the requirements for an object-oriented language.

This thesis is an incremental contribution to improve the software methodology for concurrent scientific software development. The work has also a challenging technical dimension: parallel computing is not current software practice, it is close to low-level systems software and thus requires high expertise and good understanding of non-traditional computing architectures and programming concepts (e.g. concurrent programming).

Many aspects of what we may call automatic parallelization reduce to the ability to capture and model parallelism directly into a language or a programming environment such that a system automatically accounts for the concurrency aspects. This ability relies on the possibility to discriminate between different types of parallelism. The data parallel computation model is simple, symmetric and the commonalities between the applications that employ it suggest that is possible to capture the concurrency infrastructure in a uniform concurrency model.

The main objectives of this thesis are:

- To discuss and clarify the applicability of object-based techniques for concurrent computing in the context of scientific, data parallel applications, and to demonstrate their programmability benefit by devising a concurrency model reflecting the application behaviour.

- To discuss the effectiveness of the existing concurrency models for data parallel applications and to demonstrate the potential of loosely coupled, commodity-oriented distributed architectures in delivering high performance at low cost for coarse-grain, loosely synchronous parallel applications.

1.3 Results

This work is a combination of an adaptation of relevant theoretic base, programming and computer simulations. We make several assumptions when trying to meet our objectives. First of all, we start out with a specific class of applications, i.e. the solution of Partial Differential Equations (PDEs) for general geometries. Furthermore, we focus on the Finite Element Method (FEM) for such applications. The choice of the application class is due to the participation in the CSE (Computational Science and Engineering) project at NTNU.

Throughout the thesis we try to emphasize two aspects of the applications domain. One is the distinction between the regular and irregular applications. The other is the common denominator of data parallel applications and the synthesis of their main features to enable a uniform treatment. Our goal is to devise a uniform, general framework to treat coarse-grain data parallel applications.

Our approach occupies the middle ground between automatic and manual parallelization. That is, we devise a programming model for loosely synchronous data parallel applications based on distributed and sequential objects. Then, we show how data encapsulated in distributed objects is partitioned across multiple address spaces and consistency is ensured at run-time.

The validation of our approach is built on two pillars: a prototype system for distributed numerical computations and an application test bed. The prototype system offers support for automatic data layout and consistency for PDE applications that use general geometries. The test bed consists of trivial and non-trivial data parallel applications and two test problems. The experimentation base is not extensive. More applications need to be implemented in order to fully account for the generality of our approach. We present experimental data using the prototype system and the test bed applications for the two test problems we have had available. These problems are of a relatively small size compared with realistic, grand-challenge problems. We discuss the results in the light of the application features and explain potential benefits or drawbacks when experimenting with other classes of problems.

There are several drawbacks of such an approach. First of all, the approach is tailored for data parallel applications and thus it does not cover task parallelism. Moreover, the approach is not applicable to applications that exhibit fine-grain parallelism. That is, in the presence of high communication requirements, the distributed memory model can lead to significant performance losses. Finally, the experimentation base is not exhaustive and thus, further work is needed at both techniques and applications levels.

The main contributions of this thesis are:

- An object-based concurrency model for loosely synchronous data parallel applications in the light of programmability and effectiveness demands for the high performance scientific applications.

- A general data partitioning and mapping strategy that subsumes the existing strategies and allows a uniform treatment for decomposing a parallel problem along the data space.

- A distributed consistency scheme that uses the general partitioning strategy to maintain data consistency across address spaces in distributed memory. The efficiency aspects of this scheme are discussed in detail and results are presented showing that the approach is efficient.

- A prototype implementation of an object-oriented framework for distributed scientific computations. The framework implements features relevant for building and evaluating data parallel applications in numerical analysis domain, and offers a user-friendly environment to conduct experiments.

1.4 Thesis Outline

This thesis does not require the reader to have any knowledge of numerical analysis concepts. The main concepts are introduced from the perspective of their impact on software construction and parallelization. The main concepts of concurrency are also introduced. However, it is assumed that the reader has a basic understanding of shared-memory and distributed-memory architectures, system software and object-oriented concepts. The issues of object-based concurrency and data parallelism are presented in detail, as well as programming languages and libraries examples.

Most of the results chapters were published as papers in international conferences [39–42]. The remainder of the thesis is structured as follows:

Part II Introduces the main concepts of object-based concurrency, data parallelism and concurrent programming. This part is more than a state of the art for the fields the contribution in this thesis draws upon. It discusses the existing approaches, their advantages, applicability and shortcomings, and it motivates the need for the approach in this work.

Chapter 2 presents the main approaches to object-oriented / -based concurrency. It discusses the most important models that integrate objects and processes, namely the actor model, distributed shared object models and object-oriented middleware.

Chapter 3 presents the main concurrency model for scientific applications, i.e. data parallelism. It describes the main decomposition strategies along the data space together with criteria typically used when deciding upon an effective decomposition.

Chapter 4 presents the main concurrent programming models: shared memory, distributed memory and distributed shared memory. It describes the main concurrency concepts, i.e. synchronization and interprocess communication mechanisms, and presents examples of programming languages and libraries for concurrent programming.

Chapter 5 presents a usability framework as a guideline to assess the programmability, or the ease of use for a parallelization environment (e.g. programming language, library) with respect to which concurrency aspects are explicit or visible for a user. Thus, it discusses various approaches to make one or more aspects of concurrent programming implicit for a user.

Chapter 6 concludes the first part by relating the main concepts presented to the context, approach, main results and contributions of this work. This chapter describes the research method used, the main research questions and hypotheses for this work, together with the approach and the validation plan.

Part III presents the core results and contributions of this thesis.

Chapter 7 describes a general data layout for decomposing the data space of an application into multiple partitions and map these partitions to different processing units (tasks). It further describes how such decomposition can be used to ensure implicit data consistency for loosely synchronous data parallel applications in distributed memory. A discussion of related work and experimental results is included.

Chapter 8 discusses the efficiency aspects related to the distributed consistency scheme which uses the general data layout presented in Chapter 7. It shows that this approach exploits the applications features, i.e. loosely synchronous, coarse-grain parallelism and *nearest neighbour* communication, to ensure efficient data consistency. The chapter also discusses related work and presents experimental results.

Chapter 9 discusses the data parallel programming model in the context of high-level objects as a means to increase programmability. Our object-based model uses the decomposition and consistency strategies presented in Chapter 7 and Chapter 8 at a lower level to account for effectiveness.

Chapter 10 presents an inter-object concurrency model that uses the recursive *Set* concept to express irregular, recursive data structures most common for the applications that this work addresses. This model collects all the threads from the previous three chapters into a uniform model for concurrency based on the notion of *distributed recursive sets*. The chapter also discusses related work and presents experimental results.

Chapter 11 describes the design rationale, requirements and implementation for a user-friendly programming environment for building and evaluating data parallel numerical applications.

Part IV concludes this thesis.

Chapter 12 discusses the main findings of this research and summarizes the main results and contributions.

Chapter 13 presents some future directions for research.

Appendices containing index of terms, glossary of main concepts and more experimental data can be found at the end.

Even though the thesis is organized such that the information flows logically from one chapter to another, the chapters are self-contained. Thus, minimum explanations about the main assumptions or the relations with other chapters are given.

Part II

Preliminaries

Chapter 2

Object Oriented/Based Concurrency

2.1 Introduction

This chapter presents the main approaches to object-oriented / based concurrency. The main benefit of an object-oriented / based concurrent programming model is a higher-level of abstraction in which some of the lower-level concurrency aspects are implicit. However, this is not a complete review of the existing languages and systems (see [26, 112] for complete surveys).

The chapter describes three levels of integration between objects and processes. Each of the approaches is discussed in turn by using an example of language implementation. We describe how concurrency is achieved in each of the approaches and what kind of applications can benefit from it.

The three levels of integration between objects and processes are:

1. The distinction between objects and processes is not visible. Thus, they denote the same entity, an actor. Actors communicate only through messages. We discuss the actor model [3] and the ABCL / 1 [114] language as most representative for this approach.

2. The distinction between objects and processes is explicit. Processes are active entities that communicate through passive, shared objects. The processes may run in different address spaces. We discuss the Orca [16] language to illustrate such an approach.

3. The distinction between objects and threads of execution is explicit, in a multi-threaded execution over a shared memory. We discuss Java multi-threading [71] as representative for this approach. We also introduce object-oriented middleware.

The chapter is organized as follows: Section 2.2 discusses the actor model, the type of concurrency it reveals and its applicability. This section also introduces the concepts of *active and passive objects* originating from the actor model. Section 2.3 discusses a distributed programming model based on passive shared objects and its implementation in the Orca language. It further introduces *partitioned objects* to exploit data parallelism and their implementation in an enhancement of the Orca language. Finally, it

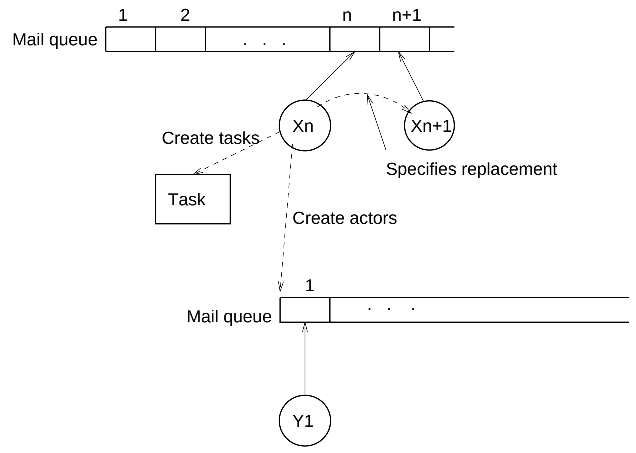


Figure 2.1: The behavior of an actor in terms of its possible actions.

discusses the Java multi-threading shared-memory concurrency model. Section 2.4 discusses the object-oriented distributed middleware and its underlying communication mechanism. Finally, Section 2.5 summarizes the main points of this chapter.

2.2 Actor-based Languages

2.2.1 The Actor Model

The actor model integrates the notions of objects and processes into object-oriented concurrency. The main benefit of the model is the implicit concurrency achieved by unifying the concepts of processes and objects under a common abstraction, i.e. an actor. In the original actor model proposed by Agha [2, 3], the basic unit in the language is an actor described by a mail address and behavior. The central idea of the actor model is to describe the behavior of an object as a function of the incoming communication.

The model: Actors encapsulate data and behavior. Actors represent history sensitive objects that allow the change of state variables. Therefore, actor languages extend the pure functional languages with "history sensitive" behavior needed to model shared mutable data objects. The "become" primitive enables state changes.

Actors take a functional view of an object's internal behavior. The behavior of an actor consists of three kinds of actions: send communications, create new actors and specify replacement behavior through *delegation*. The replacement behavior will accept the next communication. Figure 2.1 depicts the behavior of an actor. In Figure 2.1 a task is a communication together with its target actor.

Concurrency: Actors communicate only through messages in a point-to-point asynchronous buffered communication. Concurrency is achieved in actor languages in two ways:

1. By sending a number of messages in response to a single message and thus activate concurrently independent objects.

2. Through computation decomposition: an actor may create several customers which function concurrently with their creator.

The creation of a new actor enables the overlapping of communication and computation. An actor executes an action in response to a message. An actor may *delegate* another actor to take care of a request through the *replacement* behavior.

The replacement behavior **does not** update the variables of the old state. The replacement concept is a realization of the serialization mechanism, or pipelining (see Figure 2.1)and thus, it eliminates the disadvantages of the assign behavior. That is, excluding assignments to a store allows to concurrently evaluate behavior. Originally, the pure functional languages naturally provide this possibility.

Actor languages were originally designed for artificial intelligence applications. Actor languages model code sharing through replacement behavior (delegation) and modularity (actors encapsulate data and behavior). Because of this flexible structure they are suitable to rapid prototyping applications.

Actor languages express fine-grain, implicit concurrency. Moreover, they express task-level parallelism. Actor languages have no provision for expressing data-level coarse grain parallelism.

2.2.2 ABCL/1

ABCL/1 [114] is a language which has evolved from the actor model. The notion of objects is different from the notion of actors. The basic unit in ABCL/1 is an object having its own processing power and local persistent memory. Thus, in contrast to the actor model, this model is process based.

The interaction between objects is concurrent message passing. In contrast to the actor model, which allows only asynchronous communication, ABCL/1 has provisions for synchronous message passing.

The model: An object in ABCL/1 has three modes: dormant, active, or waiting. An object in the dormant mode does not perform any activity. An object becomes active as a result of an incoming request. The object decides whether to accept a request and what action to execute in response. While in active state, an object may wait for a message specified in the message pattern to arrive.

The behavior of an object is described by "scripts". The scripts specify the messages that the objects accept, and which actions to take upon receiving a message.

Concurrency: Actors find out about the identities of other actors that they "know". Actors communicate in a point-to-point message passing style. The transmission of messages is asynchronous, that is, an object may send a message whenever it desires. The reception of the messages is ordered, that is, the messages which arrive concurrently will be ordered, or serialized. Messages arriving to an object are organized in priority queues. Each object has two associated priority queues: the ordinary mode queue and the express mode queue. After completing an action in response to an incoming request, an actor selects the next request in the queue.

ABCL / 1 defines three types of messages for communication: *past* (send and no wait), *now* (send and wait) and *future* (reply to me later). The past messages model concurrency through asynchronous communication, as in the original actor model. Asynchronous communication allows overlapping of communication and computation and thus increases concurrency.

"Now" messages model synchronization of concurrent activities performed by independent objects when used in conjunction with the "parallel" construct. "Future" messages express relaxed synchronous communication. "Future" messages allow for overlapping of computation and communication, and thus increase concurrency for "now", synchronous communication. In the original actor model communication is only asynchronous and there are no "now" and "future" types of messages.

The language has provision for expressing explicit parallelism through the *parallel* and *multicasting* constructs. Parallelism can also be implicitly achieved through concurrent activations of independent objects – similar to actor languages and through "past" and "future" messages.

ABCL / 1 is suitable for applications such as distributed problem solving and planning in AI, modeling human cognitive processes, designing real-time and operating systems, and designing and constructing office information systems. The language is not designed for coarse-grain, data parallelism.

2.2.3 Concluding Remarks

The actor model introduces the notion of "active objects" that integrate the concepts of object and process. These concepts are similar in that both can be viewed as communicating encapsulated units. In ABCL / 1, an active object has its own private activity (own computational resource). Objects that are not active, are "passive". A passive object becomes active by responding to an incoming request. An active object becomes passive by completing the action associated with a request.

In the actor models there is no visible distinction between objects and processes. Such uniform models are elegant and have clean semantics. However, intra-object concurrency is difficult to implement, and it requires special mechanisms to ensure consistency. For efficiency reasons, fully concurrent objects are usually implemented as passive objects (standard objects without any activity) without any synchronization, and they are replicated on every processor [26].

Other models distinguish between data and behavioral objects [84]. Thus, one type of objects is used to structure shared memory as a collection of "passive objects", while a process is considered a special kind of "active process object". The actions on passive shared objects are performed according to their declared interface. The active objects access the shared objects using synchronization constructs. Thus, two active objects communicate only through a passive intermediary. Extending the model to a distributed-memory environment involves a hidden form of message passing. In a distributed memory setting, objects become "active" in response to communication.

In the next section we will present an object model that captures the second view of objects and concurrency, where the distinction between objects and processes is explicit, at the language level.

2.3 Distributed Shared Object Models

Many systems use the shared virtual memory (SVM) concept [74] to hide explicit message passing between different address spaces in a distributed-memory environment. This section discusses application-driven SVM implementations that ensure data consistency, rather than memory consistency. Thus the granularity of the consistency protocol is that of data objects, not transparent pages. This model of consistency can benefit from the object-oriented features with some system support.

The distributed shared object models promote the notion of virtually sharing data. These are software implementations of the virtually shared memory concept. In such approaches, data objects are replicated, instead of physical pages (as in shared virtual memory systems).

The Orca [15] language is an implementation of a shared-data object model that can be used to write parallel applications on loosely-coupled systems. Orca is an object-based language. That is, it supports objects that encapsulate data and define operations to access the data, but it does not support inheritance.

The model: Shared data is encapsulated in passive-data objects, which are variables of user-defined abstract data types. An abstract data type has two parts:

1. A specification of the operations that can be applied to objects of this type.
2. The implementation of the operations (declaration of the local variables of the object and code implementing the operations).

Concurrency: Orca supports task parallelism. Instances of an abstract data type can be created dynamically, each encapsulating the variables defined in the implementation part. These objects can be shared among multiple processes, typically running on different machines. Each process can apply operations to the object, which are listed in the specification part of the abstract type. The objects become a communication channel between the processes.

An operation execution has the following semantics:

1. All operations on a given object are executed atomically. The model guarantees serializability (loose, not strict), i.e. the sequential order is guaranteed, but not the order of the invocations.
2. All operations apply to single objects, so an operation invocation can modify at most one object. Making sequences of operations on different objects indivisible is the responsibility of the programmer.

The distribution of objects and their replication are system tasks, and thus they are hidden from the user. The user explicitly maps the tasks to processors through language constructs that specify for each process a processor to run on.

Access to remote data is done through replication to speed up access to shared data and decrease communication overhead. This also leads to increased parallelism by parallelizing simultaneous read operations.

There are several replication strategies:

1. With no replication each object is stored on one specific processor.

2. With full replication each object is replicated on all processors.

3. With partial replication each object is replicated on some of the processors, based on compile-time information, run-time information or a combination of both.

An advanced scheme is to let the run-time system decide dynamically where to replicate each object.

The system has to ensure consistency for the replicated objects. There are two main mechanisms for ensuring consistency:

1. Invalidation protocols. With this approach, an update of a data object invalidates all its replicas in other address spaces. Thus, a subsequent read access to a replicated object has to fetch the valid data from the processor that holds their last valid values.

2. Update protocols. With this approach, an update to a data object causes the update of all its replicas in other address spaces. Thus, on write, the updated data values are broadcast to all processors holding replicas of these data. A subsequent read will not require communication, since the local values are consistent.

The invalidation strategy has a write-once semantics: a write invalidates a copy. The update strategy has a write-through semantics: a write updates the copies. A design decision between invalidation or update schemes depends on the ratio between read and write operations. If read operations are more frequent than write operations, then update schemes are more efficient since they generate communication on write. If write operations are more frequent, then invalidation schemes are less expensive.

The authors indicate that Orca is "intended for high performance applications not particularly focused on banking, airline reservation systems, not for fine-grain, intra-object parallelism" [16]. However, it is obvious that the explicit distinction between shared objects and processes enables the expression of many of the distributed programming models, such as client / server, master / workers etc. Generally, task parallelism captures best distributed problem solving, in which functionality can be broken down into smaller activities that are solved separately. However, the shared objects in Orca and its consistency model increase the granularity of the concurrency of actor models.

2.3.1 Partitioned Objects

A later enhancement of Orca integrates task and data parallelism. However, data parallelism is here limited to regular data layouts and regular partitioning of data: "the advantage of data parallelism is that it is easy to use. The programmer merely specifies the distribution of data structures, and the compiler takes care of the low-level details, such as the generation of messages and synchronization" [55].

Note that the applications which use irregular data representations and general data partitioning functions are still data parallel. But they are not easy to parallelize.

The model: Data parallelism is achieved through a special kind of objects called partitioned objects. These objects partition shared information that is physically distributed over many processors.

The partitions are regularly distributed such that each processor owns a partition and replicates the remaining partitions. The computation model has the owner computes semantics: the processors do not observe updates on remote objects until the operation is completed. The owner can write the owned partition and only read the replicated data. In this approach the data is fully replicated in each address space. An update consistency scheme is employed on write operations. While the full replication of data increases concurrency on read, it results in inefficiency for applications that use very large data.

Parallelism is achieved through parallel operations defined on the partitioned objects. These operations apply to array-based structures. The user explicitly specifies the partitioning and distribution of data by using system functions.

The authors observe that "a suitable object partitioning that reflects the data locality in the program will result in lower communication overhead, because the unit of transfer will be an entire partition rather than a single element". Since the user specifies the partitioning, this is neither guaranteed nor controllable.

2.3.2 Multi-threaded Execution over Shared Memory

Concurrency in shared-memory environments is exploited through multi-threading. That is, multiple actions may be carried out concurrently by independent execution threads. A process may spawn multiple threads. Threads are light-weight processes that share the same address space. A process has its own address space. Thus, thread management (creation, termination, context switch) involves lower cost than process management.

Multi-threading exploits data parallelism by having one thread of computation per data item. The form of concurrency is fine-grain, block-level concurrency, sometimes called fork / join parallelism. That is, the sequential execution forks into multiple execution threads in the beginning of a parallel block, which join back into the sequential execution thread at the end of the parallel block. The High Performance Fortran (HPF) language models fine-grain concurrency model.

Java multi-threading is the object variant of the shared-memory concurrency. Many concurrent object models use this style of concurrency to integrate task and data parallelism in distributed shared memory environments. Thus, we discuss Java multi-threading in this section.

Concurrency: In Java, multiple activities may proceed concurrently by initiating an activity in a new thread, and causing it to proceed asynchronously. The language provides several ways to create concurrency:

- The `Runnable` interface (`java.lang.Runnable`): an interface that has only the `run()` method. In Java, interfaces say nothing about their implementation (or code). Objects of a class with a `run()` implementation assign the activity (code) implementing it to a thread of execution.

- The `Thread` class can be used to associate a runnable object with a thread. Thus, the thread will invoke the runnable objects' `run()` method on start. A thread can also be created independent of a runnable object.

There are several synchronization mechanisms in Java. Java guarantees that most primitive operations are atomic, except assignments to `long` and `double`. Java provides the `synchronize` keyword for method synchronization. By annotating methods with this keyword, one thread at the time executes the method code and obtains access to the object.

The code within a synchronized method may make a self-call to another method in the same object without blocking. However, `synchronize` does not ensure exclusive access between synchronized and unsynchronized methods.

Subclasses may override synchronized methods. The `synchronize` qualifier is not inherited, and if desired it must be specified; otherwise, the method is treated as unsynchronized. The methods declared in Java interfaces **cannot** be synchronized.

Java also allows for synchronization at the block level. That is, one thread at the time executes the code within the block and so obtains access to the object.

According to [71], passive objects model data while active objects model action (function). That is, passive objects encapsulate data that are used through their interface. Active objects model computational kernels, i.e. they do one thing at the time. Active objects may respond to action synchronously, or asynchronously. Passive objects are usually sequential objects in Java. Java threads allow active objects by creating a new asynchronous activity.

2.3.3 Concluding Remarks

The distributed shared object models target usability for distributed-memory programming by making the aspects of distribution implicit. These models either directly implement a virtually shared memory, or rely on an existing implementation of it.

In either case, a memory consistency model is considered. The consistency model can be at the hardware-, physical pages-, or data objects level. The memory consistency schemes which target general system support are conservative. However, relaxation schemes are possible as application knowledge is taken into account. Consistency models can be specialized to the extent that they only support one class of applications. While less applicable, such tailored models may offer good scalability for virtually shared memory.

2.4 Object-Oriented Middleware

The term middleware [73] refers to communication support for distributed applications. This level sits between network operating systems and applications. The middleware layer uses the primitives provided by the network operating system to provide higher-level communication support for distributed application writers. Examples of middleware include database (ODBC SQL), groupware (Lotus Notes), Internet (HTTP), and object (Corba, Java RMI) middleware.

Object-oriented middleware is based on a remote method invocation mechanism. The remote method invocation is the object variant of the remote procedure call (RPC). That is, the caller makes a call to a callee residing in a remote address space. The caller, sometimes called the client, is responsible for marshaling the actual arguments for the

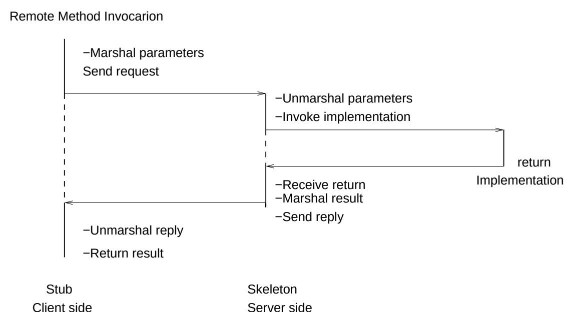


Figure 2.2: Java Remote Method Invocation.

procedure call and send it to the callee, or the server. The server is responsible for unmarshalling the parameters of the procedures, executing the procedure locally and sending the results back to the client. The client unmarshals the results and returns from the remote procedure call.

The remote call mechanism imposes overhead associated with the context switch. That is, marshaling and unmarshaling of the procedure arguments and results, message creation and the procedure execution itself. Moreover, the remote procedure call is synchronous: the client blocks on request and waits for the server to return the call.

Object-oriented middleware implementations add non-synchronous forms of communication to the RPC mechanism. These are one-way requests and deferred synchronous execution. One-way requests are asynchronous communication where the client does not wait for the result from the server. This form of communication increases concurrency by overlapping communication and computation and by allowing clients and servers to proceed concurrently. Deferred continuous execution is a relaxed form of synchronization in which the client can proceed for *a while* until the server delivers the results back. Asynchronous requests are usually implemented in object-oriented middleware by using multi-threading (e.g. a new thread may be spawned to take care of communication, concurrently with the computation thread).

2.4.1 Java Remote Method Invocation

Java RMI is designed to provide a direct foundation for distributed object-oriented computing. RMI is Java's remote procedure call (RPC). Java adds object-oriented features to the traditional RPC mechanism. That is, RMI can pass full objects as arguments, not just predefined data types (as with RPC). Java can move behavior (class implementations) between client and server.

Java RMI offers support for distributed garbage collection to collect remote server objects that are no longer referenced by any client in the network.

RMI is multi-threaded, allowing servers to exploit Java threads for the concurrent processing of client requests.

Figure 2.2 depicts the basic functionality of Java RMI. When a server is exported, its reference type is defined. When a client receives a reference to a server, RMI downloads a stub which translates calls on that reference into remote calls to the server. In Figure 2.2 the stub on the client side marshals the arguments to the method using method serialization, and sends the marshaled invocation to the server. On the server side, the call is received by the RMI system and connected to a skeleton. The skeleton unmarshals the arguments and invokes the server's implementation of the method. When the server's implementation completes, either by returning a value or by throwing an exception, the skeleton marshals the result and sends a reply to client's stub. The stub unmarshals the reply and either returns the value, or throws an exception as appropriate.

Stubs and skeletons are generated from the server implementation. Stubs use references to talk to the skeleton. The references can be for single or replicated servers, using unicast or multicast requests [56].

2.4.2 The CORBA Distributed Object Model

CORBA (Common Object Request Broker Architecture) is the OMG (Object Management Group) middleware solution and defines an open standard for distributed object computing. The central idea is to separate the specification of a class from its implementation to achieve platform independence in heterogeneous distributed environments. For this purpose CORBA defines an Interface Definition Language (IDL). IDL permits interfaces to objects to be defined independently of their implementations. After defining an interface in IDL, the interface definition is used as input to an IDL compiler which produces output that can be compiled and linked to an object implementation and its clients.

Client applications may invoke operations on object implementations in a location independent manner. CORBA adds to the RPC communication model object-oriented language features and design patterns for distributed communication.

The CORBA architecture is based on the Object Request Broker (ORB) which transparently implements RPC functionality on operation invocations. Thus, when a client invokes an operation, the ORB finds the object implementation, it transparently activates it and returns the response to the client.

The CORBA distributed object model is suitable for client / server applications, or applications based on a request / reply paradigm. One of the objectives of CORBA is to make it easier to build management and system integration applications. The idea is that distributed objects can act as gateways to existing heterogeneous information systems, integrating them into a uniform address space.

Peer (symmetric) applications (Single Programming Multiple Data - SPMD style) are cumbersome to express with the use of the CORBA client / server (asymmetric) computing model. Several studies indicate that conventional CORBA implementations have poor performance over high-speed networks [49]. The main sources of overhead are: non-optimized presentation layer conversions, data copying and memory management, excessive control information carried in request messages, and inefficient receiver-side demultiplexing and dispatching operations.

2.4.3 Concluding Remarks

Distributed object-oriented middleware offers client / server communication based on the remote procedure call mechanism. The main difference from the traditional RPC mechanism is that the middleware adds object-oriented features and thus allows application developers to easily build distributed applications. However, both CORBA and Java RMI add more layers to support location-independent invocations, and thus introduce large overheads in the remote method invocation [49, 63].

Some studies suggest [63] that for applications which transfer small amounts of information between objects, the performance of Java RMI and CORBA are similar. However, under heavy client load applications, CORBA may perform better. CORBA also offers more complex functionality than Java RMI. Therefore, it adds more architectural layers and thus overhead. These studies indicate that RMI is useful for simple applications, and CORBA is useful when the applications become more complex.

Generally, the client / server model does not directly address concurrency. While concurrency is achieved to some extent, this model is not suitable to effectively exploit task or data parallelism.

2.5 Summary

This chapter has presented three approaches to integrate concurrency into object-oriented languages. The main benefit of all these approaches is to elevate the level of abstraction and make low-level concurrency aspects transparent to the application developer. Throughout the discussion it was pointed out that each model exploits a specific type of concurrency and is suitable for certain kinds of applications. To summarize our conclusions:

- Actor languages capture fine-grain concurrency. They represent an elegant solution to implicit concurrency. These languages exploit task parallelism mostly through asynchronous message passing. However, they are hard to implement efficiently. ABCL / 1 evolved from the actor model. The main enhancement over the original actor model is to increase the concurrency granularity by adding synchronous communication and independent behavior to actors (own computation power). While achieving more flexibility in modeling realistic systems, the model makes some aspects of concurrency explicit.

- Some distributed shared object models make a distinction between process and object at the language level. Processes communicate through shared objects and thus these models also express task parallelism (since the concurrent accesses to objects are serialized). Multi-threaded fine-grain concurrency is usually employed to express data parallelism. With this approach, one thread is spawned per element computation. Java multi-threading is the object solution to fine-grain concurrency. Many concurrency aspects are explicit and a larger burden is placed on the programming side.

- Finally, object-oriented middleware is suitable to model synchronous, request / response applications based on the RPC mechanism in a distributed-memory environment. While addressing platform independence, this model does not naturally exploit concurrency.

None of the existing models directly addresses coarse-grain, data parallelism. Many existing approaches model fine-grain concurrency over shared virtual memory. We review such approaches in more detail in a later chapter.

Chapter 3

Data Parallelism

3.1 Introduction

Regular problems represent data objects as dense arrays as opposed to non-standard representations (e.g. special formats for sparse arrays). A large body of work to exploit data parallelism either through discovering parallelism into programs, or through modeling parallelism explicitly by using language facilities, addresses the regular problems.

Irregular problems represent data objects by using non-standard structures, such as indirection arrays (e.g. in Compressed Row Storage format - CRS - for sparse matrices three arrays are used: one array which lists all non-zero elements, one array which lists indices of rows for data, and one array which lists all indices of columns for data). The work on discovering parallelism in irregular data parallel applications is limited by the lack of language support to express irregular data structures. Data restructuring techniques are used for sparse array representations [22, 79, 105]. These techniques are based on the linear array representation as well. Many irregular applications use non-standard data representations such as graphs, trees, or general geometry meshes. These applications are hard or even impossible to parallelize with existing compiler and run-time support.

Data parallelism is exploited by decomposing a problem into subproblems which then can be solved concurrently. Early work on discovering parallelism based on the "independence constraint" proves that this requirement may be too strict, and it does not fully exploit the inherent parallelism in the applications. Coarse-grain parallelism in distributed memory means relaxing the independence constraint to increase the parallelism in the programs.

Throughout this chapter we use *data representation* to denote a data structure at the programming language or application level. We use *data layout* to denote the partitioning and mapping of data onto processors to exploit concurrency.

This chapter presents the main techniques to effectively exploit data parallelism in scientific applications together with their applicability. The chapter discusses the decomposition of data and computations of a program using the following perspectives:

- Data layout refers to the process of partitioning the data and mapping it onto processes in an architecture independent fashion. This chapter discusses the data layout along two axes:

  - Regular partitioning is a linear layout function that specifies a symbolic expression for mapping a variable[1] in the function definition domain to a value. There are two possible regular partitioning schemes: *block partitioning* and *cyclic partitioning*. Regular partitioning is suitable for multi-dimensional array representations.
  - General partitioning is not tied to a specific data representation or a computation model. It can be seen as a layout function that specifies a value for each variable in the function definition domain. Therefore, it does not say anything about how the values are obtained. Any partitioning function can be used, including symbolic expressions. *General graph partitioning* algorithms are suitable to express general partitioning.

- Optimal partitioning refers to the process of choosing the best partitioning strategy for a particular application, or class of applications. Some problems related to finding efficient data layouts are known to be NP complete. This chapter discusses data layouts according to the following *criteria*:
  - Execution models. One approach to finding the optimal data layout is to use a set of available data layouts together with a heuristic-based execution model that estimates the program execution time for a given layout. The data layout that requires the least execution time is the optimal layout.
  - Data locality. Another approach to efficient data layout is to preserve locality of reference when decomposing data or computations. Data locality can be tied to a particular architecture and thus exploit physical locality through registers, cache-replacement strategies, multi-word cache lines or local address space. Standard layouts (array based) exploit the linear memory organization. Non-standard layouts require special support to exploit locality. Efficient data locality strategies try to match the reference patterns to the underlying architectures (e.g. cache-based architectures).
  - Load balance. Efficient data layout requires that the same work load is mapped to each execution unit. Architectural considerations may dictate the estimation of the load. In homogeneous environments the amount of work is equally divided between the processing units. In heterogeneous environments, the load balance needs to account for the differences in the processing characteristics (speed, load, etc.).

The remainder of this chapter is organized as follows: Section 3.2 discusses data partitioning and mapping strategies. It describes the general formulation of the data partitioning problem together with some practical regular and general partitioning strategies such as *block*, *cyclic* and *general graph* partitioning. Section 3.3 discusses existing criteria for finding an optimal data layout. It discusses various execution models to select the best mapping strategy. It also discusses the data locality and load

[1]The word variable in relation to the partitioning function is used with its mathematical sense, rather than in a programming language context.

balance criteria to find the most effective data layout. Finally, Section 3.4 summarizes the chapter.

3.2 Data Partitioning and Mapping

This section formulates the data partitioning problem as described in [107]. Furthermore, it discusses various practical partitioning and mapping strategies, their rationale and applicability.

The notation in [107] is general and thus not necessarily tied to a specific execution model (architecture) or programming language. A data distribution is a mathematical function, called a map, that specifies how a bounded set of data items, called an index space, is distributed over a bounded set of processes.

General Problem Formulation

Let $\mathcal{I}_p$ be an index set as an arbitrary set of integers with the range $\overline{0, I_p - 1}$.

Definition 3.1 *A P-fold distribution of an index set $\mathcal{M}$ is a bijective map $\mu(m) = (p,i)$ such that:*

$$\mu :: \mathcal{M} \to (p,i) | 0 \leq p < P \text{ and } i \in \mathcal{I}_p : m \to (p,i)$$
$$\textit{and}$$
$$\mu^{-1} :: \{(p,i) : 0 \leq p < P \text{ and } i \in \mathcal{I}_p\} \to \mathcal{M} : (p,i) \to m$$

The definition describes a data distribution by means of a bijective map of the *global index m* to a pair (p,i), where p is the process identifier and i the *local index*.

Let $\mathcal{M} = \overline{0, M-1}$ be the set of global indices. For a given $p \in \overline{0, P-1}$, the subset $\mathcal{M}_p$ is the subset of global indices mapped to processor p by the P-fold data distribution μ. That is

$$\mathcal{M}_p = \cup_{i \in \mathcal{I}_p} \{\mu^{-1}(p,i)\}.$$

Because the P subsets $\mathcal{M}_p$ with $0 \leq p < P$ satisfy

$$\mathcal{M} = \cup_{p=0}^{P-1} \mathcal{M}_p \text{ and } \mathcal{M}_p \cap \mathcal{M}_q = \emptyset \text{ if } p \neq q,$$

they define a partition of $\mathcal{M}$. Because μ is a bijective map, the number of elements of M_p equals the number of elements of $\mathcal{I}_p$, or

$$\forall p \in \overline{0, P-1} : M_p = |\mathcal{M}_p| = |\mathcal{I}_p| = I_p$$

Practical Views on Partitioning

A concrete partitioning choice is determined by the underlying execution platform architecture, operating system and compiler support, as well as programming language support. For example, early work on partitioning and mapping requires that the sizes of data sets and the number of processing units are known at compile-time. Thus, these approaches promote static data partitioning. One advantage of static data partitioning is that it eliminates the run-time overhead. With such approaches, the index set in the general partitioning problem formulation is mapped to the iteration set for the sequential loop parallelization. Such approaches exploit fine-grain, loop-level parallelism.

A computation decomposition [58] splits the program into tasks that can execute concurrently on a multiprocessor. Data partitioning refers to the distribution of a large global data set (usually the iteration space) among concurrently executing processors. Data partitioning in sequentially iterated loops divides the index sets of the inner loops among the processors. Each processor executes the same code on a contained area of a data set, typically communicating with processors that work on neighbouring areas.

Later work adds run-time elements to the partitioning process. Wholey [110] proposes that the data mapping process is carried out in two phases, by two processes:

1. Alignment: a compile-time process that determines the relationships between mappings of different collections of data. The alignment information is derived from the structure of the program and is independent of quantities that are potentially unknown until run-time (input data size, number of processors).

2. Layout: an architecture dependent run-time process that chooses the final mapping based on run-time information (data size, number of processors) and a cost model of the target machine to minimize the estimated cost of the program. The layout process searches the space of possible data mappings, subject to the constraints of alignment, for a data mapping that minimizes the estimated cost.

High Performance Fortran (HPF) adds concurrency features to the Fortran language. HPF allows a programmer to specify how data is distributed over processors. Kennedy et. al. [68] propose an automatic data layout in HPF [68] that is specified by the alignment of data and its distribution. In HPF, arrays are aligned relative to each other by specifying a mapping of their elements to the same array of virtual processors called “template”. The array of virtual processors is then distributed over the physical processors. The HPF data layout and the concurrency model are representative of the existing data parallel frameworks.

All these views on data layout are coupled with the assumption that the information on data accesses can be computed statically, in a precise or symbolic manner. This usually holds in the context of Fortran applications with array accesses as affine expressions of loop indices. These approaches rely on the linearity and regularity of the data structures, and on the ability to align structures with respect to themselves and in memory to exploit the locality of reference. Therefore, these views apply to regular applications.

3.2.1 Block Partitioning

Block partitioning is at the core of exploiting parallelism in regular and even sometimes irregular applications. The mathematical formulation of this distribution function is given in [107], and is called linear distribution.

A linear distribution allocates consecutive array elements to consecutive local-array entries. We give the formulas for the general *load-balanced linear distribution*, where $M = PL + R$, and $0 \leq R < P$:

$$\begin{aligned} L &= \lfloor \frac{M}{P} \rfloor \\ R &= M \; mod \; P \\ \mu(m) &= (p, i), where \begin{cases} p = max(\lfloor \frac{m}{L+1} \rfloor, \lfloor \frac{m-R}{L} \rfloor) \\ i = m - pL - min(p, R) \end{cases} \end{aligned} \tag{3.1}$$

This maps $L + 1$ components to processes 0 through $R - 1$, and L components to the remaining processes. Other data-distribution quantities are given by:

$$\begin{aligned} \mathcal{M}_p &= \{m : pL + min(p, R) \leq m < (p+1)L + min(p+1, R)\} \\ I_p &= \lfloor \frac{M + P - p - 1}{P} \rfloor \\ \mathcal{I}_p &= \{i : 0 \leq i < I_p\} \\ \mu^{-1}(p, i) &= pL + min(p, R) + i. \end{aligned} \tag{3.2}$$

The block distribution is suitable for dense array structures. Due to its straightforward formulation and mapping to the linear array structure, many parallelizing compilers use this distribution to symbolically compute data accesses across multiple address spaces. However, for irregular applications, the block distribution does not reflect the locality of references. Nevertheless, some of the existing approaches to automatic parallelization for irregular applications use this distribution. In the manual parallelization approach, the application writers use non-standard distributions to obtain good scalability.

One approach that considers the application structure to adjust the standard block distribution, provides a general block distribution that generalizes the existing block distributions [36]. That is, a fully general block distribution partitions the elements of an array into rectangular blocks whose sides are parallel to the data coordinate axes. Different blocks can have different sizes. A *block* in *d*-dimensions is a rectangular parallelepiped in *d*-dimensions and can be specified by two vectors (e.g one of its vertices and the extent of the parallelepiped in each dimension). This approach addresses the requirements of a special class of applications, the semi-structured applications. The computation in semi-structured methods is characterized by irregularly organized regular computations on the underlying data. The underlying data is specified as *grid components* which are organized in an irregular fashion in a *grid hierarchy* which itself changes dynamically.

In contrast to this fully general distribution, for dense programs, block and cyclic distributions (as in HPF) are standard, and a simple rule like the owner-computes is used to determine the iterations to be performed on each processor. In this case, closed-form linear integer constraints can be used to express the local storage requirements, the local iteration sets, communication sets, as well as placement of communication. This does not apply to the semi-structured applications as described in [36]. Thus, the fully general block distribution partitions the elements of an array into regular blocks whose sides are parallel to the data coordinate axes. A fully general distribution of a data item D specifies a tuple of the form (B_i, P_i), where B_i is a block in d-dimensions defined as a rectangular parallelepiped in d-dimensions.

3.2.2 Cyclic Partitioning

Cyclic partitioning is sometimes also called scatter distribution [107]. Cyclic partitioning, as block partitioning, is a form of regular data distribution that also is applicable to linear array structures. The cyclic distribution leads to even less complex formulations than the block distribution. We give the precise mathematical formulation below, according to [107].

The *scatter distribution* allocates consecutive vector components to consecutive processors. To distribute the index set $\mathcal{M} = \overline{0, M-1}$ over P processors by the scatter distribution, let [107]:

$$\mu(m) = (p, i), where \begin{cases} p = m \; mod \; P \\ i = \lfloor \frac{m}{P} \rfloor \end{cases} \tag{3.3}$$

It follows that:

$$\begin{aligned} \mathcal{M}_p &= m : 0 \leq m < M \; and \; m \; mod \; P = p \\ I_p &= \lfloor \frac{M+P-p-1}{P} \rfloor \\ \mathcal{I}_p &= i : 0 \leq i < I_p \\ \mu^{-1}(p,i) &= iP + p \end{aligned} \tag{3.4}$$

Cyclic partitioning is useful for programs that do not require overlapping input data sets on neighbouring processors (otherwise each data item would have to be replicated on several processors).

3.2.3 General Partitioning

A general partitioning procedure maps the global data space (index set in the above formulations) onto multiple address spaces through a bijective map which specifies a unique mapping location in an address space for each global data item. Therefore, this function is not specified by a symbolic expression, but in a discrete fashion, for every variable in the input domain.

Not using a symbolic expression for the partitioning function does not mean that the partitioning should be random, as suggested in [107], where the general distribution is identified with the *random distribution*. The *random distribution* maps vector components to array entries by means of a random number generator. If the amount of work for each index is unpredictable, this is useful to balance the load statically. The main disadvantage of the random distribution is that a table of length M must be stored in every processor. This table is used to compute the maps μ and μ^{-1}, which relate global indices to local indices and processor identifiers. Moreover, the random distribution almost never outperforms the scatter distribution.

However, the argument of the *random distribution* does not hold, since the layout function can, and should be, computed by other means than random number generation. One possibility is to use an algorithm which takes as input the function definition domain (global references) and produces the mapping values (local references and processor identifiers). Graph partitioning algorithms are suitable for such situations.

The graph representation is one of the most common data representations in irregular numerical applications. Therefore, graph partitioning algorithms are a suitable partitioning function choice. The disadvantage of the "random" distribution holds for this technique too, since there is no way to relate global references to local references and process identifiers other than recording the information explicitly. However, the mapping table does not have to have length M as suggested; rather it can be distributed over the processes. The distribution of the mapping table can also be tuned to exploit the application characteristics.

General partitioning is useful for data representations that do not resemble the linear array structure. Such representations are non-standard and are usually the choice of the application writer. Therefore, the system support for such applications is very limited. The general graph partitioning problem can be formulated as follows [33]:

Definition 3.2 *Given a graph $G = (V, E)$ of $|V| = n$ vertices, partition V into k subsets, $V_1, V_2, ..., V_k$ such that $V_i \cap V_j = \emptyset$ for $i \neq j, \cup_{1 \leq i \leq k} V_i = V$, and the maximum of a cost function f over all V_i is minimized:*

$$Min(Max_{1 \leq i \leq k} f).$$

Usually the cost function f is the execution estimate of a given application in a distributed system. Graph partitioning has been proven to be NP-complete. Existing work focuses on heuristics to solve this problem.

3.3 Criteria

Traditional decompositions of data and computations to exploit concurrency try to find partitions that make the computations independent. In distributed memory, such independence means no communication. Later practice proves that this constraint is too strict and better scalability is usually obtained if some dependences between computations are allowed, and consequently communication exists. Therefore, the partitioning objective becomes how to obtain *minimum communication*.

Compiler support for parallelization in this context translates into discovering parallelism in a loop and generating communication according to data dependences. Data dependence analysis is used to generate communication. Conservative static program analysis leads to redundant communication and inefficient parallel executions. Subsequent optimization techniques are used to eliminate redundant communication.

Several optimization techniques to exploit data locality or minimize communication start with a "perfect partitioning" or an "existing user-provided" partitioning. Some researchers have concluded that such decoupling between partitioning and locality optimization leads to poor results [58]. Thus, the partitioning should be automatic.

This section presents the main directions for partitioning objectives along with examples of existing approaches. Some of the approaches assume decomposition of data and computation, while others suggest automatic data layout (partitioning and mapping). The section divides the techniques in classes according to the partitioning objective and possible heuristics to estimate it.

3.3.1 Execution Models

One way to choose between different partitioning strategies is to use an execution model. It is difficult to find an accurate static execution model. A static approximation is suitable for an initial guess that can be refined and improved using a static or dynamic analysis. This section presents commonly used partitioning strategies and execution models. The automatic layout is based on mapping techniques and execution models. Automatic data layout has many advantages. On the one hand, it alleviates the user from the difficult task of partitioning the data. On the other hand, it can use program knowledge to decide on a partitioning that preserves data locality.

Sussman [104] describes such an execution model. Here the data partitioning divides the input data sets across the processors in the linear array, so that each processor can compute its output in parallel. The automatic mapping consists of:

1. Mapping techniques. The compiler writer selects a set of techniques (block, cyclic).

2. Execution models. The compiler writer builds execution models to predict the run-time behavior of the program, using the maps from the previous step.

The main assumption in this approach is that the program is already transformed to expose parallelism. Two different cost approximation strategies are described, for block and cyclic distribution respectively. Balasundaram et. al. [18] present a static performance estimator to guide data partitioning decisions. This approach uses a static performance estimation scheme both in a compiler, which makes decisions between code alternatives, and in a programming environment to predict the implications of data partitioning decisions. The system uses a “training set” of kernel routines to test primitive computational operations and communication patterns on the target machine.

This approach is based on data partitions that are regular and assumes that the data domain is partitioned uniformly, i.e. the partitions have the same shape. The assumed execution model is that of a distributed-memory program with “loose synchronization” requirements. Therefore, all processors execute the same node program based on the strict owner-computes rule (a processor’s node program can only involve data items contained in the processor’s local memory). This means that the writing of the node program implicitly defines the partitioning of the data domain.

The authors indicate that this model may not be well suited for temporally irregular problems. The method has two initialization steps and an estimation algorithm:

1. In one initialization phase, a set of training routines is run on a target machine (e.g. arithmetic and control flow operations, etc.). The output of this phase is a file containing the average time for the control and communication operations.

2. The other initialization phase analyzes the performance data and formats this to be used by the estimation algorithm. The output of this phase is a formated file containing the execution time estimations.

3. The performance estimation algorithm uses the arithmetic / control and communication data from the previous step to determine an execution and communication time estimate. The input of the algorithm is a node program statement (simple or compound). The output is an estimate for the execution and communication time.

The static performance estimators assume that parameters such as loop iteration counts and branch frequencies are known or can be approximated [1]. Common performance metrics are: communication volume, communication frequency, parallelism achievable, extent of load-imbalance, communication latency, and communication overhead. The chief difficulty in an accurate static performance analysis is to obtain key profiling parameters such as loop iteration counts, branch frequencies, cache miss rates, etc.

In HPF the data layout is explicitly specified by the programmer. Kennedy et. al. [68] describe an approach to automatic data layout for HPF. There are four steps in automatic data layout:

1. Partition the input program into segments. A segment or a phase is the outermost loop in a loop nest such that the loop defines an induction variable that occurs in a subscript expression of an array reference in the loop body. This step constructs the *phase control flow graph* (PCFG).

2. Construct a search space of promising candidate layouts. This step uses heuristics to determine a reasonably sized set of alignment candidates that will guarantee a good overall performance for all applications. The step performs an interdimensional alignment analysis by using the *component affinity graph* (CAG). In the CAG there is one node for each array dimension, and an edge between nodes expresses alignment preferences between dimensions of distinct arrays. In a distribution analysis, a candidate distribution can map single template dimensions either by block, cyclic or block-cyclic partitionings onto the target architecture, or replicate dimensions on each processor.

3. Evaluate the candidate layout in terms of execution time. This step determines the costs of possible remappings between candidate layouts. Each candidate data layout is evaluated in terms of its expected execution time for each phase.

4. Select a single candidate layout from each search space, based on the estimated candidate layout costs and the cost of remapping between layouts. The overall cost is determined by the cost of each selected layout and the required remapping costs between selected layouts.

Fortran D and HPF compilers normally assume that they generate code for a dedicated machine of a known size [30]. However, the mapping decisions typically depend on run-time parameters. Many prototype HPF compilers (including Rice Fortran 77D) require that the number of processors are known at compile-time.

The execution models presented in this section rely on the possibility to estimate the execution time for each program statement and function call. Such an approximation is usually a guideline for an initial partitioning, and cannot reflect run-time behavior accurately. Run-time information or heuristics can be used to refine the initial guess.

3.3.2 Data Locality

Data locality is a partitioning objective or an optimization target for eliminating expensive data accesses (to different cache lines in cache-based memory architectures or remote address spaces in distributed-memory systems).

In shared-memory systems, data locality means that it is efficient to fit data that are accessed frequently or in the same proximity into the fast memory (cache). In distributed-memory systems, data locality means that it is efficient to place data whose accesses are close in time and space in the same processor or in the same address space to reduce communication.

Minimizing communication is one objective for a partitioning algorithm. However, the efficiency of a parallel algorithm is not given by the amount of communication, but by the ratio of communication to computation on a given node.

Traditional work on parallelizing compilers uses *dependence vectors* to represent data dependence information and exploit concurrency through independent executions. Hudak et. al. [58] use *access vectors* that carry communication information, rather than dependence information. The definition of such vectors is based on the array accesses and their representation as affine expressions (linear expressions) of loop indices. This approach finds a geometric partitioning that minimizes communication based on the communication information. Communication is quantified through a *weight* defined by the number of data points needed to cross partition boundaries in the newly formed partitions.

The idea of geometric partitioning is to use geometric shapes to partition a matrix. Subsequently geometrical computations are used to quantify communication. For instance, in [58] straight line perimeters are used. The perimeter of each part is composed of line segments whose slopes are rational numbers. The idea is to pair line segments that have the same orientation and measure the communication weight for a line segment pair.

Geometrical partitioning applies only if information on data references exists. That is, the vector accesses can be computed if and only if they are direct accesses of loop indexes or affine expressions of loop indices.

The linear array structure fits well with the memory linearity and linear array layout in memory. That is, in programming languages such as C and Fortran, arrays are laid out in memory in a line-wise or column-wise format. Thus, it is efficient to restructure computations so that consecutive accesses are in the same line, or column, respectively. It is not always possible to take advantage of such regularities and perform a direct mapping of the data structure to memory layout.

Chatterjee et. al. [32] describe a non-linear array layout that exploits the hierarchical memory systems. The thesis of this approach is that "the best performance results when algorithmic patterns of locality of reference match the patterns that the cache organization performs as well".

When data locality cannot be exploited by traditional means (e.g through cache replacement policies, etc.), data and computations can be restructured to fit the locality requirements. Restructuring work is classified along two axes:

1. Control flow restructuring. Loop tiling (or blocking) is a program transformation that tessellates the iteration space of a loop nest with uniform tiles of a given size

and shape, and which schedules the tiles for execution in an order consistent with the original data dependences.

2. Data structure restructuring. The array layout function changes the mapping from the array index space to the virtual address space.

Control flow restructuring is the core of such restructuring. This is, according to the authors in [32], due to the fact that programmers and compilers are more familiar and comfortable with transforming the control flow of programs; they are less likely to restructure multidimensional arrays to be "cache conscious" or "memory friendly".

Restructuring techniques have been studied for pointer-based data structures, such as heaps and trees, for profile-driven object placement and for matrices with special structures (e.g. sparse matrices formats), and in parallel computing.

The observation leading to the approach in [32] is that non-standard data layouts used by library writers and in parallel computing are applicable in more general situations, but not exploitable given the existing programming languages. The authors demonstrate, by measurements of the execution time, that the costs associated with the non-linear layout functions are minimal, and that they improve performance for representative multi-level memory hierarchy architectures.

Data restructuring is a powerful technique and it works well for programs that use non-standard data structures. Thus, it applies to situations where the static program analysis is not sufficient to find information about data accesses. Therefore, the computation can be fixed in the SPMD style, and data restructuring can be applied to ensure locality of reference.

3.3.3 Load Balance

Load balancing is dividing the amount of work that a computer has to do between two or more computers, so that more work gets done in the same amount of time, and, in general, the same application is executed faster. Load balancing can be implemented with hardware, software, or a combination of both. In the context of scientific computing, the same amount of data is roughly assigned to each processing unit.

We give a formal definition of load balance from [107], using the same notation as before.

Definition 3.3 *A P-fold distribution μ of an index set $\mathcal{M}$ is load balanced if and only if:*

$$max_{0 \leq p < P} I_p = max_{0 \leq p < P} |\mathcal{M}_p|$$

is minimal over the set of all P-fold distributions of $\mathcal{M}$.

Computations with $P \ll M$ are said to be *coarse-grain*, and computations with $P \approx M$ are said to be *fine-grain*. A good load balance may sometimes lead to poor data locality. Therefore, a compromise between the two criteria must be reached.

General Graph Partitioning Algorithms

General graph partitioning algorithms offer a general framework for decomposing data and computations among processes. There is a large body of research concerning graph partitioning for high-performance scientific applications. However, given the formulation of the general partitioning as a function defined at each input domain variable that maps one data item to one processor and location, any of the algorithms can be used. Moreover, any regular partitioning can easily be mapped to the general partitioning scheme.

We give the formulation of a general graph partitioning problem for the high performance scientific computations from [95]:

> Given a weighted, undirected graph $G = (V, E)$ that for each vertex and edge has an associated weight, the k-way graph partitioning problem is to split the vertices of V into k disjoint subsets (or subdomains) such that each subdomain has roughly an equal amount of vertex weight (referred to as the *balance constraint*), while minimizing the sum of the weights of the edges whose incident vertices belong to different subdomains (i.e. the *edge-cut*).

The general formulation of graph partitioning for multi-constraint, multi-objective graph partitioning is as follows:

> Assign a weight vector of size m ($w[m]$) to each vertex and a weight vector of size l ($e[l]$) to each edge. Find a partitioning that minimizes the edge-cut with respect to all l weights, subject to the constraints that each of the m weights is balanced across the subdomains.

3.4 Summary

This chapter has presented two major themes in two parts: partitioning strategies and criteria to choose a good partitioning. The first part of the chapter presented two main classes of data partitioning strategies, i.e. regular partitioning and general partitioning. The precise formulations were given for the general partitioning problem, along with two practical partitioning schemes: block and cyclic partitioning. Furthermore, concrete applications of these strategies were described in terms of automatic data layouts for regular applications. A general partitioning strategy was discussed as an alternative general framework to address data parallelism. It was shown that general partitioning is not a random process, but it can be effectively implemented by using general graph partitioning algorithms. One disadvantage of the general partitioning scheme is the space overhead for retaining partitioning function values instead of symbolic expressions (as with regular partitioning schemes). The overhead can be reduced by distributing the partitioning information over multiple processes.

The second part of the chapter presented three criteria for choosing a good partitioning: execution models based on heuristics, data locality and load balance. Execution models rely on the static program analysis to be able to estimate the execution time for each program statement and function call. Good approximations are hard to find. Moreover, in a programming language which supports dynamic binding of variables and functions, the static estimation may be a non-realistic approximation. Data locality relies on the programmer or compiler to restructure data or control in a program, such that algorithmic patterns of locality of reference match the patterns of memory organization or local address space. Control flow restructuring dominates the restructuring work. Data restructuring is a promising strategy in the context of complex data representations and irregular structures. Load balance ensures that the same amount of work is assigned to each computation unit. A good load balance may lead to poor data locality. Therefore, these two criteria are usually considered together. A general graph partitioning algorithm for multi-constraint, multi-objective graph partitioning enables such coupling of more criteria and objectives when deciding upon a good partitioning.

Chapter 4

Concurrent Programming

4.1 Introduction

This chapter presents the main concepts for concurrency together with existing high-level programming support for writing parallel applications. However, this is not an extensive survey of the existing concurrent programming languages and systems. The chapter presents the most important mechanisms for communication and synchronization in shared-memory and distributed-memory systems. Existing high-level programming languages and libraries are presented in the context of large-scale parallel programming. The chapter also discusses the distributed shared memory, a software implementation of a shared virtual memory on physically distributed memory.

Several aspects must be considered when deciding upon a programming model. Shared-memory multiprocessors are expensive and rely on vendor-developed software. The shared-memory concurrent programming model is close to sequential programming and thus intuitive for a programmer. Distributed-memory architectures are scalable and do not depend on proprietary software. The distributed-memory programming model is less intuitive and hardened by the nondeterminism of a program execution (race conditions). Establishing global properties for a distributed system is difficult in the absence of a shared memory.

A distributed shared memory that implements a software shared virtual memory is a cheap alternative to a shared-memory multiprocessor. Moreover, it resolves the inconveniences of a distributed-programming model. The main limitation of such a shared virtual memory is poor performance due to the large granularity of coherence (page size) and the expensive remote accesses.

This chapter is organized as follows: Section 4.2 presents the main concepts for concurrent shared-memory programming. It discusses the main communication and synchronization paradigms as well as high-level language support for programming scientific applications. Section 4.3 introduces the communication and synchronization mechanisms in a distributed-memory environment. Furthermore, it presents existing high-level language and library support for writing distributed-memory concurrent scientific applications. Section 4.4 presents the virtual shared memory concept as described by [74] and discusses several implementation issues. Furthermore, it discusses the research advances in shared virtual memory. Section 4.5 summarizes the chapter.

4.2 Shared Memory

A shared-memory multiprocessor has one global address space that is accessible for all the processors. A program consists of multiple processes that may run on different processors and perform independent computations. Processes communicate through reading and writing shared-memory locations. Concurrent accesses to shared data are serialized through synchronization.

The processes carrying out concurrent actions in shared-memory applications are usually called threads. Threads are light-weight processes sharing the same address space (context), usually that of the parent process / thread which has created them.

The main problem in a concurrent computing environment is to ensure correct execution through synchronization. Synchronizing accesses to a shared-memory location ensures that only one process at the time has access to that location.

4.2.1 Low-level Synchronization Mechanisms

There are two main mechanisms to ensure synchronization to a shared memory [11]:

- Mutual exclusion ensures that a sequence of statements is treated as an indivisible operation. Such a sequence of statements is called a *critical section*.

- Condition synchronization ensures that no action is performed when a shared data object is in a state inappropriate for executing a particular operation.

Several abstractions are used to implement the synchronization of accesses to shared memory. The early mechanisms are low-level and expose the programmer to a complex, error-prone programming style. Higher-level programming language constructs hide the low-level synchronization mechanisms and offer abstract primitives to control synchronization explicitly. Several languages make synchronization implicit and alleviate the programmer from the complex, error-prone concurrency aspects.

Some of the most important synchronization mechanisms are:

1. *Busy-waiting* uses a shared variable that concurrent processes can *test* and *set* to synchronize their accesses to shared data. Busy-waiting works well for implementing condition synchronization. The two instructions *test* and *set* make mutual exclusion difficult to design and implement. Some hardware systems implement a *test-and-set* instruction on processors to address the inconvenience of using two separate instructions to access the shared variable. However, busy-waiting wastes processor cycles by not allowing other processes run instead.

2. *Semaphores* use a non-negative integer value on which two operations are defined: P and V. Given a semaphore s, $P(s)$ delays the execution of a process until $s > 0$ and then executes $s = s - 1$. $V(s)$ is a non-blocking operation that increments the value of the semaphore variable:$s = s + 1$. It is straightforward to implement a critical region as a sequence of statements between the P and V operations. To implement condition synchronization, a shared variable represents the condition, and a semaphore is associated with it. A P operation becomes a *wait* on the condition, while a V operation becomes a *signal*. In these

circumstances, it is hard to distinguish between a critical section and a condition synchronization without a thorough examination of the code. Moreover, *P* and *V* are unstructured operations, and thus it is easy to commit errors when using them.

3. *Conditional critical regions* (CCR) offer a structured, unified notation for synchronization. Shared variables are explicitly placed in groups called resources. Each shared variable belongs to one resource only and is accessible in the CCR statements that name the resource:

```
resource r:  v1, v2, ..., vn;
    region r when B do S,
```

   where `B` is a boolean expression and `S` is a statement list. CCR can be expensive to implement given that conditions in the CCR statements can contain references to local variables, and thus each processor must evaluate its own context, leading to expensive context switches.

4. *Monitors* encapsulate a resource definition and the operations that manipulate it. Only one process at the time can entry the monitor and execute the operations on the resource mutually exclusive. There are several proposals for realizing condition synchronization. One proposal is to use a *condition variable* to delay processes executed in a monitor. A condition variable may be declared only within a monitor and has two operations, `wait` and `signal`. The processor invoking a `wait` on a condition variable will block until another processor invokes a `signal` on that condition variable, causing the latter to be suspended. A condition variable usually has queues associated with both `wait` and `signal`.

These mechanisms are low-level and expose the programmer to a complex, error-prone programming style. Ideally, a programming language or a system implicitly ensure synchronization. However, this has proved difficult to achieve. Some synchronization aspects may still be visible, in a higher-level form, at the programming language level.

4.2.2 High-level Programming

Fine-grain Parallelism

Supercomputers can achieve high performance concurrent execution over a sequential execution by exploiting parallelism at the instruction level, as well as at the multiprocessor level. Instruction-level parallelism exploits the vector and array processor architectures by allowing a single low-level machine instruction such as load, store, integer adds, and floating point multiplies to be executed in parallel. This is a fine-grain form of parallelism. At the multiprocessor level, all the active processors execute the same operations on different data items. This form of concurrency is called Single Instruction Multiple Data (SIMD) and it usually expresses trivial parallelism. The applications exposing this form of parallelism are dense matrix computations. This is also a form of fine-grained parallelism.

However, applications also exhibit non-trivial parallelism that requires special language support. Languages that exploit architectural support for parallelism are called array languages and have Fortran as a starting point. There are two approaches to support parallelism in array languages:

1. Overload arithmetic and boolean operations to perform element-wise operations [87]. For example, a Fortran-90 statement `A = B + C` performs an element-wise sum of the arrays `B` and `C` and stores the result into the array `A`.

2. Offer support for conditional vector operators that allow operations to selectively apply to only parts of arrays. There are two categories of language support:

   (a) Data oriented approaches introduce constructs to express control vectors, constant stride vectors, array triplets (e.g. `A(start:end:stride)` where `start, end` and `stride` are the integer array indices and the constant stride value) and guarded accesses to arrays.

   (b) Control oriented approaches introduce concurrency constructs, such as `forall`.

Array languages effectively exploit data parallelism for applications that perform dense matrix computations. Data parallelism may take other forms than expressing numerical vector matrix operations from linear algebra computations. It is not always obvious how parallelism can be expressed by using adequate data or control structures at the language level.

Coarse-grain, Loosely-coupled Parallelism

Trivial parallelism is to apply a function concurrently to each element of a data structure. A more subtle form of parallelism is to use operations (e.g. addition and multiplication) to combine all the elements of a data structure together. Coarse-grain parallelism exploits the loose synchronization requirements in these applications to allow large computation steps to proceed independently. All processors participate in a synchronization step to combine their local values. This form of parallelism is called Bulk-Synchronous Parallelism (BSP) [106], or Single Program Multiple Data (SPMD).

HPF (High Performance Fortran) provides a machine independent parallel programming model, and thus it can be used to express shared-memory SPMD parallelism. HPF provides the `forall` statement in conjunction with the `INDEPENDENT` directive to specify the iterations of a loop which can be executed in parallel. HPF also provides data distribution directives that allow the programmer to directly control the assignment of work to processors in block, cyclic, or block-cyclic distribution.

HPF is suitable to express data parallelism for applications that use multi-dimensional array structures. The loop-level parallelism is still fine-grain. The main addition in HPF is to allow the programmer to specify how data is distributed among processors and thus increase the grain of a computation.

Task Parallelism

Task parallelism allows two or more processors to synchronize whenever they need to without the involvement of other processors. This style of programming is called Multiple Instruction Multiple Data (MIMD).

OpenMP [37] is a set of compiler directives and run-time library routines that can be used in conjunction with a sequential programming language (e.g. Fortran, C) to express shared-memory task parallelism.

A parallel block is explicitly indicated by the use of the `PARALLEL` directive. The block is executed in a fork / join multi-threading style. A number of threads are spawned from the main control thread and cooperate in executing the body of the block concurrently. The threads join back into the main thread of control at the end of the parallel block.

The programmer can also specify how iterations can be assigned to threads. The variables in a parallel region are shared between all the threads. Declaring a variable to be `PRIVATE` gives each thread its own copy of it.

> An example of a parallel region. Variables `b` and `n` are shared, while `i` is private to each thread. A sum `reduction` operation is applied to the scalar variable `a`:

```
#pragma omp parallel private(i) shared (b, n) reduction (+:a)
{
for (i = 0; i < n; i++)
a = a + b[i];
}
```

Two or more threads can also synchronize at the entry in a critical sections. Only one thread at the time is allowed to execute the code in a critical section.

OpenMP is a standard aimed at achieving portability for shared-memory programming in face of vendor-developed extensions to Fortran and C for parallel software development. However, its fork / join model of parallelism still resembles fine-grain loop-level parallelism. OpenMP introduces the concept of *orphan* directives to specify control of synchronization from anywhere inside of a parallel region, thus making it easier to express coarse-grain parallelism.

Functional Programming Languages

Functional programming languages (Multilisp, Actors, ABCL) also express shared-memory MIMD concurrency. The *consumer-producer* synchronization is folded into the data accesses. Concurrency is exploited in two forms:

1. Task parallelism allows multiple expressions to be evaluated concurrently by concurrent tasks through procedure call / method invocation.

2. Overlapping communication and synchronization through `futures` which act like place-holders for variable values that can be returned later as a result of a communication. This is a form of fine-grain data flow synchronization at the level of data structure elements.

Functional programming is an elegant mechanism to exploit concurrency. There are some practical problems that limit its large scale use [87]. The main data structure for such languages (Multilisp) is a linked list. Unlike arrays, lists are sequentially accessed and this limits concurrency. The problem of deciding where it is safe and profitable to insert futures in a general program is non-trivial, since Multilisp is an imperative language where expression evaluation can have side-effects. “The suggested programming style is to write mostly functional code and look for opportunities to evaluate data structure elements as well as function arguments in parallel” [87].

4.3 Distributed Memory

In a distributed-memory multiprocessor each processor has its own address space of memory locations inaccessible to other processors. A concurrent program consists of multiple processes that execute in different address spaces. Processes communicate through messages. Synchronization is implicit in the message exchange routines.

In a distributed-memory environment concurrent actions are carried out by processes. A process encapsulates its own address space which is inaccessible to other processes. The `send` and `receive` interprocess communication primitives replace the `read` and `write` operations to shared variables. Synchronization is implicit in the message exchange, in that a message is received only after it has been sent.

According to their distributed computing functionality, processes can be classified in the following categories [10]:

1. *Filter* processes act as data transformers: they receive data at their input channels, perform some transformation on the data and deliver the transformed data at the output channels. A Unix command is an example of a filter process.

2. *Client* processes act as triggering processes: they invoke a request which is served by another process.

3. *Server* processes act as reactive processes: they perform some actions as a response to a request from another process.

4. *Peer* processes are a collection of identical processes.

There are two main design questions [11]:

1. How are source and destination designators specified?

2. How is communication synchronized?

There are three main naming schemes to designate processes in a distributed-memory environment:

1. *Direct naming.* With this scheme, the communicating parties name the communication channel directly. This form of identification is suitable in one-to-one communications.

2. *Mailboxes* are global names that can be used to designate the source and the destination of a communication. This naming scheme is suitable to express many-to-many type of communication.

3. *Ports* are a special case of mailboxes, in which a mailbox name can appear as the source designator in receive statements in one process only. This naming scheme is suitable in one-to-many communications.

4.3.1 Low-level Synchronization Mechanisms

Synchronization is achieved through communication primitives that can be:

1. Nonblocking invocation: its execution never delays its invoker.

2. Blocking invocation: the execution of the invoker is delayed until the invocation returns.

3. Buffered communication: messages are buffered between the time they are sent and received.

The main interprocess communication mechanisms are [10]:

- *Asynchronous message passing.* A process continues its execution immediately after issuing a communication request. The messages waiting to be sent / delivered are placed in system buffers with unbounded capacity.

- *Synchronous message passing.* A process blocks as a result of a communication request and waits for the response. There is no buffering in such approaches.

- *Buffered message passing* is in between the two previous communication forms. The system buffer has finite bounds and therefore forces processes to synchronize when the buffer is full.

- *Generative communication.* The processes share a single communication channel, called a tuple space. Different names, instead of different channels, are used to distinguish between different messages (e.g. Linda tuple space).

- *Remote procedure call.* A calling process issues an invocation and waits until it has been serviced and results have been returned. A new process is created to service the request.

- *Rendez-vous.* A calling process issues an invocation that matches an `accept` statement in another process. The caller blocks upon the invocation until results have been returned. The callee blocks in an accept statement waiting for a matching request.

All these approaches are equivalent: a program written in one notation can be rewritten in any of the other notations. However, each approach is better suited to solve some problems than others.

4.3.2 High-level Programming

Fine-grain Parallelism

Instruction level parallelism or SIMD expresses fine-grain, almost trivial concurrency. The shared-memory multiprocessors naturally express this form of parallelism. The distributed-memory multiprocessors are not suitable for such computations, since a communication between processes is much more expensive and the computation grain is too fine to amortize the cost of communication.

The existing distributed-memory SIMD languages use an assembly language called Paris (Parallel Instruction Set) that augments local sequential execution with the possibility to *send* a message. There is no matching *receive* since the processors operate in a lock-step. Thus, a send has to specify the address of the receive. This results in a very low-level programming style.

Coarse-grain, Loosely-coupled Parallelism

The distributed-memory model is best suited to exploit coarse-grain, loosely coupled parallelism. Processors operate in a loose lock-step, consisting of alternating phases of parallel asynchronous computation and synchronous communication. This model of parallel computations corresponds to the BSP or SPMD models. The existing data parallel frameworks do not capture this model of computation entirely.

HPF-2 offers support for irregular computations and task parallelism. Several vendor-supported HPF compilers target distributed-memory computers. However, the quality of the compiler-generated code is relatively poor in comparison to hand-written parallel code. Moreover, source level performance prediction has proven to be difficult since performance depends greatly on decisions about inter-processor communication made by the compiler. “For these reasons, interest in HPF is on the wane” [87].

There are libraries that implement the BSP model. The libraries have functions for sending and receiving a message as well as for barrier synchronization. These libraries do not offer as rich a functionality as the Message Passing Interface (MPI) library does. This is the reason why MPI dominates the practice in writing SPMD applications.

Even though MPI offers functionality for the MIMD model of parallelism, it is mostly used to manually parallelize SPMD bulk synchronous applications. The main disadvantage of the BSP model is that it requires global synchronization even for exchanging one message between two parties. The counter argument is that “worrying about optimizing individual messages makes parallel programming too difficult” [87]. Thus the focus should be on getting the large-scale structure in place.

This argument holds especially in the context of numerical SPMD applications that need to synchronize globally to exchange values at various points during the application lifetime. It is thus useful to concentrate the efforts towards capturing the large-scale concurrency structure.

Task Parallelism

With the MIMD synchronization model, two or more processors can synchronize whenever they need to without the involvement of other processors. The earliest distributed-memory MIMD language is CSP (Communicating Sequential Processes) [11]. CSP supports synchronous message passing and selective communication. However, CSP is not a complete concurrent language. The current practice for writing concurrent message passing applications is to use message passing libraries (MPI, PVM) in conjunction with a sequential language such as C or Fortran.

The goal of MPI is to develop a standard for writing message-passing programs in a portable, efficient and flexible style [99]. MPI is also used as run-time environment for parallel compilers and for various libraries. MPI is extensively used to develop efficient concurrent applications in conjunction with a sequential language.

An MPI program consists of a collection of autonomous processes executing their own code in a MIMD style. These programs need not be identical. The processes communicate via MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI exist. The existing implementations of MPI provide mechanisms to specify the initial number of processors and the binding of computations to physical processors. The processes are identified according to their relative rank in a group as consecutive integers in the range, i.e. `0..groupsize - 1`.

MPI offers provision for point-to-point communications, collective operations, process groups, communication domains, process topologies, and environmental management and enquiry. Through this extensive coverage of message passing communication, MPI is widely used for distributed-memory concurrent applications.

4.4 Distributed Shared Memory

A distributed shared memory is based on a shared virtual memory [74] implemented in a distributed-memory system. A shared virtual memory has a single address space shared by a number of processors. Any one processor can access any one memory location in the address space directly.

The main benefits of a shared virtual memory (SVM) are [74]:

1. The SVM pages data between physical memories and disks, just as a virtual memory does.

2. The SVM pages data between the physical memories of individual processors and thus data can migrate between processors on demand.

3. Data migrates between processors just as a conventional virtual memory swaps processes. Therefore, this provides an efficient form of process migration in distributed systems.

4. The process migration subsumes the remote procedure call (RPC) communication.

The main problem with implementing a SVM is the memory coherence. That is, ensuring that the value returned by a read operation is always the same as the value written by the most recent write operation to the same address.

The memory unit for transferring data between processors is a page. Pages can be read-only or write. The read-only pages can be replicated in each processor. The write access pages belong to individual processors and cannot be replicated. A memory reference causes a page fault when the page containing the memory location is not in the processor's current physical memory.

The central part of a SVM is memory management and mainly consists of two tasks:

1. Maintain a map between the local memories and the global address space.
2. Ensure memory coherence:
   (a) Manage the information in the map between local and global addresses using a centralized or distributed manager protocol.
   (b) Maintain memory consistency using an invalidate or update protocol upon a write operation to a local memory, such that all the processors "observe" the update consistently.

Several performance factors affect the choice of a memory coherence scheme. The factors that have direct impact on the cost of a coherence protocol are the number of parallel processes and the degree of updates for the shared data. The granularity of a page, or the size of the page depends on the application.

A coherence strategy must decide upon the page synchronization scheme (invalidate or update) and page ownership strategy. Page ownership can be fixed or dynamically established. In a dynamic scheme, the ownership of a page changes during a program execution such that the last processor that has requested a page becomes its owner.

A memory manager algorithm that is centralized maintains all the information about the mapping between the local and global address spaces in one central process that interacts with all the individual processes cooperating in the coherence protocol. The centralized manager serializes the concurrent memory requests issued by different processes. Therefore, the manager becomes a communication bottleneck.

A memory manager algorithm that is distributed moves the synchronization task from the central manager to the individual processes at each address space level. This scheme eliminates the communication bottleneck from the centralized scheme. The key to a distributed coherence protocol is to keep track of the ownership of all pages in each processor's local address space without replicating the entire memory map. Thus, each processor keeps track of the true owner, as well as of the "probable" owner, for each page. If the true ownership information is not available, the map entry for each page has a sequence of processors through which the true owner of a page can be found. If a request for a page arrives to a processor that does not own the page, the request is forwarded to the true owner in case this is known, or otherwise to the "probable" owner.

A distributed memory manager algorithm is more efficient than a centralized one. One efficiency issue of the distributed scheme concerns the amount of information at each address space level to guarantee that the true owner of a page is always found. Another efficiency issue is the number of forwarding requests needed to find the owner of a page. The benefits of a virtual shared memory are unquestionable. However, its main limitation is performance. The main performance limiting factors are [61]:

- The large granularity of coherence generates *artificial communication* since entire pages are transferred between address spaces. *False sharing* may also occur when individual memory accesses are located in the same page. *Fragmentation* arises when a requesting processor does not need all the data in a page.
- The communication between processors is through software messages. Therefore, the cost of a communication operation is high.
- Synchronization is also implemented through explicit messages. Therefore, the cost of synchronization is also high.

There exist several approaches that alleviate the high costs associated with the communication and synchronization. Their common goal is to relax the consistency constraint and thus reduce the communication and synchronization requirements.

A strict consistency protocol guarantees *sequential consistency*. That is, all shared accesses must be given the same order on all processors. Instead, the order does not matter as long as the changes are observed in the same way by all the processors at the synchronization points. As a consequence, the following relaxed consistency schemes were proposed:

- *Processor consistency* allows writes from different processors to be observed in different orders, although writes from a single processor must be performed in the order that they occurred.
- *Weak consistency* separates data accesses in regular and synchronization accesses. Therefore, coherence is postponed until a synchronization point. All regular data accesses must complete before a synchronization access is allowed to proceed.
- *Release consistency* further separates synchronization accesses into *acquire* and *release* accesses. Thus, coherence due to write operations is postponed until before the releases that follow the writes. Thus, all previous shared data updates are consistent before a release of a synchronization variable is observed by any processor.
- *Entry consistency* [21] binds data to synchronization variables. Therefore, only the data bound to the synchronization variables are subject to consistency. The main problem with this approach is that the user explicitly identifies the synchronization variables, the shared data that needs consistency, and the binding between the shared data and the synchronization variables.
- *Scope consistency* [60] tries to make the binding between shared data and synchronization variables implicit. The synchronization variables define scopes through

which the memory is viewed, and an implicit association of data (pages) with scopes is achieved dynamically when a write access occurs inside the scope.

In practice, there exist several implementations of DSM. TreadMarks [8, 75] is a user-level library implemented on top of the UNIX operating system that enables a shared memory programming model in distributed memory (e.g. networks of workstations). The library provides subroutines for process creation and destruction, synchronization (barrier and lock / unlock for critical section) and shared memory allocation.

The memory structure in Treadmarks is a linear array of bytes. The memory model is lazy release consistency. That is, the memory model enforces consistency at the time of an `acquire`. TreadMarks uses an invalidation scheme: at the time of an acquire, the modified pages are invalidated. This scheme is inefficient, since it introduces extra communication for invalidating and then updating data.

An important feature that potentially improves efficiency in TreadMarks versus other DSM implementations that use a single-writer protocol, is the use of a multiple-writers protocol. That is, multiple processes have at the same time a writable copy of a page. Processes allow concurrent modifications of a replicated page and synchronize at a barrier to ensure that differences in the page versions are propagated between them. Although simple, this scheme is dangerous since it assumes that there is no overlap between the modifications performed concurrently by the two processes. Such an overlap introduces race conditions.

Ancourt et. al. [9] also propose to build an emulated shared memory on distributed-memory machines and abandon the owner-computes rule. In order to emulate the shared memory, the processors are divided into two subsets. One half of the processors perform computations, while the other half emulate memory banks. The main advantage of this approach is that the data distribution is implicit. That is, objects are mapped in the emulated shared memory using a fixed scheme. They can be used only after they are copied in the memory of one of the compute engines. When new objects are computed, they are not available before they are copied back into the emulated shared memory. The dependence graph between program instructions is used to build parallel loops. The price to pay for the ease of use is increased communication due to the fixed mapping scheme and the load and store operations from / to the emulated shared memory to the computation engines. Serialization also occurs when two different tasks work on related block tiles.

Despite the progress in shared virtual memory, the performance of many applications is lower than that of the hardware coherence systems, especially for irregular applications involving a lot of synchronization [61]. The main guideline for future research is to integrate the application and architectural support with the communication architecture and understand and exploit the interaction among the application, protocol and communication layers rather then keeping some of them fixed. The distributed shared memory is a cheaper, still lower-performance substitute for hardware-coherent machines. The applications that best can exploit this software solution are those resulting in low-communication requirements, or coarse-grain applications.

4.5 Summary

This chapter has presented the main concepts for concurrent programming. The shared-memory programming model is more popular with programmers due to its resemblance to the sequential programming model. However, shared-memory systems are expensive and do not scale well.

The support for shared-memory programming ranges from very fine-grain instruction level parallelism to loop-level parallelism, and builds on the Fortran language. The ability to express coarse grain parallelism is limited to the loop-level, still fine-grain concurrency model. The main facility to increase the grain of a computation in the HPF model is to allow the user to specify array distributions across multiple processors, in a block, cyclic or block-cyclic fashion. This programming model is suitable to express dense array computations.

Distributed-memory programming support is also limited to the Fortran principal parallelism model (i.e. HPF style). Coarse-grain parallelism is mainly achieved through library support, in conjunction with sequential languages in manually parallelized applications. MPI library is the widely used message passing standard for writing distributed-memory applications.

The distributed shared memory is an elegant approach to combining the benefits of the shared-memory and distributed-memory systems. However, limited performance prevents a wide adoption. Many existing high-level languages or programming environments implement ideas from the shared virtual memory, adding some application specific knowledge to improve efficiency. Even though there is no strong evidence of the success of such approaches, this is a promising research path.

Chapter 5

Programmability

5.1 Introduction

This chapter discusses the usability of different concurrency models. A usability framework is presented. The framework takes into consideration the aspects of concurrency and the extent to which all or some of these can be accounted for by a given model.

There are several factors which complicate the development of concurrent programming. Ensuring the correctness of a concurrent program is difficult. The testing and debugging are complicated by the large number of possible states due to decomposition. Therefore, it is difficult to construct checkpoints, where the state of the concurrent program is saved in order to allow a debugging tool to unwind the erroneous execution states and detect the error. Dividing the concurrent computing responsibilities between the programmer and the system is one way of tackling the difficulty of writing concurrent applications. This chapter uses the term *usability* to denote the degree to which a programmer is exposed to concurrency issues. The aspects of concurrent computing are:

1. *Decomposition* specifies how the data and the control in a program are decomposed into tasks that can execute in parallel. Decisions regarding which computations are to run in parallel are of a great importance.

2. *Mapping* specifies how the parallel tasks are mapped onto physical processors; that is, which parallel unit is executed on which processor at a given point in time. A mapping strategy strongly depends on the parallelism objective: the maximum speedup, or high availability through replication, for instance.

3. *Communication* specifies how processes communicate: processes need to cooperate while executing a concurrent application. Processes can communicate either through shared memory, or through message passing.

4. *Synchronization* specifies how processes interact with each other to exchange results or reach a common state. Synchronization is explicit in shared memory and implicit in distributed memory (through message passing).

A concurrent computing model may reveal all these concurrency aspects, some of them, or none. The manual parallelization and fully implicit concurrency are the two ends of the spectrum. The former allows for great flexibility in exploiting concurrency in an application. The latter is ideally usable, but hardly achievable since it cannot account for any of the application specific requirements. In between the two ends there are several concurrency models designed with a specific class of applications in mind, and thus trying to meet special objectives. This chapter discusses the existing concurrency models and the extent to which they reveal the concurrency aspects from the perspective of data parallelism.

The remainder of the chapter is organized as follows. Section 5.2 discusses manual parallelization and illustrates the issues that a programmer must resolve in the context of the MPI (Message Passing Interface) concurrent programming model. Section 5.3 presents four approaches to increased usability by hiding one or more of the concurrency aspects gradually from synchronization to distribution. Section 5.4 discusses automatic parallelization. Section 5.5 summarizes the chapter.

5.2 Manual Parallelization

This section discusses the Message Passing Interface [51,99] (MPI) programming model. MPI allows for MIMD (multiple instructions multiple data) parallelism. However, MPI is extensively used to program data parallel applications on distributed computer systems (multicomputers, clusters of PCs, or networks of workstations), thus for programming SPMD applications. MPI programs can also run on shared-memory machines.

Decomposition. The unit of parallelism in MPI is a *process*. A process can be sequential or multi-threaded. Decisions regarding which computations are to run in parallel belong to the programmer. MPI provides the notion of process groups. The processes in a group are uniquely identified through consecutive integer identifiers in the range of the group. The number of processes is fixed at run-time and it does not change during the application execution.

Mapping. Specifies how MPI processes are to be mapped to physical processors. Mapping strongly depends on the parallelism objective. The mapping of processes to physical processors depends on the details of the underlying hardware, i.e. how the processors are connected to one another by the interconnection network. MPI provides *topology* functions to specify the relationship between the physical processors and the software processes. The most important topology is the *Cartesian* topology which specifies a decomposition along the coordinate axes, e.g. in two dimensions (x, y).

With library support for running MPI applications, mapping is usually not under the programmer's control. Data parallel applications consist of a number of peer (equal) processes that is specified at run-time. A command line argument specifies the number of processes for the application. Existing implementations transparently create the processes and map them onto physical processors. It is also allowed to specify a larger number of processes than physically available processors. In such a situation multiple processes are interleaved on a single processor. Profiling versions of MPI allow the user to monitor the execution of the various processes.

Communication. MPI communication routines define basic point-to-point communication, as well as collective communication. Point-to-point communication allows data to be transfered between two processes. MPI provides several `send` and `receive` functions for sending and receiving typed data. Both blocking and nonblocking communications are available. A `send` call specifies the receiver of the data, the type of data in the message and the message buffer. A `receive` call may not specify the sender and thus may receive from an arbitrary source. Care must be taken to distinguish between multiple receiving sequences.

The user assembles the data in the messages manually on the sender side, and writes the sequence for disassembling a message on the receiver side. Care must be taken to ensure that the types and sizes specified in send sequences match those of the receiving sequences.

The MPI specification provides several functions for collective communication: barrier synchronization, global communication functions (broadcast, scatter, gather), and global reduction operations. These communication facilities provide for one-to-many, many-to-one, and many-to-many communication paradigms.

Synchronization. Synchronization is implicit in message passing. A blocking `send` blocks the caller process until the message has been stored away from the system buffer. A blocking `receive` blocks the receiver until it receives a message that matches the call.

The blocking point-to-point communication functions are:

```
MPI_SEND(sendbuf, count, datatype, dest, tag, comm);
MPI_RECV(recvbuf, count, datatype, source, tag, comm, status);
```

Nonblocking communication allows communication and computation to overlap and thus leads to improved performance. The `send` and `receive` calls are split into `post` and `complete` sequences. A `post` sequence initiates a transfer, while a `complete` sequence completes the communication. Computation is overlapped with communication before the completion call.

The posting operations are:

```
MPI_ISEND(sendbuf, count, datatype, dest, tag, comm, request);
MPI_IRECV(recvbuf, count, datatype, source, tag, comm, request);
```

The completion operations are:

```
MPI_WAIT(request, status);
MPI_TEST(request, flag, status);
```

Complications. There are several issues related to programming MPI data parallel applications. First of all, explicit communication exposes the user to details such as data packing / unpacking, correctly identifying the communication parties, error recovery, etc. Explicit care must be taken with the ordering of the synchronous calls to avoid deadlock situations or non-deterministic program executions.

The main sources of error in point-to-point synchronous communications are:

- Non-ordered send and receive sequences. Let P_1 and P_2 be two processes that attempt to exchange two messages. The following situation would result in a deadlock because both of the programs are blocked in the receiving operations and cannot issue the send:

  Process P_1:
  ```
  MPI_Recv(msg2, ..., P2);
  MPI_Send(msg1,..., P2);
  ```

  Process P_2:
  ```
  MPI_Recv(msg1, ..., P1);
  MPI_Send(msg2,..., P1);
  ```

- Buffer limited communication. A communication that relies on the system buffer to store the messages will deadlock if there is not enough space available:

  Process P_1:
  ```
  MPI_Send(msg1,..., P2);
  MPI_Recv(msg2, ..., P2);
  ```

  Process P_2:
  ```
  MPI_Send(msg2,..., P1);
  MPI_Recv(msg1, ..., P1);
  ```

Asynchronous communication eliminates to some extent these concerns, since there is no need for buffering. However, a pending communication uses storage for communication parameters. Storage exhaustion causes a new communication request to fail.

Collective communication also requires care with the ordering of calling sequences and matching to avoid deadlock situations. Thus, the collective operations must be ordered such that no cyclic dependences occur. Different matching of sends and receives may lead to non-deterministic execution:

Process P_1:
```
MPI_Send(msg1,..., P2);
```

Process P_2:
```
MPI_Recv(msg, ..., ANY);
MPI_Recv(msg, ..., ANY);
```

Process P_3:
```
MPI_Send(msg3,..., P2);
```

All these complications have led researchers to seek means to automate the process of writing parallel applications. Even though it is hard to achieve, there is enough similarity in the structure of data parallel programs to enable capturing of the concurrency infrastructure. The complicating aspects related to an automatic solution for the parallelization of these applications are the possibly recursive data representations used to express irregular data layouts, and the limitations of the dynamic program analysis techniques.

5.3 Different Degrees of Implicit Concurrency

5.3.1 Implicit Synchronization

An elegant method to make synchronization implicit is to fold it into data accesses. This technique originates in functional languages parallelism [87]. Pure functional languages do not record state, and thus they do not allow efficient implementations. Object-based models use the same principle of folding synchronization into data accesses. Actor languages are one example. The problem with the Actor languages is that there are no guarantees for performance or cost measures given that the communication in a program may not be bounded. Also, processes are created and deleted in response to messages and thus lead to frequent, expensive context switches. This section discusses Orca [17], an object-based parallel programming language that makes synchronization implicit.

Decomposition. In Orca, parallelism is explicit. The unit of parallelism is a *process*. Processes execute sequentially on different processors. An Orca program consists of a unique process, but new processes can be created explicitly through a `fork` statement. The newly created processes are child processes and can reside on the same processor as the parent process, or they may be explicitly assigned to a processor.

A process is explicitly declared:

```
process name(arg1:type1; arg2:type2; ...; shared obj_name:obj_type);
begin
....
end;
```

A new process can be created using the `fork` statement:

```
fork name(act1, act2, ..., a_shared_object);
```

Concurrency is achieved through the creation of multiple processes and their concurrent execution. Two processes concurrently accessing a shared object will be synchronized automatically by the system.

Mapping. Parallel processes are mapped onto processors either implicitly, by the system, or explicitly, by the user. That is, upon the creation of a new process, the user can specify a logic processor for the newly created process to run on:

```
fork name(actual_parameters)[on(cpu_number)];
```

If the user does not specifically assign the newly created process to a logic processor, the system maps the new process on the same processor as the parent process that created it.

The mapping of processes to processors renounces any objective, meaning that it does not take into account locality of reference or other optimizing criteria. Thus, mapping is “random”. This may lead to many efficiency problems. Therefore, replication strategies for the shared objects are used to account for the availability of data.

Communication. In Orca, processes communicate through shared objects. Two distinct processes may take the same shared object as actual argument and thus use the shared object as a communication channel. Shared objects can also serve as communication channels between a parent process and its child. A shared-data object is a variable of an abstract data type. An abstract data type in Orca encapsulates data and specifies the applicable operations to an object of a given type. In Orca both shared (concurrent) and sequential objects are declared as instances of an abstract data type. The system distinguishes between them according to their usage.

An object specification is Orca has the following structure:

```
object specification an_object;
...; #internal data
operation op1(in_args):out;
operation op2(in_args);
...;
begin
...;#initialization sequence
end;
```

Synchronization. Concurrent processes synchronize through shared objects in Orca. The system automatically ensures *intra-object synchronization*. That is, operations on shared objects are atomic. The system ensures mutual exclusion of concurrent accesses to shared objects by serializing the concurrent operation invocations. However, the order of operation executions is non-deterministic. The model does not support *inter-object consistency*, or atomic operations across multiple objects.

The efficiency of an implementation involves both time and space efficiency. In Orca, object replication aims at improving the running time of an application by increasing the availability of data and reducing inter-processor communication. However, full replication leads to space inefficiency for large data objects and thus replication alone does not account for efficiency. The efficiency of a replication strategy is determined by the read / write ratio for a given application.

Processes can also synchronize using condition synchronization through guarded commands.

A blocking operation consists of one or more guarded commands:

```
operation op(in_args):out;
begin
guard cond1 do statements1 od;
```

```
...
guard $cond_n$ do $statements_n$ od;
end;
```

The conditions $cond_i$ are boolean expressions called *guards*. A special execution model allows for nested objects. Provisions are made such that the execution of a successful guard does not lead to the execution of blocking operations on nested objects. However, it is difficult to account for efficient execution, given that the evaluation of guards is made on deep copies of the objects (including nested objects) in order to deal with undesirable side effects.

The problem of embedding synchronization into data accesses is twofold: one issue is related to concurrent modification of data leading to the serialization of accesses, the other is related to efficient implementation of update protocols in distributed-memory systems. Pure functional concurrent languages are side effect free, but do not account for “history-sensitive” behavior and thus express fine-grained task parallelism. Actor languages introduce the “history-sensitive” behavior, but serialize the concurrent accesses to data through the replacement behavior. This also leads to fine-grained task parallelism. Orca distinguishes between processes and objects to increase the effectiveness of the implementation, i.e. accesses to objects are synchronized without creating new processes. However, serializing accesses to shared objects ensures *intra-object* consistency and accounts for task parallelism, rather than data parallelism (e.g. all the accesses to a shared matrix object would be serialized). Also, the replication of objects increases concurrency for the read operations, but may lead to expensive update due to frequent writes. Thus, some form of *inter-object* consistency is needed to account for coarse-grain data parallelism in distributed-memory systems.

5.3.2 Implicit Communication

Explicitly managing communication through message passing exposes the programmer to issues such as deadlock and non-determinism. Several approaches to make communication implicit use some form of global address or name space. At the one end, a virtual shared memory implements a global address space at the physical pages level. At the other end, distributed shared memory object models allow accesses in a location independent manner at the level of method invocation or parallel operations on collection of objects. Remote method invocation is expensive in the context of high performance applications. Also, it creates an asymmetric relation between two objects / processes, i.e. a request / reply relation. This asymmetry does not fit well with the inherent symmetry in data parallelism. Parallel object collections require the management of a global object space. Managing a global object space poses many problems, such as random object location, managing hierarchies of objects, etc. Many languages start with a sequential programming language such as C++ and add support for concurrency through parallel collections [24,35,48]. This section discusses the ICC++ [35] model.

Decomposition. In ICC++, concurrency is explicit. The unit of parallelism is a *statement* within a *block* or a *loop*. Statements within a block are partially ordered based on local data dependences. Concurrent loops expose inter-iteration concurrency. In a `conc loop`, loop carried dependences are respected only for scalar variables, but not for others such as array dependences and dependences through pointer structures.

A concurrent block is explicitly marked:

```
conc  S1; S2; ...; Sn
```

A concurrent loop (`for, while`, etc.) is explicitly marked as well:

```
conc for( ...)  ...
```

Decomposition is at the statement level and thus, fine-grain. Data parallelism is achieved by applying the same method to every element of a collection. A collection is defined by the use of standard class definitions, with the addition of `‘‘[ ]’’` to the class name. The unit of distribution is an item of the collection.

For example, a grid data structure can be defined as following:

```
class Grid [ ][ ]{
int count;
Particle* particles;
double Grid[ ][ ]::cell_size;
}
```

An operation invocation on an instance of the `Grid` type (e.g. a `grid` object) will apply synchronously on all the elements of the collection: `grid[i][j].op()`. However, the user is responsible for implementing the operation through the explicit use of `conc` loops or blocks. The parallel execution results in fine-grain concurrency.

Mapping. Concurrent threads of control are implicitly scheduled by the compiler and run-time system. There is no concern for mapping according to a particular criteria, and thus subsequent locality optimizations are employed (pointer alignment and object caching) in order to reduce traffic and synchronization. Dynamic pointer alignment is limited by the availability of static aliasing information. Object caching is also limited to static information on global states of objects and their copies. There is no performance guarantee when such information is not available.

Communication. Communication takes place through remote accesses that can result in high latencies. In order to tolerate these latencies, communication is overlapped with computation. That is, separate threads handle remote and local data accesses. Generating a large number of threads for data accesses (potentially one thread per access) is inefficient. Also, it is possible for a user to explicitly schedule threads to account for load balance.

Synchronization. ICC++ guarantees that two concurrent accesses to an object are serialized through atomic operations. However, the order of invocations is non-deterministic. The language provides explicit user control over consistency for the member variables of an object (potentially nested objects). That is, the `integral` declaration specifies that all the references to a member variable should be considered as read / write operations on that field, and thus require serialization of the accesses to that variable.

The approaches hiding communication through a global name space use serialization of concurrent accesses to data. Thus, these approaches account for intra-object consistency and express a fine-grain data parallelism or task parallelism. Moreover, any one data access can be in any one data space. Remote accesses require high-latency communication. Given that there is no control over the remote versus local accesses, there are no guarantees regarding the communication bounds.

5.3.3 Implicit Mapping

Mapping strategies that ensure spatial data locality are important for data parallel applications. In shared-memory systems, data locality exploits cache-based architectures. In distributed-memory systems, data locality reduces communication. This section discusses a mapping strategy that bounds inter-processor communication for coarse-grain data parallel applications, i.e. the bulk-synchronous parallel model (BSP) [106].

Decomposition. Decomposition in BSP is explicitly managed by the user. A parallel program in BSP consists of a number of “components”, or threads, each performing processing and / or memory functions. The user decides the data / computation for each component, or thread. The unit of concurrency is a *superstep*. In each superstep, each component performs some combination of local computation steps, message transmissions and (implicitly) message arrivals from other components. All the components synchronize at the end of a superstep.

The Oxford BSPlib implementation provides routines for dynamically creating a process and stopping all the running processes. The library routines for process manipulation are similar to those available in the existing MPI implementations. The C language synopsis is:

```
   bsp_init(void (*startproc)(void), int argc, char ** argv);
int bsp_nprocs();
int bsp_pid();
void bsp_abort(char *format,...);
```

The `bsp_init` call creates the processes dynamically. The `bsp_nprocs` and `bsp_pid` calls respectively return the total number of the processes in a program and the identifier for a specific process. All the processes stop when one process calls `bsp_abort`.

Mapping. The mapping objective in the BSP model is bounded communication. Thus, each thread is automatically assigned to a physical processor, based on a cost model that reflects the interconnection network properties. The cost model is based on the time for global synchronization (l) and the rate at which continuous randomly addressed data can be delivered (g). Then, if L is the time for an asynchronous computation (between two synchronization points), the total time for a *superstep* is $t = L + hg + l$, where h is the maximum number of messages sent or received by any processor. Random placements of threads are evaluated in conjunction with a random or adaptive routing to bound the communication time.

Communication. The communication in the BSP model is implied by the mapping. Existing implementations of BSP libraries provide routines to place data into another processor's address space and get data from another processor's address space. These resemble explicit communication. The main difference is that the `push register` and `pop register` calls allow to bind variables to names and thus enable processes to address data across address spaces by name.

> A data structure can be "published" and made visible to remote addresses and / or "unpublished":
> ```
> void bsp_push_reg (const void *ident, int size);
> void bsp_popregister(const void *ident);
> ```

Once the data has been published, the memory operations (put / get) can refer to it across address spaces directly. Therefore, the user does not need to worry about matching send / receive operations.

> The following explicit remote memory operations are available:
> ```
> void bsp_put(int pid,const void *src,void *dst,int offset,int nbytes);
> void bsp_get(int pid,const void *src, int offset,void *dst,int nbytes);
> ```

Synchronization. The BSP model defines facilities for synchronizing all or a subset of the components at regular intervals of L time units, where L is the *periodicity* parameter. After each period of L time units, a global check tells if all the processes have completed a superstep. If they have completed a superstep, the computation proceeds with the next superstep. Otherwise, a period of L units is allocated to the unfinished superstep.

The library implementation gives the user control over synchronization through a barrier synchronization routine:

```
void bsp_sync();
```

BSP is an abstract concurrency model rather than a language or library specification. Therefore, the existing library implementation is very close to the MPI manual parallelization model. The main contribution of the BSP model is to recognize a common concurrency structure, i.e. coarse-grain concurrency in loosely synchronous applications. The challenge is to exploit this fact and design a system that models the common concurrency infrastructure and enables applications to take advantage of it. Thus, the BSP model is more of a guideline to implicit mapping through communication cost estimation.

5.3.4 Implicit Decomposition

In such approaches, parallelism is explicitly expressed in an application through language constructs or compiler annotations, but software developers do not need to worry about decomposition, mapping or communication. This section discusses the implications of such a model. Generally not giving the user control over decomposition and mapping may affect performance to the extent that a compiler is limited to statically finding the best decomposition for an execution model. If the program structure is simple enough, a compiler can use simple execution models to estimate a good decomposition strategy. This is the case of static array languages. This section discusses such strategies for dialects of the Fortran language.

Decomposition. In HPF-like approaches, the unit of parallelism is a block of *loop iterations*. That is, the compiler automatically assigns loop iterations that can execute independently to processors. The `forall` statement allows simultaneous assignment to a group of array elements, resulting in a *generalized array assignment*.

```
FORALL (i=1:N,j=1:M, Y(i,j) .NE. 0.0)
  X(i,j) = 1.0/Y(i,j)
END FORALL
```

The semantics of the above `forall` loop is equivalent with the following sequence of statements:

```
DO i = 1,N
 DO j = 1, M
  IF (Y(i,j) .NE. 0.0) THEN
   X(i,j) = 1.0/Y(i,j)
  ENDIF
 END DO
END DO
```

Mapping. In HPF the user may explicitly specify one of the regular data decomposition strategies (block, cyclic). HPF provides a two-step mapping concept. First, multi-dimensional arrays align / realign with a *template*. Then, the template is distributed / redistributed on the virtual processors arrangement. For example, an alignment suitable for a transpose operation (to reflect locality) is:

```
!HPF$ ALIGN a(j,i) WITH b(i,j)
```

The user may specify a virtual processor topology and the distribution of an array onto the virtual processors, e.g.:

```
!HPF$ PROCESSORS p(1:n, 1:n)
```

One can also specify a distribution (block, cyclic, block-cyclic):

```
!HPF$ DISTRIBUTE b(BLOCK, CYCLIC) ONTO p
```

The virtual processors are automatically mapped onto physical processors. That is, the static communication structure is easily derived based on the problem size (N), and the mapping to physical processors is straightforward.

Communication. Communication is implicit in HPF programs. Communication takes place when different operands of an arithmetic statement are on different processors. By default, the computation is performed on the processor that “owns” the left hand side (LHS) of the assignment statement. Scalars are replicated and synchronized to improve performance for SPMD applications.

Synchronization. The synchronization is implicit in the communication phase.

The natural mapping of data-parallel operations to architectures (e.g. for multidimensional array representation), at least for simple types, enables a compiler and / or a run-time system to make most of the concurrency aspects implicit to the user. It also makes cost measurements possible. The challenge is to automatically handle concurrency in data parallel applications that use more complex structures (indirection arrays, graphs, etc.).

HPF and similar approaches are useful to parallelize applications that have simple concurrency structures and use static data representations. If the size of data is changing, or the data dependences are complex, it is impossible for a compiler to generate a load-balanced distribution. Several syntactic constructs may prevent data distribution as well. For instance, constructs that use mainly scalar variables impede loop parallelization. Thus, many data parallel applications cannot benefit from such support for concurrency.

5.4 Automatic Parallelization

The purpose of automatic parallelization is to discover and exploit parallelism in applications that are written in a sequential programming language. This approach has two advantages. One advantage is that a user does not need to worry about any of the concurrency aspects, and thus the approach is highly usable and cost-effective. Another advantage is that the existing (legacy) applications can be parallelized without costly human resources. The main problem with automatic parallelization is efficiency. A decomposition decision at one loop-level may be very bad later in the program, incurring high communication overheads. Also, indirect indexing for variables within a loop prevents the compiler from discovering parallelism.

Decomposition. The unit of parallelism is a block of *loop iterations*. The data is distributed automatically across multiple address spaces. The computation decomposition follows from the data decomposition and the owner-computes rule: the processor that owns the left-hand side element will perform the computation. Generally, the owner-computes rule implies [9]:

- Data structures have to be partitioned onto processors.
- Memory consistency is preserved since no data are duplicated. This is too strict and leads to poor performance due to increased communication.
- Non-local read references have to be requested from other processors.
- The control partitioning is derived from the data partitioning.

A parallelizing compiler distributes multi-dimensional arrays across multiple address spaces according to one of the regular distributions and based on data dependence analysis. There are several factors limiting the distribution of loops. First, the compiler cannot distribute arrays in the presence of array aliasing. Also, a compiler cannot distribute an array aliased to a scalar variable. Finally, the compiler cannot distribute an array of an unspecified size. In general, array distribution is limited to accesses that are affine expressions (linear) of loop indices. Indirection arrays or other non-linear accesses prevent load balanced distributions and result in poor performance.

Mapping. Mapping is automatically performed by the system, according to load balance and locality constraints. Static execution models are used to evaluate a good mapping solution.

The problem of optimal data layout (decomposition and mapping) is known to be NP hard. The existing compiler approaches use heuristics based on static execution models to choose a "good" data layout that maximizes parallelism and reduces communication and synchronization. These models are limited to regular partitioning strategies and regular applications.

Communication. Communication is automatically generated by the compiler when data on the right hand side of an assignment is not present locally. Due to the limitations of decomposition based on the owner-computes rule and static program analysis, more communication than needed is usually generated. Subsequent optimization techniques are used. Communication aggregation is one of the most important optimizations. This technique aggregates loop invariant small messages into larger ones and moves communication outside the loop.

Synchronization. Synchronization is implicit in the communication calls.

Automatic parallelization is ideally usable. Applications that can benefit from it are mainly scientific, data parallel applications. Even with static array representation, careless coding may impede the compiler in discovering parallelism even in simple cases. Moreover, a static data distribution based on a given loop structure may result in a very bad, inefficient behavior for a subsequent loop structure that uses the same arrays, but in a different access pattern (e.g. reverse order). Several optimizations exist to eliminate redundant communication and improve efficiency for parallelized programs.

5.5 Summary

This chapter has presented a usability framework for concurrency models in the context of data parallel applications. The framework analyzes the concurrency aspects and discusses possible approaches to place some of the burden of concurrent programming to a system. The approaches cover the entire spectrum, from manual parallelization to fully automatic parallelization.

Concrete examples and implementation issues show that it is difficult to find the balance between usability and effectiveness in the approaches. The approaches to hide synchronization by folding it into data accesses do not effectively model data parallelism. The main problem is that ensuring synchronization of concurrent accesses to shared data leads to serialization. On the other hand, encapsulating data into objects forces fine-grain concurrency in order to ensure that enough parallelism is exposed.

Implicit communication is typically achieved by ensuring location independence for data accesses. This can be realized by shared virtual memory, or by ensuring a global name space in a platform independent language (model). The efficiency of such approaches is difficult to guarantee, since any process may access any location at any given time.

Implicit mapping and distribution of data either renounce any parallelization objective (e.g. locality) or require some execution model based heuristics in order to decide between mapping strategies. The problem of optimal data distribution is known to be NP complete. Several strategies use static execution models based on static application structures to estimate load-balanced distributions.

Finally, automatic parallelization is limited by all these factors and applies to fairly simple parallelizable applications. The guideline is that a concurrency model may need to make special assumptions about the class of applications it addresses and express them in such a way that a system can exploit this knowledge and make concurrency implicit. The drawback of such a model is that its applicability is restricted to a particular class of problems. However, this compromise may be successful in delivering a better usability-to-performance ratio.

Chapter 6

Methodology

6.1 Introduction

The goal of this chapter is to provide a transition from the state of the object-based concurrency and data parallel programming to the research contribution of this thesis. The chapter describes the context of the work as a primary motivation for the research focus. Furthermore, it states the goal for our approach, together with the research questions and hypotheses. The research methodology is elaborated by summarizing the approach and describing the concrete evaluation and experimental basis.

6.2 Context

The context of this work is the Computational Science and Engineering (CSE) strategic project at the Norwegian University of Science and Technology (NTNU). The goal of the project is to bring together researchers with different competences to cooperate towards solving grand-challenge problems [85, 88, 91]. The problem addressed within the scope of the project is the design and analysis of fluid exposed marine structures.

The mathematics behind this problem consist of a numerical formulation of the solution of the Navier-Stokes equations. These equations are extensively used in fluid mechanics. One typical application is to model fluid-flow problems in aerodynamics and avionics to study turbulent air currents that follow a flight in a modern jet.

We use the incompressible Navier-Stokes equations in our project. A typical problem in fluid dynamics is to study the fluid flow around a cylinder. The understanding of solutions to the Navier-Stokes equations is still an active research area. However, typical solutions use discrete mathematics, i.e. the Finite Element Method (FEM) to solve the Partial Differential Equations (PDEs).

Figure 6.1 illustrates the typical solution of a scientific problem. The starting point is a physical formulation of a natural phenomena. The problem is reduced to a mathematical formulation using simplifying assumptions and identifying the physical quantities of interests. The mathematical formulation consists of a set of equations to be solved for the physical quantities of interest. For instance, the unknowns of the Navier-Stokes equations are the pressure and velocity for the fluid flow. There are no analytical solutions for such complex PDEs. Therefore, numerical analysis

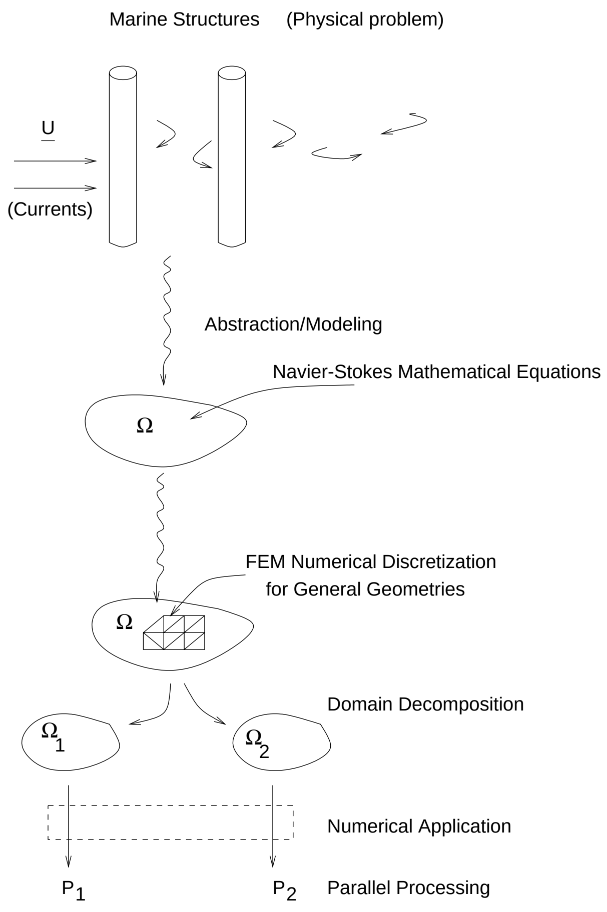


Figure 6.1: The typical solution of a grand-challenge problem. We use as example the fluid structure interaction in deep sea. The Navier-Stokes equations are used to model the currents around the marine structures.

uses discretizations of the continuous physical quantities to derive approximate solutions. A FEM discretization uses elements of different shapes to discretize the definition domain for the equations. Accurate modeling of the physical reality requires three-dimensional geometric discretizations and experimentation with various element shapes and sizes. Computer simulations solve the discrete equations. This type of computing is also called computational science or scientific computing.

The characteristics of scientific computing are:

- The data sets modeling a complex problem are large and the computations require a large number of floating point operations. These computations require parallel execution to reduce the simulation time.

- The discretized domains for the equations, called *meshes* or *grids* use complex geometries, and thus irregular data structures. This complicates the programming model and the parallelization structure for the applications.

- Domain decomposition leads to computations that are coupled for the data on the border between separate domains. This leads to *nearest neighbour* communication in parallel applications. Automatically detecting the synchronization points and the communication data is challenging in the presence of irregular data representations that do not capture locality (connectivity) information between the grid points describing the discretization domain.

The work in this thesis focuses on exploring a software methodology for writing parallel scientific, high-performance applications. That is, it addresses the following questions: What are the limitations of the current technology in supporting parallelization for numerical applications? Can commodity-oriented distributed computing platforms (e.g. clusters of PCs) replace the shared-memory multiprocessors? Finally, why do scientific programmers resort to manual parallelization?

Specifically, this work focuses on methods for hiding as much as possible of concurrency from the programmer in distributed-memory systems while enabling modular and reusable software developments through the use of object-oriented technologies.

Existing work on system support for concurrency uses Fortran as a starting point. There are two main lines of thought in approaching parallelism. One is to add concurrency features to the Fortran language (e.g. HPF). The other approach is automatic parallelization of sequential Fortran programs.

Throughout this thesis we distinguish between *regular* and *irregular* numerical applications. We use the term regular to refer to applications that use dense matrices in solving numerical problems (typically linear algebra). We use the term irregular to refer to applications that use sparse matrices or general geometries in solving numerical problems. Regular applications require support for multi-dimensional arrays and reduction operations. Irregular applications require support for recursive (e.g. graphs) and hierarchical (e.g. indirection arrays) data structures and fast access operations.

6.3 Goal

The goal of this work is to explore a means for effective implicit parallelism for scientific applications. The aim is to provide a uniform framework to address the requirements for implicit parallelism and effectiveness for both regular and irregular applications.

The main assumption is that the applications exhibit data parallelism. Fine-grain parallelism refers to the exploitation of parallelism at the instruction level for vector computers, or loop-level for ideal shared-memory multi-processors with an unbounded number of processors. Coarse-grain parallelism refers to the exploitation of parallelism in multicomputers, with a fixed number of computation nodes on which concurrent components of an application run sequentially. This work focuses on support for coarse-grain data parallelism in distributed-memory systems.

6.4 Research Questions and Hypotheses

There are two main research questions related to our goal:

Effectiveness: What is the relation between the usability and efficiency for high performance applications? Automatic parallelization of scientific applications potentially achieves ideal usability and correctness. Parallelizing compilers may not be able to achieve satisfactory performance, mainly due to their limitation in discovering parallelism and conservative analysis leading to inefficient treatment (e.g. generating redundant communication). Manual parallelization can potentially achieve the best efficiency for a given application. However, manual parallelization exposes the user to a complex, error-prone programming model.

Programmability: What can be revealed from concurrency aspects and how can it be done to explicitly express some form of concurrency in order to enable effective system support? Between fully implicit and manual parallelism a compromise is to express concurrency either through different data abstractions (e.g. parallel collections) or control statements (e.g. the `forall` statement).

The thesis answers these questions by proposing a concurrency model that occupies the middle ground between the automatic and manual parallelization. The concurrency model is based on the distinction between distributed objects that require inter-object consistency and sequential objects that do not require consistency. Thus, based on the explicit distinction between data objects in an application, a system implicitly decomposes the program into concurrent tasks, maps the tasks to processors, and generates the necessary communication and synchronization between the concurrent tasks.

The main hypotheses for our approach are:

H1 *Effective execution of data parallel applications in distributed systems.* The assumption leading to this hypothesis is that the communication structures for different classes of numerical applications can be deduced based on knowledge of both the specific application and the program implementation, and the number of synchronization points in a program is small.

H2 *Increased programmability for distributed data parallel applications.* The assumption leading to this hypothesis is that the applications share similar structures and the most difficult tasks of data distribution and communication can be solved automatically by a system. Thus, the scientific programmer has to supply only the sequential, application specific part.

6.5 Approach

This section gives an overview of the approach we use to address the two research questions: effectiveness and programmability. We propose a concurrency model that is both efficient and easy to use.

The solution steps for our approach to distributed data parallel programming are:

1. [User] Express a data parallel application using our programming model. That is, use the distributed data type to designate large data objects in the application. Besides the distinction between sequential and distributed objects, the programming model is close to sequential.

2. [System] Decompose the problem into a number of subproblems to be solved concurrently by the available processors in the system. We decompose a program along data. That is, large data objects are mapped onto a graph that reflects the connectivity between data in a complete traversal. Then, we use multi-objective graph partitioning algorithms to distribute the data across multiple address spaces such as to preserve locality of reference and thus, reduce the inter-processor communication.

3. [System] Solve the subproblems concurrently in a SPMD (Single Program Multiple Data) style. The same computation applies to the sequential and distributed data objects in each address space.

4. [System] Ensure global data consistency across address spaces. We ensure inter-object consistency for the distributed data. In our model, the processing of distributed objects is synchronized such as the individual objects' states make up a global consistent state in the absence of a shared memory view. Existing object-based concurrency models for data parallel applications use an intra-object concurrency model. That is, the accesses to a given object are synchronized such as the object is in a consistent state, independent of other objects. This means that the concurrent accesses to an object are serialized.

6.5.1 Effectiveness

There are two main aspects of effectiveness for a concurrency model in a distributed memory: locality of reference and low communication. The concurrency model addresses these issues in two steps:

1. Find a data layout that uses a general distribution of data and mapping such as to preserve data locality and thus reduce communication due to references to the remote data. The data layout is described in Chapter 7.

2. Use a data consistency scheme that incurs low space and time overhead for updating data across multiple address spaces. The efficiency aspects of the consistency protocol are described in Chapter 8.

We assess effectiveness by measuring the speedup of an application with an increasing number of processors. We use two types of measurements. One measurement is the speedup of a PDE solver for different problem sizes and increasing number of processors. Due to time limits and significant differences in expressiveness between different concurrent programming models, we do not compare the implementation of the PDE solver against other programming models. The other measurement is the comparison of the performance of a toy application implemented using various existing programming models for distributed computing, as well as our own model.

6.5.2 Programmability

We assess programmability both qualitatively and quantitatively. The qualitative assessment is based on the classification of the concurrent programming model, presented in Chapter 5, that is, according to the ability to hide one or more of the concurrent computing aspects: decomposition, mapping, communication and synchronization. This concurrency model makes decomposition, communication and synchronization implicit, and requires the programmer to indicate the data objects to which these apply. The quantitative assessment is based on a usability metric which will be described in Section 6.6.

We explore implicit parallelism in a data parallel programming model that requires the user to identify the data to be distributed across address spaces. Besides this distinction between the data objects, the programming model resembles a sequential model. Thus, we address the programmability of the concurrency model at three levels:

1. We propose a high-level, data parallel programming model that identifies the main abstractions in a data parallel application and uses *distributed* and *sequential* objects to account for implicit concurrency aspects. This high-level model is described in Chapter 9.

2. We propose a uniform data abstraction mechanism to enable the user to represent both regular and irregular data objects. This model uses recursive *sets* to allow the user to express data structures most commonly used in numerical applications and enable a system to account for data locality and reduced communication requirements. This model is described in Chapter 10.

3. We describe a prototype system implementation for experimentation with our models. The prototype system realizes some of the run-time functionality of the proposed model. However, it is not a complete system and the calls to the run-time system routines are manually inserted. We briefly describe the compiler techniques that would automatically detect and generate the insertion place for the run-time routines, but we do not implement them. The requirements for the prototype system together with its implementation are described in Chapter 11.

6.6 Experimental Methodology

The experimental base to validate the achievement of the goals consists of distributed and NUMA (Non-Uniform Memory Access) architectures, metrics to measure efficiency and usability, and a test bed consisting of data parallel applications relevant for this work.

6.6.1 Performance Metrics

The performance of a concurrent program indicates how the execution time for a given program depends on the number of processors. Thus, the number of processors becomes a study parameter, and the problem and multicomputer are fixed parameters, given the following definitions.

Definition 6.1 *[107] The speedup of a p-node computation with execution time T_p is given by:*

$$S_p = T_1^* / T_p,$$

where T_1^* is the best sequential time obtained for the same problem on the same multicomputer.

Definition 6.2 *[107] The efficiency of a p-node computation with speedup S_p is given by:*

$$\eta_p = S_p / p$$

In practice, performance data is obtained just for one sequential and just for one multicomputer program. We use the sequential time based on the parallel program running on one node and not the sequential time for the sequential program.

We measure the speedup for different problems for fixed sizes and execution platforms (multicomputer, i.e. a distributed-memory model where communication only takes place through message passing). Also, we vary the problem sizes and keep the platform fixed to study the effect of the problem size on the concurrent execution. With coarse-grain parallel applications in a distributed environment, a scalable implementation will lead to an increase in efficiency for larger problem sizes. We also fix the problem, the number of processors, and vary the execution platform.

6.6.2 Usability Metrics

The quantitative assessment of concurrent programming environments is by no means comprehensive. It uses a *usability* metric to compare a toy problem expressed in different concurrent programming environments, including our own.

The metrics used in this work are not intended to give a precise, absolute measure of usability. The purpose is to offer a guideline on how difficult it is for a programmer to implement an application using a concurrency model in the absence of a parallelizing compiler (ideally, in such cases no effort is required from the programmer). We define the following metrics to assess the usability of a concurrent programming model:

- *The learning curve* is the time spent by a user to learn a new system, library or language (in days).
- *The user effort* is the effort required from the user to have a working application by using a new system or formalism.
  - *The number of lines of code* is the number of extra lines of code the user has to write by using an existing system, library or language in order to complete an application.
  - *The number of classes* is the number of extra classes the user has to define by using an existing object-oriented language or system in order to complete an application.
  - *The number of decision points* is the number of points in the code where the user makes decisions about distributed computing (i.e. is aware of aspects such as communication, remote method invocation, data consistency, etc.).

We use different programming models to implement the same concurrent application, including our proposed programming model. We use students to collect usability data. The results strongly depend on the human factor: the programming style, the ability to exploit the expressiveness of a given model, etc.

6.6.3 Test Bed

The research in this thesis points out one significant problem: there is a lack of methodology for study and research in scientific software development. This problem stems from a number of incompatibilities between the current practice and isolated attempts to explore novel software technologies to improve the software development process. These incompatibilities are listed in Table 6.1.

Table 6.1: Factors for the lack of methodology in scientific software.

Incompatibility	**Current Practice**	**Desired Feature**
Programming Language	Fortran languages	Higher level languages (e.g. C++)
Application	Mainly Regular (linear algebra)	General (Regular and Irregular)
Input data	Regular Meshes	General Geometries
Input data format	Non-Standard	Standardized Format
Numerical Method	Finite Difference / Finite Volume	Finite Elements
Goal	Compiler Transformation / Machine Specific Tuned Libraries	Modeling and Methodological Experimentation
Numerical Expertise in Programming	Scientists	Scientists Support

Table 6.2: Characterization of the Test Bed applications.

Feature	**Non-trivial Data Parallelism**	**Trivial Data Parallelism**
Programming Language	C++	C++
Application	3D Poisson Solver	3D Body Surface Computation
Input data	Tetrahedra Mesh	Tetrahedra Mesh
Input data format	Non-Standard	Non-Standard
Numerical Method	Finite Elements, Jacobi Iteration	Vector Products
Goal	Experimentation	Experimentation
Numerical Expertise in Programming	None	Trivial

Existing benchmarks for scientific applications [20] reflect the current practice, i.e. are useful for the much researched and understood regular Fortran applications. One difficulty encountered while pursuing this research was the lack of programming examples for irregular problems, input meshes in different formats and implementations in other languages than Fortran. Thus, much valuable time was dedicated to this problem alone.

We use scientific and numeric programs for our study. Since our focus is on a special class of applications, i.e. FEM solutions of PDEs, we use the Poisson problem as representative for the class of applications we address. Table 6.2 lists the characterization of the test bed we used.

We use a non-trivial data parallel application to validate the functional and non-functional research hypotheses. We use a trivial data parallel application to assess the usability metrics and collect quantitative data on differences between various concurrent programming models. Due to the lack of benchmark suites and limited time resources, it would not be feasible to have a complex, numerical application implemented in multiple concurrent programming models within the time-frame of a thesis.

We use two test problems[1] for the applications. Their description is given in Table 6.3. These problems are relatively small compared to real applications. The problem of generating accurate 3D discretizations for various domains is a research area in itself and it does not concern our work. Thus, we have used problems we have found available for the benchmarking applications we are using. Larger problems need to be tested to sustain our experimental results on the scalability of our approach.

[1]We used two meshes from Peter Monk, Department of Mathematical Sciences University of Delaware Newark, available at http://www.math.udel.edu/ monk/

Table 6.3: Characterization of the test problems.

Mesh	**Dimension**	**Number of Elements**	**Number of Faces**	**Number of Vertices**
Tiny	3	6000	12 600	1331
Small	3	15 581	31 818	2933

6.6.4 Execution Environment

The execution model is a distributed-memory platform consisting of multiple processors linked by some interconnection medium. The communication between different address spaces is through message passing. Different architectures have different interconnecting speeds and different per-node performance, which influence the overall application performance. We use three different architectures to test the application performance:

1. SGI 3800 is a third-generation NUMA architecture, called NUMA 3. In this architecture, all processors and memory are tied together into a single logical system through the use of special crossbar switches developed by SGI. The combination of processors, memory, and crossbar switches constitute the interconnect fabric called NUMAlink. This special hardware interconnect delivers a low-latency and high-bandwidth architecture. The memory latency ratio between remote and local memory is 2:1 (less than 600 nanoseconds round-trip in the largest configuration). For message passing programming, SGI offers MPI and PVM libraries through Message Passing Toolkit (MPT) and SGI PVM 3.3 software, respectively. MPT provides versions of industry-standard message-passing libraries optimized for SGI computer systems. These high-performance libraries permit application developers to use standard, portable interfaces for developing applications while obtaining the best possible communications performance on SGI computer systems. Table 6.4 gives the system characteristics. Even though memory latency rates are given in the SGI 3000 Family Reference Guide [2], we did not find any data on the message delivery rate when using message passing.

2. The ClustIS Linux cluster consists of 36 computational nodes and one master node. The nodes are on a switched private network with 100MBit/sec between the nodes and the switch and a 1GBit/sec link between the master node and the switch. Table 6.6 lists the detailed characteristics.

3. Beowulf is a 6 processors Linux cluster which consists of 3 nodes with two Intel Pentium III 500 MHz processors and 512 MB RAM each. Table 6.7 summarizes the technical characteristics.

[2]Available at http://www.sgi.com/origin/3000/

Table 6.4: SGI Origin 3800

Processors:	220 MIPS R14000 processors
Main Memory:	220 GBytes total
Disk:	2.2 TBytes
Architecture:	Distributed Shared Memory
Peak Performance:	1000 Mflops per processor / 220 Gflops total
Operating System:	Trusted IRIX
Main Purpose:	Compute Server

Table 6.5: SGI Origin 3800

Table 6.6: ClustIS Linux cluster

Node Type	**Processor**	**Main Memory**	**Secondary Memory**	**Interconnect**
Master:	AMD Athlon XP 1700+ (1.46 GHz)	2GB	2*80GB IDE	1*Gigabit , 1*100MBit
Nodes: 1..16	AMD Athlon XP 1700+ (1.46 GHz)	2GB	1*40GB IDE	1*100MBit
Nodes: 17.. 28	AMD Athlon XP 1700+ (1.46 GHz)	1GB	1*40GB IDE	1*100MBit
Nodes: 29 .. 36	AMD Athlon MP 1600+ (1.4 GHz)	1GB	1*18GB SCSI	1*100MBit

Table 6.7: Beowulf Cluster

Processors:	Pentium III 500 MHz
Main Memory:	512 MB RAM each
Interconnect:	100Mbit

6.7 Summary

This chapter ends the first part of this thesis related to the state of concurrency models for data parallel applications. The purpose of this part is more than just to point out the state-of-the-art of our research area: it puts in perspective all the important aspects that are related to a concurrency model for the data parallel applications. Thus, it reflects on the various concurrency models, their expressiveness and limitations, performance aspects, driving design decisions, and provides an in-depth analysis of data parallelism. This first part serves as a comprehensive motivation and assessment for the approach in this thesis. After reading these chapters, the reader should be ready to explore the benefits and limitations of our approach, the new research paths it opens, and the new questions it raises.

Part III

Results

Chapter 7

A General Data Layout for Distributed Consistency

Parts of this chapter were published as a conference paper [41].

7.1 Introduction

Parallel execution for scientific applications can be employed in the context of shared-memory and distributed-memory architectures. The shared memory paradigm provides the benefit of programmability, since it is closer to the sequential programming model. The distributed-memory paradigm is appealing for its scalability and reduced cost. Since the latter is important for scientific computations, it becomes increasingly important to aim the research towards closing the gap in usability between shared- and distributed-memory programming models [11].

This chapter investigates the issues of general data layout and consistency for scientific applications in distributed-memory systems. The type of the applications and cost considerations influence the decomposition of the problem and the consistency scheme employed [10]. We address data parallel applications with general access patterns. The concurrency model is based on *peer processes*[1] that communicate by exchanging messages. Point-to-point and collective communication routines are used to maintain the consistency of data distributed across multiple address spaces.

In the SPMD model, data on each partition can be processed independently, but this does not guarantee correct results. This is because data dependences across partitions exist. One simplifying assumption that makes our approach scalable is that consistency can be ensured only at synchronization points. This means that the synchronization points have to be detected by the application programmer or some system support (e.g. compiler, run-time). Since our goal is to ensure implicit parallelism, and thus automatic data layout and consistency, we opt for the latter choice. That is, we want the system to be able to detect the synchronization points and decide what action to take upon discovering such points. Both of the problems are far from being trivial. In this chapter, we address thoroughly the second one; that is, by assuming

[1]Identical processes

that the synchronization points can be identified, what data layout can be used in a distributed data consistency algorithm.

Previous experience with data parallel applications shows that the synchronization points are loose, usually occur at the end of a large computation step, and typically involve data residing on the boundary between two different partitions. Thus, a data layout must take into consideration the spatial locality of data in order to ensure an efficient consistency scheme. We use a generic relation to reflect spatial data locality in our model. This chapter makes the following contributions:

- **A general data layout**. We describe a class of data parallel applications and the data model they employ. We formulate the problems of data partitioning, data mapping and data consistency. The layout explicitly captures the *relation* between the set elements that are to be partitioned. Furthermore, it uses this extra information to find a general solution to the transformation between local and global data spaces in the absence of a symbolic expression (e.g. as with block and cyclic partitioning).

- **A distributed algorithm for data consistency**. The algorithm we propose converges from the above formulations and shows when data consistency needs to be employed and what actions that are needed in order to achieve correctness.

- **Applicability of our algorithm**. We show how our algorithm can be employed, in conjunction with well-known solutions to our formulated problems (of data partitioning and mapping) to ensure data consistency for two important classes of applications. These are regular and irregular numerical applications for solving Partial Differential Equations (PDEs).

- **Experimental results**. We show that our approach is scalable, by providing experimental results. We use our prototype system implementation for the evaluation. We have used the system to parallelize a serial three-dimensional Finite Element (FEM) Poisson solver on a general (tetrahedra) mesh.

The remainder of the chapter is organized as follows: Section 7.2 introduces the class of applications we address, their characteristics and simplifying assumptions. Section 7.3 introduces the data layout we use, and formulates the problems of data partitioning, mapping and consistency. Section 7.4 describes a distributed data consistency algorithm. Then, in Section 7.5, we describe concrete applications of our algorithm. Section 7.6 presents results from the implementation of our layout and consistency schemes. Section 7.7 provides an overview the existing approaches to automatic data layout and consistency, and contrasts them to our approach. We conclude the chapter in Section 7.8 and indicate future directions for our work.

7.2 Background

This section describes in detail the data parallel applications, their characteristics and behavior. We discuss data parallel numeric applications. One assumption is that these applications use very large data sets.

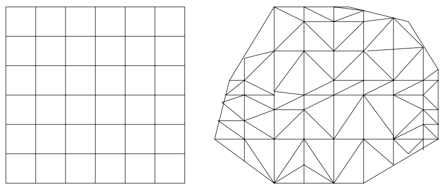


Figure 7.1: Regular and irregular data sets.

Data parallel applications consist of computations that apply independently and simultaneously to many pieces of data. The applications can be classified as *regular* or *irregular* according to the shape of the data. Figure 7.1 shows typical input data representations for numerical applications that use discrete meshes (or grids) to represent continuous, physical domains.

The regular data sets represented using multi-dimensional arrays map directly onto an iteration space describing a nested loop (e.g. for i, for j). This is the most exploited feature for instruction-level and loop-level parallelism and / or memory optimization techniques in cache-based architectures.

Another feature exploited for the parallelization of these applications is the possibility to uniformly distribute the data into alike memory blocks and align them / map them to specific memory addresses. Then, given the regularity of the data representation and its distribution, any data address can be computed as a simple function application. This means that any data item can be located from any point by applying a simple recurrence formula that describes the regularity of the data. For instance, a cyclic distribution maps a global index i of an array to processor p in a P processors multicomputer using the formula $p = imodP$.

The irregular data sets are not generated by using a recurrent relation, but by complex techniques (e.g. mesh generation). As a consequence, the first property that differentiates them from the regular sets is that much more information must be kept in order to access / locate a specific data item. Such information can be, for example, spatial coordinates for mesh data, or connectivity information for data described by graph-like structures.

The second complicating factor is that it is much harder to access / locate a data item starting from an arbitrary point based on the connectivity information. Much more complex data structures and procedures must be employed in order to make iterations over such data efficient. The gap in the difficulty of treating the two types of data gets even bigger when introducing the *distribution*, or *parallelization* factor.

So far we have described the typical data models in scientific applications. The computation is another factor that must be taken into account for parallelizing an application. For the class of applications we address, it is common that the computation can proceed independently on large regions of data. Because of the small data subsets, relating the independent data parts, global data consistency must be ensured. We assume a distributed-memory model. Therefore data consistency is ensured through message passing communication and synchronization.

In order for the parallelization to be worthwhile in a distributed memory, the independent computation steps must be large, and followed by a collective communication phase to ensure correct values for related data across processors.

To summarize, the following subproblems have to be solved when parallelizing a data intensive, numerical application:

1. Data partitioning.
2. Mapping of the partitions onto different address spaces.
3. Data consistency through:
    - Communication.
    - Synchronization.

In the remainder of the chapter we will continue to refer to the irregular data applications. This is because, on the one hand, if it is desired or useful, the regular case can be treated as a particular case of the more general irregular case. On the other hand, if an irregular problem reduces to a regular problem (e.g. the FEM formulation reduces to a linear system of equations), the two parts can be decoupled, and data remapped according to a regular data layout that reflects the new spatial locality requirements. Then the usual techniques employed for linear algebra problems can be applied.

In the following sections we will formulate the subproblems identified above[2].

7.3 The General Data Layout

7.3.1 Data Partitioning

We formulate the problem of data partitioning as follows:
Given a data set D of size N, N large, and a symmetric relation R:

$$\begin{aligned} D &= \{d_i \mid i = \overline{1, N}\} \\ R &: D \to D,\ \forall\ d_i\ s.t.\ \exists\ d_j,\ j = \overline{1, N},\ R(d_i)\ =\ d_j, \\ \exists R^{-1} &: D \to D\ s.t.\ R^{-1}(d_j)\ = d_i, \end{aligned}$$

Find a set $S = \{S_p \mid p = \overline{1, P}\}$ with P bounded and sufficiently small and a new relation R_s such that the following conditions are satisfied:

$$\begin{aligned} S_p &= \{d_{i_p} | i_p = \overline{0, |S_p|}\} && (7.1) \\ R_s &: S \to S, && (7.2) \\ \forall d_{i_p} \in S_p\ s.t.\ \exists d_{j_p} \in S_p\ R(d_{i_p}) &= d_{j_p},\ R_s \equiv R \\ \forall d_{i_p} \in S_p\ s.t.\ \exists d_{j_p} \in S_p\ R^{-1}(d_{i_p}) &= d_{j_p},\ R_s \equiv R \\ \forall p, k &= \overline{1, P},\ p \neq k\ S_p \cap S_k = \emptyset && (7.3) \\ \forall p, k &= \overline{1, P},\ p \neq k,\ abs(|S_p| - |S_k|) \leq \Delta, \end{aligned}$$

[2]All the formulations throughout the chapter are in one-dimension, for the sake of simplicity. A multidimensional, vector-based notation would complicate the formulas unnecessary.

$$\Delta \ll |S_p|, \Delta \ll |S_k| \tag{7.4}$$

$$\bigcup_{p=1}^{P} S_p \equiv D. \tag{7.5}$$

The first condition states that the initial relation between the set elements is retained for those elements being related to elements from the same subset. The second condition ensures non-overlapping [7] data partitioning. The third condition requires that the size of any two different subsets must not differ by more than a bounded quantity which is much smaller than the size of both subsets (e.g. load balance). The last condition is the prerequisite to data consistency: the initial data set and its partitions are interchangeable. That is, the initial data set can be reconstructed correctly from its subsets.

7.3.2 Data Mapping

We formulate the data mapping problem as follows:
Find a partitioning function $\mathcal{P}$

$$\mathcal{P} : D \to S, \tag{7.6}$$

such that[3]:

$$\begin{aligned} \forall\, d_i \in D,\ \mathcal{P}(d_i) = d_{i_p} \in S_p,\ \exists\, \mathcal{P}^{-1} : S_p &\to D, \\ s.t.\, \mathcal{P}^{-1}(d_{i_p}) &= d_i \end{aligned}$$

Given a partitioning function that satisfies the above, a solution to the transformation between the initial data set and its partitions is:

$$\begin{aligned} \forall\, d_{i_p} \in S_p,\ s.t. \exists\, d_{j_k} \in S_k \neq S_p,\ R(d_{i_p}) &= d_{j_k} \\ let\ R_s(d_{i_p}) &= (\mathcal{P} \circ R \circ \mathcal{P}^{-1})(d_{i_p}) \end{aligned} \tag{7.7}$$

$$\begin{aligned} \forall d_{i_p} \in S_p,\ s.t. \exists\, d_{j_k} \in S_k \neq S_p,\ R^{-1}(d_{i_p}) &= d_{j_k}, \\ let\ R_s^{-1}(d_{i_p}) &= (\mathcal{P} \circ R^{-1} \circ \mathcal{P}^{-1})(d_{i_p}) \end{aligned} \tag{7.8}$$

The formulation of the data mapping problem adds two important aspects to the data decomposition problem. One is to find a bijective function that uniquely associates an element in the initial data set with a partition. The reversibility condition ensures that from each data subset, the correspondent in the initial data set can be found. The other is to construct the relations that have been invalidated by the data partitioning based on the original relation, the partitioning function and its inverse. This information enables the transformation between the global and local data spaces.

Claim: The solution to the transformation problem is correct.

Proof: It is easy to see that $\exists \mathcal{P}^{-1}(d_{i_p}) = d_i \in D\ and\ \exists R(d_i) = d_j$. Then $\exists (R \circ \mathcal{P}^{-1})(d_{i_p}) = d_j$. It follows that given it $\exists \mathcal{P}(d_j)$, then $(\mathcal{P} \circ R \circ \mathcal{P}^{-1})(d_{i_p})$ is correctly defined. The proof of (7.8) is similar.

[3]Throughout the chapter, $\mathcal{P}$ as partitioning function is different from P as the bounded number of partitions.

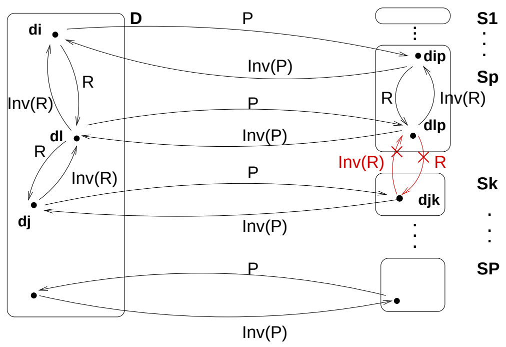


Figure 7.2: The partitioning and mapping for a data set.

The way we formulate we say that there cannot exist explicit relations among two different subsets. But since we want to preserve the equivalence and irreversibility of the initial data and its partition in subsets, we can soundly construct a relation through function composition.

7.3.3 Data Consistency

The data consistency problem requires that we ensure that applying the same transformation independently on each of the data subsets leads to the same result as when applying the transformation on the initial data set:

Let $T :< D, V > \rightarrow < D, V >$, with T a *a generic transformation* and V a set of values, be a transformation such that:

$$\exists T :< S_p, V_p > \rightarrow < S_p, V_p >,\ p = \overline{1, P} \tag{7.9}$$

The partial values V_p are the values obtained by applying the same transformation individually to each partition of data. Our goal is to ensure that the partial and the global sets of values are equivalent in order for the parallel execution to be correct. Thus, we require that:

$$\bigcup_{p=1}^{P} V_p \equiv V \tag{7.10}$$

is satisfied.

Figure 7.2 graphically illustrates the problem formulations from above. The data in the global set D is partitioned into disjoint subsets S such that the relation between elements belonging to different subsets is reconstructed to preserve the initial relation governing the global set.

The $\bigcup_{p=1}^{P} \{< S_p, R, R_s, \mathcal{P}, \mathcal{P}^{-1}, T >\}$ is necessary to the solution of the consistency problem. This means that the tuples $< S_p, R, R_s, \mathcal{P}, \mathcal{P}^{-1}, T >$ contain all the ingredients necessary to be able to ensure data consistency.

The tuples are only a prerequisite (necessary) for solving our problem. Given that we have a data set D and its partition in subsets S_p, such that the data mapping problem is solved, we are able to construct the relations that are invalidated in the partitioning process, knowing $R_s, \mathcal{P}, \mathcal{P}^{-1}$. Let us call the tuple $< S_p, R, R_s, \mathcal{P}, \mathcal{P}^{-1}, T >$ a subdomain and denote it by $Subd_p$.

Any time a transformation on a data item from a subset S_p is applied, the information from the corresponding subdomain $Subd_p$ will suffice to identify that there exists a remote data item in relation to the locally transformed data item and to correctly identify the remote subset and data item related to the local data item. This is guaranteed by the relations 7.7 and 7.8. In this case, the remote data item is the *destination* of the *relation dependency*. Conversely, we will also be able to identify any data item that is remote and in inverse relation with a local data item. This is the remote *source* of the *relation dependency*.

We call the detected *sources* and *destinations* of remote data items related to data in each individual data set the *data relation pattern* of a subdomain. These model the connection between subdomains, based on the original relation R characterizing the data set D, the partition set S_p, the partitioning function $\mathcal{P}$ and its inverse $\mathcal{P}^{-1}$.

The remaining necessary step to ensure data consistency is to generate communication. We need to establish when to generate communication, what to communicate, and where to communicate. The communication will be generated according to the computed patterns.

7.3.4 Communication

Distributed data consistency uses point-to-point communication to exchange values for data that is related across address spaces. Typical consistency protocols use either an invalidation or an update strategy. Every time a data item is updated in one address space, an invalidation scheme requires invalidation of data on all the processors that have a copy of it. Thus, the update of data is postponed until the next read. With update schemes, all the copies of data in other address spaces are updated. Communication is generated on write. An invalidation scheme requires more messages (invalidation and update) than an update scheme. Invalidation generates communication on read, while update generates communication on write.

We use an update protocol to generate communication for data that are related across address spaces. That is, a write operation to data that have relations to data in other address spaces causes a message to be sent. This eliminates the communication on read, since the updated value is guaranteed to be available.

When to Communicate

The moment of generating communication does not only depend on the data relations across address spaces according to the data representation and its partitioning, but also on the type of transformation performed (computation). This is generally strongly related to the application behavior, which we cannot know in advance. Some assumptions are usually being made for different classes of applications.

We divide the data belonging to a data set into data items that preserve their relations from the initial data set, and data items that do not preserve them (their relations were invalidated, then recomputed). We call the former data *independent data*. That is, this data is directly related (as source or destination of a relationship) only with data from the subset they belong to. We call the latter type of data *dependent data*. This is the data whose relations were invalidated by the partitioning and reconstructed later on.

Subsequently, the transformations which solely involve independent data can be carried out in parallel. For the transformations which involve dependent data, consistency must be ensured through communication. Communication is generated after transformations involving updates (writes) of such data.

What and Where to Communicate

The data to be communicated also depend on the application. Assuming that the data we treat in the above manner is *left values* (i.e. locations, data references) of application data (data address), then the *right values* of this data (data value) must be communicated according to the computed communication patterns. The right values are application and computation dependent.

The right values of the data from their original subdomain (owners) will be communicated to the subdomain indicated by the relations between different data subsets. That is, application data objects are communicated between different address spaces at the level of granularity decided by the implementation. Typically, all the data values that are remotely related need to be exchanged in a communication step. Thus, one large message is used to update all the values changed in a computation step.

7.4 A Distributed Algorithm for Data Consistency

7.4.1 Implicit Parallelism

The motivation for implicit parallelism in distributed-memory applications is to alleviate the user (programmer) from the complexity of writing and debugging concurrent applications. The development of message passing parallel programs has long been identified as a cumbersome and error-prone process.

Ideally, parallelism should be fully implicit. This cannot be achieved for any kind of application with the available tools and techniques. Nor is it always desirable, since this would impede the experimentation with novel, more performant parallel algorithms. As identified in [97], the existing languages and systems can be classified according to the ability of making implicit the following:

- Parallelism
- Decomposition
- Mapping
- Communication

- Synchronization.

Different approaches cover subsets of the above from all to nothing, i.e. ideal to manual parallelization. Also, the approaches address different classes of applications (based on certain features or assumptions). To our knowledge, the "state-of-the-art" approach for irregular applications is close to "all explicit". There are some approaches [69, 111] that make mapping, communication and synchronization implicit, while decomposition is still explicit. However, as most of the approaches for implicit parallelism in data parallel applications, this one is also restricted to loop-level parallelism.

As a solution to the problems formulated in the previous section, we present a distributed consistency algorithm for data parallel applications. We will show how decomposition, mapping, communication and synchronization can be made implicit by using our algorithm. Parallelism remains explicit to the extent that the distributed data type is explicit.

7.4.2 A Distributed Consistency Algorithm

This section presents a distributed consistency algorithm for data parallel applications using the foundation built in the previous sections. The algorithm shows how data consistency can be preserved when parallelizing a data parallel application in a distributed-memory environment.

Let D be the distributed data set holding the left values of the distributed data defined by an application. Let $T :< D, V > \to < D, V >$ be a composed transformation over the initial set D. Then we denote a data parallel (sequential) program by the tuple $< D, T, V >$.

Let $T_i, i = \overline{1, n}$ be a series of transformations that compose T, i.e. $T = T_n \circ ... \circ T_1$. Then we can write:

$$\begin{aligned} T(D, V) &= (T_n \circ ... \circ T_1)(D_1, V_1) \\ &= ... = T_n(D_n, Vn) \end{aligned}$$

There exists a decomposition of T_i into a set of atomic operations such that $T_i = Op_1 \circ ... Op_l$. The atomic operations can be classified into *read* and *write* operations. Write operations modify the values of the data.

Recall that $Subd_p =< S_p, R, R_s, \mathcal{P}, \mathcal{P}^{-1}, T >$ as defined in the previous section. Let $d_{i_p} \in S_p$ be a reference (address) of a data item in the subset S_p. Then $\forall\ d_{i_p} \in S_p\ and\ T_i(d_{i_p}) = (Op_1 \circ ... \circ Op_l)(d_i p)$ an operation $Op_l(d_{i_p})$ will be executed according to the algorithm presented in Figure 7.3.

The algorithm in Figure 7.3 ensures that all the updates of the remote data that are in relation with the local data are reflected in the local $Subd_p$. For the values that are not directly related to remote data, the computation proceeds independently in each address space.

The *Dest* pattern keeps track of the local data items that are destinations of a relation from a remote data item. The *Source* pattern keeps track of the local data items that are the origin of a relation with a remote data item. Generating communication

```
$Dest = \emptyset$
$Source = \emptyset$
$\forall\ d_{i_p}\ s.t.\ R(d_{i_p})\ \notin\ S_p$
 Construct $R_s(d_{i_p}) = (\mathcal{P} \circ R \circ \mathcal{P}^{-1})(d_{i_p})$
 Add $\{R_s(d_{i_p})\}$ $to$ $Dest$ pattern
$\forall\ d_{i_p}\ s.t.\ R^{-1}(d_{i_p})\ \notin\ S_p$
 Construct $R_s^{-1}(d_{i_p}) = (\mathcal{P} \circ R^{-1} \circ \mathcal{P}^{-1})(d_{i_p})$
 Add $\{R_s^{-1}(d_{i_p})\}$ $to$ $Source$ communication pattern
if $Op_l(d_{i_p})$ $writes$ $d_{i_p}$ then
 Gather all the right values corresponding to the addresses
  (left values) from $Source$
 Generate communication on the $Source$ pattern
 Generate communication on the $Dest$ pattern
 For all left values (addresses) from the $Dest$
  $d_{i_k} = Address(Dest[k])$
  $v_{i_k} = Value(d_{i_k})$
  $d_{i_p} = R_s^{-1}(d_{i_k})$
  $v_{i_p} = Op_l(v_{i_k})$
```

Figure 7.3: A distributed algorithm for data consistency.

on the *Source* pattern means that the changes to a local data item are reflected to the dependent, remote data item. Conversely, communication on the *Dest* pattern is generated to reflect the changes of the destination.

This is a relaxed version of the owner-computes rule. That is, since updates to remote related data can affect the locally owned data, consistency is required. With the strict owner-computes rule, the updates can affect only the locally owned data and remote data is read-only. Thus, with the strict owner-computes rule the consistency problem does not occur. With our approach, the *data relation communication patterns* keep track of the related data. An implementation typically replicates these data and uses their values to reflect the changes from remote address spaces locally and vice-versa. With the strict owner-computes rule the data cannot be replicated.

7.5 Examples

This section presents practical examples of data layouts and consistency schemes for typical computations in numerical applications. Thus, regular and irregular data layouts and computations are presented. We show in each case how remote locations are identified based on local address space information and the mapping function and how data is kept consistent.

A typical irregular application is the solution of PDEs for general geometries. The numerical applications involving PDEs use as input data a discretization of the phys-

```
for t = 0 to T do
 for i = 1 to N do
  x[i] = x[i-3]
```

Figure 7.4: A typical regular computation.

ical domain they are defined on, known as *mesh* or *grid* data. Since the size of the mesh can be fairly large, these data are usually the subject of distribution / partitioning. Moreover, the solution of the PDEs, either by Finite Difference (FD) method or by Finite Element Method (FEM) consists of iterations over the data that can mostly proceed independently on partitions of data. For some data, however, consistency has to be ensured.

Typical solutions of the data partitioning problem for FD applications are the block and cyclic distributions for regular data meshes. For irregular data meshes the data partitioning problem can be reduced to a graph partitioning problem if the mesh is mapped onto a graph. Many solutions of the latter exist [33, 95].

Typical relations for the applications modeling FD solutions of PDEs are "neighbouring" relations. This means that the FD *discretization stencils* usually result in computations that combine the values from the addresses situated at a fix, small distance in a linear data representation.

7.5.1 Regular Applications

Let us consider the code excerpt in Figure 7.4. The data representation is a linear array. Thus, the data to be distributed are the array x, i.e. the set $D = \{x + i \mid i = \overline{1, N}\}$, $R(i) = i - 3$, $R^{-1}(i) = i + 3$. Then, assuming a block distribution that assigns b (block size) consecutive entries from x to a subset $p = \overline{1, P}$ (see Figure 7.5), it follows that $b = \frac{N}{P}$[4].

We define $\mathcal{P}, \mathcal{P}^{-1}, R, R^{-1}$ as following:

$$\begin{aligned}
\mathcal{P} &: D \to S_{p=\overline{0,P-1}} = \{i_p = \overline{p \times b + 1,\ (p+1) \times b}\} \\
\mathcal{P}(i) &= (p, i_p),\ with\ p = \lfloor \frac{i}{b} \rfloor\ and\ i_p = i\ -\ p \times b \qquad (7.11) \\
\mathcal{P}^{-1} &: S_{p=\overline{0,P-1}} = \{i_p = \overline{p \times b + 1,\ (p+1) \times b}\} \to D \\
\mathcal{P}^{-1}(p, i_p) &= p \times b + i_p \qquad (7.12) \\
R &: D \to D,\ R(i) = i - 3 \qquad (7.13) \\
R^{-1} &: D \to D,\ R^{-1}(i) = i + 3 \qquad (7.14)
\end{aligned}$$

Let us consider the situation in Figure 7.5, where the subsets p and $p - 1$ are shown. Assume now that Op_l is the assignment operation and we want to update the value

[4]For simplicity assume $N = P \cdot b$.

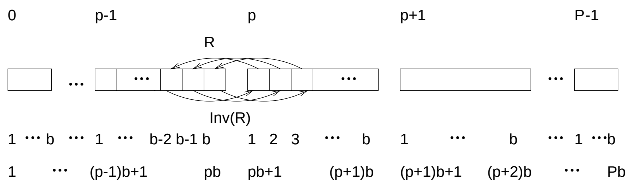

0
p-1
p
p+1
P-1
R
Inv(R)
1 ⋯ b ⋯ 1 ⋯ b-2 b-1 b 1 2 3 ⋯ b 1 ⋯ b ⋯ 1 ⋯b
1 ⋯ (p-1)b+1 pb pb+1 (p+1)b (p+1)b+1 (p+2)b ⋯ Pb

Figure 7.5: The regular partitioning of the *x* array.

$x[1]$ on processor p. Since $R(1) = 1 - 3 = -2 \notin S_p$, then on processor p, the data consistency algorithm will proceed as follows: Construct $R(i_p)$ becomes:

$$\begin{aligned}
R_s(i_p = 1) &= (\mathcal{P} \circ R \circ \mathcal{P}^{-1})(1) \\
&= (\mathcal{P} \circ R)(p \times b + 1),\ (cf.\ 7.12) \\
&= \mathcal{P}(p \times b - 2),\ (cf.\ 7.13) \\
&= (k, i_k),\ (cf.\ 7.11) \\
k &= \lfloor \frac{p \times b - 2}{b} \rfloor = \lfloor p - \frac{2}{b} \rfloor = p - 1 \\
i_k &= p \times b - 2 - (p-1) \times b = b - 2
\end{aligned}$$

Then in the *Dest* pattern the tuple $\{p, i_p, k, i_k\}$ with the values as above will be added.

Now, conversely, on processor $p-1$, when the (write) assignment operation Op_l is applied to $x[b-2]$, then $R^{-1}(i_{p-1}) = b - 2 + 3 = b + 1 \notin S_{p-1}$ is detected. Construct $R^{-1}(i_{p-1}$ becomes:

$$\begin{aligned}
R_s^{-1}(i_{p-1} = b-2) &= (\mathcal{P} \circ R^{-1} \circ \mathcal{P}^{-1})(b-2) \\
&= (\mathcal{P} \circ R^{-1})(p \times b - 2),\ (cf.\ 7.12) \\
&= \mathcal{P}(p \times b + 1),\ (cf.\ 7.14) \\
&= (k, i_k),\ (cf.\ 7.11) \\
k &= \lfloor \frac{p \times b + 1}{b} \rfloor = \lfloor p + \frac{1}{b} \rfloor = p \\
i_k &= p \times b + 1 - p \times b = 1
\end{aligned}$$

The *Source* set contains the tuple $\{p-1, i_{p-1}, p, 1\}$ with the values as above. Then, communication will be generated first on the *Source* pattern in a *sender initiated* protocol. Then, communication will be generated on the *Dest* pattern in a *receiver initiated* fashion. The distributed consistency algorithm executes symmetrically, meaning that each processor executes the same sequence of steps. Then, the update on p will be completed in the following manner:

$$\begin{aligned}
d_{i_{p-1}} &= b - 2 \\
v_{i_{p-1}} &= x_{p-1}[b-2] \\
d_{i_p} &= R_s^{-1}(b-2) = 1 \\
v_{i_p} &= Op_l(v_{i_{p-1}})
\end{aligned}$$

```
for (e = 0; e < nelems; e++) {
 vol = Vol[e];
 for (j = 0; j < NVE; j++) {
  jnode = El2MeshNo(e, j, NVE);
  p = P[jnode];
  for (k = 0; k < NVE - 1; k++) {
   knode = El2MeshNo(e, (j+k+1)%NVE, NVE);
   c1jk = ScalarProduct(a[j], a[k])/9*vol;
   B[knode] += c1jk*p;
  }
 }
}
```

Figure 7.6: A sequential irregular computation.

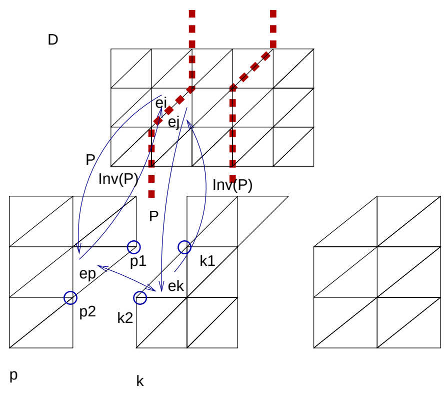


Figure 7.7: The irregular data layout.

This would lead to $x_p[1] = x_{p-1}[b-2]$. The example above shows how the regular case can be handled by our algorithm. We have implemented the algorithm for irregular data sets with general (irregular) distribution. We briefly describe our algorithm in the next section together with the experimental results.

7.5.2 Irregular Applications

This section shows how the consistency algorithm applies for irregular applications. The code excerpt in Figure 7.6 shows a typical irregular computation from the FEM iterative Poisson solver for three dimensional tetrahedra meshes. The computation is irregular because of the *indirections* it introduces (such as *knode* and *jnode*).

Figure 7.7 shows the irregular data layout. Let *B* be a data structure mapped as in Figure 7.7. That is, *B* is indexed with the vertices of the mesh. Achieving implicit parallelism means that the value of B can be updated by the same algorithm / computation replicated over a number of processors P and restricted to the data residing on each processor S_p.

In Figure 7.7 the vertices surrounded by circles mark the split in the data. These vertices denote a unique physical point $((x, y, z))$. However, the replicated computation from Figure 7.6 will attempt to compute different values (in line 9 of the code excerpt).

Analytically, the value of B at a vertex node adds the contributions from all the nodes connected to it within an element. By splitting the elements between subdomains, multiple copies of the same physical vertex reside on multiple data subsets. Therefore, the update operation Op_l, in this case $+=$ in step 9, must be executed according to our distributed consistency algorithm in order to ensure correct results.

In this case the relation between data is the *edge relation*. This means that two vertices belonging to the same element are connected by an edge. We have used the following solutions for the data partitioning and mapping:

$$\begin{array}{rcl} D & = & \{e_i \mid i = \overline{1, N}\}, R(e_i) = R^{-1}(e_i) = e_j, j = \overline{1, N} \\ \mathcal{P} & : & D \to S_{p=\overline{0,P}} = \{e_p\},\ \mathcal{P}(e_i) = (p, e_p) \\ \mathcal{P}^{-1} & : & S_{p=\overline{0,P}} = \{e_p\}, \to D,\ \mathcal{P}(p, e_p) = e_i \end{array}$$

Let e_i, e_j be two elements such that $R(e_i) = e_j = R^{-1}(e_i)$ and $R(e_j) = e_i = R^{-1}(e_j)$. This type of relation says that the *edge relation* is not *directed*. Let $\mathcal{P}(e_i) = e_p \in S_p$ and $\mathcal{P}(e_j) = e_k \in S_k$. Let i_{p_1}, i_{k_1} denote the *physically identical* nodes that belong to two different elements(e_p, e_k), that are related (through e_i, e_j). The same for i_{p_2}, i_{k_2}. Then the operation in step 9 of the code excerpt in Figure 7.6 will be executed as follows, on processor p:

$$\begin{array}{rcl} i_{p_1}, i_{p_2} & \in & e_p,\ R(e_p) = e_k \notin S_p \\ R_s(e_p) & = & (\mathcal{P} \circ R \circ \mathcal{P}^{-1})(e_p) = (\mathcal{P} \circ R)(e_i) \\ & = & \mathcal{P}(e_j) = (k, e_k) \\ Add\ \{(k, i_k)\} & to & the\ Dest\ pattern \\ R^{-1}(e_p) & = & e_k \notin S_p \\ R_s^{-1}(e_p) & = & (\mathcal{P} \circ R^{-1} \circ \mathcal{P}^{-1})(e_k) = (\mathcal{P} \circ R^{-1})(e_j) \\ & = & \mathcal{P}(e_i) = (p, e_p) \\ Add\ \{(p, e_p)\} & to & the\ Source\ pattern \\ Op_l(i_{p_1}, arg) & : & \\ d_{i_{k_1}} & = & i_{k_1} \\ v_{i_{k_1}} & = & B[i_{k_1}] \\ d_{i_{p_1}} & = & R_s^{-1}(d_{i_k}) = i_{p_1} \\ v_{i_{p_1}} & = & Op_l(v_{i_{k_1}}, arg) = (+=)(B[i_{k_1}], arg) \end{array}$$

In this case, because of the non-directed, or perfectly symmetric relation, the *Source* and *Dest* patterns are symmetric as well. This means that $Dest_p[k] = Source_k[p]$.

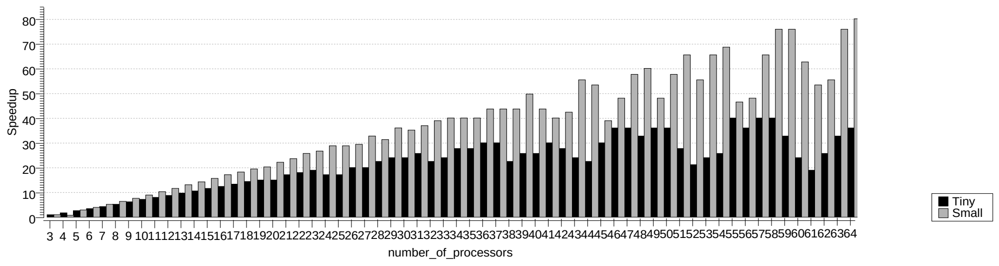


Figure 7.8: The speedup of the parallel Poisson problem for different-sized data.

We have implemented both the serial and concurrent versions of the Poisson solver. By using our system, parallelizing the serial version requires very few modifications. We will present experimental results regarding the *scalability* and *efficiency* of the parallelization in the following section.

7.6 Experimental Results

We have used the partitioning and consistency schemes presented in this chapter to implement a parallel version to the solution of the Poisson equation using a three dimensional tetrahedra mesh for a FEM discretization. The parallel version was obtained by making a few modifications to the serial one and using our prototype system for automatic data layout and consistency. The tests were run on the SGI Origin 3800 supercomputer.

As input data we have used the two meshes (described in Chapter 6) of different sizes. Figure 7.8 shows the speedup for the parallel Poisson up to 64 numbers of processors for the two different data sets. The shape of the speedup line shows that there is a threshold number of processors above which the application does not scale well. In Figure 7.8 this is visible for the smaller mesh. The explanation is that above the threshold number of processors the cost of communication becomes significant compared to the amount of computation for small data partitions. This is confirmed by the results in Figure 7.9 that show the speedup for the application using the smaller mesh as input and up to 32 processors. Also, in Figure 7.8 it is visible that speedup for the larger mesh can be achieved up to 64 processors. That is because the ratio between computation and communication is larger.

7.7 Related Work

This section presents other approaches to automatic data layout and consistency. We first review existing approaches to data layout for data parallel applications and point out differences in our approach. We then review the existing approaches to data consistency and contrast our approach against them.

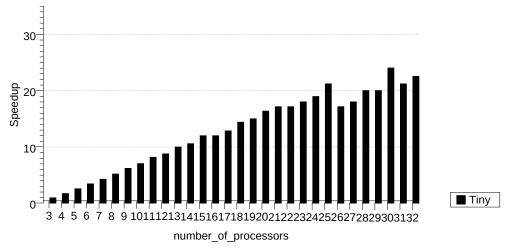


Figure 7.9: The speedup of the parallel Poisson problem for the smaller set of data.

Data Layout. Several data mapping schemes exist for Fortran programs [18, 36, 52, 57, 58, 68, 104, 110, 115]. All these approaches have two aspects in common. One is that they use Fortran-like languages and thus partition multi-dimensional arrays. The other is the regular partitioning function: block, cyclic and block-cyclic. Our partitioning model is not tied to a particular data representation. However, it assumes a generic relation between data items. Also, we use a general partitioning function that subsumes the regular partitioning strategies.

Some of these approaches use cost estimation functions and the minimum cost as a partitioning objective. Many of these strategies perform mapping in two steps: alignment of data structures in order to resemble the data relation on distribution and data layout based on problem size, number of processors and cost estimation. The approaches mainly differ in the cost estimation strategies. However, the cost estimations, static or dynamic, exploit the array data representation and use the regular partitioning functions to model the communication cost. Moreover, the automatic layout strategies are based on the strict owner-computes rule. We use a relaxed variant of the owner-computes. Then, we use the consistency protocol to ensure correct values for replicated data. With the strict owner-computes, no consistency is needed since there is no replicated data.

Our model is abstract and thus not tied with a particular language or execution. However, we use graph partitioning algorithms from the Metis family[5] to partition the data address space between different processors to minimize communication.

Data Consistency. The data consistency problem occurs when data is spread across multiple address spaces in a distributed-memory architecture. The existing techniques for maintaining data consistency across multiple address spaces range from hardware [72], to automated software data consistency, to manually ensured (application level) data consistency [8]. Automatically ensuring data consistency, by hardware or software solutions is often related to simulating a shared-memory on top of a distributed-memory architecture [16, 54, 72, 94]. Ensuring a consistent view of the shared-memory built on top of a physically distributed memory usually requires replication of physical pages or data objects and invalidation or updating of remote copies of data that is locally written. The hardware coherence schemes are triggered on read and write operations and transfer entire blocks of data. Software shared virtual

[5]http://www-users.cs.umn.edu/ karypis/metis/

memory consistency schemes are cheaper, but the performance of many applications is lower than with the hardware coherence systems.

Our approach is to maintain consistency at run-time only in specific circumstances. That is, only for data that are related across different address spaces. Also, data objects are transferred rather than physical pages. We use the data access patterns when generating communication for automatically ensured data consistency. The data access patterns are a run-time issue and are mostly determined by the application characteristics. Most of the approaches to automatic (implicit) parallelism discover the access patterns at compile-time [5]. Work in automatic parallelization uses loop-level, or instruction-level parallelism [93]. A static data distribution based on a given loop structure may result in a poor, inefficient behaviour for a subsequent loop structure that uses the same arrays, but in a different access pattern (e.g. reverse order). Our approach is to apply a global data layout scheme at the entire application level, based on the application characteristics.

7.8 Summary

This chapter has explored the issues of general data layout and consistency for SPMD programs. We have proposed a framework for data parallel applications. The main purpose of this framework is to enable a uniform treatment for a larger category than regular applications. We have shown examples of how the general formulation reduces to particular layout schemes (regular or irregular).

We have presented a data consistency algorithm based on the general data layout. The algorithm is distributed and uses the distributed information on data partitioning to locate data across address spaces. The algorithm uses the relations across address spaces to propagate changes to related data in a point-to-point message passing protocol. The run-time system generates consistency on data updates. The algorithm is general and it applies to both regular and irregular data parallel applications.

We have presented results on the scalability of our approach to data layout and consistency for a 3D FEM discretization of the Poisson problem for general geometries. These applications typically require communication for the grid points shared by elements in different processors and thus, neighbour communication. The results show that our approach is scalable for applications with loose communication requirements. Thus, the results support our hypothesis on effectiveness ($H1$ in Chapter 6).

This chapter has presented an abstract data layout and consistency model. An extension of this work is to evaluate the memory and communication requirements in the context of a concrete programming model. We use objects to encapsulate data partitions and fold synchronization into data accesses. In the next chapter we address efficiency concerns such as space requirements for maintaining the relations across different address spaces, communication generation on read / write data dependences and communication aggregation.

Chapter 8

Effective Data Consistency

8.1 Introduction

A data layout for data parallel applications specifies how a set of data is partitioned into subsets and how these are mapped onto processors. In a distributed system where the cost of communication is high, one of the partitioning objectives is to minimize communication between address spaces and thus preserve data locality. The high cost of communication also indicates that the distributed programming model is not suitable to express fine-grain parallelism or communication-intensive applications.

This chapter investigates the effectiveness of the distributed data consistency scheme for coarse-grain data parallel applications using the general data layout presented in Chapter 7. The general data layout maps a global data reference to a unique local address using a bijective function that is discretely specified (element wise), instead of a symbolic expression (as in the block or cyclic partitioning). One problem is to capture the spatial locality in a non-standard data representation. We assume that the typical non-standard data representations in data parallel applications such as sparse matrix formats, or general geometries can be mapped onto a general graph that reflects the relation between data references (e.g. connectivity information). Given a general data layout, and the same computation on each data partition, the remaining problem is to ensure that the result of a computation in the presence of partitioning is the same as the result of the computation at the level of the entire data space. This chapter addresses this problem, in the absence of a global shared memory view.

This chapter makes the following contributions:

- **A distributed data consistency scheme.** We present a *loose* consistency scheme, which folds synchronization into data accesses. One data partition is accessible to a user who expresses his / her algorithm as a set of transformations (operations) on the available (local) data. Then, consistency is ensured as if the user has applied his / her algorithm (set of transformations) on the entire data and not a single partition (a SPMD style, where the user is not exposed to concurrency).

- **Efficiency aspects of the consistency scheme.** We address three efficiency aspects related to the consistency scheme: the amount of information maintained locally at each address space level, the cost of locating an indirectly remote data

```
for (i = 0; i < NElems; i++) {
 for (j = 0; j < NVE; j++) {
  jj = Local2Global(i, j);
  a[jj] = a[jj] + c0*f[jj];
  for (k = 0; k < NVE - 1; k++) {
   kk = Local2Global(i, (j+k+1)%NVE, NVE);
   a[kk] = a[kk] + c1*f[kk];
  }
 }
}
```

Figure 8.1: A typical FEM discretization operator (assembly procedure).

access, and the cost of the consistency protocol. We show that a small amount of extra information at each address space level enables us to directly locate the typical remote accesses for data parallel applications. Moreover, the random remote accesses can be located at run-time. We show by measurements that the run-time efficiency is good for typical coarse-grain applications requiring communication for the data on the border between subdomains.

The remainder of this chapter is organized as follows. Section 8.2 introduces the context of our work and gives an overview of our approach to efficient data consistency. Section 8.3 discuses the general data layout. Section 8.4 presents the data consistency algorithm. Section 8.5 discusses the efficiency aspects of the consistency algorithm. Section 8.6 presents working examples of how the programming model we propose is different from the existing data parallel models. The section also discusses its applicability and what it takes for the user to embrace it. Section 8.7 presents experimental findings on the efficiency of our consistency algorithm for concrete applications. Section 8.8 discusses related work. We conclude and emphasize future research directions in Section 8.9.

8.2 Background

As already mentioned in the previous chapters, the solution of PDEs is based on discretizing the continuous physical domain in a regular or an irregular manner. The existing data parallel frameworks relate the parallelism in these applications to fine-grain, loop-level parallelism and linear array representations. The array representation influences the application coding, the data partitioning and mapping to processors. This approach is suitable for applications that use dense arrays or regular domains and the FD discretization scheme. Applications that use the FEM to discretize the continuous physical domain exhibit coarse-grain parallelism and a loose synchronization structure.

Figure 8.1 shows a typical computation for a FEM discretization scheme, used by most of the applications.

```
for (i = 1; i < N; i++) {
 f[i] = 0.5 * (f[i-1] + f[i+1]);
}
```

Figure 8.2: A typical FD discretization operator (stencil operator).

The existing data parallel frameworks aimed at parallelizing the iterations of a loop encounter problems for applications as in Figure 8.1 because of statements like 3 and 6. The references *jj* and *kk* can only be found at run-time. The existing run-time frameworks solve this problem [38] in the context of fine-grain parallelism, based on linear array representations and affine index expressions. Thus, they do not address the problem in statements 3 and 6. The work in [69] is an example of a unified treatment of regular, dense array computations with simple access patterns and irregular applications such as PDEs for general geometries. The regular applications are analyzed at compile-time and parallel loops are distributed across multiple processors. The irregular applications use a run-time approach based on the *inspector/executor* model. However, the approach applies in the context of multi-dimensional arrays in Fotran languages and regular data partitioning (block and cyclic). There is an important body of work on irregular applications that addresses sparse array representations [22, 105, 111].

The existing data parallel frameworks work well for numerical applications that use finite difference, or *stencil discretization* schemes (Figure 8.2). The code excerpt in Figure 8.2 shows an example of a regular code. Transformations and subsequent optimizations of regular computations for distributed-memory architectures are well understood [4, 6, 66, 67]. These do not directly apply to the type of computations such as those shown in Figure 8.1.

This chapter addresses the issue of effective distributed data consistency for irregular applications, posing problems shown in Figure 8.1. Our approach is to ensure that accesses are only to the local address space level and that the effect of a local operation is reflected to a remote data item related to a local data.

Our approach is different in many respects from the approaches that implement a global address space [9, 24, 34, 35, 48, 65, 101]. First, we distribute the data and keep a small amount of information that enables the retrieval of any data item in any address space at run-time. Moreover, instead of offering a global memory view as with a virtual shared memory, we offer a local view of one address space only. Second, we use a partitioning scheme that generalizes the existing regular and irregular partitioning schemes. The partitioning is a general function that maps global references to local references in a distributed memory. The mapping information on data partitions is distributed to allow reverse transformation from local to global references and vice versa. For all the local references which may indirectly result in remote accesses we keep local reference placeholders for the remote data values.

There are three efficiency aspects of such an approach:

- The amount of extra information maintained at each address space level about partitioning such as the indirect remote references resulting from direct local accesses can be computed statically (at compile-time) or discovered at run-time.

- The overhead incurred to discover a remote reference at run-time. For regular applications the remote references are directly (symbolically) computed and thus there is no extra communication overhead for locating a remote reference.
- The overhead of the consistency protocol to ensure that the most recent data values are available locally. This is the communication overhead incurred by the write-through (an update writes the copies) protocol.

We summarize our approach to addressing these efficiency issues as follows:

- To address the first efficiency concern, we only record information on the data *cut* when partitioning the data. Thus, we map two *neighbouring* partitions onto *neighbouring* physical processors. In our approach the *neighbouring* relation reflects the algorithmic relation between data items, based on the application knowledge, and not based on the physical data locality. In the next section we will show how we express the relation between data items at the application level.
- To address the second efficiency concern, we observe for the applications we address that the remote references typically fall on the border between partitions. Thus, these coincide with the information we keep about partitioning and can be directly found. Remote references that result in a randomly distant access (unknown locally) at run-time can be located by exchanging at most $P-1$ messages, where P is the number of processors in the system.
- To address the third efficiency concern, we employ an update consistency scheme in which the owner of a data item sends its value on update to all the processors that need it. We use a data placeholder to keep the last updated remote data value and eliminate communication on read. The assembly procedure for the FEM method consists of loop iterations which combine data values according to update operations (e.g. the arithmetic addition) that are closed under associativity and commutativity. The effect of such operations can be postponed outside the loop. Therefore, we aggregate the resulting communication and move it outside of the loop.

8.3 The Data Layout

This section puts in perspective the data layout formulations from Chapter 7. Thus, it uses the application perspective to show how the general data layout formulations generalize the existing approaches to distributing data across multiple address spaces, that is, it accounts for both regular (standard array-based) and irregular (non-standard) data layouts. Our observation is that computations like those in Figures 8.1 and 8.2 usually express a discretization scheme, and have the same substrate. The discretization scheme involves the computation of a value at a spatial point, based on the *neighbour* information or the *related* data items.

```
for (i = 0; i < N; i++) {
 for (j = R(i)) {
  a[j] = a[j] + ...;
  for (k = R(i) && k != j) {
   a[k] = a[k] + ...;
   f[i] = f[j] + f[k];
  }
 }
}
```

Figure 8.3: Discrete computations.

Let us consider the code fragment in Figure 8.3. We use a generic relation to express the reference j with respect to i as $j = R(i)$. The relation imposes an ordering for the traversal of a generic data set consisting of data items and the relation between them. We consider the data partitioning based on the relation R governing the data layout (and reflected by an implementation). Thus, both the computations from Figures 8.1 and 8.2 can be expressed in a similar way, as shown in Figure 8.3. The relation R may describe the logical, physical and algorithmic grouping of data. Therefore, we partition the data based on it, to minimize the number of relations that cross partitions boundaries. We use general graph partitioning algorithms to express the general partitioning function. Thus, we first map the global reference space and the relations between data onto a graph and then we use a graph partitioning algorithm that minimizes the *cut* [95][1]. The general partitioning formulation allows a data layout to use any symbolic or algorithmic expression to specify a partitioning function. Moreover, it can use any graph partitioning algorithm to experiment with various decomposition strategies.

The consistency model is based on this general approach to data layout that enables construction of the communication patterns and the possibility to make data partitioning and mapping implicit. Also, this general treatment allows for flexibility in selecting suitable partitioning functions and data relations to express computations in a manner that is close to sequential.

Having captured the generic relation R that governs a data set, the discretization code expressing a traversal of the data set results in accesses that are *one relation away*[2] from one another. Therefore, the only communication required after splitting the data set into partitions is due to the references where the relation was *cut*.

The data layout is illustrated in Figure 8.4, where an edge between two data items (vertices) denotes a relation (we do not consider the direction, since we require the relation to be symmetric in a multidimensional space). Then, if the partitioning function invalidates a relation, we register this in the *data relation patterns*, depicted in Figure 8.4 as *communication patterns*. This information is used to reach a data item by following

[1]We are aware of work in graph partitioning algorithms that considers more complex criteria for load balance, such as [33]

[2]In some stencils, involving more neighbours in each directions, this is also captured by one relation.

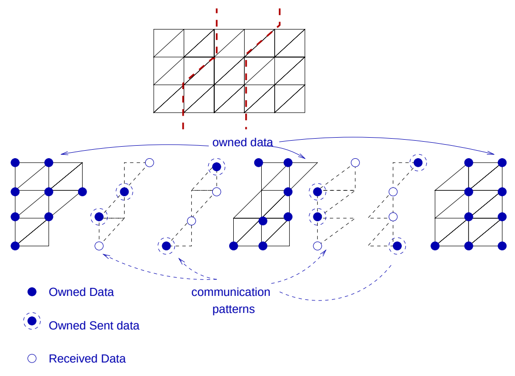


Figure 8.4: The Data Layout.

the relation in a traversal over the data set. We use the *minimum number of invalidated relations* between non-overlapping partitions as partitioning objective. The invalidated relations become the solution to the transformation problem (presented in Chapter 7). This is the extra information that each processor keeps in order to be able to locate remote references, when needed. Thus, the amount of extra information at each address space level is small.

One data partition corresponds to one address space in a distributed memory setting. Thus, we distinguish between *locally owned data* and *related data*. We sometimes refer to these also as *independent* and *dependent* data. A data access in a traversal of data based on the ordering relation R can be located in any one address space based on the *data relation patterns* information. The data relation patterns can be viewed as the communication patterns: they can serve to locate any one data in any one address space. That is, generally, if a local data access results in a remote reference, the processor first makes the request for data at the neighbouring processors. If these do not own the data or do not have a valid copy of it locally, they pass the request to their neighbours, and so on.

Most of the data accesses for the applications we have described so far follow the relation in a traversal and can be found directly (from the information about the neighbouring partitions). The accesses that cannot be found directly are located according to the procedure which we have described in the preceding paragraph. However, most of the accesses typically fall on the border region between partitions and thus it is sufficient to look at the neighbouring processors to fetch the last updated values. The experiments conducted with this class of applications show promising results. We did not experiment with applications for which accesses in one address space may require data at randomly distant locations. We can locate these data at run-time, but it may incur large overheads. Thus, an extension of this work is to employ data remapping strategies for different communication patterns.

According to our model, the data to be distributed is a bounded set consisting of data items and a relation between these data: $D = \{d_i \mid i = \overline{1, N}\}$, $R : D \rightarrow$

$D,\ \forall\ d_i\ s.t.\ \exists\ d_j,\ j = \overline{1,N},\ R(d_i)\ =\ d_j, \exists R^{-1} : D \rightarrow D\ s.t.\ R^{-1}(d_j)\ = d_i$[3]

The data partitioning function[4] $\mathcal{P}$ distributes D in P disjoint sets S_p:

$$\begin{aligned} \mathcal{P} : D \rightarrow S_p,\ p = \overline{1,P}, S_p &= \{d_{i_p} | i_p = \overline{0, |S_p|}\} \\ \forall\ d_i \in D, \mathcal{P}(d_i) &= d_{i_p} \in S_p, \\ \exists\ \mathcal{P}^{-1} : S_p \rightarrow D, \mathcal{P}^{-1}(d_{i_p}) &= d_i, \end{aligned} \tag{8.1}$$

After partitioning we use the following solution to the transformation problem (between local and global data spaces) to construct the relations invalidated during partitioning:

$$\begin{aligned} \forall d_{i_p} \in S_p,\ s.t.\ R(d_{i_p}) &\notin S_p, \\ let\ R_s(d_{i_p}) &= (\mathcal{P} \circ R \circ \mathcal{P}^{-1})(d_{i_p}) \\ &= (k, d_{i_k}), k \neq p \\ \forall d_{i_p} \in S_p,\ s.t.\ R^{-1}(d_{i_p}) &\notin S_p, \\ let\ R_s^{-1}(d_{i_p}) &= (\mathcal{P} \circ R^{-1} \circ \mathcal{P}^{-1})(d_{i_p}) \\ &= (l, d_{i_l}), l \neq p \end{aligned} \tag{8.2}$$

We have given the details of the formulations and their rational in the previous chapter. Here we only show the general layout that enables our efficient approach to data consistency.

8.4 The Data Consistency Algorithm

As explained in Chapter 7, the data consistency algorithm executes in a distributed manner, based on one address space knowledge only. Thus, if $T :< D, V > \rightarrow < D, V >$, with T a *a generic transformation* and V a set of values, is a transformation on the entire data, then $T :< S_p, V_p >, \rightarrow < S_p, V_p >, p = \overline{1,P}$ reflects the transformation locally, on a subset S_p, producing the values V_p (results). In the SPMD model a transformation is the entire program and the same transformation executes in each address space (on each data partition).

To ensure consistency, the algorithm uses the local knowledge expressed by the tuple $< S_p, R, R_s, \mathcal{P}, \mathcal{P}^{-1}, T >$, where S_p is the local data subset after partitioning the data governed by the relation R according to the bijective partitioning function $\mathcal{P}$. Note that we have already shown how we reconstruct the relation R_s to be correctly defined at each subset level (according to the equations 8.2).

The algorithm in Figure 8.5 summarizes the data consistency protocol. We present the algorithm in the context of an object model. We use objects to fold synchronization into data accesses. We use the *C*++ [103] programming language as the context for our

[3]Data relations are multidimensional. For readability reasons, we do not complicate our notation with vectored data.

[4]We use the calligraphic letter $\mathcal{P}$ to denote the partitioning function and the regular letter P to denote the number of partitions.

discussion. This chapter does not focus on the object model. Thus, we only give the preliminaries for discussing the consistency scheme in the remainder of this section. The data is encapsulated in distributed objects according to the data layout presented in the previous section. We ensure synchronization at the method execution level. That is, an object is accessed only through its methods. All the accesses to an object are either *read* (do not modify the state of an object) or *write* (update the state of an object). We ensure consistency on write accesses and eliminate communication on read. However, not all write accesses need to be synchronized. The algorithm synchronizes (through an update consistency scheme) only the methods that have read / write dependences to distributed objects.

The first step of the algorithm shows the computation of the *Send* and *Recv* sets which maintain links between independent address spaces. The owned data is the data in the S_p set. Each access is only to the locally owned data. If a local access has a direct relation to a remote data, as indicated through the invalidated relation, local write accesses may need to be propagated to the remote location. The remote location is found from the data layout information using the relations (8.1) and (8.2). Thus, each processor keeps locally a *Send* list of the local references that are remotely related. Conversely, if a local access is pointed to by a relation from remote (this means that the inverse relation of the local reference points remotely), then the value may be needed locally. The remote location is also found from the data layout information using the relations (8.1) and (8.2). Thus, the processor keeps locally a *Recv* list of remote references that are locally related.

The second step of the algorithm marks all the methods which contain *write* accesses to distributed data objects indexed by the references in D that are followed by *read* accesses to the same data object. This information is obtained from data dependence information. This phase detects the read / write dependences for distributed data objects indexed by the references in D, but does not say which accesses in S_p (references) are related. Control flow information provides dependence information for data accesses that are known at compile time (e.g. accesses that are affine expressions of loop indices). The references that cannot be known at compile time are found at run-time.

The third step of the algorithm ensures data consistency at the execution time. The consistency procedure takes into account that changes to local data must be reflected globally, according to the relation between data references. For example, in a stencil discretization scheme as in Figure 8.2, data at two consecutive locations are related. Thus, for the data on the borders between subdomains, the changes in local data are reflected in the neighbouring subdomains by adding the local contribution to the directly related data items (from *Send*). Also, changes in data from the remote subdomains are reflected locally at data locations (references) which are pointed from outside (this is given by the inverse relation information – from *Recv*).

All modifications are performed by each processor on temporary copies, and only when all the processors have finished the update phase, the changes will be committed (it can be that a value from an address is involved in multiple transfers and data races could occur due to different processing speeds).

```
1.  Construct the communication patterns:
 $\forall d_{i_p} \in S_p$ $such$ $that R(d_{i_p}) \notin S_p$
  $R_s(d_{i_p}) = (\mathcal{P} \circ R \circ \mathcal{P}^{-1})(d_i p) = (k, d_{i_k})$
  $Send_p[k] < -(d_{i_k})$
 $\forall d_{i_p} \in S_p$ $such$ $that R^{-1}(d_{i_p}) \notin S_p$
  $R_s^{-1}(d_{i_p}) = (\mathcal{P} \circ R^{-1} \circ \mathcal{P}^{-1})(d_i p) = (l, d_{i_l})$
  $Recv_p[l] < -(d_{i_l})$
2.  Mark the write methods:
 for each method
  for each basic block
   if (∃ statements S1, S2 involving distributed data dd1 and dd2 such as)
    $S1 : dd2 = ...$
    $S2 : dd1 = Op_l(dd2, ...)$
   then
    Mark write
3.  Write Method invocation:
 Entry:
  NOP
 Execute method body locally on temporary copies:
  for each basic block
   if (∃ statements S1, S2 involving distributed data dd1 and dd2 such as)
    $S1 : dd2 = ...$
    $S2 : dd1 = Op_l(dd2, ...)$
   then
    $\forall d_{i_p} \in Send_p[k]$
     $Send(dd2(d_{i_p}), k)$
    $\forall d_{i_k} \in Recv_p[k]$
     $Receive(dd2(d_{i_k}), k)$
    //local contribution:  locate and apply
    $d_i = R_s^{-1}(d_{i_k})$
    $dd1(d_i) = Op_l(dd2(d_{i_k}), ...)$
   Synchronization phase:
    Wait until everybody finished executing the block;
    Update dd
 Exit:
  NOP
```

Figure 8.5: A distributed consistency algorithm.

8.5 Efficiency Aspects

Existing data parallel frameworks use regular block or cyclic partitioning. The partitioning is thus specified by symbolic expressions which can be used in conjunction with symbolic affine expressions of array references to derive information on communication (the communication parties and data), without the need to maintain extra information. When the array accesses are not known at compile time, it is hard to find a data layout that reflects data locality.

The general partitioning technique for applications that use multidimensional arrays assigns array entries to processors by means of some function, e.g. a random-number generator in [107]. In this case the entire mapping information is maintained at each address space level. Some approaches to implicit data consistency across address spaces fully replicate the data in each address space [55].

The data layout which we use shows that a general partitioning can be controlled by a general graph partitioning algorithm instead of a symbolic expression or random number generator. The algorithm takes as input the data space mapped onto a graph that reflects the traversal relation between data and produces a load-balanced partitioning that reduces the number of remote relations between partitions. Each processor keeps only the locally owned data and links to the neighbouring processors.

This means that without using a symbolic expression to compute a data reference and using a small amount of information, we can directly know the references to the neighbouring processors and dynamically find any one random remote reference. Knowing the references to the neighbouring processors is efficient for all the FEM irregular applications and FD stencil codes. We do not claim that is efficient to find any one random reference at run-time. If the data traversal maps to a complete graph of N vertices mapped to N processors, then a table of $N-1$ (each vertex has $N-1$ neighbours) entries is needed at each address space level. This scenario is not realistic. That is, a data traversal cannot result in a completely connected graph. Additionally, it is not realistic to partition a tightly connected problem of size N onto N processors.

Other data parallel frameworks that address irregular applications use distributed schedules to keep information on irregularly distributed data [38]. In [38] the map of the global array data is block-distributed between processors and the location of an array element is symbolically computed. The accesses through an indirection array cannot be found until run-time and thus a regular partitioning cannot account for data locality. To improve the performance of a naive translation table, the authors introduce *paged* translation. However, the paged translation also binds a static number of pages to each processor and use a *replication factor* heuristic (set by the compiler writer) to dynamically replicate additional pages.

In [28] the authors propose support for automatic hybrid (regular and irregular) applications. The multidimensional array data is also distributed in a block / cyclic manner and the array references that are not symbolic affine expressions are found at run-time. The authors use extra information called *Available Section Descriptor* to represent the presence of valid data in a processor. The ASD keeps information on the array identifier, a vector of subscript values, and an array tuple that specifies for each array dimension the processor grid dimension that is mapped onto and the mapping of the corresponding array dimension.

8.5.1 Remote Location Overhead

The remote location overhead refers to the time needed to locate a reference on a remote address space based on the distributed mapping information. For the applications we address, the only remote accesses of interest are on the border between subdomains and thus directly found from the local information.

If P is the number of processors, maximum $P-1$ messages need to be exchanged in a communication protocol in order to locate a local access that is related to a remote item not residing on the border region between subdomains. That is, if the P independent partitions are connected in a linear chain, $\{(p_1, p_2,)(p_2, p_3), ..., (p_{P-1}, p_P)\}$, a random remote access will generate $P-1$ messages to locate the owner of a reference. This is the worst-case scenario. The best-case scenario, which corresponds to the applications we address, is that remote accesses occur only to the neighbouring processors. Thus, no extra messages are required to indirectly locate the owner of a data reference.

8.5.2 Communication Overhead

The consistency protocol will generate a send message on write for all the write statements to a distributed data object that are followed by read in subsequent statements. Our update scheme, however, discriminates between the locations that are related and will generate messages only for those. For the applications that typically relate data on the border between subdomains, the maximum number of generated messages on one processor is the sum of the sizes of *Send* and *Recv* sets. These sets are guaranteed to be minimum by the graph partitioning algorithm.

For loops such as in Figure 8.1, where the array a is both read and written, one message is potentially generated for each write access to a location on the border between subdomains in the assembly procedure. In such a situation we generate a single message in which all the values on the border between subdomains are exchanged outside the loop. That is because an assembly procedure for the FEM problems typically is a traversal in which the distributed data objects are playing the role of an accumulator variable: an accumulator variable accumulates values within a loop according to an arithmetic operation closed under associativity and commutativity. For example, a typical FEM assembly procedure is a loop over the elements that adds the contribution of the neighbouring elements. Thus, when distributing the data, the values on the border of the region have to be accumulated from the remote processors as well.

Treating a distributed object as an accumulator variable eliminates communication within the loop. All the computations on update are executed on temporary copies of the replicated data and summed at the exit point of the loop.

8.6 Examples

Let us consider the typical irregular code excerpt in Figure 8.1 and modify it to resemble Figure 8.6.

We introduce the *distributed data* type in order to model data that has to be globally consistent, in this case the coefficient arrays a, b. A distributed data is a parameterized

```
for (i = 0; i < NElems; i++) {
 for (j = 0; j < NVE; j++) {
  jj = Local2Global(i, j);
  b[jj] = a[jj] + c0*f[jj];
  for (k = 0; k < NVE - 1; k++) {
   kk = Local2Global(i, (j+k+1)%NVE, NVE);
   a[kk] += c1*f[kk];
  }
 }
}
```

Figure 8.6: An irregular array computation.

type that encapsulates a set of references, the initial bounds of data, and defines the methods to access data at a specific location. We only brief the necessary aspects to explain how the data consistency protocol works. The transformed code is shown in Figure 8.7.

In the transformed code fragment in Figure 8.7 the user has to identify the distributed data. Failure of correctly identifying sequential data (by declaring it distributed) results in correct execution and no unnecessary communication, but it incurs space and time overhead. Not classifying a data object that needs consistency as distributed leads to incorrect results. In the present prototype implementation, the operations to access the distributed data (set and get) are provided by the user.

The same code is executed on each processor, using only the local view of data. This is given by a *Subdomain* component which is computed by the system and contains the local view of the global data. Thus, our system computes the loop bounds and takes care of all the involved transformations. The marked lines in Figure 8.7 are the manual annotations for ensuring consistency.

The *Update* functions are system template functions that are triggered either for the update of a single value at a given location or for the collective communication phase.

The collective communication phase generates an aggregated communication, where each processor sends one message per destination found in the *Send* communication pattern and issues one receive message for each source found in the *Recv* communication patterns.

8.7 Experimental Results

Our general approach applies to regular applications that involve stencil operations resulting in remote accesses, as well as to trivially parallel applications (e.g. image processing algorithms, computational geometry, etc.).

Another class of applications that may benefit from our work are applications that use general meshes (regular or irregular) and need communication due to their relations (e.g. *neighbour* relation). One class of such applications is the solution of the

```
DistrData<double> a, b;
Subdomain m;
...
1  for (i = 0; i < m.GetNElems(); i++) {
2   for (j = 0; j < m.GetNve(); j++) {
3    jj = Local2Global(i, j);
#    b.SetAt(jj, a.GetAt(jj)+c0*f[jj]);
#    if (int p = Found(jj, CommPatterns))
#     LUpdate(b, m, jj, p);
5    for (k = 0; k < m.GetNve() - 1; k++) {
6     kk = Local2Global(i, (j+k+1)%m.GetNve());
7     a.SetAt(kk, a.GetAt(kk) + c1*f[kk]);
#     if (int p = Found(kk, CommPatterns))
#      LUpdate(a, m, kk, p);
     }
    }
   }
```

Figure 8.7: The transformation of the irregular computation.

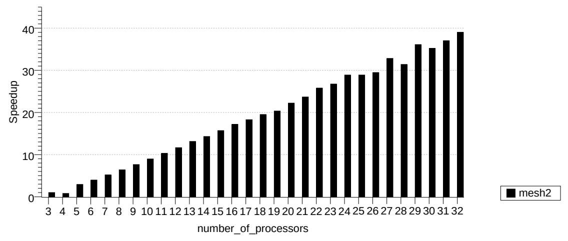


Figure 8.8: The 3-D FEM Discretization scheme using the Jacobi iteration scheme. Mesh: small.

PDEs, by either FD or FEM methods, using iterative methods for matrix vector computations. For FEM applications operating on large data, there is hardly ever the case that a matrix is explicitly constructed and manipulated. The common practice is to use implicit structures. Therefore, our model is suitable for such applications, that hardly ever use direct methods to solve a particular linear system.

The distributed consistency algorithm presented in this chapter works well for the applications that involve communication for the neighbouring processors. An extension to this work is to experiment with applications that require different communication patterns and to study the effectiveness of the consistency algorithm.

We show the results for the Poisson problem as a validating example for our approach. The FEM discretization of the Poisson problem contains all the ingredients typically found in the FEM irregular applications.

The results in Figure 8.8 show the speedup for a three dimensional FEM discretization scheme for the Poisson equation using a Jacobi iteration scheme. The results in

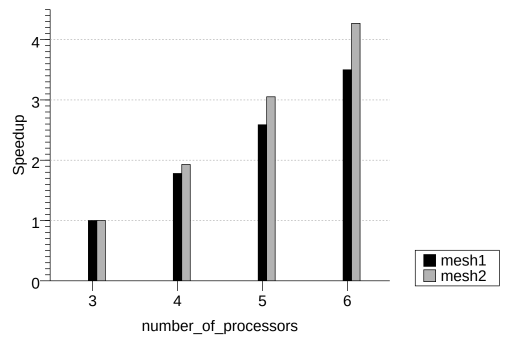


Figure 8.9: The 3-D FEM Discretization scheme using the Jacobi iteration scheme. Results on a 6 processors Beowulf cluster, for two different domain sizes.

Figure 8.8 show good scalability. This is due to the aggregation of communication. We show the speedup for the application running on SGI Origin 3800 for different mesh sizes (described in Chapter 6), using up to 32 processors.

A benefit from using the distributed-memory model is to take advantage of the commodity-oriented clusters. Thus, we are interested in experimenting the effectiveness of our model on Linux clusters, as well as on Networks of Workstations (NOWs). The results from running the Poisson solver on the Beowulf Linux cluster (Beowulf [89]) are shown in Figure 8.9.

The results are promising and show that our approach is successful in capturing the coarse-grain parallelism in scientific applications.

8.8 Related Work

The goal of our work is to make parallelism implicit by using a suitable data layout and by automatically ensuring consistency, with system support. The data layout is based on a general partitioning and a relaxed consistency scheme. Work on relaxed consistency originates in the virtual shared memory techniques [21,60,61,72,94]. One way of relaxing a strict consistency model is to ensure consistency only at synchronization points.

In a software-maintained consistency, the synchronization points can be either indicated by the user (programmer) or discovered by a compiler. The first option allows for high optimizations, but little safety (since a programmer may be wrong). The second option is less error-prone, but it can be more conservative and, as a consequence, less effective. In contrast with existing work on consistency models, the synchronization points in our approach occur in connection with read / write dependences for distributed data objects. We only ensure consistency for the related data, since independent data is consistent. Also, we can sometimes postpone synchronization even longer for certain operations (those which result in *partially correct* values for the distributed data and need just one collective communication phase to ensure global consistency).

Work on parallelizing compilers is also using (sometimes relaxed) the owner-computes rule and message passing communication in order prefetch the data read locally. Such work emphasizes fine-grain parallelism and regular data layout in the context of loop-level parallelism. Many communication optimization techniques have been proposed to improve performance for the distributed-memory machines [4–6,53,66]. These techniques work well for regular codes, or for computations where data accesses can be symbolically computed at compile time. For irregular codes work on run-time techniques for automatic parallelization considers array accesses of loop indices or indirect references as functions of the loop index, scalars that are not defined in the loop body, or arrays indexed by just the loop index (e.g.$i + 2$, $i * i + 3$, $ia(i) + i + a$) [38, 83]). The constraint on the array accesses is essential for being able to discover independent loop iterations that can execute concurrently. With our loose concurrency model all the accesses are local, and some of them may indirectly result in remote reads. Therefore, these can be found at run-time without any restriction on how a reference is computed.

An important body of work on irregular applications addresses the class of sparse matrix codes [22, 23, 45, 79, 105, 111]. These approaches make specific assumptions about the applications they address. Thus, the discretization schemes do not fit well in this context.

Finally, existing work on object models for distributed-memory architectures also focuses on the virtual shared memory concept [16,54]. While this work leaves room for flexibility by incorporating task level parallelism as well, the efficiency can be poor for the irregular, loosely synchronous applications (since the entire data can be replicated, but sometimes only parts of it kept consistent).

8.9 Summary

This chapter has presented an efficient consistency scheme based on a general data layout. Unlike a parallelizing compiler, this work does not attempt to discover parallelism. Instead, it expresses a loosely coupled parallel computing model consisting of identical computations across different address spaces. Distributed data objects in each address space correspond to partitions of a larger data object. Scalar data objects may exist at each individual space level independent of each other. Thus, with this model all computations apply concurrently on data in each address space. Synchronization is implicit and is folded into the distributed objects data accesses.

We have used a general data partitioning scheme (from Chapter 7) that subsumes existing schemes, and thus allows for a general treatment of data parallel applications. A consistency algorithm ensures loose synchronization on data accesses for data that is related across address spaces. We have shown that it is possible to locate any one data reference in a distributed address space by only keeping links between neighbouring partitions. Thus, our approach is efficient space-wise. Moreover, for many of the scientific data parallel applications each processor only needs to find out about data residing in the neighbouring processors. Thus, for these applications our approach is efficient time-wise. The loose consistency scheme generates communication only for read / write dependences for distributed data. Then, if a distributed data object behaves as an accumulator variable within a loop, communication is aggregated and moved outside the loop. This further improves the performance of our consistency scheme. These results also support our hypothesis on effectiveness, $H1$ in Chapter 6.

There are two research paths in our work complementing the approach in this chapter. One research issue is to express a general set structure that captures the relation between data. We are currently experimenting with a recursive set type that allows hierarchical data representations typically found in irregular applications (general geometries, graphs, sparse array formats). This type is defined as a template C++ class that encapsulates a bounded set of data items and defines the access to them. Hierarchical data structures are defined by allowing the template type to be a set itself. We can map an instance of a set type to a connected graph and use the approach presented in this chapter to fold synchronization into data accesses.

Another research path under investigation is to observe the behavior of the consistency algorithm for different communication patterns. The thesis of our work is that a data structure such as the set type we have described above can be linearized and restructured to reflect data locality for the recurring communication patterns in data parallel applications. Moreover, the data layout can dynamically change at run-time based on heuristic execution models.

Chapter 9

High Level Abstractions: Programmability and Effectiveness

Parts of this chapter were published as a conference paper [42].

9.1 Introduction

The integration of objects and processes into object-oriented concurrent programming is natural and useful for practitioners in fields such as telecommunications, high performance computing, banking and operating systems [82]. The current practice for scientific applications is to express concurrency orthogonal to sequential programming. The programmer annotates an application in a sequential programming language with concurrency constructs (i.e. library calls to message passing routines or shared memory directives).

This chapter investigates the issue of integrating concurrency into a distributed object model to increase the usability of distributed-memory programming for data parallel, scientific applications.

The existing distributed shared object models (that implement software virtually shared-memory on distributed-memory architectures) for distributed-memory systems pay in efficiency for the gain in expressiveness. Client-server distributed object models are suitable for applications that involve intensive data transfer. Due to the high cost associated with remote procedure calls, they are inefficient for intensive data parallel computations (scientific applications). The client-server programming model is asymmetric, and is thus limited in expressing symmetric, peer data parallel applications.

This chapter proposes an efficient object model that enables the exploitation of coarse-grain parallelism. The model uses high level abstractions to increase programmability for concurrent scientific applications and integrates the aspects of data partitioning and consistency at a lower level discussed in the previous chapters. Thus, the data partition and communication aspects are implicit in the object model. We distinguish between distributed and sequential objects. We call the former active objects and the latter passive objects. The former model non-trivial data parallel computations that require synchronization due to data dependences. The latter model trivial data parallel computations. The distributed objects fold synchronization into data accesses. The computation is replicated across address spaces in the SPMD style.

We demonstrate the use of the object model in our prototype system implementation for data parallel irregular applications. The system supports the creation and the coordination of concurrency, as well as it automatically ensures data consistency, transparent for the user. This approach ensures correctness by concealing the error-prone concurrency and distribution aspects.

This chapter makes the following contributions:

- **A high level programming model for data parallel, irregular applications**. The model is based on the notion of distributed objects. These objects are active, as opposed to sequential, passive objects[1]. Distributed objects model data that needs to move across address spaces in a communication protocol. This ensures consistency, transparent for the user. Sequential objects model data that is relevant for a single address space. Moreover, distributed objects model BSP computation kernels [106]. In the BSP model computation steps alternate with synchronization phases. The distributed objects synchronize loosely to exchange data between processing nodes. The sequential objects express “ideal” SPMD computation kernels, which are independent and do not require global consistency, and thus, communication.

- **An evaluation of the object model**. We use our prototype implementation of an object-oriented framework([47], [62]) to illustrate how the object model can support concurrency for the solution of PDEs on irregular meshes (described in Chapter 11). We evaluate the resulting programming model versus other approaches to distributed programming for scientific applications. We use two perspectives for our evaluation. *Usability* measures the user effort involved in writing a distributed-memory scientific application. *Efficiency* involves the scalability, and the speedup of an application implemented using the approaches compared. We present evaluation data from a didactic experiment we have conducted as a part of a student project.

The remainder of this chapter is organized as follows: Section 9.2 gives an overview of the transparent concurrency support issues, together with their realization in the object model we propose. Section 9.3 describes in detail a distributed object model to support transparent concurrency and the enabling techniques. Section 9.4 discusses the realization of the object model as a prototype object-oriented framework. Furthermore, it illustrates the resulting programming model in the prototype system and shows a concrete example from a typical data parallel application. Section 9.5 presents the evaluation of the object model, together with preliminary results. Section 9.6 reviews the existing approaches to system support for concurrency. Section 9.7 concludes the chapter.

[1]The notions of “active” and “passive” do not correspond exactly to their normal usage in parallel object-oriented programming. By active objects we mean objects that know how and where to communicate themselves (i.e. pack, unpack). By passive objects we mean objects that do not take the existence of multiple address spaces into account.

9.2 Background

Many scientific applications share a similar parallel structure. The parallel structure is coarse-grain, and set up manually. The programmer decomposes and maps data and computations to processors in one “fork phase”, at the beginning of the application. The execution of the application consists of independent *heavy-weight processes* executing in different address spaces. The programmer takes care of data consistency between different address spaces by means of calls to message passing routines (e.g. Message Passing Interface - MPI [51, 99]). The independent computations merge at the end of the application. Thus, the programmer collects the final results in a “join phase” before the application ends. This parallel structure is different from the fine-grain, fork / join loop level parallelism. In such a model the sequential computation forks into multiple threads, i.e. *light-weight processes*, at the beginning of a parallel block (loop). These threads join back into the sequential thread at the end of the parallel block.

The object model we present captures the coarse-grain parallelism model rather than the fine-grain or HPF concurrency style. We describe a distributed object model for transparent concurrency that addresses the following issues:

1. Data model.
2. Data distribution.
3. Data consistency.
4. Computation model.

The data model consists of distributed and sequential objects. The distinction between these is visible to the user who indicates the distributed objects upon declaration. The system treats any data that is not declared distributed as sequential. The distributed active objects maintain information about data location and data dependences across address spaces. They have communication capabilities, including serialization (packing) and deserialization (unpacking) of their data. The system treats sequential passive objects as independent, data parallel objects that have copies at each node and do not need consistency. Distributed objects model large application data that is involved in data intensive computations. Sequential objects model parallel independent computational kernels used in conjunction with distributed objects to model a complex application.

Data distribution is performed automatically by the system. The system distributes data across multiple address spaces according to a general partitioning function that assigns each data item to a unique partition. The user instantiates / creates the distributed objects. Typically for data parallel computations, the user either stores large data from an external storage (usually a file) into the application structures, or employs large data structures (e.g. very large multidimensional array objects) to produce application output. In either case the user specifies the real instantiation data (e.g from file, or real data ranges from application parameters). The system then partitions the data across multiple address spaces in a load-balanced manner. After the initialization (instantiation), the user accesses the distributed objects through their interface, and therefore only the local data (partition). The system replicates the sequential objects at each address space level.

Data consistency is automatically ensured by the system for distributed objects only. Upon data partitioning, these maintain information on data location and data dependences across address spaces. Thus, for each object the system records data dependence information on data partitions residing in other address spaces. The system uses this information to keep data consistent. The system propagates the local changes in the data to the dependent remote objects. Conversely, it reflects the changes in remote objects data to the local objects according to data dependences.

The computation model from the user perspective is the trivial Single Program Multiple Data (SPMD). In the SPMD style data objects reside in different address spaces, and the same computation (program) is replicated across multiple address spaces. The system automatically ensures consistency for the distributed objects. The user has the illusion of an“ideal SPMD” model, and can thus write an application in a “close to sequential” programming style.

9.3 A Distributed Object Model for Transparent Concurrency

The model described so far is suitable for data parallel applications, where computations can proceed independently and consistency has to be ensured only at certain points in the application control flow. Thus, we refer to these applications, consisting of large computation steps followed by a communication phase, as loosely synchronous [31, 106].

A process[2] controls all distributed and sequential objects at one address space level. There is no intra node concurrency. That is, at one address space level the computation is sequential. Distributed objects encapsulate partitions of data, i.e. data that are distributed among processors according to a general partitioning function. The partitioning function is a bijection and uniquely maps one data item on one partition corresponding to one physical processor. Thus, the object data are disjointly assigned to processors.

In order to avoid excessive traffic due to consistency, the system classifies the data around distributed objects into *truly owned data* and *replicated data*. This distinction is not visible to the user. The truly owned data are the object data assigned to a specific processor in the distribution process. The replicated data are the remote data needed locally to correctly compute a value, due to data dependences. Instead of performing a remote access every time the system detects a data dependence to a remote location at run-time, the system uses the local replicas of data. The system updates the replicas periodically to ensure that the latest updated values are available locally. From the user point of view only the truly owned data are available. That is because the user accesses objects through their interface only.

An operation executes sequentially within a single distributed object, but the same operation applies in parallel on all distributed object partitions in the system. In the presence of data dependences inter objects, the state of the object is kept consistent

[2]One process corresponds to one physical processor in the system. Throughout the chapter we may use terms processor and process interchangeable.

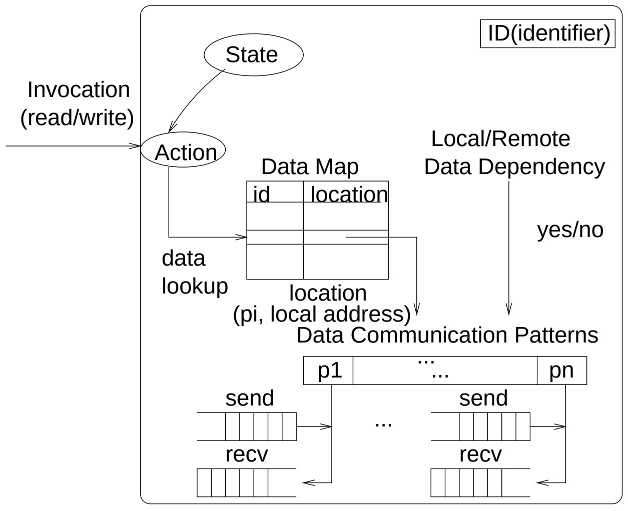


Figure 9.1: The distributed object model.

through collective update synchronization. The owner of each set of data computes the newest values for data replicated remotely and propagates them in a collective update phase. The update communication protocol is point-to-point. The objects have the ability to pack (“serialize”) and unpack (“deserialize”) their data (owned or replicated) and to communicate it in a message. In order to automatically generate communication, the system computes the communication patterns . That is, based on the data dependences and the partitioning function, the system records the communication blueprint of the objects. The objects cooperate by sending and receiving messages along the communication patterns to ensure data consistency.

In Figure 9.1 the *state* of a distributed object is the “truly owned” and replicated data. The user accesses the truly owned data of the object through its interface. The system distinguishes between the read and write accesses in order to be able to handle consistency.

The data dependences of interest are only for distributed objects. We can see these as ordered collections of elements encapsulated together. Due to partitioning, some of the accesses can be remote. To make the remote values available the system has to know their location, and thus it keeps information on partitioned data location.

Upon partitioning, a *map of the data* (data map in Figure 9.1) records the identifier and the location of the distributed objects partitions. Each processor only keeps track of the location of data partitions that are directly related (i.e where the cut in the data occurs upon partitioning). Based on the local knowledge of each processor, any one data access can be located at any one time in a distributed cooperative search. However, for iterative numerical algorithms it is sufficient to look at the neighbour processors to find the data. If the location search becomes an efficiency bottleneck, data reorganization is possible.

The identifier in the data map serves as the lookup index for retrieving the address of a remote object from the data map. The location consists of a processor identifier (corresponding to a physical processor address) and the address at the remote location. The processor identifier is an index for data communication patterns.

In Figure 9.1 the system constructs the data communication patterns statically, i.e. the *send* and *receive* patterns. These are, for each address space, data to send / receive to / from remote locations. The system uses these patterns in the data consistency protocol.

The system takes different actions on operation invocation for distributed objects. The system uses the access parameters and the inferred read / write dependences to decide the operation execution strategy. That is, if a write access precedes a read access, inconsistencies may occur if the latest values of replicated data are not available locally. Thus, all processes synchronize upon write. Each process propagates the locally updated owned data values to the remote replicas (according to the data map). Therefore, the system ensures that the latest updated values are available for the subsequent local read accesses that point indirectly to replicated data.

Typically, data parallel computations uncover dependences between data residing on the region at the boundary between the data partitions. By carefully tuning data partitioning [33] to preserve data locality (e.g. such as to reduce the “edge cut” in a general graph partitioning algorithm [95]), the data duplicated due to partitioning are small. Moreover, the aggregation of communication decreases the number of generated messages.

Sequential objects model data and behavior that do not need to be consistent or synchronized. That is, they model parallel independent computations. Both distributed and sequential objects are accessed via their interfaces, in a location-transparent fashion[3]. This means that they can be employed in the same manner, as in a sequential programming model. This is how the objects can be used. The only distinction between the two types of objects is when one or another type should be used.

The distributed objects represent large data involved in storing large application input data and / or computing values that contribute to producing the application output data according to the user computation. The sequential objects represent data parallel independent computations. The distinction is visible for a user. This implies that a programmer (user) has to make the right choice. The distinction between the type of objects breaks the uniformity of an object model.

9.4 Applicability to Data Parallel Programs

In this section we demonstrate the use of object-oriented techniques and generic programming concepts, in conjunction with a message passing formalism to solve the problem of transparent concurrency for irregular data parallel applications, i.e. FEM solvers for PDEs.

[3]There is a subtle difference between location-independence and location-transparency in a distributed memory. The former allows any processor to access any address in any one remote address space. The latter controls the accesses such that they are only local and the remote accesses due to data dependences are transparently dealt with.

9.4.1 Data Representation

The data parallel applications typically use large data either to store large input data in application specific structures or to contribute to computing the application output. In either case the data are distributed onto processors to decrease the application running time. For the applications we address, the input data to be stored in application specific structures are a discretized physical domain. The output data are the values of a physical attribute (e.g. fluid pressure, temperature, etc.) defined at each spatial location in the discretized physical domain. Consequently, our system provides two different data representations for the two cases.

A preprocessing phase usually places the discretized physical domain data in an input file for the application, called *mesh* or *grid*[4]. Traditionally, regular data parallel applications that employ regular domain discretizations and regular data partitioning, store the mesh or grid data in multidimensional arrays. However, more complex representations are required in practice to express irregular, multidimensional discretized physical domains.

The prototype system implementation provides a predefined representation of the unstructured mesh data that the system can use to store input data. The user has access to a partition of the input data (local to each processor) through the *Subdomain* object. The system also provides a representation of the generic user data that participates in non-trivial data parallel computations, called *UserData*. The user subclasses the *UserData* class and provides the access functions to read and write at a specific location. Instances of this specialized type are distributed user objects. Details on the class interfaces and their implementation are given in Chapter 11.

The *Subdomain* data is not changing during the application lifetime. Thus, no read / write dependences occur in connection with the input data stored by an application. Therefore, a computation using the *Subdomain* data does not involve consistency. The distributed user objects participate in computations that may use the geometrical data from the *Subdomain* object and / or other sequential or distributed data in iterative computations (e.g. coefficient matrix computations, computing the unknown, etc.). Thus, the system automatically ensures data partitioning, distribution and consistency for the distributed user objects.

9.4.2 Data Consistency

The system ensures data consistency using a point-to-point update protocol in which owners of data send the newest values to the processors that replicate these data. The system provides a generic function, *Update*, that uses the generic user data (subclassing the *UserData* class) and the information on data dependences across different address spaces to update the actual user data. That is, the *Update* function transparently packs / unpacks the actual user data in messages that are exchanged among all the distributed objects. After an update phase, the latest values are available for replicated data in each address space. Thus, a new computation step can be safely executed.

Figure 9.2 depicts a typical communication pattern for an application using general geometries in the FEM solution process. The computation uncovers dependences

[4]This work does not address preprocessing phase aspects, such as mesh generation techniques.

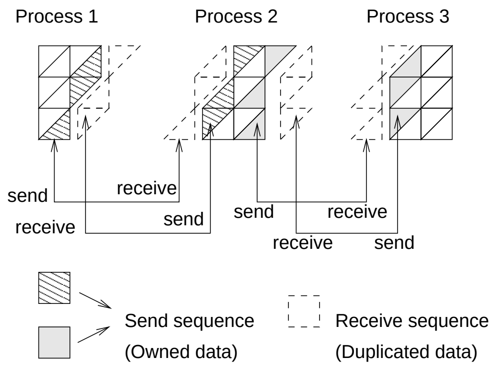


Figure 9.2: Point-to-point communication triggered by an update phase. Internal boundary data is packed / unpacked and sent / received in one message.

among data residing on the boundary between partitions. Thus, each processor holds replicas of the remote boundary data. On write invocations the processes cooperate in a point-to-point protocol to exchange the latest values of the owned data. Therefore, the system ensures that the replicas of data in each address space are consistent.

9.4.3 A Typical Computation

The C++ code excerpt in Figure 9.3 illustrates the use of the distributed object model in a concrete prototype framework implementation. The framework provides transparent concurrency support for data parallel applications that use general geometries. We show a code fragment from an implementation of a 3-D FEM discretization scheme for the Poisson equation using meshes of tetrahedral elements.

In Figure 9.3, variable m designates a *Subdomain*, i.e. the local view of the geometrical mesh data. The user defines the parameterized $DistrData < T >$ type as a specialization of the provided $UserData < T >$ type. The user provides the *Set* and *Get* operations for the $DistrData < T >$ type. The user declares variables a and c as $DistrData < double >$ arrays of floating point numbers. These are coefficient matrices[5] that contribute to the computation of the *pressure* values. The system distributes and keeps consistent data for a and c objects. The variable f represents the right hand side vector field for the Poisson equation $\triangle p(x) = f(x)$. Since the values of the right hand side vector field[6] are known and not modified by the solver, the variable f is declared as an array of floating numbers. Thus, the system treats f as a passive sequential object, i.e. does not care about its consistency. On the other hand, the programmer uses the ranges of the local data to initialize f, i.e. only at the discrete points in the local sub-domain, thus, not for the entire mesh data.

For illustrative purposes we only show the computation of the matrix coefficients

[5]Typically, one does not have to store large two-dimensional arrays since they are very sparse. Hence, in this case we use one-dimensional data structure to represent the coefficient matrices.

[6]In numerical analysis terminology.

```
class Poisson {
 Subdomain m;
 DistrData<double> *a, *c;
 ...
 double *f;
public:
 ...
 Poisson& Compute_A();
 Poisson& Compute_C();
 ...
}
...
Poisson& Poisson::Compute_A(){
 int dim = m.GetDim();
 ...
 for (int e = 0; e < m.GetNElems(); e++){
  double c0ii = 0.10*m.ElemVol(e);
  double c0ij = 0.05*m.ElemVol(e);
  for ( int i = 0; i < m.GetNve(); i++){
   int inode = m.El2MeshNo(e, i);
   double tmp = a->GetAt(inode);
   tmp += c0ii*f[inode];
   a->SetAt(inode, tmp);
   for ( int j = 0; j < m.GetNve() - 1; j++){
    int jnode = m.El2MeshNo(e, (i+j+1)%m.GetNve());
    tmp = a->GetAt(jnode);
    tmp += c0ij*f[jnode];
    a->SetAt(jnode, tmp);
   }
  }
 }
 ...
 Update(*a, m);
 return *this;
}
```

Figure 9.3: A typical computation combining stored and computed distributed data, as well as sequential data objects.

a. In Figure 9.3 *a* is accessed through the *Set* and *Get* operations, and thus, accesses are local. Moreover, the system uses the operations in the *read/write* invocation distinction to detect when to initiate a consistency phase. The computation of the coefficient matrix involves reading and writing the values of *a*. The consistency protocol triggers an update phase after each write that precedes a read (here a read will occur in the next loop iteration). We use an optimized variant of the *Update* function to move the update phase beyond the loop. Showing how this optimization can be detected automatically is outside the scope of this chapter. In the current version of our prototype we manually insert the calls to initiate the update phase.

The example in Figure 9.3 shows the programming model for writing data parallel applications. In the current version of our prototype we use the two representations for storing input, geometrical mesh data and for the generic user data to demonstrate the distributed objects concept. We are currently developing a uniform mechanism for expressing irregular, parameterized recursive data structures (graphs, trees, irregular meshes) so that both input (geometrical mesh) data and generic user data can be expressed in a common format. The motivation for defining such a mechanism lies in enabling the system to implement the distributed object model presented so far and many of the optimizations currently used in the loop level parallelism frameworks [4, 6, 38, 53, 66, 67]

9.5 Evaluation and Experimental Results

We analyze our system from two perspectives. From the *usability* point of view we want to show how simple it is for a user to learn and use our system to write data parallel scientific applications. From the *efficiency* point of view we want to show that our approach is scalable and efficient.

We describe a student experiment from a 5th year project course in our department[7]. Two groups consisting of two students each worked on two projects focusing on system support for scientific, data parallel irregular applications. One group focused on object-oriented support for scientific applications. The other group focused on distributed systems support. The evaluation of the expressiveness and run-time efficiency of the existing object-oriented languages and systems and distributed programming environments was a part of the study.

The evaluation was based on expressing and implementing a toy problem in different existing formalisms as well as using our system. The problem was computing the area of the surface of a three dimensional body described by a general geometry. Therefore the complexity of the problem does not reflect the full functionality of our system.

The problem was implemented using the C++ programming language, together with the *mpich* [80] implementation of MPI, OOMPI (Object-Oriented Message Passing Interface [102]) and CORBA.

We have performed our tests on the Beowulf Linux cluster (Beowulf [89]). In the following we present the evaluation procedure together with the results of our analysis.

[7]Department of Computer and Information Science, Norwegian University of Science and Technology

9.5.1 Usability

We asses the usability of the observed systems, languages or libraries using the metrics described in Chapter 6.

Table 9.1 shows the results of the usability measurements. The listing is not in a particular order (i.e. implementation or learning of the languages, systems or libraries), but relatively random. Moreover, reusable parts (components) from our system were used by the students (directly or adapted) to implement the other versions. More time would probably have been required otherwise. We use the metrics de-

Table 9.1: Usability

Approach	**Learning Curve (days)**	**Number of classes**	**Lines of code**	**Decision Point**
MPI	3	2	751	4
OOMPI	3	2	728	4
CORBA	7	3	952	4
OODFw [8]	1	1	74	0

scribed above for the evaluation of our high-level solution, versus other lower level solutions. In Table 9.1 the name OODFw denotes our system. The data in Table 9.1 emphasize that it will take less effort to write a concurrent application when the concurrency infrastructure is ready-made and can be reused *as is*.

The results (decision points in Table 9.1) also show that ready made design solutions implemented in a system reduce the complexity of writing irregular data parallel scientific applications. It also follows from here that our solution would increase the correctness of the design since most of the low-level, error-prone tasks are taken over by a system.

This evaluation is limited and is not meant to be statistically significant. Larger evaluations require much more time. Thus, we hope that the results are sufficient to give an idea about the goals of our approach and what kind of benefits can be expected.

9.5.2 Efficiency

We show performance figures for the toy problem implementation using the observed systems, languages or libraries by measuring the application speedup. Figure 9.4 depicts the speedup of the application for its different implementations. The results in Figure 9.4 show that MPI, OOMPI and our system (OODFw) have a very similar performance. This means that the overhead we add to OOMPI is negligible[9]. The insignificant difference in performance between MPI and OOMPI confirms the results of [90].

[9]Our system uses the OOMPI library for message passing communication.

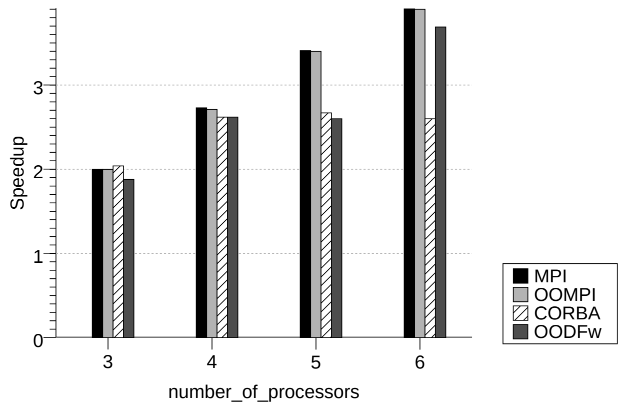


Figure 9.4: Toy problem implementation speedup for different implementations.

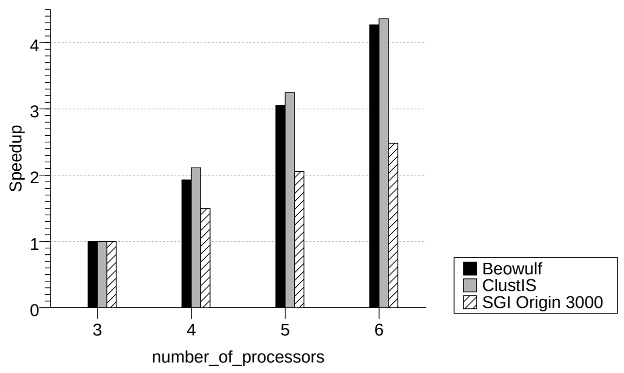


Figure 9.5: The 3-D Poisson solver scalability on different distributed platforms.

Table 9.2: Platform Description

Platform	Performance			Interconnect
	Ghz	Ratio	MFlops	
Beowulf	500 MHz	(SSE) 4	2000 MFlops	Fast Ethernet 100 MBit / sec
		1	500 MFlops	
ClustIS	1.4 GHz	2	2800 MFlops	Fast Ethernet 100 MBit / sec
SGI Origin 3800	NA	NA	1000 MFlops	Hardware Switch

In Figure 9.4 the CORBA version shows poor scalability with the increase of the number of processors. This result indicates that the object-oriented middleware solution is not scalable.

These results are preliminary. The toy problem does not realistically model the large numerical computations. The number of iterations is small compared with realistic solvers. We have simulated large iteration loops in order to get more realistic results. Also, we have only tested for the two mesh sizes that we had available. We give the results for the smaller mesh.

More results on efficiency are shown in Figure 9.5. We show comparative speedup figures for different platforms from the 3-D Poisson solver on the larger mesh.

We analyze the results from Figure 9.5[10] using data from Table 9.2. In Table 9.2 we give the peak floating point performance for the three platforms we use for speedup measurements. We derive the floating point performance from the cycle time using the constant *Ratio* for each processor type (e.g. for Pentium III with ISA extension SSE - Streaming SIMD, the ratio is 4). In Table 9.2 we also give the interconnect characteristics.

The results in Figure 9.5 and the data in Table 9.2 indicate that the speedup results are better on the cluster than on the SGI. We need to compare the actual absolute wall clock times for the platforms in order to interpret these results. Figure 9.6 shows the absolute wall clock times for the three platforms. The results show that the best running time is obtained on ClustIS cluster, followed by SGI and then by the Beowulf cluster. These results are surprising. One would expect that the floating-point performance for the application is lower on the clusters than on the SGI. This is true for the Beowulf cluster, which is the least performant of all. But, surprisingly, the results on ClustIS are extremely good, compared with the SGI. One possible explanation is that the vendor optimized MPI implementation for SGI is poor. We did not find any data on message-passing performance for the SGI architecture. The vendors give only the remote versus local access rate (typically 2:1 for SGI 3000 family).

The results on the cluster platforms are promising. The scalability of the distributed systems (cluster technology) can compensate for the potential overhead added by the system (setting up the concurrency infrastructure, problems with object-oriented languages effectiveness, etc.). There are other benefits from the distributed-memory

[10]We show results for only 6 processors because we do not have a larger Beowulf cluster available for this study. For other studies, we experiment with all the available processors for each platform.

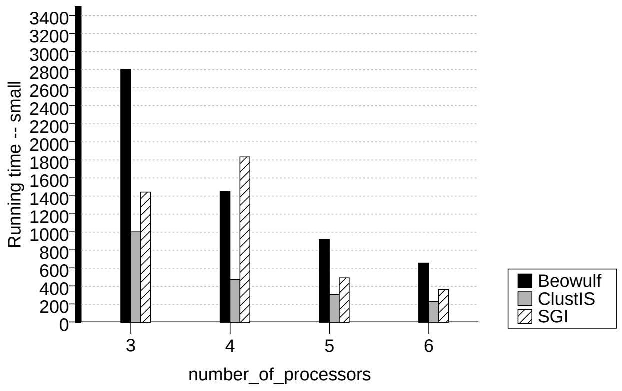


Figure 9.6: The absolute wall clock times for the 3-D Poisson solver on "small" mesh for the compared platforms.

architectures over the shared-memory architectures, such as scalability, economy (especially of the cluster architectures and NOWs - Network of Workstations) that we do not analyze here. However, these benefits and the results in this chapter strongly motivate our approach.

9.6 Related Work

There are two aspects of our distributed object model. One aspect is the integration of objects and processes into object-oriented concurrent programming (see [26, 112] for complete reviews). Traditionally, object-oriented concurrency distinguishes between active and passive objects. The other aspect is the support for non-trivial data parallelism in the context of irregular data representations and irregular communication patterns. Below we discuss each of these aspects and compare them with our approach.

Concurrent object-oriented computing has roots in actor languages [2,3]. The actor model has a functional[11] view of concurrency. Actors are objects that only communicate through messages. The communication between actors is asynchronous. Actors allow full concurrency within the objects (by not allowing changing states). The actor model expresses task parallelism.

The ABCL/1 [114] object-oriented concurrent language evolved from the actor model. An object in ABCL/1 encapsulates data and its *own activity*. Objects communicate through concurrent message passing. The objects become "active" in response to communication. In contrast to the actor model, ABCL/1 adds mechanisms for synchronous communication through *now* and *future* types of messages. Now messages have "send and wait" semantics. Future messages have "reply to me later" semantics.

[11]In the sense of functional programming.

The future type of messages helps increasing concurrency by overlapping communication and synchronization.

Both models of integrating objects and processes into object-oriented concurrency capture task parallelism and not data parallelism. Moreover, objects communicate *explicitly*. Accesses to objects may be concurrent. These models have provisions to serialize the concurrent accesses to an object.

Our model associates a process with all the distributed and sequential objects at one address space level. Distributed objects communicate *implicitly*. There is no explicit remote access or message passing mechanism between objects. The objects allow concurrent state updates and state consistency is *implicit*. Our model captures data parallelism and not task parallelism.

Orca [15] is an object-based language in which processes communicate through shared objects. Orca supports only passive objects. Concurrent accesses to shared objects are serialized. Thus, the system ensures that an operation execution is atomic. The original model of concurrency in Orca supports task-level parallelism. A later enhancement of the language [54] adds support for data parallelism. The language adds support for partitioning array-based structures into regular partitions (blocks). The user *explicitly* specifies the type of partitioning (block, cyclic) and the mapping of data to processors. Parallel operations apply concurrently on partitions. Each processor has a copy of the entire data, but it *owns* just a partition of it. This approach is inefficient for applications that store very large data. The owner processor has write access to the owned partition and read only access to the replicated partitions. The run-time system ensures consistency through an invalidation, or an update protocol (existing implementations support both mechanisms). The system transfers entire partitions, rather than individual elements.

In our model the partitioning and the mapping (distribution) of data to processors are *implicit*. Moreover, the system partitions the data in a load-balanced manner. In the enhanced Orca language there is no provision on the system side for load-balancing. Moreover, the enhanced Orca language provides regular partitioning of array-based structures. Our model provides general partitioning, for data with irregular layout.

Irregular applications refer to data parallel applications that use irregular data layout (structures). Fortran-like array languages express irregular data layouts with support of indirection arrays. The indirection arrays complicate the loop-level parallelism. The *inspector/executor* run-time support addresses this issue. Similar to the *inspector/executor* strategy in CHAOS [96], we generate all data that needs to be sent and received based on the knowledge of data dependences. This approach addresses fine-grain loop-level parallelism, while we address coarse-grain parallelism.

CHAOS++ [31] extends CHAOS to support objects and thus, coarse-grain parallelism. The object model in CHAOS++ supports *mobile* and *globally addressable* objects. The mobile objects model communication. The globally addressable objects model shared data between processors. The user is responsible for packing and unpacking data into *mobile objects* and for explicitly specifying message exchange. With our approach we isolate the user from these aspects.

Finally, there are many object models that add concurrency constructs to the C++ programming language to support data parallelism [24, 34, 48]. These models support HPF-like concurrency in an object-oriented setting. In contrast with the fine-grain HPF concurrency style, we address coarse-grain concurrency.

9.7 Summary

This chapter has presented a distributed object model for data parallel irregular applications. The model is based on the notion of distributed active and sequential passive objects. The distinction between the two types of objects is visible in terms of "when" to use them. Distributed objects express coarse-grain non-trivial data parallelism for applications with irregular data layout. Sequential objects express coarse-grain, trivial parallelism. The main benefit of our model is the resulting "illusionary" sequential programming style. Therefore, the distributed computing aspects are transparent to the user.

We have presented an evaluation of the usability and the efficiency of the object model together with preliminary results. The results show that our approach requires the least effort for writing a new concurrent scientific application. The results also show that our approach is efficient and exploits the scalability of the distributed-memory architectures (clusters of PCs). These results support both research hypothesis on effectiveness and programmability from Chapter 6, $H1$, $H2$.

In the present prototype implementation we use two different representations to store input data and to express application specific data. An extension of this work is to provide a uniform data representation for irregular data layouts. This common representation can be used by the system to implement the distributed object mechanisms presented in the chapter.

Chapter 10

Inter-Object Concurrency

10.1 Introduction

The object-oriented principles of abstraction and encapsulation have been extensively explored for the software development process. There is broad acceptance of the object-oriented method as an organizing principle for developing reusable software artifacts. A large variety of object-oriented languages and systems exists. Beyond their proved success through wide adoption, they bring valuable insight to the practice of the object-oriented method.

There are several implementation aspects of the object-oriented languages and systems that have shadowed their popularity for some classes of applications. One class of applications for which object-oriented practices are still in the incipient phase is the class of performance sensitive, concurrent applications.

Our goal is to prove that, besides the benefits of abstraction and encapsulation, techniques such as generic programming and dynamic binding can be profitable for raising the level of abstraction while increasing concurrency for data parallel applications. We elevate the level of abstraction for parallel applications by making data mapping and consistency implicit in a distributed object model. We increase concurrency by having a distributed object exploit data locality through encapsulation of the data local to one address space.

The existing object-oriented concurrent languages and systems promote *intra-object* concurrency. Several complications arise when ensuring that an object accessed in a concurrent manner is in a consistent state, or that the principles of object orientation are not violated (e.g. inheritance breaks encapsulation [81, 100]). Moreover, for applications that manipulate large data through intensive computations[1], the fine-grain intra-object concurrency conflicts with the need of partitioning the data into coarse-grain, manageable sub-sets.

We propose an *inter-object* concurrency model that exploits data parallelism for computationally intensive applications. This chapter makes the following contributions:

[1]One has to distinguish between applications that store large data and involve data transfer through queries (i.e. client-server style), and applications that use large data in computationally intensive tasks (e.g. computational biology, weapon simulations, climate modeling, etc.).

- **A data model for expressing recursive data layouts:** We use the C++ programming language to implement the *Set* abstraction as a recursive data abstraction for the efficient representation of large, irregular data layouts. We use the *DistrSet* abstraction to represent large, distributed data sets among multiple address spaces.

- **A data mapping scheme that ensures data locality:** The distributed data sets are automatically mapped to different processors to ensure data locality. Our original mapping scheme uses a general partitioning function that uniquely maps a data address to a partition (processor) together with information on data accesses to ensure data locality.

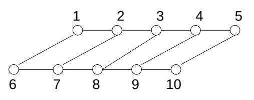
- **A dynamic inter-object consistency scheme:** Upon the distribution of data based on our mapping algorithm, the system maintains consistency information on distributed objects, such as location, references to different address spaces and references from different address spaces. The consistency scheme uses this information and a relaxed *owner-computes* rule to automatically generate communication for the most recently updated values across address spaces. The minimum number of messages is generated due to the controlled mapping scheme and granted write accesses to non-owned locations.

The remainder of this chapter is organized as follows: Section 10.2 overviews the inter-object concurrency model. It describes the data model, the concurrency model and the computation model. Section 10.3 presents the data model. It introduces the *Set* abstraction as a means to represent recursive data layouts. It also describes the distributed set concept in detail. Section 10.4 elaborates the notion of parallelism in our model. It describes the mapping of data to processors, as well as the inter-object consistency algorithm. Section 10.5 presents the computation model from the user perspective. Furthermore, it contrasts the user perspective from the system perspective in order to highlight the usability of our model. In Section 10.6 the efficiency of our techniques is evaluated. Scalability results on a 28 processors Linux cluster show that our approach is efficient. An overview of the related work is given in Section 10.7. Section 10.8 concludes the chapter and indicates future research directions.

10.2 An Overview of the Model

In this section we overview the object model for data parallel applications. The main assumption is that such applications use large, irregular data structures required in scientific, graphics or database applications (e.g. unstructured meshes, graphs, B-trees, etc.). Such data are usually represented by recursive (trees, graphs) or indirection structures (indirection arrays).

In this chapter we consider SPMD or BSP [106] model of parallelism, as usual. In a straightforward data parallel model, the computation can be carried out independently, on each data item, in a consistent manner. However, this is rarely the case for realistic applications. Thus, computations on “large parts of data” can be consistently carried out independently. Consistency has to be ensured for the remaining



Vertices Representation:
1 2 3 4 5 6 7 8 9 10

Adjacency information representation:

1	2	3	4	5	6	7	8	9	10
2 6	1 3 7	2 4 8	3 5 9	4 10	1 7	2 6 8	3 7 9	4 8 10	5 9

Figure 10.1: An example of hierarchical data representation.

data. Therefore, in the BSP model, large computation steps proceed in parallel on independent data items and are followed by a common synchronization phase.

The synchronization phase is usually the bottleneck for usability and efficiency when writing data parallel, distributed-memory applications. In our approach usability is achieved by making synchronization implicit. We achieve efficiency through coarse-grain, *inter-object* parallelism, and a *relaxed inter-object consistency* model.

The Data Parallel Model: There are two aspects of our data model. On the one hand, it is cumbersome to represent the recursive structures required by the applications. On the other hand, the recursive representation features poor locality, which can lead to poor performance. On the other hand, it requires complex, dynamic analysis to compute the location of a data reference. Another hard aspect is to ensure inter-object consistency for the application distributed data.

We use the *set* representation in order to model the recursive data. In our model a set becomes a “first class” object that expresses the recursive data layout. A set is a collection of items with a given range. Items in the set can be sets themselves. Coarse-grain data parallel objects result from the partitioning of a set into subsets and the distribution of subsets across multiple address spaces. Thus, computations proceed independently on each subset. Consistency is guaranteed inter sets. While not extending the C++ syntax, we define the set class interface to be used for expressing data parallelism in an application.

We distinguish between *dependent data* and *independent data*. In a SPMD model the independent data are manipulated by a process, concurrently with other processes. Computations on independent data require no communication. In an enhanced form of SPMD, however, conglomerations of data are handled concurrently and communication is required for ensuring consistency. Thus, in our model the dependent data is expressed by the distributed recursive sets of data. When distributing the data we keep track of potential inter-set data dependences. Consistency is guaranteed for distributed sets.

The independent data can be distributed sets with no inter-set dependences, non distributed sets or any other user defined data. There is no consistency notion for the independent data objects.

Figure 10.1 illustrates an example of hierarchical data representation commonly

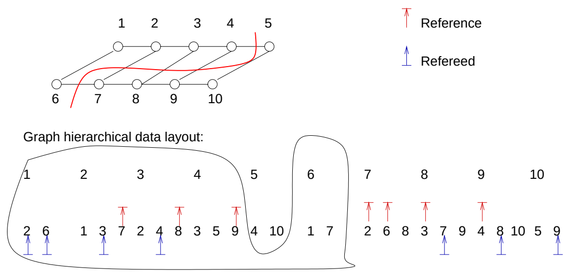


Figure 10.2: Data partitioning. Reference and referee marking.

used for representing undirected, unweighted graph information. It is straight forward to represent such data with only one dimensional indirection arrays. However, in order to exploit coarse-grain parallelism when partitioning such data across multiple address spaces, a more flexible representation is needed. For instance, a Fortran parallelizing compiler would have problems in parallelizing traversals over indirection arrays. Also, the traditional data parallel models (High Performance Fortran - HPF) lack expressiveness and flexibility for complex, hierarchical data layout.

The Concurrency Model: Parallelism is achieved through inter-object concurrency. Upon partitioning a set, its new range is automatically recomputed. There is no notion of intra object concurrency, or shared memory view. Instead, a local address view is available. Due to the recursive references, local data accesses may refer to remote addresses. Thus, upon partitioning each distributed set keeps track of all the references to a different address space. It also keeps track of all the refereed locations from other address spaces to the local address space. Therefore, data accesses to these locations are subject to the consistency protocol.

For instance, in a graph an edge relates two vertices. Thus, for each vertex, the connected vertices model the relations existing in a graph structure. While this information is usually expressed by an application, a system cannot make use of it. Therefore, we require explicit information about the relations implicit in a data representation so that the system can register the references on partitioning and use them in the consistency protocol. This is enabled in our model by declaring the graph as a set of vertices which point themselves to sets of nodes indicating connectivity information. Therefore, this would be a two level data structure with linked references. Figure 10.2 shows how the referring and refereed locations are detected upon partitioning.

The Computation Model: The computation model is an enhanced owner-computes rule. This means that write accesses are granted to non-owners of data as well. In the relaxed consistency protocol we locally duplicate the references to remote data objects. Our consistency algorithm is based on an update protocol in which the owner of the data initiates an update phase upon write. Thus, the system accounts for the availability of the correct value upon read.

The same computation is applied on different data sets, in the SPMD style. One process is spawned for each address space and the computation within a single address space is sequential. Therefore, the user has access to a sequential programming

```
template <class T> class Set{
 public:
  Set(size_t range);
  reference& operator[] (size_t pos);
  Set& operator= (const Set&);
  Set& operator () (size_t size);
  size_t Range();
  size_t Pos (const T& val);
  ˜ Set();
}
```

Figure 10.3: The set class interface.

model and one address space only. Employing the distributed sets structures, the user indicates to the system that the set should be distributed across the available processors in the system. Employing regular (non-distributed) data objects, the user is aware that the values are only relevant to the one address space.

10.3 The Data Model

Our data model is a distributed, coarse-grain data parallel model. Data objects are visible at one address space level only. There is no shared memory view, and thus, no intra-object concurrency (i.e. an object is accessed by a single thread / process). Coarse-grain parallelism is exploited through inter-object concurrency only. Inter-object consistency is ensured through message passing.

10.3.1 The Set Data Structure

The introduction of sets in our model is motivated by the lack of a uniform mechanism to efficiently support irregular, recursive data representations that are used in data parallel applications.

The multidimensional array representation does not enable the exploitation of coarse-grain parallelism characteristic to many data parallel applications. Nor does it enable a flexible representation for complex, irregular structures. Moreover, its linear structure and regularity call for regular partitioning functions such as block and cyclic distributions.

This approach does not apply to data parallel applications that use irregular, complex data due to the fundamental differences with respect to the regular structures. That is, consecutive elements in an irregular sequence are not for instance at contiguous addresses as elements in an array. Thus, there is little inherent spatial locality between consecutively-accessed data. As a consequence array data representations and regular distributions do not always work for irregular data layouts. Instead, a more flexible, recursive data representation that properly reflects the underlying data layout has to be used in order to exploit *logical data locality*, rather than *physical data locality*.

```
Set<int> conn(3);
Set<Set<int>> Graph(10);
...
for (int i = 0; i < 10; i++){
 for (int j = 0; j < 3; j++)
  conn[j] = int_val;
 Graph[i] = conn;
 }
...
}
```

Figure 10.4: An example of using sets in graph problems.

Figure 10.3 shows the interface of a general data set. A set has a range and elements in that range. The type of elements in the set is specified through the template parameter class type. An element in a set can be a set itself. A set has its dimension specified upon instantiation. The set has operations defined for subscripting (fast access) and assignment. Sets cannot be extended to add new methods. This is the provision we make for safe manipulation of distributed sets by the system. Also, sets are not handled through the use of pointers. Therefore, simple program analysis can provide precise information on data access patterns for sets.

Figure 10.4 shows an example of the representation and traversal of the graph in Figure 10.1 by using sets.

10.3.2 The Distributed Sets

The second reason for introducing sets is to facilitate the mapping of data to processors in a controlled, optimal way, such as to minimize communication across different address spaces. Thus, when instantiating large data sets, the user specifies that the set is to be distributed by using the *DistrSet* class. A distributed set can only be parameterized with a set type and not with another distributed set type. Besides this restriction, the interface for the distributed set class is identical to the interface for the set class. This workaround is meant to avoid extending the C++ language by introducing the keyword *distributed* as type qualifier for distributed sets.

For each distributed data set object the system registers the *automatic consistency information* upon partitioning, as shown in Table 10.1.

In Table 10.1 an address of any one set contains a unique partition number (corresponding to a physical address space), and the memory location. The *list of references* holds any of the references (addresses) that this set might have to addresses that contain a different partition number (i.e. to different address spaces). The *list of referees* holds the list of all the references belonging to a set whose referees contain a different partition number (i.e. from a different address space).

The user can define and use more complex data layouts, existing components from libraries, or ad-hoc simple layouts. Upon instantiation of such a data objects, only local addresses and values are used. They can be used to store values from distributed sets,

Table 10.1: The distributed set class descriptor.

`address`	the address of the memory location
`list of references`	the list of references to remote address spaces
`list of referees`	the list of references from remote address spaces

to compute values for distributed sets, or parameterize algorithms with distributed sets.

10.4 Parallelism

Parallelism is achieved in our model through concurrent application of functions on distributed data objects. A process is associated with all the data objects residing in one address space and the computation on these objects. Processes synchronize loosely to exchange the most recent updated values for data duplicated in other address spaces.

The mapping of data to processors and the replication of remote data locally are controlled by the partitioning algorithm. The algorithm, based on heuristic criteria such as *minimizing the remote references*, uniquely assigns a partition number to a data item. Thus, our consistency protocol results in minimum communication while maximizing data locality.

The sets are the key to exploiting parallelism. Sets can be viewed as a means for the user to express large, irregular structures. Even though regular structures can be successfully expressed and manipulated with the use of sets (for instance a distributed array can be a set of *range* elements), the sets were motivated by the need to express irregular, recursive large data.

There are two key ideas behind the set structure. One is to express recursive data layouts to exploit coarse-grain parallelism. The second is to enable the partitioning of sets such as to minimize the number of references to remote address spaces.

Coarse-grain parallelism is obtained by applying a general partitioning function for a large data set in conjunction with information on data accesses. Each partition corresponds to a physical processor. Then, the references to remote locations are replaced by local references, which will act as placeholders for the remote values. The most recent value will be available when the true owner of the remote location updates this.

The minimum number of remote references is ensured by having the system linearize the recursive data structure and apply a load-balanced partitioning to data. Upon partitioning all the addresses are computed and consist of a partition number (process identifier) and a local address. The number of potentially remote references is used as minimizing criteria for a load-balanced partitioning algorithm. Thus reference and referee lists are minimal. The lists will be used as communication patterns for the subsequent consistency phase.

10.4.1 Mapping of Data to Processors

In this section we elaborate the data mapping procedure. The mapping of data to processors uses a general graph partitioning algorithm and information on data dependences to preserve data locality. Therefore, the mapping strategy relies on being able to infer the data access patterns for the distributed sets.

We use the linearization technique [77] to account for data locality. Hence, theoretically, any recursive structure expressed with the help of pointers can be potentially linearized. Thus, references can be computed and mapped such as to exploit data locality. In practice, though, the cost of such a procedure for an arbitrary changing recursive data structure would far overcome the gain.

We use a simplification in order to make linearization practical. That is, we restrict the application of the technique to the structures that can aggressively benefit from it, i.e. the set structures. On the other hand, the set structure is designed with two purposes in mind: to express irregular, recursive data layouts while facilitating the linearization.

The linearization of the distributed sets enables us to compute a data access address based on a subscripting index. Moreover, the data can be reordered such as to match the traversal of recursive sets at compile time. When partitioning we can use the data accesses patterns to minimize potentially remote references.

The data accesses for the distributed sets can be the result of various expressions. Usually, data accesses follow the traversal order and are bounded by the ranges of sets. The extreme case is having accesses as an arbitrary function call that returns the access index (also bounded by the set range). In the first case, we can compute the reference and referee lists at compile time. These lists are bound to be small since the remote references are subject to minimization criteria when partitioning data.

For the second case, we cannot know anything about the accesses until run-time. Initially, these references and referees are marked as unprocessed. At run-time we use the information on partitioning in a collective, distributed protocol for computing the unprocessed references and referees lists. We exploit the symmetry of the computation model (SPMD) to obtain these data accesses at run-time. That is, the processes cooperate in a point-to-point exchange of messages to construct the reference and referee lists for the distributed data sets associated with each address space.

There is overhead associated with the run-time computation of the reference and referee lists. Our approach is feasible given that most of the applications access the recursive data through traversals. Thus, the reordering of data and mapping such as to minimize remote references will work alone for most of the applications. For the special cases when we cannot possibly know anything about the shape of the function computing an access index at compile time, it is very probable that once the information is computed at run-time, it can be reused for subsequent unprocessed access patterns. Given that the structure of data does not change frequently (or at all) for the applications we look at, the overhead associated with the run-time computation of the reference and referee lists is small.

Our partitioning scheme applies recursively. The partitioning starts with distributed sets of depth n, with a *null* set of reference information, by applying the *minimize references* partitioning function. With no reference knowledge the partitioning will result in a disjoint load-balanced partitioning of data. Then, according to dependences across distributed sets, the set of references becomes the number of potentially remote references for depth n sets (estimated at compile time). The partitioning of depth n distributed sets is now fixed (each data is mapped to a partition) and the procedure continues for depth $n-1$, mapping each data to a partition such as to minimize the references at level n. References for distributed sets of depth $n-1$ are subject to minimization when applying the general partitioning function for depth $n-2$, etc.

The depth of a recursive set is established according to the parameterization type. If the type of the elements in a set is a set itself, then the level is incremented with one. As soon as the type of the set elements is a built-in type, or a type other than set, the maximum depth is reached.

We use general graph partitioning algorithms [95] that assign a unique partition number to each node, such as to minimize the *edge cut*. In our case nodes are data addresses, and edges are data dependences. Our mapping scheme is original by using the number of potentially remote references as minimizing criteria for mapping the data to processors. Moreover, most of the existing mapping schemes used to parallelize applications are based on regular distribution functions.

For illustrative purposes, let us assume that the graph in Figure 10.1 represents an irregular mesh information where nodes represent vertices and edges connect the vertices of an element. Through the new perspective, there are 4 elements, each having 4 vertices. Figure 10.5 illustrates how our reference-count based partitioning technique works on the irregular mesh structure. In Figure 10.5, the elements are level two data, while vertices are level one data. Let us assume that both, elements and vertices, are large data, declared as distributed sets and given their dimensions by the user. Then, based on the control flow information, the number of potentially remote references can be computed for the elements. Therefore, the level one data, i.e. the vertices, are mapped such as to preserve data locality.

Our mapping procedure is driven by the scalability concerns and it relies on knowledge of real world, irregular, data intensive applications. That is, in SPMD applications the recursive data structures holding large data sets change very slowly, if they change at all. Thus, our optimal placement of data to locations such as to improve data locality works extremely well. We have experimented with our techniques for data intensive numerical applications that use large data sets as input and are used throughout the entire simulation process. The results show extremely good scalability.

10.4.2 The Inter-Object Consistency Protocol

At run-time the mapping procedure results in having assigned each data to a partition. Based on this information and the information on data accesses, we can dynamically compute the information for the distributed sets, i.e. their new mapping address, the references and referee lists. Moreover, for each remote reference we duplicate the data locally to serve as the place-holder for the remote data. These distributed set objects

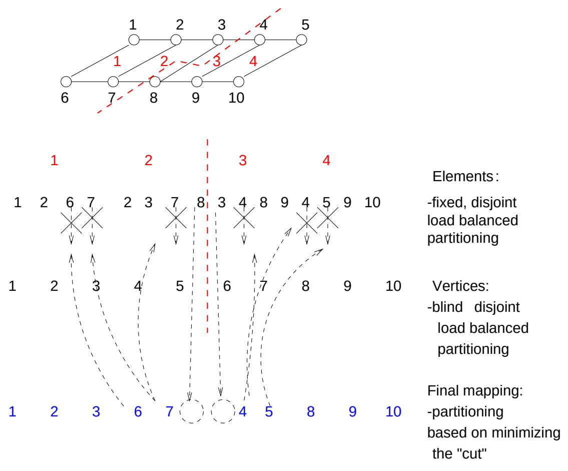

Elements:
-fixed, disjoint load balanced partitioning
Vertices:
-blind disjoint load balanced partitioning
Final mapping:
-partitioning based on minimizing the "cut"

Figure 10.5: Reference count-based mapping scheme.

information is the input for the consistency algorithm.

The inter-objects consistency algorithm is as follows[2]:

```
1.  Classify the distributed set accesses as read
and write.
2.  For each distributed data object:
 2.1.  Detect read/write dependences for distributed
 sets.
 2.2.  Mark object as dependent (treat as in step 3).
 If not, mark independent(do nothing).
3.  For all the dependent data objects:
  3.1.  Insert run-time checks to read/write
  dependent accesses.
  3.2.  Upon write to a refereed location,
  send the last value to all referees.
  3.3.  Prefetch the latest values for references,
  if any.
```

Our consistency scheme is extremely simple and efficient at the same time. First, we use the local placeholder to allow non-owners to write to not owned locations without incurring a data transfer. Moreover, based on the *referee information*, we insert

[2]The steps in the algorithm indicate logical rather than temporal sequencing. Thus, some of the steps are performed at compile time, while others are at run-time.

run-time checks and trigger a data transfer initiated by the owner of data. Furthermore, we take advantage of this synchronization point to prefetch the newest data as well, based on the *reference information*. We can do that because of the symmetry of the SPMD model. This simple trick guarantees that the next read access to a location from the reference list will have the newest, consistent value. Thus, upon a read operation no synchronization is needed.

10.5 The Computation Model

The concurrency model we have presented results in potential benefits for the user. These are: high usability and high scalability for the SPMD data intensive, irregular applications. Besides using the *distributed set type* to declare sets of data that are to be distributed across multiple address spaces, the programmer is not aware of the distributed computing details. This achievement can be considerable, given that real world, scalable applications explicitly code all the mapping (including partitioning) and consistency (including communication) details we have presented in the previous sections.

There are two views of the computation model:

I System view: From the system perspective the computation model is SPMD (single computation applied concurrently to multiple instances of symmetric data) with optimal mapping of data onto processors and relaxed consistency model.

II User view: From the user perspective the computational model is sequential and restricted to a one *local address space* only (as opposed to a software shared memory view of multiple address spaces). That is, we ensure location-transparency, instead of location-independence for accesses to distributed data.

Our computation model is original due to the special user/system perspectives. Generally, our approach belongs to the class of techniques aimed at achieving usability for the distributed programming style, while retaining the scalability of the underlying distributed system. The existing approaches can be classified along two main axes:

- Fully automatic, or guided (through compiler directives) parallelizing (restructuring) compilers for Fortran languages [5, 9, 57, 111]. Along this axis, an existing sequential program can be transformed to run in parallel. The great advantage of this approach is the possibility to parallelize legacy code without involving costly human resources. The main limitation resides in their restricted applicability. Such techniques are successful for regular data layouts and less successful for irregular data layouts.

- Distributed Shared Memory model (virtually shared memory) [21, 60, 61, 72, 74, 94]. Along this axis, all the existing realizations, from hardware to software, attempt to exploit the usability of the shared memory model. A virtual shared memory view is built on top of a physically distributed memory. While the approach is successful in achieving usability, its scalability is still a challenge to prove for researchers.

```
typedef Set<double> vertex;
typedef Set<vertex> elems;
DistrSet<vertex> Verts(nverts);
DistrSet<elems> Elems(nelems);
DistrSet<double> a(nverts);
TetraMesh m;
...
for (int e = 0; e < Elems.Range(); e++){
 double c0ii = K1* m.Vol(e);
 double c0ij = K2*m.Vol(e);
 for (int i = 0; i < I; i++){
  int inode = m.El2MeshNo(e, i);
  a[inode] += c0ii*f[inode];
  for (int j = 0; j < J; j++){
   jnode = m.El2MeshNo(e, j);
   a[jnode] += c0ij*f[jnode];
  }
 }
}
...
```

Figure 10.6: A typical irregular computation. Example of integrating sets and user defined components.

Our approach offers a flexible programming model that enables the expression of the application knowledge such that a system can exploit it at best. Legacy programs cannot benefit from our approach to parallelization. Nor can they be applied to task parallel applications. However, there is always a trade-off involved when trying to achieve a goal. We trade-off the applicability of our techniques to data parallel applications only, in order to achieve high usability and scalability.

Figure 10.6 shows a simple example of a computation expressed using distributed sets and user defined data. This code excerpt is a typical computation from a numerical application that involves large, irregular data. The data describes an irregularly discretized physical domain by using geometrical shapes. In Figure 10.6 an element of such a mesh is described by the number of vertices making up the corresponding geometrical shape (e.g. 3 for triangles, 4 for tetrahedra, etc.). A vertex is described by its spatial coordinates. The distributed set of elements is used to hold and access large data. A user-defined class called mesh is used to express complex computations with geometrical data (e.g. computation of volume, area, surface normal, etc.). The mesh computations, such as element volume and surface area, are involved in computing a coefficient matrix.

The elements and vertices are given their real range, upon their declaration as distributed sets. Then these are only accessed through the *Range*() operation which returns the local range recomputed automatically by the system upon distribution. The user mesh data will be instantiated by the user with the elements from the distributed large data sets and thus, only with the local subset. The computation is therefore kept within the small range of the actual data as a part of a much larger data. The coefficient matrix a[3] is also a distributed set since it defines a data structure corresponding to the number of nodes from the entire mesh (and thus, very large).

The example in Figure 10.6 emphasizes two aspects. One is the ease of the programming model and its conformance to the sequential programming model. The other aspect is that the existence of sets does not prevent the programmer from using the power of the C++ language to express much more complex data layouts or computational kernels. It merely helps the programmer to store the large data in a distributed manner and use just a part of it as desired.

10.6 Evaluation and Experimental Results

There are two aspects involved when establishing the appropriateness of the object-oriented languages for data parallel, performance sensitive applications. One is related to the sequential efficiency, and translates into the overhead added by dynamic binding. The other efficiency aspect is related to the parallel execution scalability.

In general, the scalability of a system indicates how well the system can adapt to increased demands. For concurrent applications scalability is usually measured by the *speedup* of an application. The speedup of an application indicates how the parallel version performs with respect to the sequential one.

The efficiency of the sequential object-oriented languages and systems is a research topic in itself. Efficiency issues such as the abstraction penalty and inheritance [29] are addressed by techniques such as procedure / method in-lining. Optimizations such as dead code elimination, code motion, loop merging, etc. are also explored by research into optimizing compilers. Here we are interested in showing the effectiveness of our technique in reducing communication and increasing data locality. Therefore, we show results on the parallel execution scalability.

We present scalability data from running a large, irregular numerical simulation on ClustIS Linux cluster [43].

The speedup results are shown in Figures 10.7, 10.8[4]. The input data size used for the measurements in Figure 10.8 is approximatively twice the input data size used for the measurements in Figure 10.7. As expected, the larger the data size, the better the gain in efficiency when adding more resources (processors). However, the scalability of a distributed computation is also limited by the communication aspects. As a consequence, when communication becomes significant compared with the computation, there is little, if any, performance gain in adding more computational resources (i.e. processing nodes). Hence, for different application sizes, there is a different threshold number of processors, above which the system does not scale well (this number is 21 for results in Figure 10.7 and 27 for the results in Figure 10.8).

There is also the aspect of the scalability of the underlying architecture as well. For tightly coupled, shared-memory architectures, the communication between different address spaces is through (expensive) fast hardware switches or memory bus, and

[3]In numerical applications sometimes matrices are not kept in two dimensional structures, even though, conceptually, they do represent the elements of a two dimensional mathematical matrix.

[4]Note the difference in scaling along the speedup axis in the figures.

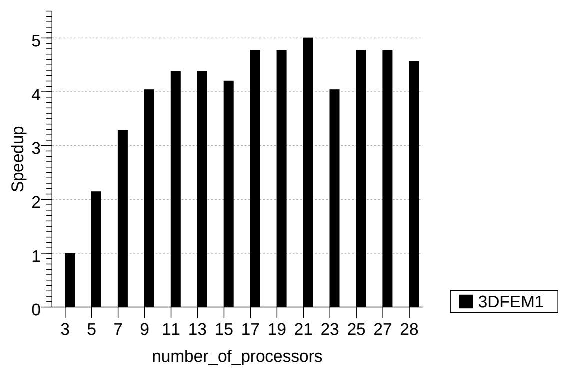


Figure 10.7: The speedup of the 3DFEM application for "tiny" mesh.

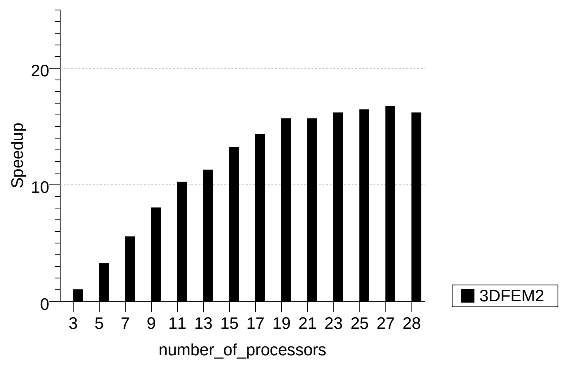


Figure 10.8: The speedup of the 3DFEM application for "small" mesh.

therefore much faster. For loosely coupled systems, such as PC clusters or NOWs (Networks of Workstations), the communication is limited by the network bandwidth. We have tested for different platforms (from tightly coupled, shared-memory machines to Linux clusters).

However, all the results show very good scalability for large data sets, proving the effectiveness of our techniques. Moreover, these results show that the sequential overhead of an object-oriented language cannot be held against their adoption for parallel computing. That is, if a system scales well, the sequential overhead is amortized in the parallel execution.

10.7 Related Work

There are numerous approaches to integrating objects and concurrency (see [82, 112] for complete surveys). First, we briefly review the main concepts. Then we present the approaches to high performance applications in more detail.

"Pure" object-oriented concurrent languages satisfy the requirements for object-orientation [84], such as inheritance. In languages like [3, 114] an object can accept concurrent requests through messages. These models support *intra-object concurrency*. In such a model an object is shared or visible to concurrent processes. In a distributed-memory environment this translates into support for shared memory view, or explicit remote method invocation.

Intra-object concurrency is suitable for exploiting fine-grain parallelism, in the context of a shared memory view. Neither the shared memory view nor the remote method invocation answer well the scalability requirements for the data intensive, SPMD applications. Our model supports *inter-object concurrency* only in order to efficiently exploit the SPMD computation style.

Object-oriented distributed frameworks [63, 71] such as Corba and Java RMI are based on the remote method invocation mechanism and support the client-server style of communication. These models work best for data transfer, and not very well for data intensive computations. Even though peer (equal) distributed applications can be expressed in these models, they do not show good scalability. Efforts have been made to improve Java efficiency and use the shared-memory Java multi-threading to high-performance applications [113].

Object-oriented/based Distributed Shared Memory (DSM) models implement a software shared memory view on top of physically distributed memory. Hawk [54] is a system based on ORCA [15,16] that has the notion of partitioned objects for supporting regular, data parallel applications. There is one thread of control per data access. In this DSM implementation the entire data is replicated on each address space, and only parts of it are truly owned. The parts that are not owned are invalid and updated by a consistency protocol. Due to the replication strategy, the system is inefficient for intensive data applications.

Many researchers have explored the use of objects for writing parallel programs. One commonality for all these efforts, as well as ours, is to prove that high performance applications can benefit from object-oriented techniques such as abstraction and encapsulation. The challenge is to provide evidence on scalability for complex,

Table 10.2: Comparison of various object-oriented / based concurrent languages and systems.

Model vs. Criteria	**Data Model**	**Concurrency**	**Computation**	**Explicit/Implicit Parallelism**
Our Model	Distributed Memory	SPMD	Relaxed owner computes	Implicit
OMPC++	DSM	Shared Memory SPMD	SPMD	Implicit
EPEE	Distributed and shared	BSP (Block Synchronous)	NA	Explicit (API designer); Implicit(user)
Charm++	DSM	Message passing, RPC	MIMD[5]	Explicit
ICC++	DSM	Concurrent blocks & loops	NA	Explicit
Concert	DSM	Concurrent statements	Multi-threaded	Implicit (intra objects); Explicit (task parallelism)
pC++	DSM	HPF like	Owner computes	Explicit (intra object)
Mentat	Data-flow model	RMI	Data flow graph	Implicit
HPC++	DSM	HPF loop directives and multi-threading	SPMD	Explicit

high performance applications.

OMPC++ [101] is an implementation of DSM using OpenC++ reflection. The focus is on achieving portability for parallel execution environments. A parallel multithreaded shared-memory C++ program is transformed to execute on various environments (e.g. message passing). Our model has a local memory view, as opposed to a shared memory view. Also, our consistency scheme is based on an update, rather than an invalidation protocol. Both the local memory view and the update scheme are more efficient. Yet, the shared memory view is more flexible.

EPEE [64, 92] is an Eiffel parallel execution environment aimed at offering a high level API developer a platform for incorporating new components as common behavioral patterns are detected. The final goal is to hide parallelism, while hiding data partitioning is optional. This idea is similar to the approach taken by COPS [78]. Both of these systems differ fundamentally from ours in that they are relying on the developer to find and incorporate new solutions to hard problems such as hiding parallelism from the user. Instead we present a simple, elegant solution to support efficiently implicit data parallelism for irregular, data intensive applications.

Charm++ [65] is a concurrent object-oriented system based on C++. Parallelism is explicitly expressed as an extension to the C++ language. Parallel processes (chares) communicate through message objects that are explicitly packed / unpacked by user. The system also features special shared objects and remote accesses through remote procedure calls. Having the user explicitly pack and unpack messages is exactly what we are trying to avoid.

ICC++ [35] is a C++ dialect for high performance parallel computing. It supports expression of irregular and fine-grain concurrency. The data model is based in intra-object concurrency, where concurrent method calls can be made to an object. The system guarantees that the method calls do not interrupt each other. But the programmer must reason about progress and deadlock. Collections allow distribution to be explicitly specified. Instead of fine-grain HPF like parallelism, we support coarse-grain, loosely synchronous concurrency.

Concert [34] is a parallel programming environment based on ICC++. The concurrency model is HPF like, with one thread for each access to an object. Optimizations are added, such as static object in-lining, method in-lining and access region expansion. Dynamic optimizations such as pointer alignment and view caching are also supported.

pC++ [24] is an object parallel language based on HPF for managing data and computations. Template (in the HPF sense, not in the C++) and alignment directives are supported. Similar to the HPF style, this model supports regular applications and is based on regular partitioning functions (clock, cyclic, whole).

The Mentat [50] system provides data-driven support for object-oriented programming. The idea is to support a data-flow graph computation model in which nodes are actors and arcs are data dependences. The programmer must specify the classes whose member functions are sufficiently computationally complex to warrant parallel execution. The data-flow model is enhanced to support larger granularity and a dynamic topology. Parallelism is supported through having multiple actors execution on multiple processors.

HPC++ [48] is based on PTSL (Parallel Standard Template Library) parallel extension of STL, Java style thread class for shared-memory architectures, and HPF like directives for loop level parallelism. A context is a virtual address space on a node. Parallelism within a context is loop level parallelism. Parallelism across multiple contexts allows one thread of execution on each context. Low level synchronization primitives (semaphores, barriers, etc.) coexist with high level collection and iterators.

Table 10.2 classifies the approaches according to the data model, support for concurrency, computation model and the visibility of parallelism (implicit / explicit).

There are two major differences between our model and the existing object-oriented models. One is that we take a fundamentally different approach in increasing usability for the distributed-memory programming model. That is, we avoid paying the price in scalability by implementing a software shared-memory layer on top of a physically distributed memory. We obtain the same usability effect (sequential programming model) by exposing the user to a local memory view only. Also, we avoid exposing the user to any of the low-level concurrency aspects such as synchronization to shared-memory access or explicit communication.

We place a strong emphasis on scalability. Our data mapping and consistency schemes maximize data locality while minimizing communication. None of the abovementioned systems employs such techniques. However, our model may be less general than other models by not supporting task parallelism.

10.8 Summary

In this chapter we have presented a scalable concurrent object model for data intensive applications with complex layout. We have introduced the set data abstraction as the key to exploiting coarse-grain parallelism. Moreover, we have described the distributed set abstraction as a means for the user to indicate large data that should be distributed across multiple address spaces. We have shown how these abstractions enable us to exploit the *logical data locality* in the context of recursive data abstractions, as opposed to *physical data locality* inherent in linear, multidimensional array representations.

We have presented an original data mapping scheme that uses the potential number of remote references as minimizing criteria in conjunction with a general graph partitioning algorithm. We have explained how our mapping scheme that preserves locality is used by the consistency algorithm to generate the minimum number of messages. We have further optimized our consistency algorithm by relaxing the owner computes rule.

We have shown that our model results in high usability by exposing the user to an “almost sequential” programming model. Thus, we have presented programming examples that support our hypothesis on programmability from Chapter 6, $H2$. We have shown how our approach retains the scalability of the distributed-memory model by avoiding implementation of a virtually shared memory view.

Our mapping scheme relies on compile-time estimates of data accesses to potentially remote locations. Dynamic remapping schemes are usually based on the runtime information about remote references. Dynamic remapping can be expensive. However, for the worst-case scenario applications with many data accesses impossible to process, re-mapping is desirable. Even though this is rarely the case in practice, we plan to experiment with dynamic remapping when the initial partitioning exhibits poor locality and results in high traffic (communication).

The benefits of our approach come at the cost of applicability. That is, our concurrency model does not accommodate task parallelism. We intend to explore the issue of integrating task level parallelism in the concurrency model. Since we want to preserve the scalability of the concurrent object, we need more insight into the behaviour of our system for different classes of applications.

Chapter 11

A Prototype System for Implicit Concurrency

Parts of this chapter were published as conference papers [39, 40].

11.1 Introduction

This chapter describes a prototype system that achieves implicit concurrency for numerical applications. The focus is on providing a general framework that can be used by practitioners in the application domain to implement their numerical algorithms without being concerned with distributed computing aspects, such as data partitioning and communication generation. Efficiency is an important requirement for scientific numerical applications. We focus on the problem of concurrent Finite Element Method (FEM) solution of Partial Differential Equations (PDEs) for general (unstructured) meshes. Numerical abstractions and algorithms are not addressed in the implementation of the prototype system.

Parallel computing has been employed extensively in the scientific computing field in an explicit manner. Most of the parallel scientific applications use the Fortran language in conjunction with the message passing paradigm (MPI[1]) to specify the decomposition, mapping, communication and synchronization. Reuse has been explored in its incipient phase as function libraries. Even though Fortran libraries account for some reuse, Fortran applications are hardly extendible. As a consequence, despite their similar structure, most of the parallel applications are re-designed from scratch. Given that the process of writing distributed-memory applications is complex and error-prone, this means low productivity.

As stated, our goal is to abstract away from the user the distributed computing aspects and thus give the user the illusion of a sequential programming model.

With our approach we take advantage of the application specific features to automate the parallelization process. We separate the user data (numerical application specific data) from the parallelization algorithm. Therefore, we capture the concurrency infrastructure for the class of applications at hand (PDE solvers) and dynamically use

[1]http://www.mpi-forum.org/

it during the user solution process. We use generic programming techniques in order to couple user data and the workload partitions for the transparent distributed solution process.

In the remainder of the chapter we will refer to the FEM solution process, since we treat the FD case as a particular case of the general solution process. The **key features** of FEM applications are:

- The applications are data parallel and loosely synchronous. Domain decomposition is a technique used to break down the physical domain into smaller sub-domains that can be treated separately. With this method data on the border between sub-domains is logically connected with data from other sub-domains. In a distributed memory setting this means that data residing on remote processors are needed locally. The computation steps consist of independent local computations followed by communication.

- The physical domain is described by a geometrical, discretized structure. This usually translates into nodes, elements or faces (edges, i.e. the connection between the elements). Any user application specific abstractions (matrices, vectors, etc.) or attributes (e.g. pressure, temperature) are indexed according to the nodes, elements or faces. The numerical computation consists mainly of iterations over the entities (nodes, elements, edges).

- The applications are inherently dynamic: experimentation with different geometrical data structures (degrees of freedom, element shapes) or different numerical algorithms (time discretization schemes, iteration schemes, etc.) is at the core of the physical phenomena simulation (numerical applications).

This chapter describes the prototype system used to implement the techniques and the numerical applications presented in this thesis. Thus, it describes the design and the implementation of the component-based framework [62] that captures the concurrency infrastructure for dynamic, distributed numerical applications. Furthermore, it discusses the main design choices, and it motivates our approach. The system uses the techniques described throughout this thesis to account for efficiency.

The remainder of the chapter is organized as follows: Section 11.2 gives an overview of the system. Section 11.3 describes the component-based framework implementation. It also discusses the design rationale that drove our decisions. Section 11.4 overviews the existing approaches to the problem we attempt to solve (transparent concurrency) and motivates our approach. Section 11.5 concludes the chapter.

11.2 An Overview of the System

In this section we describe the system for transparent concurrency of the distributed solution of the PDEs from the user perspective. In this perspective, we emphasize the following requirements for our system:

- Applicability - the class of the applications we address is the parallelization for the general solution of PDEs (FEM, FD etc.). PDEs are at the core of most of engineering and natural science problems. With such a large class of applications, our system is most likely to be highly relevant for a large applied research community.

- Usability - we expose the user to a small, well-tested, well-documented component set, together with the control thread (the way the components play together) that can be easily learned and used.
- Extensibility - the user should be able to use our system in conjunction with his/her own data and algorithms in order to complete the solution process.
- Efficiency - last, but not least, efficiency is an important requirement of the scientific applications. Our architectural approach is driven by efficiency as well.

In succeeding at meeting our requirements, we have accounted for the following:

- To achieve large applicability, we only focus on capturing the concurrency infrastructure, that is the load-balanced data distribution, finding the data that needs to be communicated (i.e. computing the communication patterns), knowing when the communication takes place. We only employ a high-level mathematical interface in the form of the geometrical data description for the discretized physical domain. Our solution is general and it does not involve any algorithmic or discrete mathematics interface. With our solution, any existing computational kernels (e.g. BLAS [44], Lapack [12]) or numerical library (e.g. Diffpack [70]) can be employed by the user in the solution process.
- Most of the existing numerical libraries for the distributed solution of PDEs are still low-level and complex (e.g PETSc[2]). By low-level we mean that the user is still aware of the data distribution and communication aspects, as well as many other low-level aspects (renumbering, data access offset, etc. [98]). By complex we mean that the libraries provide a rich, mixed functionality. Part of the functionality accounts for numerical abstractions (linear algebra solvers, time discretization schemes, etc.). More general functionality (sometimes duplicated by each library, in its own "philosophy") such as geometrical data representation, local element to global mesh renumbering, etc. is mixed in also. Therefore, the libraries become large, hard to use and learn. We separate the parallel solution process and the application specific data and algorithm. We achieve high usability by designing a small set of well-tested, well-documented components, with a narrowly focused functionality.
- Our solution is high-level and reusable through the use of encapsulation. We hide the details of concurrency from the user, while achieving reuse *as-is* of the entire concurrency infrastructure. We also hide the tedious details of the geometrical data representation from the user. At the same time, the component-based design accounts for the reuse of any existing (numerical) software artifacts.
- Our solution is efficient because we employ a *truly distributed object model*. That is, there is no notion in our model of a global name or address space or remote

[2]http://www-fp.mcs.anl.gov/petsc/

invocations. The distributed objects are loosely synchronous. Communication only takes place at particular times during the application life time. We optimize communication by knowing when communication takes place and aggregating data into large messages.

The main concepts/steps involved in the distributed solution of the PDEs are:

1. Geometric data representation.

2. Data structure creation.

3. Off-processor data updates.

4. Local numerical computation.

Our system accounts for the first three phases, while the user is responsible for providing the numerical algorithm for a particular PDE.

11.2.1 Geometrical Data Representation

Geometrical data representation is one of the hard aspects of scientific applications that use general meshes. Even though the applications use similar geometries, many different representations coexist in practice, making existing scientific codes hard to understand, modify, maintain and extend. We isolate the geometrical data representation in a component with a well-defined interface for accessing all the needed geometrical attributes. At the system level, the geometrical data representation can be easily replaced, without affecting the system functionality, or requiring any modifications in any other system modules.

The user specifies the structure of the input domain as an input file, called *mesh* or *grid*, for the system. The user also specifies the number of the processors available in his/her system. The file describing the domain has a prescribed, well-documented format. Such files are usually obtained by means of tools called *mesh generators*[3].

The system reads the data from the file into the internal components. Different element shapes used for the discretization of the input domain can be specified in the mesh file.

The system uses a load-balanced partitioning algorithm (as provided by *METIS*[4]) for breaking down the input mesh structure data into smaller regions. All the details related to the geometrical data representation are encapsulated by our system component. The user gets access to all geometrical data aspects through our component interface, which is the *Subdomain*. At the system level the geometrical data representation can easily be replaced without affecting the system functionality, or requiring any modifications in any other system modules.

[3]An example of such a tool can be found at http://www.sfb013.uni-linz.ac.at/ joachim/netgen/

[4]http://www-users.cs.umn.edu/ karypis/metis/

11.2.2 Data Structure Creation

The system creates a number of regions from the input data structure equal to the number of the processors available. Then it associates each data region with a process[5], transparently for the user. The *internal boundary* geometrical mesh data is duplicated on each process, so that the system has access, locally, to all the off-processor data needed during one computation.

The user has access to the geometrical data local to one processor through the *Subdomain* component. The component presents the user with a uniform view of the geometrical data structure that can be employed in a sequential programming model for implementing a numerical algorithm (solver). All the distributed computing aspects that the component incorporates are invisible to the user.

The user has to subclass a system provided component *UserData* for defining any attribute (e.g. pressure, temperature) or data abstraction on the mesh structure which is involved in the computation of the final result. The user provides the concrete interface for storing and retrieving a user defined data item to/from a specific mesh location (element, node, etc.).

11.2.3 Off-processor Data Updates

The concurrency structure of the applications we address (the solution of PDEs) consists of independent local computation, followed by communication phases. Therefore, they are loosely synchronous [31]. In order to automatically provide for the off-processor data updates, we need to know when and what to communicate. That is, we need to know the communication patterns and when to generate communication. Our system computes the communication patterns, but it is the user that explicitly invokes the update phase. Then, the system performs the updates transparently.

Each region associated with a process is stored in the *Subdomain* component. The "invisible" part of this component makes use of the local data in order to account for the distributed computing part. The component computes the communication patterns, creates the communication data container objects and maintains the global data consistency transparent for the user. The user has to call a system-provided generic function, *Update*, which makes sure that the user-defined data is globally consistent.

11.2.4 Local Numerical Computation

Our system treats the user data differently, depending on the *dependence* attribute:

1. We call *dependent data* any defined user property that is the final result (i.e. the unknown for which the equations are solved, e.g. pressure, temperature, etc.), or updated by another *dependent data* item (e.g. some of the coefficient matrices computed based on the unknown).

2. We call *independent data* any user data that are not assigned the result of an expression computed based on *dependent data*.

[5]In our model a single process runs on a single processor.

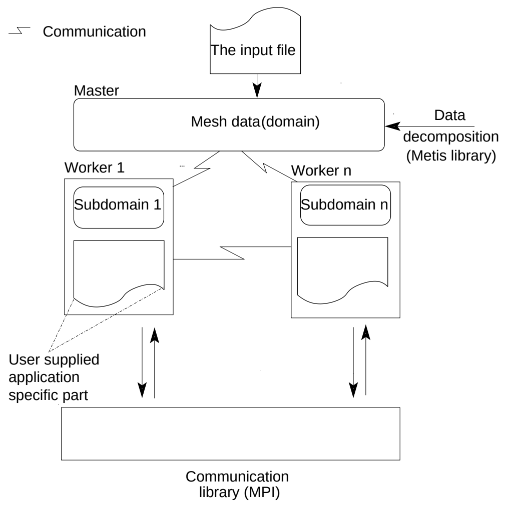

Communication
The input file
Master
Mesh data(domain)
Data decomposition (Metis library)
Worker 1
Worker n
Subdomain 1
Subdomain n
User supplied application specific part
Communication library (MPI)

Figure 11.1: The main building blocks of the system.

Figure 11.1 shows how the system supports the above tasks transparently to the user, and what the user contribution is in completing the solution process. As shown in Figure 11.1, we employ a Master / Worker concurrency model. A master process is responsible for reading the domain data from an input file and distributing the subdomains to the workers. We use a hybrid master / worker and SPMD concurrency model. In the SPMD model all the worker processes execute a similar task on the local domain (data).

In Figure 11.1 we associate each subdomain with a unique process. A subdomain has *communication capabilities*, that is, it “knows” how to “pack / unpack itself” and send / receive its data. Inside the system there are other components with similar features. We call these components *active components*. In contrast, some other components, *passive* components, capture only the structure data, and / or the algorithm associated with it, without having any communication capabilities[6].

The user sees only the “passive interface” of the *Subdomain*. This will allow the user to manipulate the appropriate geometrical data. We hide the geometrical data representation from the user. The system instantiates and “computes” the “right” subdomain for each worker. The user acts only at a subdomain level, in a sequential fashion. The system replicates the user algorithm and data over all the workers. The workers communicate transparently.

In Figure 11.2 we show the difference between a code excerpt written in a “classical sequential manner”, and the same code excerpt enriched with our system functionality to look sequential, but execute distributed. We do not show the “classical concurrent model” for such an application (i.e. the MPI) version, since we assume

[6]The active components are distributed objects, while the passive components are sequential objects.

that its complexity is evident to the reader and the space would not allow for such an illustration.

```
Classical sequential model:
A FEM Poisson Solver using a tetrahedral mesh
 void main() {


  while(abs(p[n+1] - p[n]) < EPS) {
   ComputeB();
   ComputePressure();
  }
 }
 void ComputePressure() {
  for (i = 0; i < nverts; i++)
   P[i] = (A[i] - B[i])/C[i];




 void ComputeA() {
  ...
 }
 void ComputeC() {
  ...
 }
 void ComputeB() {
  for (i = 0; i < nverts; i++) {
   B[i] = 0;
   for (e = 0; e < nelems; e++) {
    vol = Vol[e];
    for (j = 0; j < NVE; j++) {
     jnode = El2MeshNo(e, j, NVE);
     p = P[jnode];
     for (k = 0; k < NVE - 1; k++) {
      knode = El2MeshNo(e, (j+k+1)%NVE, NVE);


      c1jk = ScalarProduct(a[j], a[k])/9*vol;
      B[knode] += c1jk*p;


     }
    }
   }
  }
 }
```

```
Our non-classical sequential model:
A FEM Poisson Solver using a tetrahedral mesh
 void main() {
  P.Init();
  Update(P, loc_data);
  while(abs(p[n+1] - p[n]) < EPS) {
   ComputeB();
   ComputePressure();
  }
 }
 void ComputePressure() {
  for (i = 0; i < loc_data.GetNVerts(); i++) {
   temp = A[i] - B.GetAt(i)/C[i];
   P.SetAt(i,temp);
  }
  Update(P, loc_data);
 }
 void ComputeA() {
  ...
 }
 void ComputeC() {
  ...
 }
 void ComputeB() {
  B.Init;
  Update(B, loc_data);
  for (e = 0; e < loc_data.GetNElems(); e++) {
   vol = Vol[e];
   for (j = 0; j < loc_data.GetNve(); j++) {
    jnode = El2MeshNo(e, j, NVE);
    p = P.GetAt(jnode);
    for (k = 0; k < loc_data.GetNve() - 1; k++) {
     knode = El2MeshNo(e,
     (j+k+1)%loc_data.GetNve(),
     local_data.GetNve());
     c1jk = ScalarProduct(a[j], a[k])/9*vol;
     temp += c1jk*p;
     B.SetAt(knode, temp);
    }
   }
  }
  Update(B, loc_data);
 }
```

Figure 11.2: A comparison of two sequential programming models.

In Figure 11.2, we emphasize the difference between the two models by using grey for our model required modifications. We use black for the similar parts. It is easy to see that the data items `B` and `P` are candidates for the *dependent data*. Therefore, in the code excerpt above, these data specialize our component *UserData*. Also, the `loc_data` variable reflects the data the user sees, i.e. a *Subdomain*. On the other hand, the data items `A` and `C` are *independent*, and they do not require any modification to the sequential algorithm.

An important observation here is that the user is the one who has to "observe" the difference between the *dependent* and the *independent* data. With proper guidance provided by the system documentation and the user experience, this task is straight forward.

11.3 Discussion

We have implemented a prototype framework([62], [19]) to demonstrate our approach, using the C++ programming language [103]. We have used the *METIS* library for the load balanced (graph) partitioning of the irregular mesh. We use the Object Oriented Message Passing Interface (OOMPI[7]) library for communication. Figure 11.3 depicts a view of the prototype, using the UML [46] (Unified Modeling Language) formalism. For brevity we only show the key components and interfaces in the UML diagram.

Our design is based on a careful study of the application domain (PDE solvers) and their behavior. Our key observation is that the applications share a similar structure, with differences residing in the numerical problem parameters rather than the concurrency infrastructure. Therefore, by separating the parallel algorithm and geometrical data from the application specific parameters, data and algorithms, we were able to come up with a solution to automate the parallelization process.

In Figure 11.3 the *UserAlg* component is "the hook" for the user to "anchor" his / her computation into the framework. The user *subclasses* the abstract component *UserAlg*[8] and provides his / her main control flow which complements the control flow of the framework in completing a particular application. The user also hooks the data representation for his / her application here. As we have mentioned earlier(9.4.3), the data can be *independent* or *dependent*. Furthermore, it can be user-defined, or components imported by the user from other libraries. In this case, we say that the user extends the framework through *composition*. *Composition* is the first mechanism used to extend the framework. The user *dependent data* has to be defined by subclassing the framework container component *UserData*. *Subclassing* is the second mechanism used to extend the framework.

The user has access to the geometrical data for a subdomain through the interface of the *Subdomain* component. The user algorithm component is parameterized with the *Subdomain* component. The *Subdomain* is created by the framework, so that the user has a contiguous, consistent view of his / her data. The user writes his / her application for a single *Subdomain*. In a SPMD fashion, the framework instantiates a number of *workers* which replicate the user provided computation for all the subdomains. A worker process is modeled by a *Worker* component, which is not visible for the user.

[7]http: / / www.mpi.nd.edu / research / oompi /

[8]In this particular implementation, components and classes are the same.

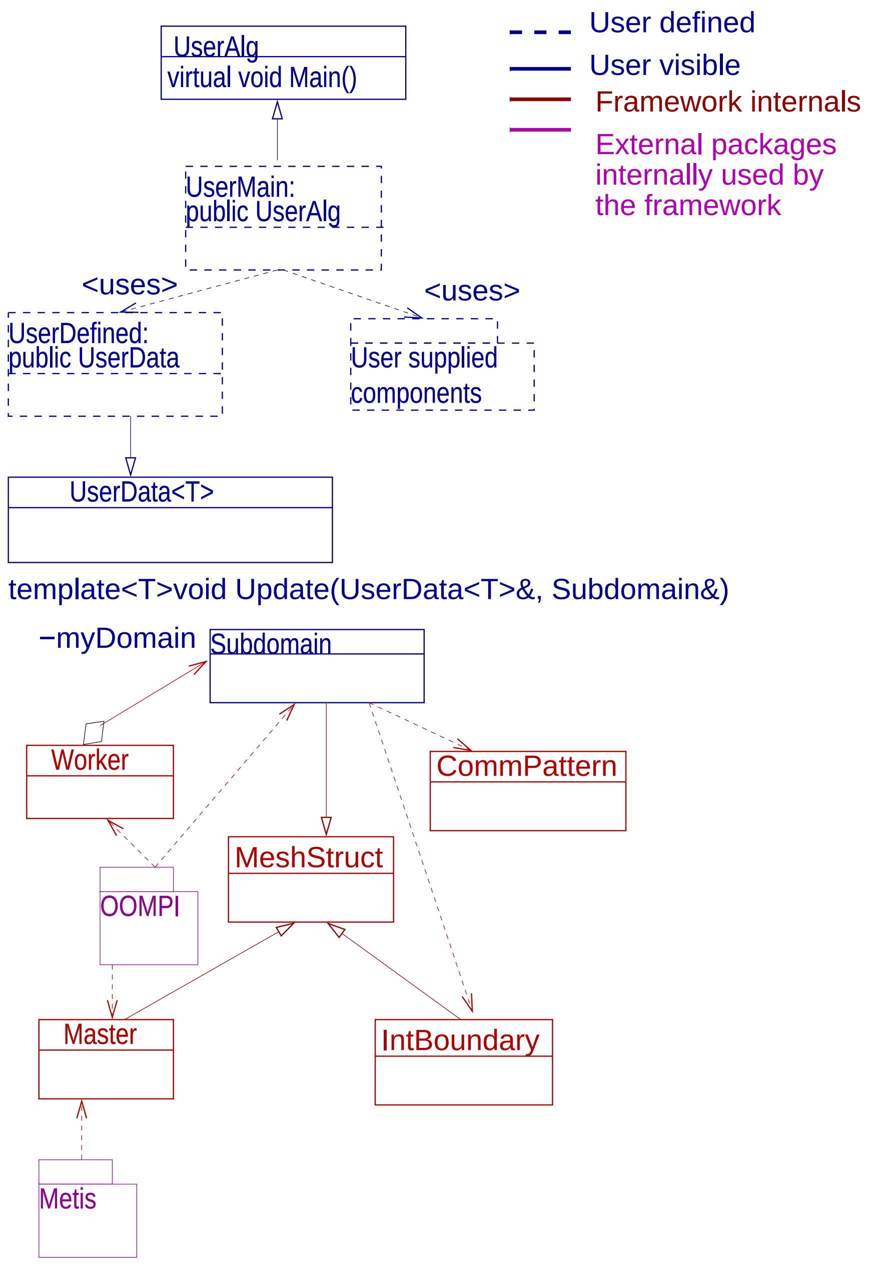


Figure 11.3: The infrastructure of the framework.

The component receives its work from a master process, in a Master / workers fashion. The *Master* component reads the input data as provided by the user (the discretized physical domain) and breaks it down into smaller partitions that it sends to the workers. Based on their local data provided by the *Subdomain* component, a *worker* sets up the communication patterns in cooperation with other workers. The generic function *Update* uses the communication patterns for a *Subdomain* and the container component *UserData* to automatically generate communication every time the user calls the function, after updating a user *dependent data* item.

The component-based design for our framework we have chosen due to our *constructional* approach, i.e. we construct part of the solution process. The *generative* approach would be to analyze an existing solution and generate a new one (e.g. compiler-driven parallelization). The compiler techniques (data flow analysis, dynamic analysis) are limited to the regular applications, that use simple data layout. Frameworks fulfill best our need for the user to be able to *plug in* his / her own data representations and algorithms, i.e. to provide the remaining part of the solution process.

The benefits of our architecture choice range from the ability to "automate" the parallelization process for a general class of applications (regular and irregular applications) to the *data locality* and *communication optimization*, the data encapsulation concept naturally provides. These benefits come at the cost of whatever makes object-oriented languages slow: abstraction penalty, dynamic binding penalty and inheritance penalty. At last, but not least, the (object-oriented) framework has been the most effective route to reuse [86].

Genericity is an important aspect of our design. Because the parallel structure of the numerical applications we refer to can be expressed independently of the representation, the concurrency infrastructure is based on the concept of generic programming. We use generic programming to be able automatically generate communication for user data. We use containers as placeholders for later defined user data to be able to pack / unpack and send / receive the data. This solution enables us to free the user from any distributed computing aspects, from data distribution, and data consistency, to data communication.

The concurrency model we use is the hybrid master / workers and SPMD. SPMD is the widely-used concurrency model for numerical applications. We use a special process, a master, to evenly divide the data and send the work to the worker processes. This way the workers will have approximately the same amount of work load, and the computation will be balanced.

The validation of the framework has two aspects. At the *usability* level our system is available to students and researchers for experimenting. We have implemented a test suite to test the functionality of our framework as well as for documentation purposes.

From the *efficiency* point of view, we are interested in the applications running time and speedup. We have implemented few applications using the framework on clusters of PCs, NOWs and NUMA architectures. The results show that our approach is efficient.

```
Compilers - regular case:
Example:
  for i = 1, N
    x[i] = x[i-3]
1.Gather data dependence info
  (i.e.  {[+,3],[0,3]})
2.Data and computation decomposition
  (i.e.  the iteration space).
3.Code (communication) generation
  (based on data flow information
  and owner-compute rule.)
```

```
Run-time support - irregular case:
Example:
  for i = 1, N
    x[i] = x[i] + y[ia(i)]
1.Build the communication schedule
  (i.e.  a translation table lists the
  home processor and the local address
  for each array element)
2.Move the data based on the schedule
The transformed code:
  DS = Schedule(...)
  Call DataMove(y, DS)
  for i = 1, Nlocal
    x[i] = x[i] + y[iaLocal(i)]
```

Figure 11.4: Compile-time and run-time support for parallelization.

11.4 Related Work

Several approaches exist to support the parallelization of scientific applications. The spectrum of the approaches lies between manual parallelization, at the one end, to the fully automatic parallelization, at the other end. Despite its inconveniences, manual parallelization is still the most popular approach to writing concurrent scientific applications, due to its efficiency and taylorability. On the other hand, writing concurrent scientific applications is an error-prone, complex task. Automatic parallelization is one way to tackle the complexity and the reliability of concurrent applications. Research into parallelizing compilers for scientific applications has been successful for Fortran applications with simple data representations and regular access patterns [105], [5]. Compile-time analysis cannot handle arbitrary data access patterns that depend on run-time information. Run-time analysis has been used to address the issue of compiling concurrent loop nests in the presence of complicated array references and irregularly distributed arrays [111]. However, these approaches are limited to loop level parallelism and simple data layouts (arrays) in the context of Fortran languages. The code excerpt in Figure 11.4 exemplifies the applicability of the compiler support for parallelization.

The loop-level parallelism is fine-grain and results in a lot of communication. Also, the compiler support can only handle arbitrary access patterns in the context of simple data layouts, such as arrays.

Object-oriented / -based distributed systems that support data parallel applications have been proposed [31], [54]. Either they do not support complex data representations with general distribution, or many of the concurrency aspects are still visible to the user. A complete survey of models and languages for parallel computation can be found in [97]. We will only refer to the object-oriented models. In [97] they are classified into *external* and *internal* models according to whether the parallelism is orthogonal to the object model, or integrated with the object model. We are interested in the internal object models, because these are closely related to data parallelism, since the language appears sequentially at the top level. Existing approaches require communication to be explicit, but reduce some of the burden of the synchronization associated with it. Our model aims at hiding communication and synchronization from the user. With our prototype implementation we succeed at achieving this to the extent that the synchronization phase has to be triggered by the user. It is possible to extend our system implementation to detect and trigger the synchronization phase when necessary automatically.

Chaos++ [31] is a library that provides support for distributed arrays and distributed pointer-based data structures. The library can be used directly by the application programmers to parallelize applications with adaptive and / or irregular data access patterns. The object model is based on *mobile* and *globally addressable* objects. A *mobile object* is an object that knows how to pack and unpack itself to and from a message buffer. A globally *addressable object* is an object that it is assigned to one processor, but allows copies to reside on other processors (referred to as *ghost objects*).

The first problem with this approach is that the user is expected to provide implementations of the *pack* and *unpack* functions (that support *deep copy*) when subclassing the *mobile* component. On the one hand, the packing and unpacking tasks are low-level operations that expose the user to many of the concurrency aspects (what to pack, when to pack, where to unpack, what to unpack). On the other hand, more *mobile objects* may contain a pointer to same-sub-objects. The user has to make sure that only one copy of a sub-object exists at any point during the program execution. The use of *globally addressable objects* can alleviate some of these problems. The *global object* is intended to support the global pointer concept. The main problem is that the contents of the ghost objects are updated by explicit calls to data exchange routines. Our approach is different from this one by not mixing the concurrency aspects at the user level. We do not let the user see the internals of the active, distributed components of our model. Also, we do not need naively to transport the entire content of an object for every communication. We only update the parts of the objects that are needed for subsequent computations and avoid the overhead associated with communicating entire objects.

In contrast with the *generative approach*, such as compiler support for parallelization we use a *constructional approach*. Therefore, we construct part of the solution. We want to achieve the “illusion” of a sequential programming environment, as opposed to transforming sequential programs to run in parallel. Our approach is due to the limited compiler support for dynamic analysis and efficiency considerations. Our approach is based on capturing the concurrency infrastructure of a class of applications and reusing it for every new application. This idea is similar to the notion of *skeletons*, ready-made building blocks [97], or abstractions characteristic to a class of applications. In our case we are closer to *algorithmic skeletons*, those that encapsulate structure. [25] explore the approach of algorithmic skeletons or common parallelization patterns for another class of applications, i.e. adaptive multigrid methods. Moreover, [25] list a series of requirements for a language supporting algorithmic skeletons, among which data access control and polymorphism. The authors introduce the notion of *parallel abstract data type* (PADT) in order to account for the required features. Furthermore, the host language used for experimentation is an imperative language, i.e.

the C programming language. We argue that object-oriented models and generic programming naturally fulfill the requirements for implementing algorithmic skeletons. Therefore, we concentrate on an efficient data parallel object model suitable for high performance, parallel applications.

11.5 Summary

In this chapter we have presented a new approach towards the parallelization of scientific codes, i.e. a *constructional approach.* In contrast to the *generative* approach (e.g.compiler-driven parallelization), we construct part of the solution, instead of generating a new solution based on an existing one. We use a component-based architecture in order to be able to allow the user to build on our concurrency infrastructure. With our approach we get closer to the ideal goal of not having the user dealing with the concurrency at all, without restricting the generality of the application class. Therefore, we are able to handle the distributed solution of PDEs for general geometries (meshes).

Given that efficiency is an important constraint of the class of the applications we address, we show how a "truly distributed" component model can alleviate the efficiency problems of the object-oriented middleware (e.g. Java RMI, CORBA, DCOM).

This research explores the appropriateness of objects in conjunction with concurrency (a much desired association [82]) in the context of high performance computing. High performance scientific computing is known as a community traditionally reluctant to object-oriented techniques because of the poor performance implementations of object-oriented languages and systems show.

The work presented in this thesis provides evidence that our approach is scalable. We intended our system architecture for a cheap, flexible distributed computing platform, consisting of clusters of Linux PCs or NOWs. With a scalable approach, a potentially "unlimited" number of computers can be used for gaining computational power.

Part IV

Synopsis

Chapter 12

Summary and Results

12.1 Discussion

This work has explored a high-level programming model for scientific applications, in conjunction with lower level enabling techniques (e.g. data distribution and consistency). Moreover, it was pointed out a need for modern software technology and software development methodology in a domain that traditionally took on the technical dimension, rather than the methodological aspects. That is, the domain concerned itself with the underlying architectural issues and low-level system software for parallel computing, rather than software tools and programming environments for application writers. This was mainly due to the non-traditional nature of both the application domain (scientific computing) and the computing style (concurrent computing).

The shift in high performance architectures for delivering high computation power towards commodity-oriented cluster computing forces a shift in trends towards both system software and programming environments for scientific computing. The industry is already moving in this direction and in the past years several projects took on migrating legacy software to modern techniques (e.g. object-oriented wrappers, component-based development, etc. [13, 14, 19, 27, 59]).

It is still unclear to what degree performance losses due to the software layer (system and programming environment) are unacceptable. Our thesis is that the main gain in performance comes from the architectural and algorithmic scalability. Thus, the software layer may negatively affect performance, but not in a more dramatic way than it does in a traditional computing setting (i.e. sequential computing).

The work in this thesis explores these issues of where the gain in performance comes from and what an acceptable compromise between effectiveness and programmability is. Thus, the following aspects need to be explored:

- Evaluate the performance degradation due to the software environment. We have not directly attacked this issue. We point out that answering the question in the context of present software methodology for realistic applications is not feasible. While numerous applications and benchmarks exist for Fortran languages, porting those to different programming environments for the purpose of comparison is not sufficiently attractive to invest effort into this, neither for the research community, nor of for the industry. We assume that such loss is not crucial for the overall application performance [108, 109].

- Discover and exploit parallelism inherent in the applications. We have tried to show that the parallelism inherent in the applications and the parallelism exposed under the constraints of an architecture-dictated model are different. Thus, starting with the system platform and exploiting parallelism in a "bottom-up" approach that incrementally augments the trivial parallelism at instruction level, and loop level with non-trivial features is limited by fixed assumptions. These are the array language (Fortran), partitioning function (regular) and parallelization grain (fine). We have pointed out that a "top-down" approach, starting with the application knowledge and devising the concurrency model according to the application features invalidates some of the fixed assumptions. Thus, the following mismatches occur when trying to exploit parallelism:

  – Regular partitioning does not fully reflect and model data locality in applications.

  – The Fortran language is not rich and flexible enough to express the common data representations in scientific applications.

  – Static program analysis is limited to discovering and exploiting parallelism across the entire program.

  Thus this thesis has investigated:

  – A general distribution and mapping strategy, together with a consistency scheme to enable a general layout and automatic consistency for the coarse-grain data parallel applications.

  – The power of expressiveness for generic, dynamic programming (in C++) to enable more flexible data representations and subsequent exploitation of parallelism.

  – The power of run-time techniques to complement compiler analysis in discovering and exploiting parallelism.

There are several problems with our approach. First of all, we have tailored our solution to a specific class of concurrent applications, i.e. data parallel applications. Thus, the solution is not general enough to cover task parallelism. Moreover, we have tested our approach mostly for irregular problems, in particular for the FEM solution of PDEs for general geometries. Even though we have discussed the FD solution process as well and showed how our approach applies to regular applications that require neighbor communication, we did not complete our tests for these problems. Also, some of the techniques are not finalized in a complete system. We have implemented them by hand to produce experimental results. We have not directly addressed regular problems and dense matrix operations typical in linear algebra. One of the reasons is that there is an enormous amount of work in this direction. However, we would like to study more communication patterns typical for such computations and experiment with dynamic data remapping due to changing communication patterns at run-time.

The investigation in this thesis is far from complete. This is due to the non-traditional path taken and the departure from a different set of assumptions than traditional investigations in the same research area. Other work exploring the novel software development techniques differs from ours in that it only varied the high-level dimension (software development techniques) while it retained the other dimensions (data partitioning, mapping, fine-grain concurrency, etc.).

This thesis raises more questions than it answers. Many of the pieces linking various techniques presented here we assumed rather than completed in a system or a comprehensive analysis. However, research directions and future work will be presented in Chapter 13.

12.2 Validation

Chapter 6 introduced two main hypotheses for our approach. These are:

H1 *Effective execution of data parallel applications in distributed systems.* The assumption leading to this hypothesis is that the communication structures for different classes of numerical applications can be deduced based on knowledge of both the specific application and the program implementation, and the number of synchronization points in a program is small.

H2 *Increased programmability for distributed data parallel applications.* The assumption leading to this hypothesis is that the applications share similar structures and the most difficult tasks of data distribution and communication can be solved automatically by a system. Thus, the scientific programmer has to supply only the sequential, application specific part.

To validate our first hypothesis, the results presented throughout the thesis have shown good effectiveness for our approach. We have presented speedup (the absolute wall clock times for the measurements are available in the Appendices) measurements for the data layout and consistency schemes.

We have manually restructured the application input data to reflect the traversal order in the application and mapped it onto a graph. Then we have applied our general layout scheme to compute the mapping between the local and global address spaces and the space requirements to duplicate only the related data across the address spaces. We have measured the effectiveness of this technique for the implementation of the 3D Poisson FEM solver. The speedup results in Chapter 7 (see the corresponding wall clock times in the Appendices) show good effectiveness for our technique. We have only tested for the two small test problems (meshes) available, because we did not have access to larger test problems. Generating large, accurate 3D test problems (meshes) is a different research area and is not the focus of this work.

Chapter 8 discusses in detail the efficiency aspects of the consistency scheme that uses the layout presented in Chapter 7. It was shown that the space overhead incurred is small since we only keep the links between different partitions. This scheme works well for applications that require nearest neighbour communication since information about potentially remote accesses is available locally and no extra run-time overhead is required. However, it was pointed out that different communication patterns that cannot be symbolically computed at compile time would incur extra run-time overhead. The chapter presented speedup measurements for the applications that use nearest neighbour communication (iterative FEM solver). We have used an optimized consistency scheme that employs message aggregation in these measurements. The speedup results (together with the absolute wall clock times presented in the Appendices) show that the consistency scheme is effective, and it does not incur large run-time overhead.

To validate the second hypothesis, the programming examples using our model show that the hard aspects of distribution and concurrency are automatically taken care of and the user is exposed to a close to sequential programming model.

Chapter 9 has presented preliminary results on the usability of our programming environment for the rapid prototyping of distributed data parallel applications versus other distributed programming environments (MPI, CORBA and OOMPI). The investigation conducted for collecting the data is far too small to have statistical relevance. We have used a four-student project to asses the usability and effectiveness of our system in contrast with other programming environments and tools. The time dedicated to this project was already significant (two thirds of the entire semester work) and it shows that a comprehensive investigation would require ample resources (student years). However, the results give a sense of what the goal of our approach is and what the potential benefits are. That is, using our programming environment was the easiest of all with respect to learning time and programming effort. Moreover, our system showed good effectiveness. That is, it showed comparable speedup with the manual MPI implementation, despite the extra architectural layers it adds for automatic layout. However, the test problem was simple and it did not reflect the full functionality of our system. Moreover, it did not involve the consistency scheme and thus, the measurements do not reflect the full run-time behaviour.

Chapter 10 has introduced a structured notation for irregular data structures most commonly used in data parallel applications. We have shown how recursive sets can be used to express graphs, general geometries, indirection arrays used in sparse codes and even regular arrays. Moreover, we have shown how the layout and consistency schemes presented in Chapters 7 and 8 can be used to implement inter-object consistency for distributed recursive sets. We have presented examples of the resulting programming model. We have also presented results of applications on loosely coupled distributed platforms. These results are promising. However, more extensive experimentation is needed in this case as well.

The validation of our results is not complete. Yet, enough results were presented to show that we have good foundations for our research hypotheses and that our approach is promising. Further research and experimentation need to be conducted.

12.3 Main Results and Contributions

For a concurrency model to gain acceptance in the scientific computing community and generate impact, it is important that the application domain is carefully analyzed and the model meets the realistic application demands.

This thesis has tried to show that an object-based concurrency model can be equipped to posses several qualities needed to ensure effectiveness and programmability for data parallel applications. The main results and contributions of this thesis can be summarized as follows:

- A general data layout scheme that subsumes the existing schemes and applies to a larger class of applications than traditional schemes apply to. The abstract, mathematical formulations are given together with some practical examples of applications in Chapter 7.
- An efficient distributed data consistency algorithm that incurs little time and space overhead to automatically ensure consistency across data partitions in a distributed-memory environment. Experimental results show that numerical applications using the consistency scheme based on the general data layout strategy are scalable. Chapter 8 discusses in detail the efficiency aspects of the algorithm.
- An object-based concurrency model for loosely synchronous data parallel applications. The model, its usability and efficiency against other concurrent programming models were presented in Chapter 9. An inter-object concurrency scheme based on the *distributed set* abstraction is presented in Chapter 10. The *Set* data type generalizes the multi-dimensional array type and enables recursive data representations. This model collects the threads of the techniques and concepts developed in the previous chapters into a uniform concurrency model for data parallel loosely synchronous applications.
- A prototype implementation of an object-oriented framework for distributed scientific computations is presented in Chapter 11. The chapter presents the main features of the applications and the goal and rationales for designing such a system, together with the requirements. Then it discusses in detail the design and implementation of the system together with the various design decisions. This system was utilized for experimentation of all the techniques presented in this thesis and for the rapid prototyping of the test bed applications used to produce empirical data on effectiveness for various architectures.

Chapter 13

Further Work

13.1 Techniques

13.1.1 Data Layout

The general data layout presented in Chapter 7 assumes that any large, regular or irregular data can be mapped onto a graph that reflects the relation between data items in a traversal. While this may be possible, it is important to indicate a common data representation and its mapping onto a graph structure.

Thus, we must first demonstrate that such a common representation is sufficient to express the most relevant irregular / recursive data structures for a class of applications, i.e. graphs, trees, linked lists, etc. That is, when we link the data partitioning and mapping strategy for the *distributed set* type in Chapter 10, we need to prove that:

1. Sets are expressive enough to allow the flexible representation of graph, trees, linked lists, etc.

2. The linearization technique for sets works well for most of the applications. That is, based on the partitioning at level n, data at level $n-1$ can be restructured to ensure locality of reference at the same address space level.

In order to address the first issue, experimentation with various data representations for scientific or non-scientific applications that use large irregular data sets is needed. This requires further experimentation with various applications types.

The second issue is related to compiler optimizations to restructure data in order to improve data locality. Data and computation restructuring were traditionally employed by optimizing compilers to reduce memory latency by minimizing cache misses. Examples of such optimizations are loop tiling (block loop iterations such as to fit in the cache line) and loop interchange (interchange the loop dimensions accordingly, to exploit locality – e.g. for line, column major multi-dimensional array addressing for C and Fortran).

Linearization is an optimization which potentially improves locality for recursive data structures [76]. It has mainly been explored for compiler-based prefetching of data structures linked through pointers. We did not implement the linearization technique, but manually restructured data and fed it into a graph partitioning tool to experiment with recursive set partitioning to preserve locality. Implementing such a technique in a compiler is an important and interesting future research direction. This technique can enable the application of many optimizations traditionally employed in Fortran programs to dynamic programming environments.

13.1.2 Data Consistency

The data consistency algorithm described in Chapters 7 and 8 relies on the data layout to ensure the minimum data dependences across address spaces. Thus, given the application characteristics and the general data layout scheme, it is possible to show by measurement that the space and time overhead of the run-time consistency scheme is not significant. However, the algorithm is general and applicable to programs with more complex access patterns.

A future research direction is to capture the data access patterns for different types of scientific applications. This is hard in the presence of pointers. Let us assume that either through restricting the programming environment not to allow pointers, or through the advances in pointer analysis techniques for heap-allocated objects, pointers are not an obstacle anymore. Then, a dynamic remapping strategy should be employed to account for changing access patterns. Questions regarding the run-time efficiency of the dynamic remapping would need to be answered when pursuing this research direction.

13.2 Applications

When experimenting with techniques to improve efficiency of concurrent applications, it is important to classify and analyze the application domains and understand their various requirements. We have studied a restricted application domain, but we wanted to devise sufficiently general techniques in order to be applicable in a larger context. However, further experimentation may lead to changing requirements for efficient parallelization.

To prove the general applicability of the recursive set representation for various applications in banking, scientific computing, etc., the applications in these domains need to be analyzed and benchmarks need to be implemented using sets.

To experiment with applications with changing access patterns in the context of scientific programs is important for the study of the dynamic data remapping strategies. Such changing patterns do occur in some applications. For instance, the FEM formulation reduces a PDE to a linear algebra problem. Some of the linear algebra operations require direct methods that employ vector matrix computations and exhibit the behaviour typical for regular applications. Thus, it would be efficient to employ remapping to reflect this change in the application behaviour.

Part V

Appendices

Appendix A

Glossary

The main concepts are defined here to improve the readability of the thesis. The entries marked by * were produced using a compiled list of terms by Dongarra. The remaining entries were produced by the author either to define original concepts introduced by this research, or give general definitions of common terms in computer science and scientific computing.

***** Aliases: Two or more expressions that denote the same memory address. Pointers and procedures introduce aliases.

Affine transformation: A transformation that preserves lines and parallelism (maps parallel lines to parallel lines).

Automatic data layout : The process of having a system to partition and map data onto different threads of computation in concurrent processing.

Automatic parallelization : The process of having a system (usually the compiler and the runtime system) to perform the decomposition and the concurrent execution of a program.

***** Basic block: A sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end, without halt or the possibility of branching except at the end.

***** Cache: Small interface memory with better fetch speed than main memory. Cache memory is usually made transparent to the user. A reference to a given area of the main memory for one piece of data or instruction is usually closely followed by several additional references to that same area for other data or instructions.

***** Cache hit: A cache access that successfully finds the requested data.

***** Cache line: The unit in which data are fetched from memory to cache.

***** Cache miss: A cache access that fails to find the requested data. The cache must then be filled from main memory at the expense of time.

***** Coarse-grain parallelism: Parallel execution in which the amount of computation per task is significantly larger than the overhead and communication expended per task.

***** Compiler directives: Special keywords specified on a comment card, but recognized by a compiler as providing additional information from the user to use in optimizations.

Concurrency model : A conceptual specification of how a program is to be decomposed and run in parallel. Task parallelism, data parallelism, etc. are concurrency models. A specific concurrency model specifies how the computations are to be carried out in parallel and how synchronization is ensured.

***** Concurrent processing: Simultaneous execution of instructions by two or more processors connected through memory or network.

Control flow analysis: A process that identifies loops in the flow graph of a program.

***** Data dependence: The situation existing between two statements if one statement can store into a location that is later accessed by the other statement.

Data flow analysis: A process of collecting information about the way variables are used in a program.

***** Dependence analysis: An analysis (by compiler) that reveals which portions of a program depend on the prior completion of the other portions of the program.

***** Distributed memory: A form of storage in which each processor can access only a part of the total memory in the system and must explicitly communicate with other processors in the system to share data.

Distributed processing: Processing on a network of computers that do not share main memory.

Effectiveness : An abstract notation that denotes the set of metrics to establish the success of a technique. Speedup is a metric that can be used to indicate the effectiveness of a parallel algorithm / program.

***** Explicitly parallel: Language semantics or software constructs (e.g. library calls, components, etc.) that describe which computations are independent and can be performed in parallel.

***** Fine-grain parallelism: A type of parallelism where the amount of work per task is relatively small compared with the overhead necessary for communication and synchronization.

General data layout : A data layout for which the partitioning is specified by a general function, rather than a particular symbolic expression. The general function can have symbolic expressions as values or can be specified at each point in the input domain.

* Granularity: The size of the tasks to be executed in parallel. Fine granularity is illustrated by execution of statements or small loop iterations as separate processes; coarse granularity involves subroutines or sets of subroutines as separate processes. The greater the number of processes, the "finer" the granularity and the greater the overhead required to keep track of them.

* Implicitly parallel: Programming environment (language, library) that does not allow the user to explicitly describe which computations are independent and can be performed in parallel.

Indirection arrays : Arrays whose indices are array expressions themselves, rather than expressions of loop indices.

Inter-object concurrency : An object-based concurrency model in which concurrent processing of independent objects is synchronized such that the individual objects' states make up a global consistent state.

* Interprocessor communication: The passing of data and information among the processors of a multicomputer during the execution of a parallel program.

Intra-object concurrency : An object-based concurrency model in which concurrent accesses to a given object are synchronized such that the object is in a consistent state, independent of other objects.

Irregular applications : Applications that manipulate irregular data structures, such as graphs, trees, indirection arrays, etc..

* Load balancing: A strategy in which the longer tasks and the shorter tasks are carefully allocated to different processors to minimize synchronization costs or startup overhead.

Locality: The grouping together of objects to reduce the cost of memory accesses (e.g. remote accesses require communication).

Middleware : An intermediate level that sits between network operating systems and applications to support communication in distributed applications. Its main role is to alleviate the user from the low-level communication layer programming.

Multicomputer: A computing architecture consisting of several autonomous processors that do not share memory, but rather communicate by sending messages over a communication network.

Multiprocessor: A computing architecture consisting of several autonomous processors sharing a common primary memory.

* MIMD: A multiple-instruction stream / multiple-data stream architecture. Multiple instruction streams in MIMD machines are executed simultaneously. MIMD terminology is used more generally to refer to multiprocessor machines.

MPMD: A multiple-program / multiple-data computation structure. Multiple programs (processes) are executed simultaneously on multiple sets of data. MPMD terminology is used to refer to a high-level computation model, independent of the machine architecture.

Nearest neighbour communication : A simple communication pattern that involves the concurrent processes accessing the data directly related in a given representation (*neighbouring accesses*).

Object-based : An object model that is more general than an object-oriented model. That is, it supports encapsulation through objects, but it does not support all the features in an object-oriented model (i.e. inheritance).

Owner computes rule : A rule for parallel processing specifying that operations writing the same location must be executed by the same processor.

* Page: The smallest managed unit of a virtual memory system. The system maintains separate virtual-to-physical translation information for each page.

* Parallel computer: A computer that can perform multiple operations simultaneously, usually because multiple processors (that is, control units or ALUs) or memory units are available. Parallel computers containing a small number of processors (less that 50) are called multiprocessors; those with more than 1000 are often referred to as massively parallel computers.

* Parallel processing: Processing with more than one CPU on a single program simultaneously.

Parallelizing compilers : Special compilers that discover parallelism in a sequential program based on program analysis and generate the instructions for its parallel execution.

* Partitioning: Restructuring a program or a algorithm into independent computational segments. The goal is to have multiple CPUs work simultaneously on the independent computational segments.

Process: A logical processor that executes code sequentially and has its own state and data.

Programmability : An abstract notation that denotes the set of metrics (e.g. usability) to establish how easy to use a programming environment is. The number of lines of code can give a rough indication of programmability.

Regular applications : Applications that manipulate regular data structures, such as multi-dimensional arrays whose indices are not expressed using array expressions.

* Scalar processing: The execution of a program by using instructions that can operate only on a single pair of operands at a time (as opposed to parallel processing).

Scientific computing : A type of computing aimed at solving scientific problems which cannot otherwise be solved by humans, using a computer.

Serialization : A sequential scheduling of concurrent operations due to synchronization. Serialization limits parallelism.

SIMD: Single-instruction stream / multiple-data stream architecture.

* Speedup: The factor of performance improvement over pure scalar performance. The term is usually applied to performance of one CPU versus multiple CPUs or vector versus scalar processing. The reported numbers are misleading because of an inconsistency in reporting the speedup over an original or revised process running in scalar mode.

* Synchronization: An operation in which two or more processors exchange information to coordinate their activity.

* Thread: A lightweight or small-granularity process.

Update consistency scheme : A consistency scheme in which updates to data replicated across multiple address spaces are propagated to all the replicas.

Usability : A subjective metric indicating how easy to use a programming environment is. Depending on a particular model, specific metrics can be used. For instance, when programming with classes, the number of classes is a typical metric to indicate how easy it is to write an application using a specific environment (programming language, library, etc.).

Vector computers: A computer that uses many processors that simultaneously apply the same arithmetic operations to different data.

* Vector processing: A mode of a computer processing that acts on operands that are vectors or arrays. Supercomputers achieve speed through pipelined arithmetic units. Pipelining, when coupled with instructions designed to process all the elements of a vector rather than one data pair at a time, is known as vector processing.

* Virtual memory: A memory scheme that provides a programmer with a larger memory than that physically available on a computer. As data items are referenced by a program, the system assigns them to actual physical memory locations. Infrequently referenced items are transparently migrated to and from secondary storage – often disks. The collection of physical memory locations assigned to a program is its “working set”.

Appendix B

Experimental Data

The experimental results presented in this thesis have mainly concerned speedup measurements for the parallel execution. Here we include the absolute floating-point performance for the speedup results presented in this thesis. We give results for the two test problems described in an earlier chapter.

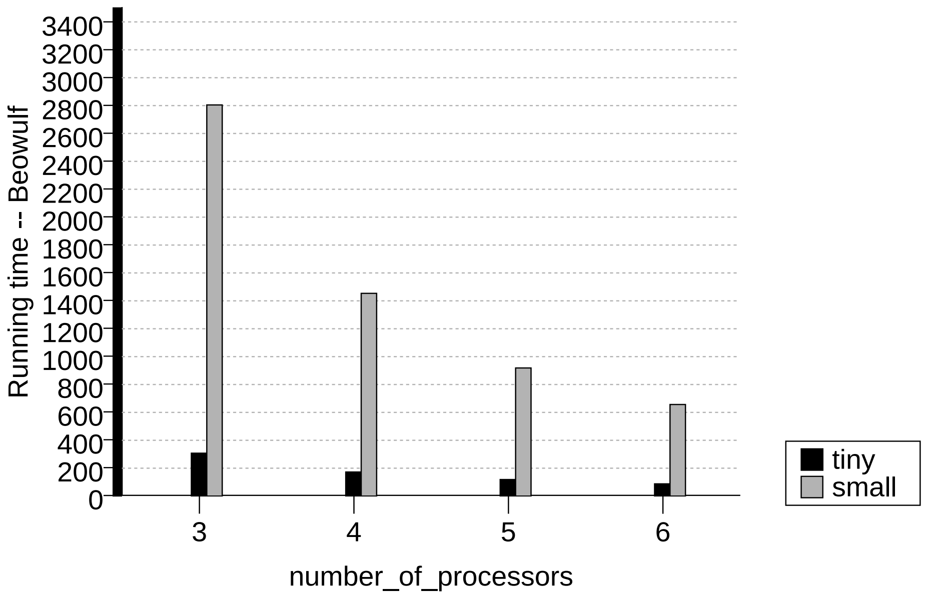


Figure B.1: Wall clock times for the Poisson solver on the Beowulf cluster.

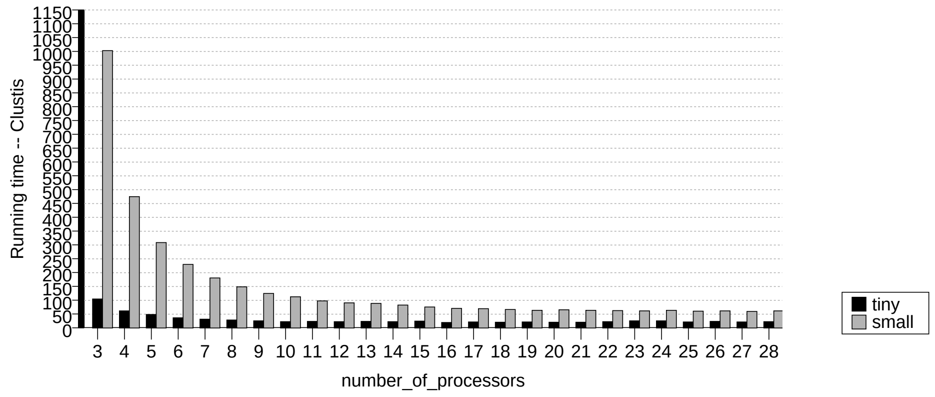


Figure B.2: Wall clock times for the Poisson solver on the ClustIS cluster.

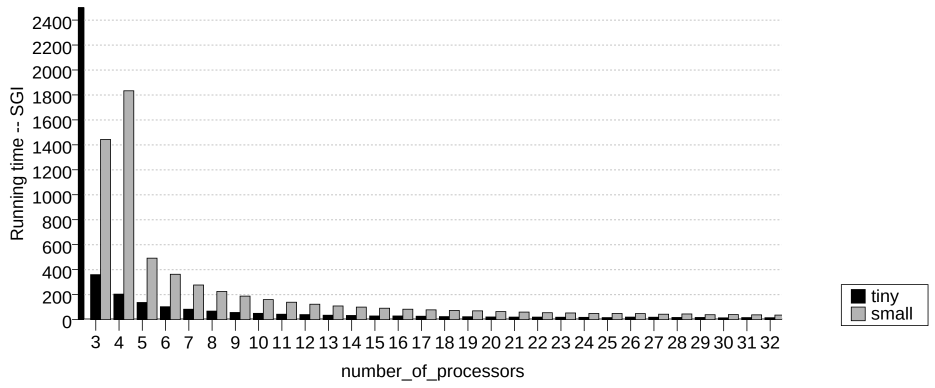


Figure B.3: Wall clock times for the Poisson solver on SGI Origin 3800 – up to 32 processors.

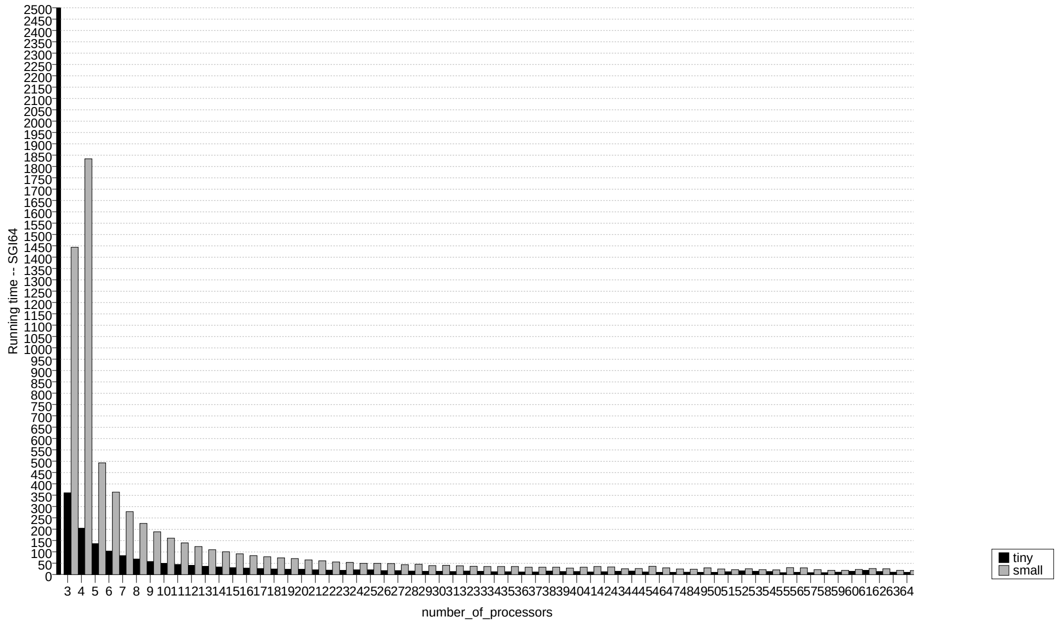


Figure B.4: Wall clock times for the Poisson solver on SGI Origin 3800 – up to 64 processors.

Index

Bibliography

[1] Vikram Adve, Alan Carle, Elana Granston, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, John Mellor-Crummey, Scott Warren, and Chau-Wen Tseng. Requirements for Data-Parallel Programming Environments. *IEEE Parallel and Distributed Technology: Systems and Applications*, 2(3):48–58, Fall 1994.

[2] Gul Agha. An overview of actor languages. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 58–67. ACM Press, 1986.

[3] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[4] Gagan Agrawal. Interprocedural communication and optimizations for message passing architectures. In *Proceedings of Frontiers 99*, pages 123–130, February 1999.

[5] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Amy W. Lim. An overview of a compiler for scalable parallel machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 253–272, Portland, OR, August 1993. ACM press.

[6] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of ACM SIGPLAN'93, Conference on Programming Languages Design and Implementation*, June 1993.

[7] Jerome Galtier amd Stephane Lanteri. On overlapping partitions. In *Proceedings of the 2000 International Conference on Parallel Processing*, pages 461–468. IEEE, 2000.

[8] Christiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[9] Corinne Ancourt and Francois Irigoin. Automatic Code Distribution. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[10] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[11] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[12] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, and D. Bischof, C. ans Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of Supercomputing '90*, pages 2–11, 1990.

[13] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. http://www.mcs.anl.gov/DOE2000. Work partially supported by the MICS Division of the U.S. Department of Energy through the DOE2000 Initiative.

[14] Susan Atlas, Subhankar Banerjee, Julian Cummings, Paul J. Hinker, M. Srikant, John V. W. Reynders, and Marydell Tholburn. Pooma: A high performance distributed simulation environment for scientific applications. In *Proceedings of SuperComputing 95*, 1995.

[15] Henri E. Bal and M. Frans Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, volume 28 of *SIGPLAN Notices*, pages 162–177, New York, NY, 1993. ACM Press.

[16] Henri E. Bal, M. Frans Kaashoek, Andrew S. Tanenbaum, and Jack Jansen. Replication Techniques for Speeding up Parallel Applications on Distributed Systems. *Concurrency Practice and Experience*, 4(5):337–355, August 1992.

[17] Heri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[18] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 213–223, Williamsburg, Virginia, United States, 1991. ACM Press.

[19] Federico Bassetti, Kei Davis, and Daniel Quinlan. Optimizations for parallel object-oriented frameworks. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object-Oriented Methods for Interoperable Scientific and Engineering Computing*, pages 303–312. In Proceedings of the 1998 SIAM Workshop.

[20] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker,

C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[21] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (USA), 1991.

[22] Aart J.C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Leiden University, Netherland, May 1996.

[23] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[24] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic Ideas for an object parallel language. *Scientific Programming*, 2(3), 1993.

[25] George Horatiu Botorog and Herbert Kuchen. Algorithmic skeletons for adaptive multigrid methods. In *Proceedings Irregular'95*, volume 980, pages 27–41. Springer LNCS, 1995.

[26] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[27] David L. Brown, William D. Henshaw, and Daniel J. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyon, editors, *Object-Oriented Methods for Interoperable Scientific and Engineering Computing*, pages 245–255. Proceedings of the 1998 SIAM Workshop, 1998.

[28] Dhruva R. Chakrabarti and Prithviraj Banerjee. Global optimization techniques for automatic parallelization of hybrid applications. In *Proceedings of the 15th International Conference on Supercomputing*, pages 166–180. ACM Press, 2001.

[29] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, Stanford University, March 1992.

[30] Mani Chandy, Ian Foster, Ken Kennedy, Charles Koelbel, and Chau-Wen Iseng. Integrated Support for Task and Data Parallelism. *The International Journal of Supercomputer Applications*, 8(2):80–98, Summer 1994.

[31] Chialin Chang, Alan Sussman, and Joel Saltz. Object-oriented runtime support for complex distributed data structures. Technical Report UMIACS-TR-95-35, University of Maryland, 1995.

[32] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 13th international conference on Supercomputing*, pages 444–453. ACM Press, 1999.

[33] Jian Chen and Valerie E. Taylor. Mesh partitioning for efficient use of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):67–79, January 2002.

[34] A. Chien, J. Dolby, B. Ganguly, V. Karamecheti, and X. Zhang. Supporting High Level Programming with High Performance: The Illinois Concert System. In *Proceedings of the Second International Workshop on High-Level Programming Models and Supportive Environments (HIPS '97)*, April 1997.

[35] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. ICC++ — A C++ dialect for high performance parallel computing. *Springer LNCS*, 1049:76–94, 1996.

[36] Nikos Chrisochoides, Induprakas Kodukula, and Keshav Pingali. Compiler and run-time support for semi-structured applications. In *Proceedings of the 11th international conference on Supercomputing*, pages 229–236. ACM Press, 1997.

[37] Leonard Dagum and Remesh Menon. OpenMP: An industry standard API for shared memory programming. *IEEE Computational Science & Engineering*, pages 46–55, January-March 1998.

[38] R. Das, M. Uysal, J. Saltz, and Y. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[39] Roxana Diaconescu. An object-oriented framework for distributed numerical computations. In *OOPSLA 2001 Companion, Doctoral Symposium*, page 9, Tampa Bay, Florida, October 2001. ACM press.

[40] Roxana Diaconescu. Distributed component architecture for scientific applications. In James Noble and Eds. John Potter, editors, *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia.*, volume Vol. 10. of *Conferences in Research and Practice in Information Technology*. Australian Computer Society, Inc., 2002.

[41] Roxana Diaconescu. A general data layout for distributed consistency in data parallel applications. In *Proceedings of the International Conference on High Performance Computing (HiPC 2002)*, Bangalore, India, 18-21 December 2002. Springer LNCS. To appear.

[42] Roxana Diaconescu and Reidar Conradi. A data parallel programming model based on distributed objects. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2002)*, Chicago, Illinois, 23-26 September 2002. IEEE. To appear.

[43] Intelligent Systems Division of Intelligent Systems. ClustIS - Intelligent Systems Cluster. Computer Science Department, Norwegian University of Science and Technology, http://clustis.idi.ntnu.no/, 2002.

[44] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, March 1988.

[45] G. Edjlali, G. Agrawal, A. Sussman, J. Humphries, and J. Saltz. Compiler and runtime support for programming in adaptive parallel environments. *Scientific Programming*, pages 215–227, January 1997.

[46] Martin Fowler, Kendall Scott, and Grady Booch. *UML Distiled: A Brief Guide to the Standard Object Modeling Language*. The Addison-Wesley Object Technology Series. Addison-Wesley, 2nd edition edition, August 20 1999.

[47] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing. Addison-Wesley, 18th printing edition, September 1999.

[48] Dennis Gannon, Peter Beckman, Elizabeth Johnson, Todd Green, and Mike Levine. HPC++ and the HPC++Lib Toolkit. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 73–108, 2001.

[49] Aniruddha Gokhale and Douglas C. Schmidt. Measuring the performance of communication middleware on high-speed networks. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 306–317. ACM Press, 1996.

[50] Andrew S. Grimshaw. The mentat computation model data-driven support for object-oriented parallel processing. Technical Report CS-93-30, University of Virginia, May 1993.

[51] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, Cambridge, Massachusetts, London, England, second edition, 1999.

[52] Manish Gupta and Prithviraj Banerjee. Paradigm: a compiler for automatic data distribution on multicomputers. In *Proceedings of the 7th international conference on Supercomputing*, pages 87–96. ACM Press, 1993.

[53] Manish Gupta, Edith Schonberg, and Harini Srinivasan. A unified framework for optimizing communication in data-parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):689–704, 1996.

[54] Saniya Ben Hassen, Irina Athanasiu, and Henri E. Bal. A flexible operation execution model for shared distributed objects. In *Proceedings of the OOPSLA'96 Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 30–50. ACM, October 1996.

[55] Saniya Ben Hassen, Henri E. Bal, and Ceriel J. H. Jacobs. A task- and data-parallel programming language based on shared objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(6):1131–1170, 1998.

[56] Michael Hicks, Suresh Jagannathan, Richard Kelsey, Jonathan T. Moore, and Cristian Ungureanu. Transparent communication for distributed objects in java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 160–170. ACM Press, 1999.

[57] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, 1992.

[58] David E. Hudak and Santosh G. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 4th international conference on Supercomputing*, pages 187–200. ACM Press, 1990.

[59] IBM Research, http://www.research.ibm.com/nao. *Numerical Analysis Objects - NAO*.

[60] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, 1996.

[61] Liviu Iftode and Jaswinder Pal Singh. Shared virtual memory: Progress and challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):498–507, 1999.

[62] Ralph E. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[63] Matjaz B. Juric, Ales Zivkovic, and Ivan Rozman. Are distributed objects fast enough? *Java Report, SIGS*, (5):29–38, May 1998.

[64] J.-M. Jzquel. An object-oriented framework for data parallelism. *ACM Computing Surveys*, 32(31):31–35, March 2000.

[65] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based On C++. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 91–108, 1993.

[66] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Transactions on Programming Languages and Systems*, 21(6):1251–1297, 1999.

[67] K. Kennedy and N. Nedeljković. Combining dependence and data-flow analyses to optimize communication. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, 1995.

[68] Ken Kennedy and Ulrich Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of the 1995 conference on Supercomputing (CD-ROM)*, page 76. ACM Press, 1995.

[69] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[70] Hans Petter Langtangen. *Computational Partial Differential Equations: Numerical Metheds and Diffpack programming*, volume 2 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag Berlin Heidelberg, 1999.

[71] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*, chapter 5 (Concurrency Control). Java Series. Addison-Wesley, second edition, January 1997.

[72] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. *Computer*, pages 63,79, March 1992.

[73] Scott M. Lewandowski. Frameworks for component-based client/server computing. *ACM Computing Surveys (CSUR)*, 30(1):3–27, 1998.

[74] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 229–239, New York, NY, 1986. ACM Press.

[75] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Proceedings of Supercomputing*, December 1995.

[76] Chi-Keung Luk. *Optimizing the Cache Performance of Non-Numeric Applications*. PhD thesis, Graduate Department of computer Science, University of Toronto, 2000.

[77] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems*, pages 222–233, 1996.

[78] Steve MacDonald, Duane Szafron, Jonathan Schaffer, and Steven Bromling. From patterns to frameworks to parallel programs. Submitted to the Journal of Parallel and Distributed Computing, 2000.

[79] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *International Conference on Supercomputing*, pages 88–99, 2000.

[80] Mathematics and Computer Science Division Argonne National Laboratory. Mpich-a portable implementation of mpi. Available at http://www-unix.mcs.anl.gov/mpi/mpich/. Argonne, Illinois, USA.

[81] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[82] Bertrand Meyer. Systematic concurrent object-oriented programming. *Communication of the* ACM, 36(9):56–80, September 1993.

[83] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In *Conference proceedings on International conference on supercomputing*, pages 140–152. ACM, November 1988.

[84] Oscar Nierstrasz. A survey of object-oriented concepts. In W. Kim and F. Lochovsk, editors, *Object-Oriented Concepts, Databases and Applications*, pages 3–21. ACM Press and Addison Wesley, 1989.

[85] Dianne P. O'Leary. Teamwork: Computational science and applied mathematics. In *IEEE Computational Science and Engineering*, April-June 1997. Based on a presentation at the IEEE Computer Society Workshop on Computational Science and Engineering, Oct. 1996, Purdue University.

[86] David Parsons, Awais Rashid, Andreas Speck, and Alexandru Telea. A "framewok" for object oriented framework design. In *Technology of Object-Oriented Languages and Systems*, Proceedings of, pages 141–151, 1999.

[87] Keshav Pingali. *Parallel and Vector Programming Languages*. Wiley Encyclopedia of Electrical and Electronics Engineering. John Wiley & Sons, 1999.

[88] John R. Rice. Computational science as one driving force for all aspects of computing research. *ACM Computing Surveys*, 28, 1996.

[89] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the power of parallelism in a pile-of-pcs. In *Proceedings, IEEE Aerospace*, 1997.

[90] Peter W. Rijks, Jeffrey M. Squyres, and Andrew Lumsdaine. Performance benchmarking of object oriented mpi (oompi) version 1.0.2g. Technical report, University of Notre Dame Department of Computer Science and Engineering, 1999.

[91] A. Sameh, G. Cybenko, M. Kalos, K. Neves, J. Rice, D. Sorensen, and F. Sullivan. Computational science and engineering. *ACM Computing Surveys (CSUR)*, 28(4):810–817, 1996.

[92] Naohito Sato, Satoshi Matsuoka, Jean-Marc Jezequel, and Akinori Yonezawa. A methodology for specifying data distribution using only standard object-oriented features. In *Proc. of International Conference on Supercomputing*, pages 116–123. ACM, 1997.

[93] Toshinori Sato and Itsujiro Arita. Partial resolution in data value predictors. In *Proceedings of the 2000 International Conference on Parallel Processing*, pages 69–76. IEEE, 2000.

[94] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *OSDI94*, pages 101–114, Monterey, CA, November 1994. USENIX Association.

[95] Kirk Schloegel, George Karypis, and Vipin Kumar. Graph partitioning for high performance scientific simulations. In J. Dongarra et al., editors, CRPC Parallel Computing Handbook. Morgan Kaufmann, 2000 (in press), 2000.

[96] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan shin Hwang, Raja Das, and Joel Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing 94*, pages 97–106. IEEE, November 1994.

[97] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123,169, June 1998.

[98] Barry F. Smith. The transition of numerical software: From nuts-and-bolts to abstractions. This work was supported by the Mathematical, Information and Computer Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under contract W-31-109-Eng-38, May 1998.

[99] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, London, England, 1996.

[100] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In OOPSLA*'86*, pages 38–45. ACM Press, September 1986.

[101] Yukihiko Sohda, Hirotaka Ogawa, and Satoshi Matsuoka. OMPC++ - A portable high-performance implementation of DSM using openc++ reflection. In *Reflection*, pages 215–234, 1999.

[102] Jeffrey M. Squyres, Brian C. McCandless, and Andrew Lumsdaine. Object oriented mpi: A class library for the message passing interface. In *Proceedings of the POOMA conference*, 1996.

[103] Bjarne Stroustrup. *The* C++ *Programming Language*. Addison-Wesley Reading, Massachusetts, third edition, 2000.

[104] A. Sussman. Model-driven mapping onto distributed memory parallel computers. In *Proceedings of the 1992 conference on Supercomputing '92*, pages 818–829. IEEE Computer Society Press, 1992.

[105] M. Ujaldon, S. Sharma, J. Saltz, and E. Zapata. Run-time techniques for parallelizing sparse matrix problems. Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'95), Lyon, France, September 1995.

[106] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[107] Eric F. Van DE Velde. *Concurrent Scientific Computing*, volume 16 of *Texts in Applied Mathematics*. Springer-Verlag New York, Berlin, Heidelberg, London, Paris, Tokyo, Hong Kong, Barcelona, Budapest, 1994.

[108] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[109] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[110] Skef Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proceedings of the 6th international conference on Supercomputing*, pages 25–34. ACM Press, 1992.

[111] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6):737–753, June 1995.

[112] Barbara B. Wyatt, Krishna Kavi, and Steve Hufnagel. Parallelism in object-oriented languages: A survey. *IEEE Software*, 9(6):56–66, November 1992.

[113] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

[114] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA '86 Proceedings*, pages 258–268, September 1986.

[115] A. Zaafrani and M. R. Ito. Partitioning the global space for distributed memory systems. In *Proceedings of the 1993 conference on Supercomputing*, pages 327–336. ACM Press, 1993.