The Pennsylvania State University

The Graduate School

Department of Computer Science

Parallel Programming with Guarded Objects

A Thesis in

Computer Science

by

Babak Bagheri

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 1994

We approve the thesis of Babak Bagheri.

Date of Signature

Mary Jane Irwin
Professor of Computer Science
Chair of Committee

L. Ridgway Scott
Professor of Computer Science and Mathematics
Thesis Adviser

Robert M. Owens
Associate Professor of Computer Science

William H. Winsborough
Assistant Professor of Computer Science

Jinchao Xu
Associate Professor of Mathematics

Joseph M. Lambert
Professor of Computer Science
Head of the Department of Computer Science and Engineering

Abstract

This dissertation presents the semantics and pragmatics of a *quasi-explicit* model of parallel computation called IP. In IP, the expression of parallelism itself is explicit, but the expression of coordination—namely, synchronization and communication—is implicit. The IP rules for implicit coordination are encapsulated in a model of memory called *guarded memory*. Parallel programming with guarded memory avoids the two main difficulties of traditional explicitly parallel programming models, namely nondeterminism and deadlock. In the IP model, it is in fact impossible to express parallel algorithms which may lead to nondeterministic behavior. At the same time, computations which may lead to deadlock can be easily identified. The dissertation includes formal semantics of a simple sequential language extended to include the IP model, proof of determinism, and characterization of deadlock. The dissertation also includes descriptions of three languages, IPC, IPfortran, and IPC++, which use the IP model together with a general approach to code motion which can be used to optimize the compilation of communication and synchronization code in these languages.

Table of Contents

List of Figures

List of Tables

Notation

$:=$	Equal by definition
$\mathbb{N}$	The natural numbers
$\mathbb{R}$	The real numbers
$\mathbb{Z}$	The integers
$\{x_1, \ldots, x_n\}$	A set
$\mathcal{P}(S)$	The set of subsets of set S
$\underline{n}$	{1,2,. . .,n}
$\mathcal{E}(G)$	Edges of graph G
$\mathcal{N}(G)$	Nodes of graph G
$\boldsymbol{P}(g)$	Points of cfg g
$p \prec q$	$\exists$ a directed path from point p to point q
$p \succ q$	$\exists$ a directed path from point q to point p
$p \prec\!\!\!\succ q$	$p \prec q$ or $p \succ q$
S^Ω	Tuples of members of S: $\bigcup_{n=1}^{\infty} S^n$
$[x_1, \ldots, x_n]$	An n-tuple
t_i	i^{th} element of tuple t
$[]$	The empty tuple
$\mid a \mid$	Length of tuple a
τa	Last element of tuple a
$a \oplus b$	Concatenation of tuples a and b
$\langle x_1, \ldots, x_n \rangle$	A configuration
$A \times B$	Cartesian product of sets A and B
$A \rightarrow B$	Set of functions from set A to set B
$\triangleright$	Transition relation between configurations
$\overset{*}{\triangleright}$	Transitive closure of $\triangleright$

Preface

The work reported in this dissertation grew out of my advisor's dissatisfaction with low-level languages for parallel scientific programming in the early to mid 1980's when programmers were limited to operating system primitives for programming the few existing parallel processors. The resulting investigations have lead to the $\mathbb{P}$ model which is presented in this dissertation.

Having put together a simple strategy for "symmetric copying" of data between the processors of these systems in SPMD programs, Professor Scott asked me to investigate its use. During that summer, I put together a simple translator for what turned out to be an early version of static $\mathbb{P}$ model as an extension of FORTRAN 77.

Although that work, and a later (fall 1988) implementation of the static $\mathbb{P}$ extension of C, were successful in simplifying the task of parallel scientific programming, both the programming model and the implementation strategies had many shortcomings. The programming model was incomplete and only informally specified; this lead to much confusion about what various constructions meant. The implementation strategy lead to deadlock in many common cases. This dissertation reflects an attempt to understand and extend the underlying model and improve the implementation techniques. The organization of the dissertation reflects the long journey which has lead to the general $\mathbb{P}$ model and its implementation as an extension of the C++ programming language.

In this dissertation, I will explore the $\mathbb{P}$ model in great detail by

1. presenting two classes of language extension embodied in three languages (IPC, IPfortran, and IPC++) based on the IP model and presenting implementations of many algorithms which form the kernels of scientific computations,

2. providing a critique of the IP model by examining two large application programs from two of the main areas, chemistry and fluid dynamics, of scientific computation,

3. giving the exact operational semantics of a small language, called IPtiny, based on the IP model,

4. proving that IPtiny is deterministic and characterizing deadlock for it, and

5. presenting compiler optimization techniques for IPC++.

Part I is introductory; it presents the IP model informally and provides an evaluation and critique of IP suggested by programming experience. Part II contains the formal semantics of a small language and its *general* IP extension. This work shows that the original intuition about the IP model, although incomplete, did contain the germ of a simple and elegant model of parallel computation. Part III presents a description of the compilation techniques needed to translate IPC++ programs into efficient code for distributed-memory multiprocessors.

While developing these ideas, I have benefited from many co-workers' help. I will take this opportunity to acknowledge that help. First and foremost, I thank my advisor Professor L. Ridgway Scott who, with great patience, has taught me how to conduct research and supported me in many ways for many years. I thank the members of my thesis committee for their patience and helpful comments. I especially thank Professor

Mary Jane Irwin for serving as chair of the committee. I thank Professors Irwin and Owens for teaching me about computer hardware and parallel architectures.

I thank Terry Clark for writing and using a translator for IPFortran. I thank Balaji Raghavachari for helping with the implementation of the IPC translator. I thank Reinhard v. Hanxleden for using IPC and IPfortran, providing valuable comments, and sharing early drafts of his work on code placement. I thank Andrew Ilin for his meticulous work on the IPfortran version of the University of Houston Brownian Dynamics program.

I thank Ms. Vicki Keller without whose vigilance and help I would not have survived.

Some of the simulations were performed at the Cornell National Supercomputer Facility at Cornell University. Program development was performed on an iPSC/860 provided by Intel and housed at the University of Houston's Institute for Molecular Design. Delta simulations were performed on the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium with access provided by the Center for Research on Parallel Computation. This work was supported in part by the National Science Foundation, award number ASC-921734 (which includes funds from DARPA).

Part I

The $\mathbb{P}$ Model of Computation

I usually think of language design as "the Pit of Despair."

— **Preston Briggs**[1]

[1] In a message posted to `comp.compilers` on USENET

Chapter 1

Introduction

In recent years, the need to use many cooperating processors to solve large problems has become clear. But as many of their users have discovered, parallel processors are significantly more difficult to program than sequential ones. Many of the difficulties are conceptual and algorithmic; others, however, are linguistic and clerical. The linguistic difficulties come about because cooperating parallel processes need to share information and resources. The resulting management difficulties are especially acute for complex applications which demand very efficient use of computing resources. Such complex applications often require explicit management of parallelism. For these applications, the challenge is to design a language which allows the programmer to control parallelism explicitly without concern about troublesome details.

This dissertation presents the semantics and pragmatics of a *quasi-explicit* model of parallel computation called $\mathbb{P}$. In $\mathbb{P}$, the expression of parallelism itself is explicit, but the expression of coordination—namely, synchronization and communication—is implicit. The $\mathbb{P}$ rules for implicit coordination are encapsulated in a model of memory called *guarded memory*.

Parallel programming with guarded memory avoids the two main difficulties of traditional explicitly parallel programming models, namely nondeterminism and deadlock. In the $\mathbb{P}$ model, it is in fact impossible to express parallel algorithms which may lead to nondeterministic behavior. At the same time, computations which may lead to deadlock can be easily identified.

1.1 Goals and Progress

The main goal of the project reported in this dissertation was to design and implement a simple, expressive model of parallelism for scientific computation. The sheer number of projects with the same goal which have been undertaken in the past decade brings up the following question. Why design a new parallel programming model? The simple answer to this question is: "Because no satisfactory way to program parallel processors has yet emerged." Thus it still pays to explore promising new models. In a few decades experience may single out one as superior to others for scientific computation. Hence this dissertation aims to do no more than present one promising model and to evaluate the model as much as possible at present. The author has come to believe that it is essential to pay careful attention to the design of the semantics and use of a language in the early stages of its development. This belief is strongly reflected in this dissertation; the expressiveness of the main language (IPC++) presented here is assessed much more carefully than the efficiency of its implementation. A very detailed performance evaluation is left for future phases of the evaluation process.

Both the conception and the evaluation of a programming model depend very much on the kind of programming we want to do. What makes the design of a parallel programming model for scientific computation nontrivial is the combination of goals which must be met. For scientific programming, the most important goals are familiarity and conciseness of expression and efficiency of execution. Programs which express algorithms in a familiar and concise way go a long way toward ensuring their own correctness. The need for efficiency in scientific computation is, of course, evident. Indeed, this need is why parallel processors exist. Our challenge, then, is to design a language in which parallel algorithms can be expressed in a notation familiar to scientists and which can be translated into efficient code.

How close we have come to the first goal (familiarity of expression) is difficult to quantify and measure. Over 30 years of experience have made it clear, however, that arrays (or indexed collections of objects in general) play a special role in scientific computing. As will be shown in chapter 4 on IPC++, the IP model can be taylored to meet this need.

The second goal (conciseness of expression) is easier to measure. We want a language which does not force programmers to repeat similar patterns of computation. In the context of parallel programming, we want a way to express parallelism which does not work against abstraction. Elegance also demands simplicity. We want a language which is as spare as possible. The IP model goes a long way toward all of these goals. The implicit coordination mechanism of guarded memory simplifies the expression of parallel algorithms common in scientific computing. The conciseness with which important and complex algorithms can be expressed in IPC++ is examined in chapter 5.

Surprisingly, efficiency of execution is also difficult to measure effectively for a programming language. It is relatively easy to measure the efficiency of particular programs, but much harder to decide what the source of any inefficiency is. Is it the algorithm? Is it the compilation technique? Is it the surrounding operating system? Often algorithms which form numerical kernels of scientific programs are used as a first test; this has been done in IPC, IPfortran, and IPC++. However, realistic scientific programs pose much greater programming problems. In IPC++, a library of classes for finite element computation, called `fec`, was developed to study these difficulties. Both IPC and IPfortran are based on a static version of IP which is particularly easy to translate into efficient code, but not as expressive as one would like. Thus execution times for IPC and IPfortran programs come very close to hand-coded message-passing programs for the algorithms we have implemented. IPC++ is based on a more ambitious and expressive version of the IP model. Considerably more effort is required to generate hand-coded quality code from IPC++ programs. Part III reports on progress in this direction.

1.2 Parallel Processors

The IP model is motivated by an interest in easy and efficient use of resources in parallel processing systems. To understand the efficiency issues, one must be familiar with the capabilities of parallel processing hardware. This section gives a brief overview of parallel hardware from the point of view of efficiency.

Parallel processors arise naturally from the limitations of single-processing systems. If we have only one processor which executes one instruction at a time, then the speed of that processor will dictate how fast we can use any other resource in the system. In a strictly single-processing system, most of the hardware would be idle most of the time waiting for the central processor to finish using slower resources.

System designers recognized these limitations early on. Thus even single processors exploit parallelism within the processor itself. Such processor-level parallelism has been responsible for the speed of traditional supercomputers, and large scale integration has made it possible to incorporate similar techniques in microprocessors. The amount of parallelism available at the processor level is, however, limited. This dissertation is concerned with systems (often called “massively parallel”) which allow large numbers of simultaneous computations.

The central efficiency issue for massively parallel hardware design is the management of communication and synchronization overhead. The most straight-forward way to provide for communication and synchronization among parallel processors would be to connect them all to a single bank of physically shared memory. It is not hard to see, however, that this global connection would serve as a bottleneck and slow down interaction. How best to avoid this bottleneck continues to be an area of research. Two popular approaches have emerged.

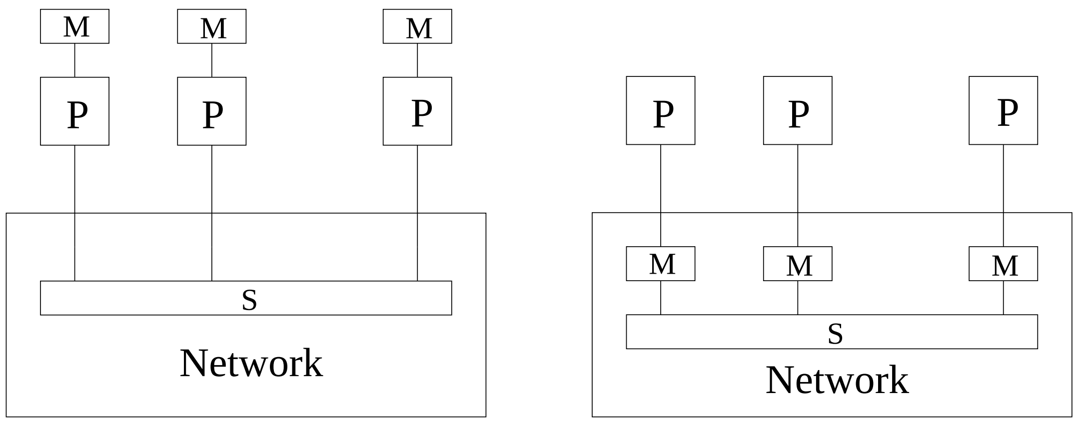

Fig. 1.1. Two Popular Architectures for Parallel Processors

Figure 1.1 shows rough schematics for these two designs which approach interaction overhead in very different ways. The left-hand-side design is a pessimistic one.

The intuition behind the design is that as little should be automated in hardware as possible. The processors share no resources except the network which connects them. In this "distributed-memory" design, all responsibility for communication is in software. When processes running on the various processors need to share information, they must explicitly form messages and put them on the network. In practice, forming messages in software has been rather an expensive operation. The overhead for sending and receiving a message on most existing distributed-memory systems is on the order of a thousand or so floating point operations performed by a processor in the same system. Thus on distributed-memory systems, it is expensive to communicate small amounts of information using popular message-passing primitives. This fact has a dramatic effect on the design of compilers for programming models, like ℙ, in which processes interact through shared objects (as will become clear in part III).

The right-hand-side design in figure 1.1 is a more optimistic one. The processors' local memories are connected directly *and* there is hardware support for automatic movement of data among these memories. In this "shared-memory" design, each processor sees a shared address space; when it accesses a location which is not in its local memory, the interconnection hardware fetches or modifies its contents at its remote location. Various hardware cacheing schemes are used to make these nonlocal accesses efficient. These schemes are effective enough that in practice, average memory access times are often as low as a few floating-point operations even when large numbers of small pieces of information are exchanged [8]. If such systems can be scaled to large numbers of processors, then the job of compiling languages like ℙC++ will be simplified considerably. To date, however, the scalability of such systems beyond one hundred or so processors.

Shared memory hardware support is not the only way to lower communication overhead in parallel processors. It is possible to devise faster message-passing primitives by changing the semantics of message passing. For example, as David Culler and others have shown, interrupt-driven message-passing semantics allow implementors to remove much of the overhead from the message-passing layer [20, 63]. These techniques will also be valuable for simplifying the implementation of languages like IPC++.

1.3 Partitioning and Coordination

The world of parallel programming has grown so vast that even a document of this size can provide a detailed account of all proposed models and languages. Instead of attempting to do so, I will present a simple view of parallel programming which puts IP in perspective by distinguishing two main aspects of parallel programming—namely, partitioning and coordination of tasks. The partitioning aspect deals with mechanisms for specifying how a computation is divided into sequential subcomputations. The coordination aspect deals with mechanisms for controlling the interaction between subcomputations. I identify three ways that parallel programming models can deal with partitioning and coordination: *explicitly*, *implicitly*, or *not at all*. To decide which category a given model falls into, we must answer the following very simple questions. "How do programmers express partitioning and coordination? Directly? Indirectly through hints and annotations? Or not at all?." This taxonomy divides the vast world of parallel programming into very different territories. Programming in a language which allows the programmer to express partitioning or coordination directly is very different in character from programming in a language which partitioning or coordination is expressed indirectly or not at all. This taxonomy is only meant to orient the reader; it is by no means formal or precise. There can be debate about where some languages and models fit. Nevertheless, going through the exercise of trying to classify a model in this way often enhances one's understanding of the model.

Table 1.1
A View of Parallel Programming Models

Language(s)	Partitioning	Coordination
Sisal, Fortran 77	Absent	Absent
Fortran (90)D	Implicit	Absent
IP	Explicit	Absent
Jade	Explicit	Implicit
Occam, Linda, SR	Explicit	Explicit

Table 1.1 gives a rough taxonomy of programming languages from the partitioning and coordination point of view. The two features of partitioning and coordination and their three possibilities make up nine categories. Not all nine categories have models in them (which suggests that more models may be forthcoming). In the first column of the table, I have mentioned languages which belong to the given category. I have only mentioned those languages for which there have been serious compilation efforts (whether successful or not).

According to the school of thought which advocates the first category, one should avoid "parallel programming." Proponents of this school would relegate such "details" to the system (compiler, operating system and hardware). A successful effort in this direction requires techniques for identifying parallelism and coordination without any help from programmers. If such techniques are successfully developed, then this approach would be ideal. There is already a great deal of software written in languages with no mention of parallelism. Perhaps more importantly, programmers are familiar with models in this category. Much effort has been expended in this direction. The results obtained so far are not encouraging.

Compilers which extract fine-grained parallelism from traditional imperative programming languages for vector and wide-instruction processors have been quite successful [3, 26]. This success led to the expectation that parallelizing compilers could be designed using similar techniques. But the extraction and exploitation of coarse-grained parallelism has proven to be more difficult. Such a compiler must analyze programs to see which of the operations that are explicitly sequenced in the program are in fact independent of each other. There have been successful parallelization attempts for simple classes of programs with predictable patterns of memory access. Extracting such dependence information from imperative languages becomes quite complicated in the presence of arrays, procedure calls, and other aliasing mechanisms. More importantly, there may not be enough parallelism available in the *algorithms* implemented in typical scientific programs. It is often necessary to develop explicitly parallel algorithms. It is awkward to express such algorithms in sequential form and hope that a compiler will discover the parallelism one has in mind.

The next category in table 1.1 belongs to languages which allow programmers to give indirect hints as to how the computation should be partitioned. The most popular of these are languages in which programmers indicate the partitioning of computation by giving directions for partitioning arrays. Such array partitioning annotations for implicit partitioning of sequential programs were first proposed in the mid 1980's for multicomputers in Fortran [56, 67, 14], and for nonuniform-access multiprocessors in the language Blaze [46]. Since then, many languages and systems using this approach have been proposed and some implemented: Kali [49] (a descendent of Blaze), Fortran D [28], PANDORE [4]. More recently, the High Perfomance Fortran Forum was formed to standardize array partitioning annotations and constructs for expressing loosely-synchronous parallel algorithms [37]. Annotations for partitioning pointer-based data structures have also been studied [33]. All of these data-driven approached have difficulties expressing nested parallelism and parallel computations which require simultaneous execution of very different tasks (so-called "task-parallel" computations).

The programming language Jade was designed by Martin Rinard and Monica Lam at Stanford [47]. It also provides annotations to guide the compiler in partitioning and coordinating sequential tasks. Jade annotations are more appropriate for coarse-grained task-parallel applications than fine-grained data-parallel ones. Jade also has difficulties with expression of nested parallelism.

1.4 Partitioning

Explicit partitioning mechanisms are often quite simple. In most languages with such mechanisms, programmers simply delimit portions of a program as separate processes in a structured way. Such structured constructs include parallel loops (which are perhaps the most commonly proposed). The general IP model follows this approach.

Non-structured approaches to program partitioning have also been proposed and explored. The *fork* and *join* operations made up one of the earliest non-structured mechanisms for expressing partitioning explicitly. They were proposed in the early 1960's [19]. The surprising fact is that this aspect of explicitly parallel programming has received little attention.

1.5 Coordination

The coordination aspect of a parallel programming model is intimately related to the way processes share information which is in turn intimately related to the way the model deals with memory. This is especially true for the $\mathbb{P}$ model, hence this section presents various models of memory and points out their advantages and disadvantages.

Private The most trivial memory model is for each subcomputation to have its own private memory without any possibility of sharing information in mid computation. At first glance, this model of memory may not seem very useful. But if we allow a computation which invokes parallel subcomputations to use their results *after* they are finished, then we have a reasonable model of memory for parallel computation. This model of memory usually goes together with function- or procedure-based (indirect) mechanisms for synchronization. Remote procedure calls, for example, fall into this category.

Distributed In a distributed memory programming environment, subcomputations which proceed in parallel still have their own private memory. But they can use the results of other subcomputations in mid computation by sending and receiving messages explicitly. The disadvantages of this model include the following.

- Programs which do not send and receive messages correctly can produce nondeterministic or deadlocked computations. This combination of deadlock and nondeterminism makes programs very difficult to understand and debug because their behavior depends on factors other than those (e.g. inputs) which the programmer can control.

- The programmer must place message-passing primitives to ensure not only correctness, but also efficiency. That is, the programmer must structure the program in such a way that the low-level message-passing mechanisms are used efficiently.

The message-passing coordination paradigm first appeared in the late 1960's in the context of multiprogramming [51]. Brinch Hansen [34] then suggested the use of messages for synchronization (useful even with a global name space). C. A. R. Hoare [38, 39] systematized the idea of synchronous unbuffered message passing in his language CSP. A variant of CSP (called Occam [43]) was later implemented on networks of transputers. The message passing mechanism can be used with a purely local name space if we insist that processes do not share any information other than through messages. This is the approach taken by manufacturers of distributed-memory multiprocessors. They provide libraries of (usually buffered and asynchronous) message passing routines [44] and no provisions for shared names.

Shared In a parallel programming model with shared memory, parallel subcomputations have unrestricted access to all memory locations. The programmer is responsible for synchronizing memory access to avoid conflicts. Many synchronization constructs and mechanisms have been designed for this purpose. The shared memory model of parallel computation has the following advantages.

- The programmer need not map data structures onto a segmented address space.
- Thus, coordination problems aside, memory management is close to the familiar sequential case.

But the shared memory model has the following disadvantages.

- Programs which do not use synchronization constructs correctly can produce non-deterministic and deadlocked computations.
- Truly shared memory to which all processors have uniform access is difficult to build. Thus programming languages which view shared memory locations as indistinguishable can mislead programmers into writing inefficient programs.

The most basic coordination mechanism for models with shared memory is the semaphore which was identified by Edsger Dijkstra [25]. Semaphores which get their name from transportation signals allow processes to coordinated access to shared objects by acquiring and releasing locks.

Active An interesting variation on shared memory comes about if we bundle data and computation together. The result is what might be called active objects. Such objects interact with each other by sending each other signals and reacting to signals they receive. The most prominent example of an active object model is the actor model designed by Gul Agha [1]. The distinction between active objects and communicating processes is a subtle one. The main advantage of such models over shared or distributed memory models is the encapsulation and modularity inherent in them. Their main disadvantage is that they do not deal well with collections of objects very well. The active object abstraction forces programmers to serialize access to members of collections; this can affect performance very negatively especially for scientific computation. Various proposals have been made to overcome this difficulty. The most promising of these is to add aggregates to the model as a new kind of active object. The resulting model, called Concurrent Aggregates (CA), by its inventors [17, 15] has been successful enough to inspire the design of an extension to C++ called pC++ [13, 48].

Content-Addressable Content-addressable memory is another model of memory useful for parallel computation. In such a model, memory locations are identified by their contents. The language which codified this model is Linda designed by David Gelernter [30]. The advantages of parallel programming with content-addressable memory include the following.

- Parallel subcomputations are essentially decoupled, so that the only information they shared is in content-addressable memory.

The disadvantages of the content-addressable model of memory are similar to the disadvantages of distributed memory. Coordination through content-addressable memory still requires a matching (albeit partially matching) pair of operations (`in` and `out` in Linda). Orchestrating these operations can pose the same non-determinism and deadlock problems as orchestrating message send and receive operation.

Guarded Guarded memory is made up of a collection of guarded objects which can be accessed in parallel. Each guarded object has a unique process as a guard. Access to guarded objects is through a simple protocol which ensures determinism and easy detection of deadlock. From the programmer's perspective guarded objects behave very much like shared objects. In fact, in most cases, the programmer can ignore the guarding protocols and program in a shared-memory style. The following chapter 2 below gives a detailed but informal presentation of guarded memory and the $\mathbb{P}$ model.

Chapter 2

The IP Model

The IP model builds on existing procedural models of sequential computation. Given such a model of computation, we can define its IP extension by including a few new ingredients. The main ingredient is an access protocol for guarded objects. Guarded objects are shared by all processes (guards), but parallel operations on them must conform to a set of rules (*i.e.* a protocol). The next few sections introduce these new ingredients and explain how the syntax for some of these ingredients can be borrowed for the new purpose from typical base languages and not be added anew.

2.1 Labels

IP requires a set of labels to identify both data objects and tasks which operate on these objects. Labels can be added to the language as a new data type, but they can also be an existing data type which already serves a labeling function in the base language. In all existing IP languages, labels are tuples or nested tuples of integers. It must be mentioned that the labels are not absolutely necessary. What we need is a way to assign a unique guard to each object. In practice, however, assigning guards without labels or names can be cumbersome especially if we want nested guards.

2.2 Guards

In IP, we need mechanisms for creating and destroying processes or tasks which are called *guards* in IP jargon. Guards are so called because each guard is responsible for maintaining a set of guarded objects according to a simple protocol which guarantees determinism. The guard creation mechanism must have a provision for labeling the guard or guards being created. The labeling rules depend strongly on how flexible we want the guard creation and destruction operations to be. For example, allowing nested guards may complicate these rules as we will see in chapter 4 on IPC++. The fundamental piece of information the IP model needs about guards is the set of expressions or statements which make up the guard as a task. Again, this ingredient of IP can often be readily derived from mechanisms meant for other purposes in the base language.

2.3 Guarded Objects

Finally, IP requires a way to identify objects in the base language and to label them by labels from the same set of labels as those used to label guards.

2.4 Rules of Behavior

The semantics of the IP model can be explained informally by the following simple rules of behavior for guards.

1. There can be at most one guard with a given label executing any statement or evaluating any expression at any given time during the computation.

2. A guarded object is only modified by its guard.

3. On its n-th access to an object o guarded by g, a nonguard sees o in its state after g's $\min(n, N)$-th access to o where N is the number of times g accesses o. (N can be infinite.)

2.5 Design Rationale

The main goal in the design of the IP model was to provide a model of parallelism for programmers writing programs with simple synchronization requirements. The most prominent examples of such algorithms come from numerical and scientific computation. The driving force behind the design of IP was the belief that synchronization and communication details should be hidden from programmers as much as possible while allowing them to express parallelism directly. This belief was the result of experience with models with explicit synchronization and communication mechanisms, especially message-passing and semaphore-based models.

Experience with explicit synchronization showed that the kinds of synchronization needed in most scientific computation could be encapsulated in a simple protocol. Originally, the protocol was formulated in terms of message-passing as "the golden rule [56]." However, that protocol was both difficult to formalize and too restrictive. Simple conditional statements which contained access to shared objects caused deadlock contrary to programmers' intuition. These experiences lead directly to the implicit synchronization mechanism in IP.

Synchronous semantics also meet this criterion but restrict the kinds of algorithms which can be expressed easily. The simple rules of behavior for the IP model allow programmers to express a variety of algorithms including those with some asynchrony.

2.6 Static IP

In chapter 7, I will make the informal rules given in section 2.4 precise by giving formal operational semantics for a IP extension of a small language. In this introductory chapter, however, I will present a *static* version of the IP model. This version of IP is static in the sense that guards cannot be created and destroyed by programmers and the assignment of labels to guarded objects is fixed. There are static IP extensions of C and Fortran called IPC and IPfortran respectively. Most of the presentation in this chapter will use IPC, but some of the examples are given in IPfortran. Appendix A contains a detailed description of IPfortran and its implementation. I will use "static IP," "IPC," and "IPfortran" almost interchangeably in the sequel.

Static IP does more than illustrate concepts. Both IPC and IPfortran have been used in parallel programming courses, and several students have completed programming projects using them. Most prominently, Reinhard von Hanxleden used IPC to investigate load balancing strategies in a particle simulation code [64, 65]. In each project, students spent their time experimenting with various algorithms instead of debugging message–passing code. In addition, IPfortran has been used to parallelize large applications, most notably GROMOS (Gröningen Molecular Simulation) [18] and UHBD (University of Houston Brownian Dynamics) [7]. Chapter 3 presents the use of IPfortran in parallelizing UHBD in detail and shows both its strengths and weaknesses.

2.7 Guards in Static IP

In static IP, there are a fixed number of guards labeled by consecutive integers starting at 0 and continuing to `numnodes`-1. Thus the value of the special variable `numnodes` is the total number of guards and is fixed at load time. Each guard also has a special variable `thisnode` which evaluates to its label. All guards are created and begin executing at the beginning of `main` and are destroyed when they reach the end of `main` or when they execute the `exit` or `abort` functions (if they exist). With `numnodes` fixed at, say, 4, the program

```
main() {
  printf("I am %d of %d processes.\n", thisnode, numnodes);
}
```

will produce an output of the form

```
I am 1 of 4 processes.
I am 0 of 4 processes.
I am 2 of 4 processes.
I am 3 of 4 processes.
```

Up to this point, the static IP model is nothing more than the familiar Single Program Multiple Data (SPMD) model with no provision for coordination. The order of the output statements depends on the semantics of I/O which the IP model leaves unspecified. As we will see, the only sources of nondeterminism and deadlock in IP programs are in their interactions with the surrounding system (through I/O, system calls, *etc.*).

2.8 Guarded Objects in Static IP

In static IP, all objects are guarded. Each expression refers to a sequence of guarded objects. The labels of these objects are the same as those of the fixed sequence of guards (*i.e.* from 0 to `numnodes`-1). Each element of such a sequence is called a *version* of the expression which refers to it. The new "dereferencing" operator `@` allows access to specific versions of expression. It can appear between two C expressions. With the `@` operator left off, each expression evaluates to the object guarded by the guard evaluating the expression. Thus, in IPC, an expression e always has the same value as e`@[thisnode]`. The following simple examples show how the `@` operator is used in practice.

In the simplest case, we can use the `@` operator to access data with a specific guard. For example,

```
x@[2] = y@[0];
```

has the effect of copying the element of `y` with guard `0` onto the element of `x` with guard 2. By omitting the operator from the left hand side, we get a *broadcast* from guard `0` to all other guards. Thus,

```
x = y@[n+1];
```

means: assign the element of `y` with guard `n+1` to all elements of `x`.

Even with these simple operations, synchronization can be done easily in static IP through access to "dummy" data. For example, one can simply copy an unused object (a "baton") from location to location to print in a given order. Each guard waits until it has copied the baton from the guard preceding it in the given order before incrementing the baton and printing the data it guards.

```
void Baton() {
  int baton, i;
  if (thisnode == 0)
    printf("hello from\n",thisnode);
  baton@[0] = 1;
  for (i=1; i<numnodes; ++i) {
     baton@[i] = baton@[i-1];
     if (baton == thisnode) {
       printf("hello from\n",thisnode);
       baton = baton +1;
     }
  }
}
```

Fig. 2.1. Synchronization by passing batons in IPC

2.9 Column oriented LU decomposition

This example illustrates the implementation of "data decomposition" in static IP for an LU decomposition algorithm. Geist and Romine [29] describe the algorithm shown in figure 2.2 for column-to-processor assignment of a matrix using *LU decomposition* on an $n \times n$ matrix.

The natural unit of decomposition, columns of an $m \times n$ matrix, leads to an efficient implementation of this algorithm using *cyclic assignment*; $column_j$ belongs to processor, p, where $p = mod(column_j, numnodes)$ and $0 \leq column_j < n$. An alternative

```
for k = 0, n-1
 if (I own column k) then
  for i = k+1, n-1
     l(i,k) = a(i,k) /  a(k,k)
  end for
  broadcast l
 else
  receive l
  for (all columns j>k that I own)
   for i = k+1 to n-1
    a(i,j) = a(i,j) - l(i,k) * a(k,j)
   end for
  end for
 end if
end for
```

Fig. 2.2.   Distributed Gaussian Elimination

decomposition, *contiguous assignment*, leads to increased processor idling since the lower indexed columns are eliminated first.

Figure 2.3 demonstrates distribution of both data and computation among processors are in IPfortran. The distribution of data and computation are intimately related, and depend of course on the arrangement of data in memory.

2.10 Aggregate Operations

As mentioned before, static IP expressions refer to sequences of "versions." It is often useful to perform an operation on such sequences of objects. To allow such operations, IPC allows statements of the form

```
y = op{x@[1:5]};
```

The term `op` may be the identifier of any function of two parameters both of which have the same type as `x`. The function must return a value of the same type as `y` and assignment must be defined for this type. Of course, since the order of application of the function to the set of values is not specified, the function must be associative and commutative. More precisely, we are assuming that the set of values with the type of `x` forms an Abelian semi–group with respect to `op`.

For example, the sum of all version of a variable `y` can be calculated succinctly using

```
sum = +{y};
```

This statement has the effect: "Copy to every version of `sum` the sum of all versions of `y`." Similarly,

```
   REAL*8 a(64), m(64)
C distribute columns cyclically
   do  i = 1, 64
     if (thisnode.eq.mod(i,numnodes)
  .      call getcolumn(i,a)
   enddo
   do k = 1, N-1
C owner computes column of multipliers
     if (thisnode.eq.mod(k,numnodes)) then
      do i = k+1, n-1
        m(i,k) = a(i,k)/a(k,k)
      enddo
     endif
C broadcast column of multipliers
     m(k+1:N) = m(k+1:N)@mod(k, numnodes)
C reduce columns
     do i = k+1,N
       if (mod(i,numnodes).eq.thisnode) then
         do j = k+1, N
          a(j,i) = a(i,j) - m(j)*a(k,j)
          enddo
       endif
       a(i,j) = a(i,j) - m(i)*a(k,j)
     enddo
   enddo
```

Fig. 2.3. Gaussian Elimination in IPfortran

```
x = max{y@[0:7]};
```

has the effect: "Copy to every version of `x` the maximum of versions of `y` with guards 0 through 7," provided we have defined a function called `max` appropriately for the objects of the type of `x` and `y`.

More complex operators are often useful. Consider the following approach to computing the product of a matrix and a vector. To begin with, we define "vector" as a type and a corresponding vector addition (`addvec`) in figure 2.4.

```
typedef struct {double v[D];} VECTOR;

VECTOR addvec(a,b)
VECTOR a,b;
 {
        int i;
        VECTOR r;
        for (i=0;i<D;i++) r.v[i] = a.v[i] + b.v[i];
        return(r);
 }
```

Fig. 2.4. Vector Addition in IPC

The function `matvec` shown in figure 2.5 uses a reduction based on `addvec`. Note that `D` is the dimension of the matrix, and `N` is the number of columns guarded by each of the `numnodes` guards. The function `addvec` is clearly associative and commutative; its global version adds all the components among all processes. A similar approach can be used to "accumulate" other distributed objects, such as pixel values in a digital image.

```
void matvec(res, mat, vec)
double res[N], mat[D][N], vec[N];
 {
         int i,j,li,lj;
         VECTOR s;
         for(i=0;i<D;i++) {
           s.v[i] = 0.0;
           for(j=0;j<N;j++)
           s.v[i] += mat[i][j] * vec[j];
         }
         s = addvec{s};
         for(i=0;i<N;i++) res[i] = s.v[global(i)];
}
```

Fig. 2.5. Matrix-Vector Multiplication in IPC

Chapter 3

An Evaluation of Static IP: UHBD

This chapter presents a critical evaluation of the static IP model. I will examine the use of IPfortran in parallelizing a large computational chemistry application and show clearly where its shortcomings lie. The program is the University of Houston Brownian Dynamics (UHBD) program [23].

Brownian dynamics simulation in UHBD has two complimentary phases. In the first, an electrostatics problem is solved for a macromolecule in solution. Then "Brownian dynamics" simulates the diffusion of molecules in the solution subject to the electrostatic forces. Ideally, we would like to recalculate the electrostatic potential during the calculation of Brownian dynamics trajectories. These two computational phases—of electrostatic potential and Brownian dynamics trajectories—make competing demands on both the hardware and the available software tools. This interplay makes Brownian dynamics simulation a good test case for comparing parallel systems.

I will present two IPfortran versions of the UHBD program: one with "replicated" arrays and one with "distributed" arrays. This exploration shows that in large applications one needs more flexibility in the expression of parallelism than IPfortran provides, especially for expressing algorithms which scale with respect to memory. It does turns out that IPfortran guarded objects are convenient for expressing communication and coordination in UHBD. But the rigid naming structure of IPfortran makes it difficult to express algorithms which require parallel operations on segments of large data structures. The general IP model is designed to address precisely this weakness of the static IP model embodied by IPfortran. In addition to evaluating IPfortran, UHBD makes it possible to illustrate the use of IPfortran for typical finite-difference and Monte-Carlo algorithms.

3.1 Phases of UHBD

The two major phases of UHBD are quite different. Both phases use iterative algorithms. The electrostatics phase is a typical finite-difference-based partial differential equation solver. The data access pattern of this phase is predictable and fixed from one iteration to the next (for up to a few hundred iterations). The Brownian Dynamics phase, on the other hand, has unstructured and unpredictable data access patterns which can change from one iteration to the next (for some thousands of iterations). The combination of two such different types of computation in one code makes UHBD an interesting testbed for studying and contrasting different programming models and hardware architectures [8]. In particular, UHBD offers an interesting test for different approaches to expressing parallelism.

The electrostatics phase of the UHBD program calculates the solution of the Linearized Poisson-Boltzmann Equation (LPBE) [24]. In three dimensions and with the standard seven-point difference stencil, the resulting linear system of equations has a sparse coefficient matrix, $\boldsymbol{A}$, which is symmetric and has the nonzero structure shown in figure 3.1. Numerical experiments have shown that the Preconditioned Conjugate Gradients (PCG) method, shown in figure 3.1 (where $\boldsymbol{M}$ is the preconditioning matrix), is preferable to other iterative techniques [24, 40, 42] for solving the corresponding system of linear equations. The evaluation presented here uses diagonal preconditioning and a Parallel version of the (Modified) Incomplete Cholesky preconditioner, which we denote by P(M)IC, which is easy to implement and scalable for reasonable numbers of processors [7, 9].

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{p}_0 = \boldsymbol{M}^{-1}\boldsymbol{r}_0$
For $i = 0$ to convergence
  $\alpha = (\boldsymbol{r}_i \cdot \boldsymbol{M}^{-1}\boldsymbol{r}_i)/(\boldsymbol{p}_i \cdot \boldsymbol{A}\boldsymbol{p}_i)$
  $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha\boldsymbol{p}_i$
  $\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha\boldsymbol{A}\boldsymbol{p}_i$
  $\beta = (\boldsymbol{r}_{i+1} \cdot \boldsymbol{M}^{-1}\boldsymbol{r}_{i+1})/(\boldsymbol{r}_i \cdot \boldsymbol{M}^{-1}\boldsymbol{r}_i)$
  $\boldsymbol{p}_{i+1} = \boldsymbol{M}^{-1}\boldsymbol{r}_{i+1} + \beta\boldsymbol{p}_i$
End For

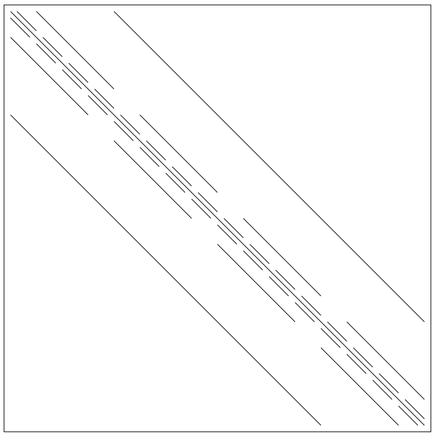

Fig. 3.1. Preconditioned Conjugate Gradients (PCG) for approximating the solution of $Ax = b$, with preconditioning matrix M, and sparsity structure of matrix A for LPBE

The phase of UHBD which computes Brownian Dynamics trajectories simulates the movement of a test particle toward the molecule being studied (see figure 3.2). The BD algorithm uses the electrostatic potential to compute trajectories of a diffusing test particle under the influence of a combination of random and electrostatic forces (see figure 3.3).

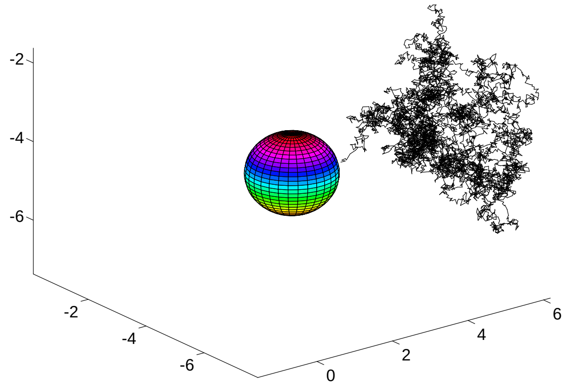

Fig. 3.2. Brownian Dynamics Simulation of a particle diffusing via random motion, starting in the upper-right quadrant, toward a single atom, depicted as a sphere

3.2 The Replicated Version

In the replicated version, all data structures are duplicated on all processors. This duplication allows us to use the same naming scheme as in the original sequential program. The major drawback of this approach is the memory constraint it imposes. The advantage of this approach is that it can be easily expressed in static $\mathbb{P}$. Figures 3.4 and 3.5 show the sequential and replicated versions of the action of the discretized operator $\boldsymbol{A}$ for LPBE. The variables `im`, `imjm`, and `imjmkm` hold the first dimension, the product of the first and second dimensions, and the product of the first, second and third dimensions of the discretized operator $\boldsymbol{A}$. The variables `imjmkmlo` and `imjmkmhi` are the lower and upper bounds of the portions of the replicated arrays which are to be updated by each processor. the product of the first and second dimensions, and the product of for each trajectory

1. compute starting position of test particle
2. initialize reaction indicators
3. do while the test particle is inside the test volume and there is not a reaction
   (a) get the time step (either constant or user defined)
   (b) exit the while loop if any limit is exceeded
   (c) initialise all forces
   (d) calculate electrostatic forces
   (e) calculate forces on subunits in each flexible unit
   (f) calculate forces between subunits of different units due to excluded volume and solvent interaction
   (g) take a bd step:
      i. for each subunit
         A. check to make sure this is not a fixed subunit
         B. calculate the product of the Q matrix and the force vector for this subunit
      ii. if not a monomer then reconstrain
      iii. calculate distance from protein and check for reaction
      iv. for each subunit
         A. see if it bumped into protein
      v. for each subunit
         A. if bumped and bumping unit only contains two particles keep them both in their original positions
         B. else if didn't bump then move it
      vi. if reconstraining after positioning subunits to avoid bumps then
         A. if unit with more than 2 subunits and using SHAKE constraints then reconstrain
      vii. calculate statistics on forces
   (h) if there was a reaction then increment reaction counter
   (i) calculate mean force and largest force step

Fig. 3.3. Calculation of Brownian Dynamics Trajectories

the first, second and third dimensions of the grid respectively. The arrays `epsc`, `epsi`, `epsj`, and `epsk` are the nonzero diagonals of the matrix $\boldsymbol{A}$.

```
         do  100  i = 1 , imjmkm

            zv = epsc(i) * pvec(i)
            if (i .lt. imjmkm) zv = zv - epsi(i)*pvec(i+1)
            if (i .le. imjmkm-im) zv = zv - epsj(i)*pvec(i+im)
            if (i .le. imjmkm-imjm) zv = zv - epsk(i)*pvec(i+imjm)
            if (i .gt. 1) zv = zv - epsi(i-1)*pvec(i-1)
            if (i .gt. im) zv = zv - epsj(i-im)*pvec(i-im)
            if (i .gt. imjm) zv = zv - epsk(i-imjm)*pvec(i-imjm)

            zvec(i) = zv

     100 continue
```

Fig. 3.4. Sequential Version of Application of Discretized LPB Operator

The only changes programmers need to make for implementing replicated versions of such finite-difference grid-based algorithms is to change loop boundaries and add some simple code to read arrays elements which have been updated by their guards. Figure 3.6 shows the read operations needed in this case. (The subroutine `pcomm` applies `exch` many times to accommodate reads from multiple processes. In all subsequent communication procedures, the function `gynv` returns the inverse gray code corresponding to its argument if UHBD is running on a hypercube and the appropriate switch is turned on.)

```
      call pcomm(pvec)
      do  100  i = imjmkmlo , imjmkmhi

         zv = epsc(i) * pvec(i)
         if (i .lt. imjmkm) zv = zv - epsi(i)*pvec(i+1)
         if (i .le. imjmkm-im) zv = zv - epsj(i)*pvec(i+im)
         if (i .le. imjmkm-imjm) zv = zv - epsk(i)*pvec(i+imjm)
         if (i .gt. 1) zv = zv - epsi(i-1)*pvec(i-1)
         if (i .gt. im) zv = zv - epsj(i-im)*pvec(i-im)
         if (i .gt. imjm) zv = zv - epsk(i-imjm)*pvec(i-imjm)

         zvec(i) = zv

  100 continue
```

Fig. 3.5. Replicated Version of Application of Discretized LPB Operator

```
subroutine exch(kl, kr, ip, j, vec)
c ...
integer kl
integer kr
integer ip
integer j
REAL vec(1)
integer gray, ginv

  if(j .ge. 0 .and. j .lt. numnode .and. j .ne. ip) then
  jg = gynv(j)
  ipg = gynv(ip)
  vec(kl : kr) @ ipg = vec(kl : kr) @ jg
endif
return
end
```

Fig. 3.6. Exchange of Array Elements in IPfortran for Replicated UHBD

Coding the trajectory calculations in the replicated version turns out to be quite easy. The trajectory calculations are embarrassingly parallel and the aggregate operations of IPfortran can be used to accumulate statistics for them very easily.

Figure 3.7 shows timings for the replicated version on an iPSC/860. The test case is a single-atom molecule with a single diffusing particle on a grid of size 48^3. Note that for this simple case and with a random initial distribution of trajectories, load-balancing is not an issue. Although the performance for both phases of UHBD are acceptable, the replicated nature of the implementation rules out the use of this program for grids of size greater than 48^3 on a typical iPSC/860 with 8 MB of memory per processor.

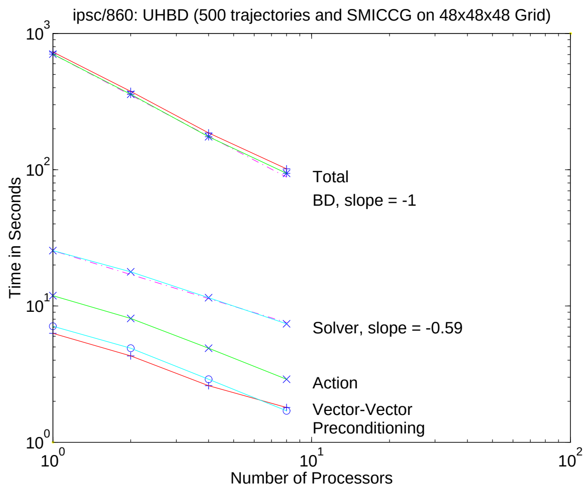

Fig. 3.7. CPU Times for calculating trajectories with a single-atom target molecule on a 48^3 grid with calculations done on an Intel iPSC/860

3.3 The Distributed Version

In the distributed version only the array holding the electrostatic potential is replicated and the rest are distributed. The distribution is by strips (a two-dimensional diagram of a strip distribution is shown in figure 3.8). Each box of the figure represents one element of an array; the bold lines surround the elements stored at each processor. Each processor also stores a buffer which contains elements fetched from neighboring processors. This approach makes the distributed version more scalable with respect to memory.

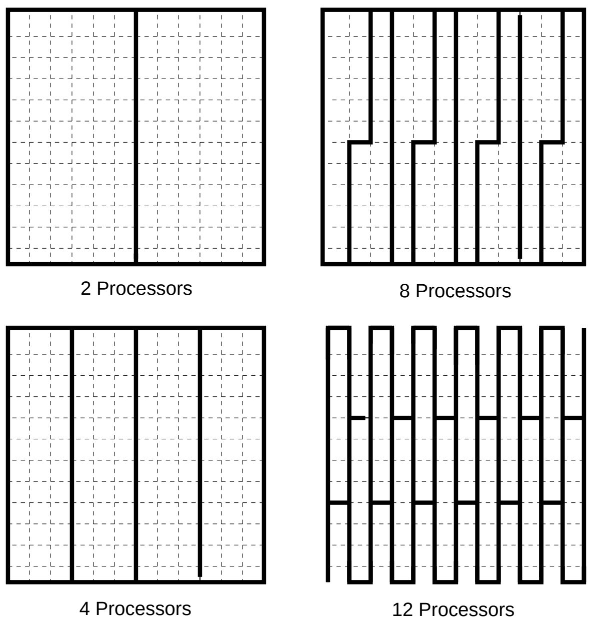

Fig. 3.8. Strip Distribution of a 2D Index Set

Figure 3.9 shows the implementation of the action of the discretized LPB operator for the distributed case. The exchange of array elements guarded by other processors is written out more fully in this case. One of the exchange routines is further elaborated in figure 3.10.

Figure 3.11 shows timings for the distributed version on an iPSC/860 for a grid of size 100^4 and 5000 trajectories.

3.4 Assessment

The replicated version of UHBD is relatively easy to code in ℙfortran because arrays have their natural shapes. That is, the i-th element of `epsc` (for example) is still called `epsc(i)` in the replicated version. In the distributed version, the i-th element of `epsc` on processor p is called `epsc(LOCAL(p,i))`. This forced name change is manageable in small programs, but for large programs like UHBD, it can be the source of many errors and complicates the programming considerably. The `foreach` statement introduced in ℙC++ (see 4) gives the programmer the flexibility to program in the natural name space of the problem.

The difficulties of programming with the forced naming restrictions of ℙfortran have in fact prevented us from distributing the electrostatics potential array among processors during BD simulation. This in turn means that there is an absolute limit to the size of problems we can solve on any given parallel machine, and this size is given by the size of memory on each processor. On a shared memory machine with a naturally global name space expressing the scalable (embarrassingly) parallel algorithm for this phase of the program is still far easier than a scalable version of the trajectory calculations would be in static ℙ [8, 64, 65].

```
      do i = 0, numnode - 2
        call exchr(i, pvec)
      enddo
      do i = numnode - 1, 1, -1
        call exchl(i, pvec)
      enddo

      do  100  i = 1 + border , imjmkmlcl + border

         zv = epsc(i) * pvec(i)
         if (i .lt. imjmkm) zv = zv - epsi(i)*pvec(i+1)
         if (i .le. imjmkm-im) zv = zv - epsj(i)*pvec(i+im)
         if (i .le. imjmkm-imjm) zv = zv - epsk(i)*pvec(i+imjm)
         if (i .gt. 1) zv = zv - epsi(i-1)*pvec(i-1)
         if (i .gt. im) zv = zv - epsj(i-im)*pvec(i-im)
         if (i .gt. imjm) zv = zv - epsk(i-imjm)*pvec(i-imjm)

         zvec(i) = zv

  100 continue
```

Fig. 3.9. Distributed Version of Application of Discretized LPB Operator

```
subroutine exchr(ip, vec)

c        ...

      integer kl, lkl
      integer kr, lkr
      integer ip, j
      REAL vec(1)
      integer gray, ginv
      kl = hithere(ip) + 1
      kr = hithere(ip) + imjm
      j = ip + 1
      jg = gynv(j)
      ipg = gynv(ip)
      lkl = LOCAL(ip, kl)
      nkl = 1 + border
      nkr = LOCAL(j, kr)

      do while(j .lt. numnode .and. nkr .gt. nthere(j) + border)
        lkr = lkl + nthere(j) - 1
        nkr = nthere(j) + border
       vec(lkl : lkr) @ ipg = vec(nkl : nkr) @ jg
        j = j + 1
         jg = gynv(j)
        lkl = lkr + 1
        nkr = LOCAL(j, kr)
      end do
      if (j .lt. numnode) then
        lkr = LOCAL(ip, kr)
        vec(lkl : lkr) @ ipg = vec(nkl : nkr) @ jg
       endif
       return
end
```

Fig. 3.10. Exchange of Array Values for Distributed UHBD

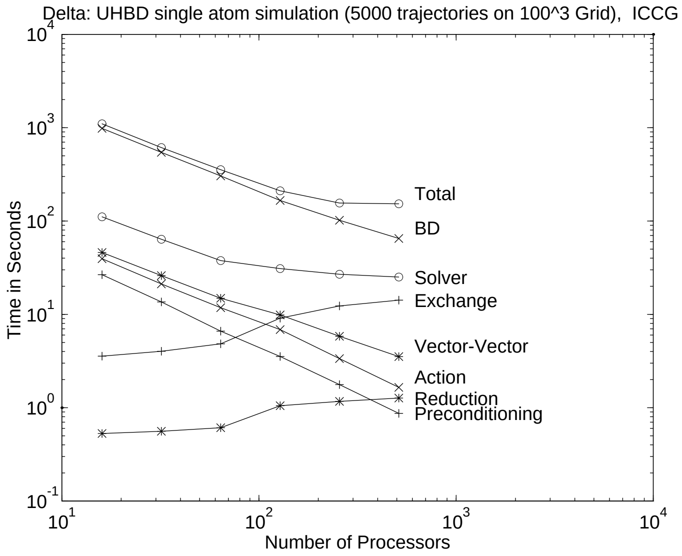

Fig. 3.11. Total CPU time for PCG and BD calculation with 5000 trajectories and a single-atom target molecule on a 100^3 grid as a function of the number of processors on the iPSC/860 with arrays distributed by strips.

There is, however, a silver lining to this major shortcoming of IPfortran. The IPfortran naming structure makes difficult the expression of algorithms precisely when they do not enjoy locality of reference. This means that how easily an algorithm can be expressed in IPfortran is a reasonably good measure of how good a parallel algorithm it is.

Chapter 4

IPC++

This chapter presents the IP extension of the C++ programming language. C++ was chosen as a base language to study the use of the interaction between the data abstraction mechanisms it provides and the IP model. An immediate advantage is that C++ objects can serve directly as the guarded of objects of IP.

4.1 Labels in IPC++

The set of labels assigned to both guarded objects and the guards which guard them are nested tuples of integers in IPC++. The nested nature of labels reflects the fact that we can have nested collections of guarded objects in IPC++. One of the design goals for IPC++ was to make the labeling process as natural as possible. Thus ss will be seen in the following sections, declaration of labled objects is patterned after declaration of arrays and nested labels are closely tied to the specification of nested guards.

4.2 Guarded Objects in IPC++

To specify a guarded object, IPC++ introduces a new storage class with the keyword `guarded` which has the same syntactic status as other storage specifiers such as `const`, `static` and `extern`. An object which is declared `guarded` can only be manipulated according to the rules for guarded memory (see chapter 2). In the current implementation, the only way to create guarded objects is to create arrays of them.

Such arrays of guarded objects are called IP arrays. Each element e of a IP array A is guarded by any guard whose label is the index of e in A. Arrays of guarded objects are declared with special syntax using the new symbol '`@`'. Just as the declarations

```
int x[5], y[2][3];
```

declare `x` and `y` to be arrays of size 5 and 2×3, the following declarations declare `x` and `y` to be IP arrays of size 5 and 2×3.

```
int x@[5], y@[2][3];
```

The syntax of IP array allocation also differs from the syntax of C++ array declaration only in the presence of the '`@`' symbol. And just as

```
new int[3][4];
```

expresses the allocation of a C++ array of size 3×4,

```
new int@[3][4];
```

expresses the allocation of a 3×4 IP array. Just as `y[1][2]` refers to an element of the array `y`, `y@[1][2]` refers to an element of the IP array y. In the existing implementation, IP array allocation is analogous to the allocation of class instances and not C++ arrays. For example, `new float@[n]` evaluates to a pointer to a IP array containing `n` floating point numbers and *not* to a pointer to the first element of a IP array. The type of `new double@[n]` is `(*double)@[]` not `*double`. Dynamically allocated IP arrays can be deallocated using `delete` in the same way as class instances.

4.3 Guards in IPC++

The `foreach` statement is the IPC++ construct for creating guards. To evaluate a foreach statement

`foreach` (e) S

is to create a guard for each element x of the IP array-valued expression e, labeled by the index of x, and to compute S in parallel in parallel with other guards. A guard which completes the computation of S or terminates while trying is said to be dead.

During the evaluation of the statement S, the label of the current guard (or the "current label") is given by the function `thisnode`, and the total number of active guards is given by the function `numnodes`. The function `thisnode` takes one or two integer arguments and returns a single integer. The argments pick out the level and dimension of the current label. That is, the arguments just indicate which component of the current nested lable it should return. The first argument indicates the level numbered from bottom up and the second argument the dimension. Both levels and dimensions are numbered beginning at 0. For example, `thisnode(0,0)` is the component of the current label at the bottom-most level and along dimension 0.

4.4 Implicit Indexing

For convenience and for historical reasons, IPC++ allows the use of the following shorthand lexical conventions. The special identifiers `$`i and `$#`i stand for `thisnode(`i`)` and `numnodes(`i`)` for $i \in \{0, 1, \ldots\}$. Adding extra dollar signs picks out components of the current label at successively higher levels; thus `$$`i stands for `thisnode(1,0)`. For example, the following statements assign the values of `x` to `y` except where `$0` has the value 1 in which case `y` is set to 42.

```
int y@[3];
foreach (y) {
  int x;
  x@[$0] = 1;
  y@[$0] = $0 == 1 ? 42 : x@[$0];
};
```

At the end of the `foreach` statement, `y` will have the values $[1, 42, 1]$.

The programmer is allowed to write g instead of g`@[$0][$1]`$\cdots$`[$`n`]` within the lexical scope of the foreach statement if n is the same as the rank of e. As the IP rules of behavior (see section 2.4) imply, expressions which involve indexing into IP arrays are points of synchronization. For example, in the statement

```
foreach (x) {
  for (int i=0; i < 3; ++i)
    x@[i] = x@[(i-1) % 3];
}
```

the effect of shifting the contents of `x` cyclically to the left *sequentially*—that is, there are 3 points of synchronization. In contrast, the statement

```
foreach (x) {
  x = x@[($0-1) % 3];
```

```
}
```

which also shifts the values of `x` to the left, contains only one point of synchronization.

These rules for implicit indexing are convenient, but they can be confusing at times. The convenience comes from the fact that within the syntactic scope of a `foreach`, programmers can pretend that they are programming in static IP except with the index set specified by the argument of the `foreach`. Confusion can arise because symbols or expressions can now stand for both a collection as a whole or a single element depending on the syntactic context and on their index set. Experience with IPC++ has shown that pretending to program in static IP within a `foreach` is a good heuristic for avoiding this confusion.

4.5 Aggregate Operations

Suppose `x` is the IP array $[1, 2, 3, 4]$. Then for every guard, the expression

```
+ { x }
```

evaluates to 10 which is the sum of the elements of `x`. User-defined operators are allowed in addition to the builtin ones (`+`, `-`, `!` , *etc.*). The order in which reductions are calculated is unspecified, thus if the operator is not associative or commutative, then the result will be implementation-dependent. There is a subtle point about aggregate operations and implicit indexing which should be noted. Aggregate operations apply only to collections, and so when implicit indexing says that a symbol stands for a single element and not for a the entire collection, aggregate operations cannot be applied. For example, to sum the numbers from one to n, one should write

```
int x@[n];

foreach (x) x = $0 + 1;

int sum = +{x};
```

It would not be legal to put the global operation inside the `foreach` statement.

4.6 IPC++ Syntax

To extend the syntax of C++ to that of IPC++, we add a small number of grammar rules to a standard grammar of C++ (see [59]). As these rules indicate, IP array declarators can appear wherever array declarators can.

array-abstract-declarator ::= *IP-abstract-declarator*

IP-abstract-declarator ::= `@` [ ]

IP-abstract-declarator ::= *IP-abstract-declarator* [ ]

IP-abstract-declarator ::= `@` [*constant-expression*]

IP-abstract-declarator ::= *IP-abstract-declarator* [*constant-expression*]

operator-new-array-declarator ::= *new-IP-declarator*

new-IP-declarator ::= @ [*expression*]

new-IP-declarator ::= *operator-new-IP-declarator* [*expression*]

foreach-statement ::= foreach (*expression*) *statement*

reduction-expression ::= *operator-or-function-name* { *expression* }

4.7 Bibliographic Notes

The programming language C was designed and implemented at AT&T Bell Laboratories by Kenighan and Ritchie [45] in the 1970's. Since then, it has become one of the most widely used programming languages. Bjarne Stroustrup has defined an extension of C, called C++[59]. This extension of C is very useful for scientific computation because of its support for defining and using abstract data types.

The C⋆ programming language resembles IPC++ more than most other existing languages. It was designed by Guy Steele and others for use on Thinking Machine Corporation's CM2 parallel processor. C⋆ also has parallel arrays and a `with` statement which is similar to `foreach` in IPC++. There are important semantic differences between the two languages, however. C⋆has a flat, synchronous model of computation, very much like the machine for which it was intended. Control can be at only one place in a C⋆ program. As a consequence, algorithms which require looser synchronization are difficult to express in C⋆. Divide and conquer algorithms make up one set of such algorithms. Strassen's matrix multiplication algorithm (see section 5.5) is one of these.

Chapter 5

Computational Kernels in IPC++

5.1 Matrices in IPC++

A matrix data type can be implemented as a class in IPC++ in much the same way as in C++ as shown in figure 5.1. Such an implementation will have the same advantages and disadvantages as in the C++ case. Operator overloading will allow a user of the class to manipulate matrices using familiar syntax. On the other hand a straight-forward implementation will introduce copying and multiple-loop overhead. The IP model allows the library implementor to express the parallelism in matrix operations easily, but it does not alleviate the other shortcomings of C++.

A typical parallel elementwize operation on matrices can be expressed simply as follows in IPC++.

```
matrix& operator+=(matrix& a) {
  foreach (s())
    s() += a.s();
  return *this;
}
```

Note that the syntactic conventions which apply to `foreach` allow us to express this operation with even simpler syntax that the typical `for` loop which would be used here.

```
typedef double Pdouble@[][];

class matrix {
  Pdouble* s_;
  int m_;
  int n_;
  Pdouble& s() {return *s_;}
 public:
  matrix(int i, int j) {
    s_ = new double@[m_=i][n_=j];
  }
  matrix(matrix&);
  ~matrix() {delete s_;}
  int m() {return m_;}
  int n() {return n_;}
  matrix& operator=(matrix& a);
  matrix& operator+=(matrix& a);
  matrix& operator-=(matrix& a);
  matrix operator+(matrix& a);
  matrix operator-(matrix& a);
  matrix operator*(matrix& a);
};
```

Fig. 5.1. A Matrix Class in PC++

5.2 Jacobi Iteration

Let us begin with one of the simplest numerical algorithms, namely the Jacobi relaxation iteration. Iterative schemes of this kind arise naturally as methods of solution for partial differential equations (see appendix B).

```
float x@[8][7];
foreach (x) {
  for (int i=0; i < iterations; ++i) {
    x = c * (x@[($0-1)%8][$1] +
             x@[($0+1)%8][$1] +
             x@[$0][($1-1)%7] +
             x@[$0][($1+1)%7]);
  }
}
```

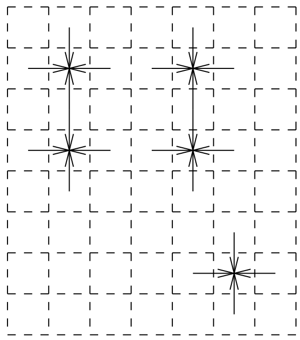

Fig. 5.2. Jacobi Application of the Five-Point Difference Stencil in IPC++

Figure 5.2 contains a program fragment, which applies the five-point difference stencil in parallel to the elements of x, and the corresponding pattern of data movement. This pattern of references is only shown for five elements. It is repeated for each element with wrap-around at the edges.

5.3 Gaussian Elimination

Gaussian elimination is one of the fundamental algorithms of numerical computation. An implementation in IPC++ is shown in figure 5.3. The algorithm is expressed in an element-by-element (as opposed to a blocked) form. The matrix a is assumed to be n by n; the iterations of gaussian elimination work on successively smaller submatrices of `a` anchored at the corner point `a@[k][k]`. For every such submatrix, the gaussian multipliers are computed and stored in the column under `a@[k][k]`. That column is then used to update the rest of the `n-k+1` by `n-k+1` matrix below `a@[k][k]`.

```
void lu(matrix& a) {
  for (int k=0; k<n; ++k) {
    foreach (a) {
      if ($0 > k) {
        a@[$0][k] /= a@[k][k];
        if($1 > k)
          a -= a@[$0][k] * a@[k][$1];
      }
    }
  }
```

Fig. 5.3. Gaussian elimination in IPC++

5.4 Matrix Multiplication

The typical algorithm for matrix multiplication illustrates the use of reductions in IPC++; it is shown in figure 5.4. Notice that we have allocated a temporary foreach element of the resulting matrix `c` to store the partial products used to compute that element. Thus the storage requirement for this implementation is greater than the usual sequential matrix multiplication algorithm.

```
matrix matrix::operator* (matrix& b) {
  matrix c(b.m(),b.n());
  foreach (b.s()) {
    c.s() = 0;
    double t@[n()];
    foreach (t)
      t = s()@[$$0][$0] * b.s()@[$0][$$1];
    c.s() = +{t};
  }
  return c;
}
```

Fig. 5.4. Matrix multiplication in IPC++

5.5 Strassen's Matrix Multiplication

This section presents a function called `smm` which implements Strassen's matrix multiplication algorithm [2] and illustrates dynamically nested parallelism in IPC++. The implementation simply divides the input matrix into four submatrices, applies Strassen's technique for multiplying two by two matrices using seven multiplications instead of eight by applying `smm` recursively. The levels of parallelism are nested as deeply as this recursive process requires. Implementing this divide and conquer algorithms without support for nested parallelism would be considerably more involved.

The function `piece` shown below is an auxiliary function which breaks a matrix into submatrices.

```
matrix piece(matrix& a, int l0, int l1, int h0, int h1) {
  int n = h0 - l0 + 1;
  matrix p(n,n);
  foreach (p.s()) {
    p.s() = a.s()@[$0+l0][$1+l1];
  }
```

```
  return p;
}
```

The function `smm` shown in figure 5.5 implements Strassen's algorithm proper. Note that the `foreach (a)` and `foreach (p)` constructs exhibit some "task-parallelism." The algorithm also illustrates nested parallelism. Each invocation of `smm` creates new sets of guards for each guard which does the invocation. Thus we have a set of trees of arrays of guards.

5.6 Fast Fourier Transform

The implementation of the fast fourier transform shown in figure 5.6 illustrates the use of guarded objects to expose a particular communication pattern, namely the FFT "butterfly" network. The algorithm is taken almost verbatim from Aho, Hopcroft and Ullman's textbook [2]. The implementation in IPC++ is in fact more concise than the psuedocode given in that textbook.

5.7 Nonbonded Force Calculation

The calculation of nonbonded forces is an example of *parallel accumulation*. In algorithms of this kind, parallel subcomputations compute contributions to a large data structure. The contributions then have to be combined in some (preferably deterministic) order. An implementation of cut-off based algorithm for non-bonded force calculation is shown in figure 5.7.

```
matrix smm(matrix& A, matrix& B) {

  int n = dimof(A, 0);
  if (n <= 8) return A * B;

  //first, pick out the pieces:
  matrix a@[2][2], b@[2][2];
  foreach (a) {
    int n0 = $0*n, n1 = $1*n;
    a = piece(A, n0,n0+n, n1,n1+n);
    b = piece(B, n0,n0+n, n1,n1+n);
  }

  //next, find the partial results:
  matrix p@[7];
  foreach (p) {
    p@[0] = smm(a@[0][1]-a@[1][1], b@[1][0]+b@[1][1]);
    p@[1] = smm(a@[0][0]+a@[1][1], b@[0][0]+b@[1][1]);
    p@[2] = smm(a@[0][0]-a@[1][0], b@[0][0]+b@[0][1]);
    p@[3] = smm(a@[0][0]+a@[0][1], b@[1][1]);
    p@[4] = smm(a@[0][0], b@[0][1]-b@[1][1]);
    p@[5] = smm(a@[1][1], b@[1][0]-b@[0][0]);
    p@[6] = smm(a@[1][0]+a@[1][1], b@[0][0]);
  }
  foreach (a) {
    a@[0][0] = p@[0] + p@[1] - p@[3] + p@[5];
    a@[0][1] = p@[3] + p@[4];
    a@[1][0] = p@[5] + p@[6];
    a@[1][1] = p@[1] - p@[2] + p@[4] - p@[6];
  }

  //last, but not least, put the pieces of the result together:
  matrix c(n,n);
  foreach (c.s()) {
    c.s() = a@[$0/n][$1/n].s()@[$0%n][$1%n];
  }
  return c;
}
```

Fig. 5.5. Strassen's matrix multiplication algorithm in IPC++

```
int k = 100;
int n = pow2(k);
complex a@[n], b@[n];
foreach (a) {
  complex r = a;
  for (int l=0; l<k; ++l) {
    complex s = r;
    r = s@[$0 & ~(1 << (k-l))] +
        root($0 << l) * s@[$0 | (1 << (k-l))];
  }
  b = r@[reverse($0)];
}
```

Fig. 5.6. Fast fourier transform in IPC++

```
// one dimensional non-bonded force calculation:

int Maxatoms = 10000;
int Maxneighbors = 10;

double nbf(const double);

main () {
  double x@[Maxatoms];  // the coordinates
  double force@[Maxatoms];  // the forces to be computed

  int no_of_neighbors@[Maxatoms];
  int neighbor@[Maxatoms][Maxneighbors];


  foreach (force) {
    double localforce@[no_of_neighbors];
    foreach (localforce) {
      localforce = nbf(x@[$0] - x@[neighbor]);
      localforce@[neighbor] -= localforce;
    }
    force = +{localforce};
    force = 1;
  }
}
```

Fig. 5.7. Non-bonded force calculation in IPC++

Chapter 6

An Evaluation of IPC++: `fec`

The finite element method is a systematic approach to the discretization of partial differential equations (see appendix B). It is a powerful and widely applicable method, but it is difficult to code in the programming language (Fortran) traditionally used for numerical computation.

This chapter describes a set of data types, called `fec`, designed for use in programs using the finite element approach to solve partial differential equations [6]. These data types have been implemented in both C++ and IPC++. The capabilities of C++ are used to encapsulate mathematical concepts as abstract data types. In this way, the finite element method has been implemented without imposing restrictions on the kinds of elements and meshes used. `fec` is designed as a hierarchical set of classes. At the bottom are classes for manipulation of vectors and matrices of low (two or three) dimensions. Above these are classes which manipulate large vectors and matrices both dense and sparse. The next level is made up of classes which manipulate objects specific to the finite element method. Besides having the usual software engineering advantages, this approach allows us to assess the expressiveness of IPC++ at each level of the hierarchy.

It turns out that the differences between the parallel and sequential versions of `fec` follow a simple theme. Where the sequential version uses arrays and `for` loops to manipulate objects in sequence, the parallel version uses IP arrays and —foreach— loops to manipulate objects in parallel. This experience with IPC++ suggests that the main contribution of IPC++ is to expand the range of applicability of "parallel loops." The use of parallel loops in shared memory languages is limited by the fact that the programmer needs to take great care to avoid non-determinism and deadlock.

6.1 Basic Classes

The basic abstract data types of linear algebra are scalars, vectors and matrices. In the finite element method (as well as other scientific programs) vectors and matrices of dimension two and three play a special role. They represent values in and linear maps on the underlying physical space of the problem. In our library of basic IPC++ classes, we have special classes for these basic objects: `Scalar`, `Vector`, and `Matrix`. All of the operations on these classes are done with small `inline`d functions. Thus low-dimensional arithmetic operations, which occur often, can be expressed easily and efficiently. The implementation of these classes in IPC++ is identical to their implementation in C++.

6.2 Container Classes

We need various collections whose elements are instances of our basic classes. We refer to these as container classes. In the sequential version of `fec`, there are five major kinds of container classes: `Vec`, `Mat`, `spMat` and `Tensor`. Arithmetic operations are defined wherever they make sense. It turned out that for the IPC++ version of `fec`, the most convenient way to incorporate parallelism at this level of the hierarchy was to introduce special "parallel" container classes `PVec`, `PMat`, and `PTensor`. The numerical functionality of these classes is identical to the functionality of their usual sequential counterparts. The difference is that the parallel classes allocate IP arrays where the sequential ones would allocate normal C++ arrays, use `foreach` statements where the sequential classes would use `for` statements and use aggregate operations where the sequential classes would accumulate values in loops.

6.3 Finite Element Classes

The finite element classes form the core of `fec`. They allow the user to build finite element discretizations of partial differential equations. A pictorial description of these classes is shown in figure 6.1. The arrows in the picture show inheritence and membership relations between classes. The major classes have hashed backgrounds; a solid arrow means that the class at tail of the arrow is derived from the class at its head; a dotted arrow means that the class at the head of the arrow appears as a member in the class at its tail (multiple dotted arrows mean that more than one such member exists).

A user of `fec` can compute with a notation very similar to the notation a mathematician would use. With the declarations

```
LagrangeElement E(4);
Mesh M(meshfile);
Fes F(M,E)
Interpolant u(F);
Interpolant v(F);
KOperator a(F,F);
```

we can use `a` in our programs in the same way as we would use the form $a(\cdot,\cdot)$ (of appendix B) to describe an algorithm mathematically. For example, figure 6.2 contains the

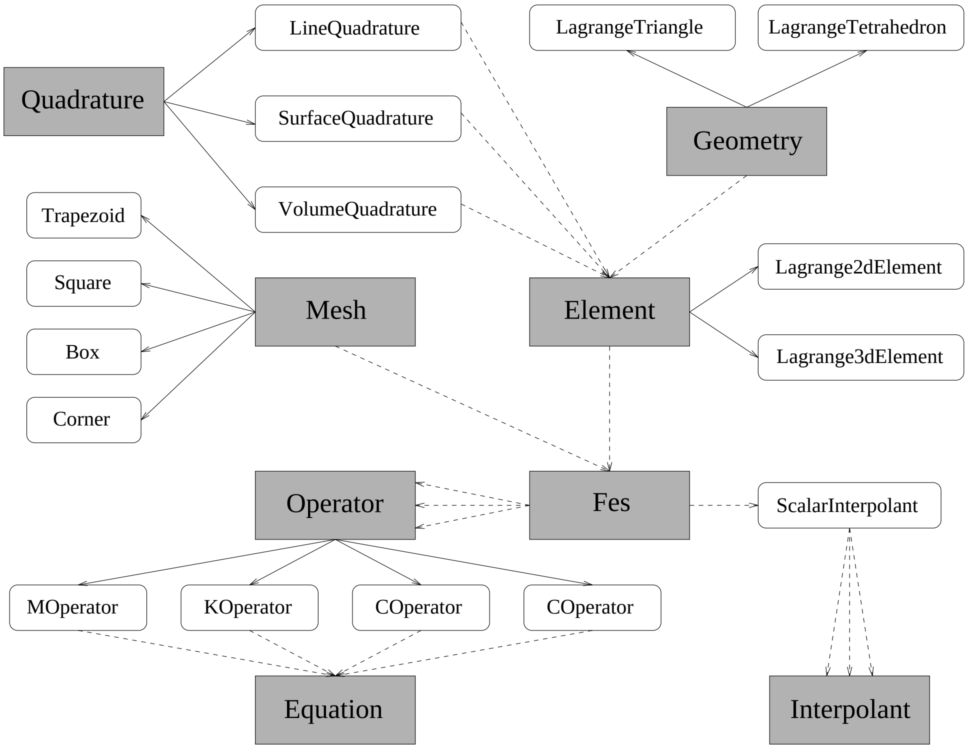

Fig. 6.1. The finite element classes of `fec` and their relations

conjugate gradients algorithm for solving an equation whose left-hand side is discretized by the `Operator a`. with initial guess `x` and maximum number of iterations `iters`.

In this way, many different solvers have been implemented and investigated for various formulations of the Stokes and Navier-Stokes equations which model fluid flow [12, 11]. We have found that the solvers are the simplest part of the code and the easiest to write. In fact at this level of programming, the $\mathbb{P}$ model aspects of $\mathbb{P}$C++ do not play a role. Iterative solvers in $\mathbb{P}$C++ look the same as they do in C++. The more challenging part of each implementation is to write efficient code for new `Operator`s (see below) which appear in the equations.

The following subsections give a description of the finite element classes of `fec` and an assessment of the impact of parallelism as expressed in $\mathbb{P}$C++. The basic design of the parallel `fec` is to compute the local bilinear forms sequentially, but in parallel across elements and to accummulate the results. This approach is similar to the one used in section 5.7 to accumulate nonbonded forces in molecular dynamics calculations.

6.3.1 Quadrature and Descendants

Objects of class `Quadrature` contain the information necessary for numerical integration. (Currently, we use Gaussian quadrature.) The same data structures can be used for volume, surface and line integration, thus `Quadrature` serves as the base class for `ElementQuadrature` and `LineQuadrature`. These two classes then implement quadrature over a reference element (e.g. a triangle) and a reference line respectively. All existing quadrature classes maintain two arrays holding quadrature points and weights and implement quadrature using a simple `for` loop. Expressing parallelism within the

```
Interpolant cg (Operator& a, Interpolant& x,
                Interpolant& b, int iters)
{
  Interpolant newx = x;
  Interpolant w(b, 0.0);
  Interpolant r = b - a(x);
  Interpolant p = r;
  Scalar rho = r * r;
  Scalar nu = rho;
  Scalar alpha = 0;
  for(int k=0; k<iters && sqrt(rho)>tolerance; ++k) {
    if (k>0) { p = r + (rho/nu) * p;}
    w = a(p);
    alpha = rho/(w * p);
    newx += alpha * p;
    r -= alpha * w;
    nu = rho;
    rho = r * r;
  }
  return newx;
}
```

Fig. 6.2. Conjugate Gradients in `fec`

quadrature class would involve changing the data structures which holds the points and weights from arrays to $\mathbb{P}$ arrays and changing the `for` loop to a `foreach` loop. As quadrature over a single element is carried out sequentially, parallelization using $\mathbb{P}$C++ introduces no change into this subhierarchy of classes.

6.3.2 ElementGeometry and Descendants

The class `ElementGeometry` encapsulates the geometry of a reference element. This geometric information consists of a description of the boundary of the element and the location of the nodal points which determine shape functions. As with the quadrature classes, element geometries are left unchanged after parallelization.

6.3.3 Element and Descendants

The class `Element` encapsulates the mathematical notion of finite element. Objects of class `Element` contain three objects which determine their geometry and define surface and line quadrature. They use the information in these members to calculate shape functions and gradients of shape functions and nodal variables. The usual mathematical definition of a finite element does not explicitly include the notion of integration. That notion is implicit; we assume the relevant functions are integrable and the relevant domains are measurable. For computational purposes, however, it is convenient to separate out the notion of integration. Again, as operations on single elements are not parallelized, the reference element class and its descendents are left unchanged by parallelization.

6.3.4 Mesh and Descendants

A `Mesh` represents a collection of elements (restricted to be identical polygonal domains) which together comprise a polygonal domain in $\mathbb{R}^2$. The representation is such that we can carry out the necessary mesh operations efficiently. These operations include: local to global vertex numbering, mesh subdivision, identification of boundary edges, etc. It is convenient to generate certain meshes at run time. The descendants of `Mesh` have constructors do so. For example, `Square` and `Trapezoid` generate "standard" square and trapezoidal meshes. The effect of parallelization on the mesh class and its descendents is small. Only two data structure which hold information for each general element (the coordinates and connectivity of the vertices of the element) change from C++ arrays to $\mathbb{P}$ arrays. And sequential `for` loops manipulating these structures are changed to `foreach` statments.

6.3.5 Fes

A finite element space (called `Fes`) is what we get by putting together a `Mesh` and an `Element`. Thus it includes all of the information necessary for discretizing functions and operators. `Fes` contains the Jacobians of the maps from the reference element to a general mesh element, the local to global basis numbering, and information for evaluating nodal variables. The effect of parallelization on this class is similar to the effect on the mesh class. Data structures which hold information for each element change from C++ arrays to $\mathbb{P}$ arrays. There are four such data structures in this class, namely the structures which hold the

6.3.6 Interpolant

The class `Interpolant` has two members: a pointer to a `Fes` and a vector of vectors containing the components of the interpolant of a function on that `Fes`. The two members allow us to define operators on `Interpolant`s using the discretization information in their `Fes`.

6.3.7 Operator and Descendants

The class `Operator` represents the action of a discretized variational form on an `Interpolant`. Its main member is a routine which carries out the action on a given `Interpolant`. To compute bilinear forms efficiently, it is often necessary to precompute various matrices or tensors on the reference element. Thus each particular `Operator` should have its own initializing routine. For this reason, it is convenient to implement each multilinear form as a class derived from `Operator`. Figure 6.3 contains a routine which performs the action of the form $a(\cdot, \cdot)$.

This routine "assembles" the action of $a(\cdot, \cdot)$ by summing the contributions of each element in the mesh on which `u` is defined . The matrix `elementmat` is the "stiffness matrix" on the reference element and has been precomputed by the constructor of class `KOperator` which is derived from `Operator`. We have implemented `Operator`s for all of the terms appearing in many formulations of the Navier-Stokes equations [31, 62].

Each descendant of `Operator` is a direct translation into C++ of an element-by-element assembly of a bilinear form as discussed in section 2. It is not a trivial task, however, to perform the element-by-element assembly of a multilinear form in such a

```
Interpolant KOperator::
operator () (Domain d, Interpolant& u)
{
  Interpolant v(d, u, 0.0);
  int gi, gj;
  int m = elementmat.rsize();
  int n = elementmat.csize();
  Matrix (&jd)@[] = space().jacobian_determinants();
  Matrix (&ji)@[] = space().jacobian_inverse();
  foreach (jd) {
    Matrix jijit = jd * (ji * transpose(ji));
    for ( LocalBasisNum i=0; i<m; ++i )
      if ( valid(gi = v[0].index(e,i)) )
        for ( LocalBasisNum j=0; j<n; ++j )
          if ( valid(gj = u[0].index(e,j)) ) {
            Scalar eij = jijit ^ elementmat[i][j];
            for (register k=0; k<v.capacity(); ++k)
              v[k]@[gi] += eij * u[k]@[gj];
          }
  }
  return v;
}
```

Fig. 6.3. Assembly of Bilinear Form a in `fec`

way that a direct translation will produce efficient code. One must be careful about factoring the sums to minimize the amount of arithmetic to be performed.

Part II

Understanding $\mathbb{P}$

The truth of the matter is that putting languages together is a very tricky business. When one attempts to combine language concepts, unexpected and counterintuitive interactions arise. At this point, even the most experienced designer's intuition must be buttressed by a rigorous definition of what the language means.

— **John C. Reynolds**[1]

[1] In [61].

Chapter 7

IPtiny and its Semantics

This chapter presents the semantics of a small parallel language, called IPtiny, which is based on the IP model. There are two equally important reasons for defining the semantics of IPtiny precisely. Firstly, it is important to make clear what is meant by "the IP model" independently of any detailed implementation. Secondly, it is equally important to be sure that the semantics we have in mind do in fact meet the objectives we have in mind. To these ends, the operational semantics of IPtiny are defined here using an elegant approach introduced by Plotkin and Henessy [36]. In this style of semantic definition, one defines the operational semantics of a language by specifying inference rules for deducing the behavior of a construct from the behavior if its subconstructs. The inference rules hide the details of traversing the syntax trees of program constructs and allow one to concentrate on the interactions among program units.

7.1 Preliminaries

In this chapter, function applications are denoted by juxtaposition and group to the left when parentheses are left out. More precisely, a string of applications $f_1 f_2 \cdots f_{n-1} f_n$, stands for $((\ldots(f_1 f_2)\ldots f_{n-1})f_n)$. This convention in turn makes it natural that expressions denoting function spaces group to the right. More precisely, if $X_0, X_1, \ldots, X_{n-1}, X_n$ are sets, then $X_0 \to X_1 \to \ldots \to X_{n-1} \to X_n$ stands for $X_0 \to (X_1 \to (\ldots(X_{n-1} \to X_n)\ldots))$.

Inference rules are used to define both the syntax and the semantics of $\mathbb{P}$. I will use the following notation for inference rules:

$$\frac{P_1 P_2 \cdots P_n}{Q_1 Q_2 \cdots Q_n}.$$

This notation means that if the premises P_i hold, then one can deduce the conclusions (or consequences) Q_i for $i \in \underline{n}$. If there are no premises, then the rule is in fact an axiom.

The letters E, V, v, and a stand for complex expressions (*i.e.* expressions other than simple variables, values or addresses), variables, values and addresses respectively. For a subset S of a function f, $f\colon S \mapsto x$ means that f maps every element of S to x.

7.2 Types and Syntax

There is one basic type in $\mathbb{P}$tiny, namely $\mathbf{i}$ (the integers). The full set of types is specified by the following BNF-like rules.

$$\begin{array}{lllllllllll}
\boldsymbol{\beta} & ::= & \mathbf{i} \\
\boldsymbol{\nu} & ::= & \boldsymbol{\beta} & \mid & \boldsymbol{\beta} \to \boldsymbol{\nu} \\
\boldsymbol{\theta} & ::= & \boldsymbol{\nu} & \mid & \mathbf{l}_{\boldsymbol{\nu}} & \mid & \mathbf{r}_{\boldsymbol{\nu}} & \mid & \mathbf{e}_{\boldsymbol{\nu}} & \mid & \mathbf{s}
\end{array}$$

In other words, the set of basic types, over which $\boldsymbol{\beta}$ will range, is $\boldsymbol{B} := \{\mathbf{i}\}$, the set of value types, over which $\boldsymbol{\nu}$ will range, is

$$\boldsymbol{V} := \boldsymbol{B} \cup (\boldsymbol{B} \to \boldsymbol{B}) \cup (\boldsymbol{B} \to (\boldsymbol{B} \to \boldsymbol{B})) \cup \ldots,$$

where $X \to Y := \{x \to y \mid x \in X, y \in Y\}$, and similarly the set of all types is $\boldsymbol{T}$ over which $\boldsymbol{\theta}$ will range.

For each value type $\boldsymbol{\nu}$, there is a set of constants $\mathcal{C}_{\boldsymbol{\nu}}$ and a set of variable identifiers $\mathcal{V}_{\boldsymbol{\nu}}$. The set of constants of type integer $\mathcal{C}_{\mathbf{i}}$ is just the set of integers $\mathbb{Z}$. Constants of array type and variables are syntactic domains whose lexical structures are unimportant to us and so will remain unspecified.

The syntax and type system of IPtiny are defined at the same time by sets of inference rules in table 7.1. In this context, an inference rule says that if the constructs mentioned in its premise are of the given types, then the construct mentioned as its consequence is in the language and has the given type. For example, a parenthesization rule,

$$\frac{E : \mathbf{r}_{\boldsymbol{\nu}}}{(E) : \mathbf{r}_{\boldsymbol{\nu}}},$$

would say that if E is an expression of type $\mathbf{r}_{\boldsymbol{\nu}}$, for some value type $\boldsymbol{\nu}$, then (E) is syntactically valid and is also of type $\mathbf{r}_{\boldsymbol{\nu}}$. The notation $X\colon \tau_1, \tau_2$ means that X is either of type τ_1 or of type τ_2.

The following shorthand is useful. A parallel sequence all of whose members are S is denoted by

$$||^n\, S := S||S||S||\ldots||S := \{\{\{\{S||S\}||S\}\ldots\}||S\}.$$

7.3 Values, Addresses, Labels, and States

Labeled processes and objects (represented by their addresses) are the basic ingredients of the semantics of IPtiny. The particular set of labels and labeling mechanism used here are somewhat incidental. Here labels are sequences of integers and are tied

Table 7.1
Syntax of $\mathbb{P}$tiny

$\dfrac{E : \mathbf{e}_{\boldsymbol{\nu}}}{(E) : \mathbf{e}_{\boldsymbol{\nu}}}$	Parentheses
$\dfrac{}{C : \boldsymbol{\nu}} \quad C \in \mathcal{C}_{\boldsymbol{\nu}}$	Constants
$\dfrac{}{V : \mathbf{l}_{\boldsymbol{\nu}}} \quad V \in \mathcal{V}_{\boldsymbol{\nu}}$	Variables
$\dfrac{V : \mathbf{l}_{\boldsymbol{\nu}}, \mathbf{r}_{\boldsymbol{\nu}}}{V : \mathbf{e}_{\boldsymbol{\nu}}}$	Coercion
$\dfrac{E : \mathbf{e}_{\boldsymbol{\beta}\rightarrow\boldsymbol{\nu}} \quad i : \mathbf{e}_{\boldsymbol{\beta}}}{E[i] : \mathbf{l}_{\boldsymbol{\nu}}}$	Selection
$\dfrac{E_1 : \mathbf{l}_{\boldsymbol{\nu}} \quad E_2 : \mathbf{e}_{\boldsymbol{\nu}}}{E_1 = E_2; : \mathtt{s}}$	Assignment
$\dfrac{E : \mathbf{e}_{\boldsymbol{\nu}} \quad S_1 : \mathtt{s} \quad S_2 : \mathtt{s}}{\mathtt{if}\ (E)\ S_1\ \mathtt{else}\ S_2 : \mathtt{s}}$	Choice
$\dfrac{E : \mathbf{e}_{\boldsymbol{\nu}} \quad S : \mathtt{s}}{\mathtt{while}\ (E)\ S : \mathtt{s}}$	Repetition
$\dfrac{E : \mathbf{e}_{\mathrm{i}\rightarrow\boldsymbol{\nu}} \quad i : \mathbf{e}_{\mathrm{i}}}{E@[i] : \mathbf{l}_{\boldsymbol{\nu}}}$	Indexing
$\dfrac{E : \mathbf{e}_{\mathrm{i}\rightarrow\boldsymbol{\nu}'} \quad S : \mathtt{s}}{\mathtt{foreach}\ (E)\ S : \mathtt{s}}$	Parallelism
$\dfrac{S_1 : \mathtt{s} \quad S_2 : \mathtt{s}}{S_1 \| S_2 : \mathtt{s}}$	Parallelism

to indices of special arrays (as we shall see in the next section). This approach fits well with the domain of applications we have in mind, namely scientific computation. Other labeling mechanisms may be useful for other application domains.

Definition 1 *The set of* labels *is defined to be the set of all nested sequences of integers. The empty label, ϵ, is defined to be the empty sequence: $\epsilon := []$.*

The *semantic domains* which help define the meaning of $\mathbb{P}$tiny expressions are quite simple. The lvalue domain of all types is a set of addresses $\mathcal{A}$, while the rvalue domain of all types is just the set of integers $\mathbb{Z}$. In principle, we could avoid the extra set of addresses but this would obscure and complicate the semantics of labeling.

Definition 2 *The set of addresses is defined to be the set of integers:*

$$\mathcal{A} := \mathbb{Z}.$$

The set of counters is defined to be pairs of addresses and labels:

$$\mathcal{K} := \Lambda \times \mathcal{A}.$$

The counting function $K : \Lambda \to \mathcal{A} \to \mathcal{K}$ is defined by

$$K \lambda a := [\lambda, a].$$

Counters are used to keep track of the "generation" to which a computation belongs. According to IP semantics, readers of guarded objects see the version which belongs to their own generation.

Definition 3 *The domain of "lvalues" is defined to be the set of addresses:*

$$\mathcal{L} := \mathcal{A}.$$

The domain of "rvalues" is defined to be the same as the set of constants of type integer:

$$\mathcal{R} := \mathcal{C}_{\mathbf{i}}(= \mathbb{Z}).$$

The set of all values is defined by

$$\mathcal{D} := \mathcal{L} \cup \mathcal{R}.$$

Our semantic domains are ordinary sets because the expressions in IPtiny are simple: they do not contain recursion or looping constructs. There is no domain of values for statements because the style of operational semantics we use does not require such values.

A few more functions are needed to define the meaning of IPtiny constructs. These functions are left unspecified and are only required to satisfy certain conditions. The labeling function $L\colon \mathcal{A} \to \Lambda$ gives the label of each address. The address function $A\colon \mathcal{V} \to \mathcal{A}$ gives the address of each variable. For variables of collection (array and IP array) type, there is an indexing function $I\colon \mathcal{A} \to \mathcal{D} \to \mathcal{A}$ maps the base address of a collection and an index value to the address of a single element.

Definition 4 *The set of states for IPtiny is defined to be the set of maps from addresses to sequences of values or from counters to the set of integers plus the special value ∞ which obeys the usual formal rules for a positive infinite ordinal added to the integers ($\infty > i$, $\infty + i = \infty$, $\infty - i = \infty$ for any $i \in \mathbb{Z}$).*

$$\Sigma := (\mathcal{A} \to \mathcal{D}^{\Omega}) \cup (\mathcal{K} \to (\mathbb{Z} \cup \infty)).$$

7.4 Semantics

The meaning of a IPtiny program is a sequence of transitions between configurations of an abstract machine. More precisely, we have the following definition.

Definition 5 *A* labeled transition system *T is a triple $[\Xi, T, \rhd]$ where Ξ is a set of configurations, $T \subseteq \Xi$ is a set of* terminal configurations *and $\cdot \overset{\cdot}{\rhd} \cdot \subseteq \Xi \times \Lambda \times \Xi$ is a* labeled transition relation. *The middle component of a transition, placed above the symbol $\rhd$, is its* label. *A configuration κ is said to be* maximal *(in Ξ) if there is no configuration (in Ξ) κ' such that $\kappa \to \kappa'$.*

A labeled transition $\kappa \overset{\lambda}{\rhd} \kappa'$ models one computational step carried out by a process whose label is λ. The labeled transition relation which defines the semantics of IPtiny will be defined by a set of inference rules. *The* transition relation defined by a set of inference rules $\mathcal{I}$ is the smallest relation (according to set inclusion) which satisfies $\mathcal{I}$.

I will use the following shorthand for inference rules involving transitions between configurations (see section 7.4). A rule of the form

$$\frac{\kappa \ \triangleright\ \kappa_1 \mid \kappa_2 \mid \cdots \mid \kappa_n}{\kappa' \ \triangleright\ \kappa'_1 \mid \kappa'_2 \mid \cdots \mid \kappa'_n}$$

stands for the set of rules

$$\frac{\kappa \ \triangleright\ \kappa_i}{\kappa' \ \triangleright\ \kappa'_i} \quad i \in \underline{n}.$$

If transition labels are missing in a rule, then the transitions in the premise(s) and those in the consequence are required to have the same label.

Definition 6 *The transition system $\mathcal{T}_p$ for $\mathbb{P}$tiny is the triple $[K, T, \triangleright]$ where the set of configurations is $K := (\mathcal{S} \cup \mathcal{D}) \times \Sigma$, the set of terminal configurations is $T := (\{;\} \cup \mathcal{D}) \times \Sigma$, and the labeled transition relation $\dot{\triangleright}$ is the labeled transition relation defined by the rules in tables 7.2, 7.3, 7.4, 7.5, and 7.6. The initial configuration for a $\mathbb{P}$tiny program P is $\langle P, \{\mathcal{K} \mapsto 0, \mathcal{A} \mapsto []\}\rangle$.*

7.4.1 Access to Memory

At the heart of $\mathbb{P}$tiny semantics is the meaning of access to (guarded) objects, so the rules which define such access come first (in table 7.2). The crucial point about these semantics is that we have augmented the state of the computation to keep a counter for each address-label pair representing a guarded object. Both guards and non-guards increment their counter as they access objects. Non-guards wanting to read an object then wait if necessary for the value of the guard's counter to reach the value of their own counter before they complete their read. In the case when the guard modifies a guarded object and moves ahead, a reader of that guarded object just picks out its value corresponding to the reader's own counter for that object. States keep track of sequences of values instead of single values to simplify specification of these semantics. This matching of counters is the crux of the $\mathbb{P}$ model.

7.4.2 Basic Expressions

The basic expressions of $\mathbb{P}$tiny are those which are atomic, parenthesized, or the special variable `thisnode`. The semantics of atomic expressions are delegated to the unspecified transition relation $\underset{a}{\triangleright}$. The rules labeled "Inquiry" say that the special variable `thisnode` evaluates to the label of current subcomputation in a parallel context, but that we get an absurd value (-1) if we try to evaluate `thisnode` in a sequential context. Note that this semantic choice is arbitrary; `thisnode` could evaluate to an error state or to some other agreed-upon value.

7.4.3 Indexing

The rules for indexing into arrays simple say that the index expression must be evaluated first and that the address of an element is calculated from the "base" address of the array.

7.4.4 Basic Statements

The basic statements of $\mathbb{P}$tiny are those statements which $\mathbb{P}$tiny has in common with basic sequential languages. The semantics of these are shown in figure 7.5. The only remarkable aspect of the formalism here is that the rules labeled Conditional together

Table 7.2
Access to Memory

$$\frac{}{\langle V, \sigma\rangle \rhd \langle AV, \sigma\rangle} \qquad \text{Variables}$$

$$\frac{}{\langle a, \sigma\rangle \overset{\lambda}{\rhd} \langle \tau(\sigma a), \sigma\rangle} \quad \begin{cases} L(a) = \epsilon & \text{or} \\ \sigma(K\lambda a) = \infty \end{cases} \qquad \text{Ordinary}$$

$$\frac{}{\langle a, \sigma\rangle \overset{\lambda}{\rhd} \langle \tau(\sigma a), \sigma[K\lambda a \mapsto \sigma(K\lambda a) + 1]\rangle} \quad L(a) = \lambda \qquad \text{Guard}$$

$$\frac{}{\langle a, \sigma\rangle \overset{\mu}{\rhd} \langle (\sigma a)_{\sigma m}, \sigma[m \mapsto \sigma m + 1]\rangle} \quad \begin{cases} m := K\mu a, \\ \lambda := L(a) \neq \mu, \quad \text{and} \\ \sigma m \leq \sigma(K\lambda a) \end{cases} \qquad \text{Others}$$

$$\frac{\langle E_2, \sigma\rangle \rhd \langle E_2', \sigma'\rangle \mid \langle v, \sigma'\rangle}{\langle E_1 = E_2;, \sigma\rangle \rhd \langle E_1 = E_2';, \sigma'\rangle \mid \langle E_1 = v;, \sigma'\rangle} \qquad \text{RHS}$$

$$\frac{\langle E_1, \sigma\rangle \overset{\lambda}{\rhd} \langle a, \sigma'\rangle}{\langle E_1 = v;, \sigma\rangle \overset{\lambda}{\rhd} \langle ;, \sigma'[a \mapsto \sigma' a \oplus [v]][l \mapsto \sigma' l + 1]\rangle} \quad \begin{cases} l := K\lambda a \quad \text{and} \\ L(a) = \lambda, \end{cases} \qquad \text{LHS}$$

$$\frac{\langle E_1, \sigma\rangle \overset{\lambda}{\rhd} \langle a, \sigma'\rangle}{\langle E_1 = v;, \sigma\rangle \overset{\lambda}{\rhd} \langle ;, \sigma'\rangle} \quad L(a) \neq \lambda \qquad \text{LHS}$$

Table 7.3
Basic $\mathbb{P}$tiny Expressions

$$\frac{\langle X,\sigma\rangle \underset{a}{\triangleright} \langle X',\sigma'\rangle}{\langle X,\sigma\rangle \triangleright \langle X',\sigma'\rangle}$$	Atomic
$$\frac{\langle E,\sigma\rangle \triangleright \langle E',\sigma'\rangle}{\langle (E),\sigma\rangle \triangleright \langle E',\sigma'\rangle}$$	Bracketing
$$\frac{}{\langle \texttt{thisnode},\sigma\rangle \overset{\lambda}{\triangleright} \langle \lambda,\sigma\rangle}$$	Inquiry
$$\frac{}{\langle \texttt{thisnode},\sigma\rangle \overset{\epsilon}{\triangleright} \langle -1,\sigma\rangle}$$	Inquiry

Table 7.4
Indexing in $\mathbb{P}$tiny

$$\frac{\langle E_2,\sigma\rangle \triangleright \langle E_2',\sigma'\rangle \mid \langle v,\sigma'\rangle}{\langle E_1[E_2],\sigma\rangle \triangleright \langle E_1[E_2'],\sigma'\rangle \mid \langle E_1[v],\sigma'\rangle}$$	Index
$$\frac{\langle E_1,\sigma\rangle \triangleright \langle E_1',\sigma'\rangle \mid \langle a,\sigma'\rangle}{\langle E_1[v],\sigma\rangle \triangleright \langle E_1'[v],\sigma'\rangle \mid \langle Iav,\sigma'\rangle}$$	Element

with the interleaving semantics for parallelism (given in the following subsection) imply that replicated conditionals can make independent choices about which branch of the conditional to evaluate.

7.4.5 Parallel Execution

The rule labeled "Parallelism" just says that to deduce a transition for a parallel sequence, we must be able to deduce a transition with an extended label for one of its components. Thus a transition for the whole parallel construct can follow from a transition for *any* of its components. This fact leads to the interleaving model of parallelism embodied by the IPtiny semantics. The letter **S** in table 7.6 stands for any parallel sequence os statements.

7.5 Simplifications

To make the presentation more readable and concise, I have made IPtiny simple compared with a language (like IPC++) which would be useful. The most drastic simplification is that there are no functions or procedures in IPtiny. Adding them would require some additional rules and a way to deal with local variables. The basic framework of the semantics would remain the same, however. The less drastic simplifications are as follows. The access variable (or function in the case of IPC++) `numnodes` which exists in practical IP implementations and gives the shape of the current set of active guards is missing. Adding it would only complicate the semantics without contributing to an understanding of IP. IPtiny has only one-dimensional arrays; but they can be nested, so nothing is lost. If we were to distinguish multidimensional arrays from nested ones, then

Table 7.5
Basic IPtiny Statements

$\dfrac{\langle A, \sigma\rangle \underset{a}{\rhd} \langle A', \sigma'\rangle}{\langle A, \sigma\rangle \rhd \langle A', \sigma'\rangle}$		Atomic
$\dfrac{}{\langle \{S\}, \sigma\rangle \rhd \langle S, \sigma\rangle}$		Bracketing
$\dfrac{\langle S_1, \sigma\rangle \rhd \langle S_1', \sigma'\rangle \mid \langle ;, \sigma'\rangle}{\langle S_1 S_2, \sigma\rangle \rhd \langle S_1' S_2, \sigma'\rangle \mid \langle S_2, \sigma'\rangle}$		Sequencing
$\dfrac{\langle E, \sigma\rangle \rhd \langle E', \sigma'\rangle}{\langle \texttt{if}\ (E)\ S_1\ \texttt{else}\ S_2, \sigma\rangle \rhd \langle \texttt{if}\ (E')\ S_1\ \texttt{else}\ S_2, \sigma\rangle}$		Conditional
$\dfrac{\langle E, \sigma\rangle \rhd \langle v, \sigma'\rangle \mid \langle 0, \sigma'\rangle}{\langle \texttt{if}\ (E)\ S_1\ \texttt{else}\ S_2, \sigma\rangle \rhd \langle S_1, \sigma'\rangle \mid \langle S_2, \sigma'\rangle}$	$v \neq 0$	Conditional
$\dfrac{}{\langle \texttt{while}\ (E)\ S, \sigma\rangle \rhd \langle \texttt{if}\ (E)\ \{S\ \texttt{while}\ (E)\ S\}\ \texttt{else}\ ;, \sigma\rangle}$		Repetition

Table 7.6
Parallel Execution of IPtiny Statements

$\dfrac{\langle S, \sigma\rangle \overset{\lambda\oplus[i]}{\rhd} \langle S', \sigma'\rangle \mid \langle ;, \sigma'\rangle}{\langle \mathbf{S}, \sigma\rangle \overset{\lambda}{\rhd} \langle \mathbf{S}[i \mapsto S'], \sigma'\rangle \mid \langle \mathbf{S}[i \mapsto ;], \sigma'[Ki \mapsto \infty]\rangle}$	$S = \mathbf{S}_i\ 1 \leq i \leq \lvert\mathbf{S}\rvert$	Parallelism
$\dfrac{}{\langle \lvert\rvert^n ;, \sigma\rangle \rhd \langle ;, \sigma[K \mapsto 0]\rangle}$		Join
$\dfrac{\langle E, \sigma\rangle \rhd \langle v, \sigma'\rangle}{\langle \texttt{foreach}\ (E)\ S, \sigma\rangle \rhd \langle \lvert\rvert^n S, \sigma'\rangle}$	$n := \lvert v\rvert$	Fork

the set of labels would have to reflect this distinction because the shapes of computations are determined by the shapes of arrays. In that case, labels would naturally be nested sequences. Finally, declarations are missing.

7.6 Bibliographic Notes

At this writing, many theoretical approaches toward defining the meaning of computer programs have been explored by many writers. In the denotational approach, one defines the meaning of a program to be a mathematical object, usually by induction on the syntax of the program and the use of fixed point techniques to deal with loops and recursion. In the operational approach, one defines the meaning of a program by specifying the behavior of a machine executing the program. The denotational approach to the semantics of programming languages was invented by Scott and Strachey [58]. By now, there are many textbooks on the subject [55, 57, 61].

The denotational approach works well for sequential imperative languages, but the construction of the semantic domains becomes much more complicated for the statements of explicitly parallel programs [53]. It is both easier and more helpful to define the semantics of such languages operationally. A style of defining operational semantics well suited for this purpose was introduced by Plotkin and Henessy. In that style, one defines operational semantics by giving inference rules for deducing the meaning of a statement from the meaning of its constituents. This is the approach followed in this dissertation. This technique was introduced by Plotkin and Henessy [36], later applied to CSP [54], and refined by others [5].

Chapter 8

Determinism and Deadlock

The IP model was designed to be deterministic. In this chapter I show that this is so by proving determinism for IPtiny. Not surprisingly the proofs are straight-forward (although perhaps tedious). Although IP programs can in general lead to deadlock, it is easy to identify computations which may do so. In this chapter, I give sufficient conditions which can be used to detect these computations.

8.1 Determinism

A deterministic computation is distinguished by the fact that a single configuration cannot lead to many different final configurations.

Definition 1 *A transition system* $[K, T, \rhd\,]$ *is* deterministic *iff for any configuration* κ*, if there is a maximal configuration* μ *such that* $\kappa \overset{*}{\rhd} \mu$*, the* μ *is unique; i.e. if* $\kappa \overset{*}{\rhd} \mu'$*, then* $\mu' = \mu$*.*

It is possible to reduce the global determinism property to a local "diamond property" which can then be proven using the IPtiny inference rules.

Definition 2 *Let* $\prec$ *be a binary relation on a set* X*.* $\prec$ *has the* diamond property *iff for all elements* $x, y, z \in X$ *with* $y \neq z$*, if* $x \prec y$ *and* $x \prec z$*, then there is an element* $w \in X$ *such that* $y \prec w$ *and* $z \prec w$*. A binary relation* $\prec$ *is* confluent *iff its transitive closure* $\prec^*$ *has the diamond property. An element* $x \in X$ *is* $\prec$-maximal *iff there is no element*

$y \in X$ such that $x \prec y$. A maximal finite $\prec$-sequence *is a sequence $x_1 \prec x_2 \prec \ldots x_n$ where x_n is maximal.*

Lemma 1 *If a binary relation has the diamond property, then it is confluent.*

Proof: See [52]. ∎

Lemma 2 *A transition system $\mathcal{T} = [K, T, \rhd]$ is deterministic if it has the diamond property.*

Proof: Take any configuration κ. The determinism condition says that all maximal finite sequences beginning with κ must lead to a *unique* maximal configuration μ. Suppose there are maximal configurations μ_1 and μ_2 such that $\kappa \overset{*}{\rhd} \mu_1$ and $\kappa \overset{*}{\rhd} \mu_2$. Now $\rhd$ is confluent, so there must be a configuration ν such that both $\mu_1 \overset{*}{\rhd} \nu$ and $\mu_2 \overset{*}{\rhd} \nu$. But μ_1 and μ_2 are maximal, so we must have $\mu_1 = \nu$ and $\mu_2 = \nu$. Hence $\mu_1 = \mu_2$. ∎

Theorem 1 *The IPtiny transition system $\mathcal{T}_p$ given in definition 6 is deterministic.*

Proof: By lemma 2, we only need to show that the transition relation $\rhd$ of $\mathcal{T}_p$ is confluent. The only configurations which can have more than one successor in $\rhd$ are of the form

$$\langle S_1 || S_2 || \ldots || S_n, \sigma \rangle .$$

This is so because configurations of this form are the only ones which appear as the left hand side of a transition which is a consequent of more than one rule all whose premises can be true for a given state. The set of rules in question is the one labeled "Parallelism" in figure 7.6.

Let

$$\Xi := \langle S_1 || S_2 || \ldots || S_n, \sigma \rangle$$

Suppose we have

$$\Xi \rhd \kappa_1 \quad \text{and} \quad \Xi \rhd \kappa_2,$$

while $\kappa_1 \neq \kappa_2$. We want to show that there is a configuration κ such that $\kappa_1 \rhd \kappa$ and $\kappa_2 \rhd \kappa$. Suppose then that $\kappa_1 = \langle S, \sigma_1 \rangle$ and $\kappa_2 = \langle S', \sigma_2 \rangle$. By the Parallelism rule, it must be that

$$S = S_1 || \ldots || S_i' || \ldots || S_n,$$

and

$$S' = S_1 || \ldots || S_j' || \ldots || S_n,$$

for some $i \neq j$. Let

$$S'' = S_1 || \ldots || S_i' || \ldots || S_j' || \ldots || S_n.$$

Now all we need to complete our construction of the configuration $\kappa := \langle S'', \sigma'' \rangle$ is to define a valid state σ'' which is a merge of σ_1 and σ_2. Simply, define σ' to be

$$\sigma'' x := \begin{cases} \sigma_1 x, & \text{if } \sigma_1 x \neq \sigma x, \\ \sigma_2 x, & \text{if } \sigma_2 x \neq \sigma x, \end{cases}$$

for $x \in \mathcal{A} \cup \mathcal{K}$. Now we need to show that σ'' is a valid state, and that we can deduce $\kappa_1 \rhd \kappa$ and $\kappa_2 \rhd \kappa$.

To show that σ'' is a valid state, it is sufficient to show that transitions with different labels do not interfere; that is, we cannot have two transitions with different labels modify a common address or a common counter. This can be verified readily by examining the rules in table 7.2 which are the only rules modifying the state.

To show that we can deduce $\kappa_1 \rhd \kappa$ and $\kappa_2 \rhd \kappa$, we must show that we can deduce

$$\langle S_j, \sigma_1 \rangle \overset{j}{\rhd} \left\langle S_j', \sigma'' \right\rangle \tag{8.1}$$

and

$$\langle S_i, \sigma_2 \rangle \overset{i}{\rhd} \left\langle S_i', \sigma'' \right\rangle \tag{8.2}$$

respectively. By the Parallelism rule, to have been able to deduce $\Xi \rhd \kappa_1$ and $\Xi \rhd \kappa_2$, we must have been able to deduce

$$\langle S_i, \sigma \rangle \overset{i}{\rhd} \left\langle S_i', \sigma_1 \right\rangle \tag{8.3}$$

and

$$\langle S_j, \sigma \rangle \overset{j}{\rhd} \left\langle S_j', \sigma_2 \right\rangle \tag{8.4}$$

respectively. Take transition 8.1; I claim that the same rule, say r, which allows us to deduce 8.4 allows us to deduce 8.1. The argument of the previous paragraph applies to show that if r applied to deduce a transition from $\langle S_j, \sigma_1 \rangle$, then the target configuration of the transition would have σ'' as its state. Thus all we need to verify is that r applies. To see this, it is sufficient to see that the change of state from σ to σ_1 does not invalidate the use of r. The only rules which have side conditions which depend on the state are the Ordinary and Others rules so r must be one of these. Suppose r is the Ordinary rule and it applied because of the side condition on the state ($\sigma(Kja) = \infty$). We know that no transition with a different label than j can modify the state in such a way that this side condition no longer holds. This is so because a counter can be modified only by transitions with the same label as its (the counter's) first component. Thus r applies to deduce 8.1. Similarly, if r is the Others rule, then no transition with a different label than j can invalidate the $\sigma(Kja) \leq \sigma(K\lambda a)$ side condition. This is so because counters can only increase in value.

The same argument with j replaced by i applies to show that 8.2 can be deduced. Thus we have shown that we can deduce $\kappa_1 \rhd \kappa$ and $\kappa_2 \rhd \kappa$. ∎

8.2 Deadlock

A computation is deadlocked if it cannot make any progress while it is in a non-terminal state. Deadlock often occurs when processes wait for resources which will never become available or for conditions which will never hold. For example, in distributed-memory systems deadlock occurs when processes wait for messages which will never arrive. More precisely, we have the following definition.

Definition 3 *Given a transition system $[K, T, \rhd]$, if a transition sequence s reaches a configuration which is maximal but not in T, then s is said to be* deadlocked.

To formulate the hypothesis of the deadlock theorem 2, we need a notion of "loose synchrony" in $\mathbb{P}$tiny programs. Computations made up of loosely synchronous guards cannot lead to deadlock. It is possible to formulate narrower tests, but these are difficult to apply in practice. Also, the $\mathbb{P}$ model was designed for computations which tend to be loosely synchronous. Computations which are truly asynchronous and tightly coupled are somewhat awkward to express in $\mathbb{P}$. Thus it seems natural to draw the boundary at loosely synchronous.

Definition 4 *A computation of $\mathcal{T}_p$ beginning with the initial configuration $\langle P, \sigma_0 \rangle$ is* loosely synchronous *iff whenever the predicate for a conditional (*`if` *or* `while`*) within P is true (> 0) for one set, say T, of guards and false ($= 0$) for another set, say F, then*

$$obs(T) \cap obs(F) = \emptyset,$$

where $obs(X)$ is the set of guarded objects which a set of guards X access.

Theorem 2 *No computation K of the transition system $\mathcal{T}_p$ beginning with configuration κ_0 can be deadlocked if every configuration $\langle A, \sigma \rangle$ with A an atomic construct has a successor in $\underset{a}{\triangleright}$ and if K is loosely synchronous.*

Proof: Let $\kappa := \langle C, \sigma \rangle$ be a configuration such that $\kappa_0 \overset{*}{\triangleright} \kappa$ where κ_0 is an initial configuration and such that C is not the empty statement (;) or a value (so that κ is not terminal). I will show by induction on the structure of C that κ has a successor in $\triangleright$. For the base case of the induction we need to examine those C which have no subconstructs.

If C is atomic, then by the Atomic rules in tables 7.3 and 7.5 and the first hypothesis of the theorem, κ has a successor in $\triangleright$. If C is a variable, then the Variables axiom of table 7.2 applies to give a successor for κ.

The interesting base case occurs when C is an address, say a, then the Ordinary, Guard, and Others rules of table 7.2 apply. We must show that given any state, σ, derivable from the initial state and any label, λ, we can find a configuration κ' such that $\kappa \overset{\lambda}{\triangleright} \kappa'$. If $L(a)$ is λ or ϵ or if $\sigma(K\lambda a) = \infty$, then this follows from the Ordinary and Guard rules. If $L(a) \neq \lambda$ and $\sigma(K\lambda a) \neq \infty$, then we must show that $\sigma(K\lambda a) \leq \sigma(KL(a)a)$. Formally, this is done by induction on the length of the derivation from the initial configuration κ_0 to κ with the help of the hypothesis of loose synchrony. For the base case, if κ is κ_0, then all counters are zero and we are done. For the induction case, we need to show that if the inequality holds for all transition sequences of length n from κ_0 to κ, then it holds for all such sequences of length $n + 1$. The only way that an additional single transition can invalidate this inequality is for the following to hold. We must have $\sigma(K\lambda a) = \sigma(KL(a)a)$ and the transition must increase $\sigma(K\lambda a)$ without increasing $\sigma(KL(a)a)$. This is ruled out by the loose synchrony hypothesis implis that each transition which increases $\sigma(K\lambda a)$ must be accompanied by a matching transition which increases $\sigma(KL(a)a)$.

The outer induction case then follows simply from the fact that the only rules for complex expressions and statements with side conditions which can prevent them from being applied are the rules labeled RHS in table 7.2, and the side conditions on these rules are complementary (*i.e.* their disjunction is always true). ∎

Part III

Engineering IP

Chapter 9

The IPC++ Translator

The IPC++ translator translates a IPC++ program into a C++ program with calls to a set of routines which manage low-level communication and synchronization. The compilation strategy used in the IPC++ translator is as follows. Each guarded object is stored by the physical processor executing its guard. The compiler then partitions the computation in such a way that each processor only computes values for the objects it guards and other objects are sent to it when needed.

More precisely, for each IP array a, the compiler and the run-time system keep track of the indices of the elements, say $G(p,a)$, of a guarded by (guards simulated by)]each physical processor p. Through a process called localization, a parallel statement `foreach(`a`) S;` is translated to repetitive constructs which ensure that each processor p executes any expression E in S if and only if the guarded objects being modified in E have indices in $G(p,a)$. Thus the partitioning of the `foreach` argument a determines the partitioning of computation within the `foreach` statement. Another phase of the compiler generates code which communicates objects to processors which are not responsible for storing them.

There are two factors which complicate the implementation of this simple strategy. One factor is that `foreach` statements can be nested and complex control structures can appear in their bodies. Another factor is that straight-forward generation of communication statements produces inefficient code.

The overall structure of the IPC++ compiler is shown in figure 9.1.

The translation process begins with the construction of an abstract syntax tree for each declaration in a IPC++ program. This structure is then checked for undeclared and misused variables, The next step is to analyze the program to determine static information.

9.1 Program Representation

Experience gained from early work on the IPC++ compiler suggested that the best representation for its purposes is a combination of abstract syntax trees, local value graphs, control flow graphs, static single assignment information, control dependence information. This approach is not unusual; many other projects of this kind have similar internal representations. For completeness, and because there are some unusual details in it, the representation is discussed in detail in chapter 11.

9.2 Data Management

Data management is done through a runtime library of C++ classes and macros which provide a target "instruction set" for the compiler. These instruction, or *codes* as they are known in IPC++ jargon, correspond directly to the set of codes used in the theory developed in chapter 13. They are discussed in some detail in chapter 10.

9.3 Process Management

Mapping the IPC++ process structure onto a distributed-memory machine efficiently is a challenging problem. The fundamental difficulty is in simulating a flexible

1. parse,
2. check types and sizes,
3. for each function:
   (a) construct CFG,
   (b) compute control dependences,
   (c) build SSA-PDG,
4. for each function:
   (a) propagate nonuniformity information,
5. for each function:
   (a) find codes to be moved,
6. for each function:
   (a) move codes,
   (b) generate contexts (relative slices) for moved code,
   (c) if possible, evaluate the relative slices.
7. for each function:
   (a) expand declarations,
   (b) localize control,
   (c) place code.

Fig. 9.1. Sturcture of IPC++ Compiler

and dynamically changing set of processes on a fixed number of disjoint physical processors. The most natural way to do this is to simulate the processes assigned to a single processor as interleaved threads. This approach does, however, have a number of serious disadvantages.

- A standard thread package has not been available on most parallel processors.
- Most thread packages are still too slow to accomodate scientific computation.
- There is no standard and efficient message-passing package which interfaces well with available thread packages.

For these reasons, I decided to follow an alternate route toward process simulation. In the approach I have chosen, the burden of simulation is almost entierly on the compiler. The compiler generates simulation loops which repeat statements of the simulated program once for each process being simulated. This approach is simple enough for straight-line programs, but it becomes more complex in the presence of complex control structure and nested parallelism. In fact, the technique presented here works only for structured control flow.

The method used by the compiler to simulate many processes on a single processor is the same as that used by Quinn, Hatcher, and others [35] to compile C⋆. The parallel `foreach` loops are distributed over control structures which contain expressions needing values computed by other processes. The only difference in our case is that we need to allow nested parallel loops. This is done by maintaining the current index set as a stack of index sets at run time. At the beginning of each `foreach` statement a pointer to the index set of the `foreach` argument is pushed onto the stack, and the stack is popped at the end of each `foreach` statement. In addition, the runtime system computes bounds for the simulation loops as index sets are pushed and popped. The pushing and popping operations are called "Fork" and "Join" in the section 10.4 on the runtime system's instruction set.

9.4 Code Motion

The main optimization technique used by the IPC++ compiler is a general code motion technique which moves and combines expensive operations. The idea for this technique is to

- identify expensive operations,
- apply a heuristic to compute new, less expensive, positions, in the program for these operations,
- generate program slices to compute expressions used by the moved operations at their new location,
- store the result of the operation in a buffer, and
- use the contents of the buffer in place place of the moved operation.

The formulation of the code motion problem and a greedy heuristic for finding an approximate solution is explained in detail in chapter 13. The only other part of the above procedure worthy of note is the generation of program slices.

The notion of program slicing was introduced by Mark Weiser for the purpose of program understanding and debugging [66]. A slice of a program P with respect to an expression e at a point p is any program which computes the same values for e at p as P does. Note that the point p may not be the point where e occurs in the program P; this is the sense in which slices discussed here are relative. Note also that if p *is* the point at which e occurs, then the original program P fits the definition of a slice. One is, of course, interested in slices which have some desirable property. For example, we might want a slice which is minimal in some sense. For the purpose of compiling ℙC++, we are interested in (relative) slices which have minimal running time.

It turns out that computing an approximation of slices with minimal run time is not difficult if we have the correct program representation. In fact, with the SSA-PDG (see section 11.3), the problem of computing near-minimal slices is just the problem of computing reachable subgraphs. Such slices are only nearly minimal because the SSA-PDG encodes a static estimate of the control and data dependences in the program. An algorithm which computes such slices is given in figure 9.2. The algoriths is similar to the worklist algorithm which computes nonuniformity information (see chapter 12). The algorithm is shown in figure 9.2. Note that the heuristic it uses to approximate minimal running time is to include in the slice only those expressions (and hence statements) which SSA-PDG says are dependent on the source node (representing the origin of the moved operation).

This algorithm works well within a single procedure; to take into account interprocedural information, we would have to resort to more complex algorithms which match procedure parameter with actual arguments (see for example [41]).

Algoritm: Slice

Input: SSA-PDG G of a program and nodes s and d of G representing an expression e and a target point p

Output: SSA-PDG S of slice with respect to e at p

```
for i from 1 to |𝒩(G)|   A[i] = False
X ← {s}
W ← {s}
while W ≠ ∅
   take and delete v from W
   for every edge ⟨v, w⟩ in G
      if not A[w] and w ≠ p then
         A[w] = True
         W ← W ∪ {w}
         X ← X ∪ {w}
S ← subgraph of G induced by X
```

Fig. 9.2. Computing a Slice with Respect to an Expression and a Target Point

Chapter 10

The Target Model for IPC++ Compilation

The target of the IPC++ translation process is a program in the *distributed-memory* SPMD model of computation. In the SPMD model, copies of a single program run on many similar processors. They communicate with each other either by sharing memory and using synchronization primitives or by sending each other messages. In the former case, the model is called *shared-memory* SPMD, and in the latter case *distributed-memory* SPMD. The SPMD model is especially appropriate as the target of a compilation process because it is close to parallel processing hardware (see section 1.2 of the introduction).

10.1 Design Goals

The design goals for the library which implements the IPC++ run-time model are *completeness*, *simplicity*, and *portability*. By completeness I mean that the set of instructions should be designed in such a way that the compiler can generate efficient code for a variety of computations. As is often the case, these goals make conflicting demands on the designer. The challenge is to find a reasonable compromise.

10.2 The Message-Passing Layer

At the lowest level, the SPMD model which is the target of the pC++ compilation process is particularly simple. A program P in the target model is a C++ program with the following predefined functions.

- `long numnodes()` which returns the number of physical processors executing copies of P.
- `long mynode()` which returns a unique identifier in the range $0 \ldots$`numnodes()`-1.
- `void send(long tag, long to, void* buf, long size)` which sends the contents of `buf` to processor `to` with integet tag `tag`.
- `void recv(long tag, long from, void* buf, long size)` which waits until a message with tag `tag` has been received from processor `from` and stores the contents of the message in `buf`.

Copies of P execute asynchronously on each process. They are synchronized and information is exchanged among them using the `send` and `recv` pair of functions. This low-level model, with slight variations, is familiar to programmers who have programmed existing distributed-memory parallel processors.

Above this low-level model is the pC++ run-time library which is made up of C++ classed and macros. The purpose of this library is to separate, as much as possible, the details of message-passing from the compiler. This approach both simplifies the compiler and increases portability.

10.3 Segmented Index Sets

The class central to the runtime system is the Segmented Index Set (SIS). A SIS represents a rectangular subset of $\mathbb{Z}^n$ subdivided into segments. The segments are represented by triples (similar to those in Fortran 90) which contain a lower bound, an upper bound, and a step size. Each segment represents the elements of the index set stored on a processor. The use of triples does mean that the runtime system cannot represent unstructured distributions of index sets among processors. There are provisions for assigning a "manager" to each index so that dynamic assignments of indices and segments to processors can be handled. The current compiler does not make use of this possibility however.

Each $\mathbb{P}$ array is represented by a class with a SIS. The instructions of the runtime system take as arguments pointers to $\mathbb{P}$ arrays and guarded objects (as well as other auxiliary values, *e.g.* message tags). Thus the segmentation information is entirely encapsulated the single class representing a segmented index set.

10.4 The Instruction Set

The runtime system provides a virtual instruction set for the compiler. This instruction set is based on table 10.1. Each instruction (or code) has one field corresponding to each column of the table. The instructions are in fact C++ macros, and their fields are just portions of their names. For example, there is a macro called

```
read_none_basic_all_unknown_yes
```

An explanation of each field follows.

Operation	Function	Type	Part	Pattern	Barrier
Fork	None	None	None	None	Yes
Join	Arithmetic	Basic	All	Unknown	No
Read	Logical	Class	Send	PToP	
Write	User		Recv	Broadcast	
Combine			Find	TShift	
Global				CShift	

Table 10.1. Table of Target Codes

Operation: This field is the major operation the code represents. Fork and Join represent pushing and popping the current index set. Read and Write represent reading and writing elements of IP arrays. Combine represents combining operations (like global sum). Global represents the calculation of global indices from local simulation loop indices.

Function: This field represents the function to be applied to elements to which code applies. For example, for a global sum operation this field would be Add.

Type: This field distinguishes between operations applied to objects of basic type and those which have user-defined type. This distinction is crucial because objects of user-defined type must be packed and unpacked when they are communicated.

Part: This field represents the part of the operation which the instruction represents. For example, the instruction could just represent the send part of Read operation.

Pattern: The compiler can generate communication instruction for which the pattern of communication is known. This filed represents that information. If the pattern is unknown, then it is supplied as an operand for the instruction.

Barrier: This field is to indicate if there should be an explicit synchronization point among the participants after the instruction is carried out.

Chapter 11

Program Representation

11.1 Control Flow Graph

A control flow graph (cfg) is a directed graph whose nodes are basic blocks and whose edges represent control flow between those blocks. Basic blocks themselves are represented by local value graphs described in the next section. Figure 11.1 contains a program fragment which computes the LU decomposition [32] of the matrix `a`.

Description 1 *A* cfg *is a directed acyclic graph with the following properties. Two of its nodes are distinguished and labeled by* **start** *and* **stop***. The* **start** *node has indegree 0 and outdegree 2. The* **stop** *node has outdegree 0 and indegree* ≥ 1*. There is a path from* **start** *to every node, and there is a path from every node to* **stop***.*

Description 2 *The set of* points *of a cfg G is defined to be*

$$\boldsymbol{P}(G) := \mathcal{A}(G) \times \{b, a\}.$$

For each $N \in \mathcal{A}(G)$, N_b and N_a are defined to stand for $[N, b]$ and $[N, a]$ respectively.

11.2 Local Value Graphs

Basic blocks are loop-free segments of a program. They are represented by directed graphs whose nodes represent operations, variables or constants. Edges in the

```
k=0;                                   (1)
while ( k < N ) {                      (2)
  ainv =  1.0/a[k][k];                 (3)
  i = k + 1;                           (3)
  while ( i < N ) {                    (4)
    a[i][k] *= ainv;                   (5)
    i++;                               (5)
  }
  i = k + 1;                           (6)
  while ( i < N ) {                    (7)
    j = k + 1;                         (8)
    while ( j < N ) {                  (9)
       a[i][j] -= a[i][k]*a[k][j];     (10)
       j++;                            (10)
    }
    i++;                               (11)
  }
k++;                                   (12)
}
```

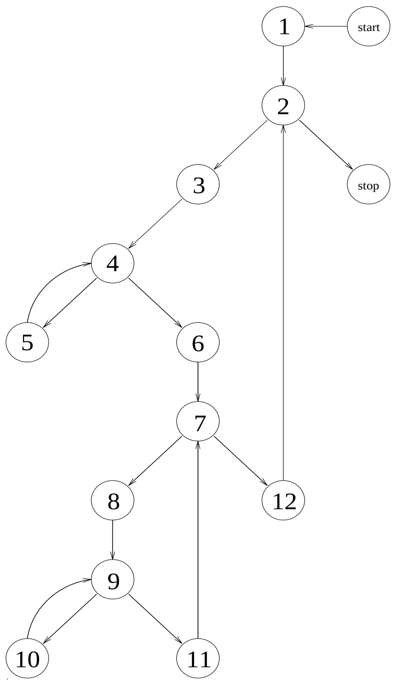

Fig. 11.1. Gaussian Elimination for a Dense Matrix

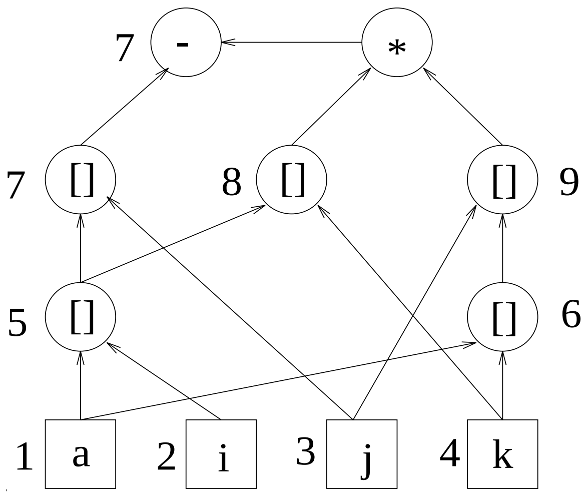

Fig. 11.2. Local Value Graph for Basic Block (10) of Gaussian Elimination

graph represent the flow of values among operations and variables. This representation of basic blocks decouples subsequent compiler analyses from the source language; it also identifies locally common subexpressions. Most importantly, local value graphs have a simpler and more uniform structure than the abstract syntax trees representing basic block. This simplicity streamlines the next phase of the compilation process. A precise description of local value graphs is given in description 3 below. Figure 11.2 shows the local value graph corresponding to basic block 10 of the Gaussian Elimination program shown in figure 11.1.

Description 3 *A* local value graph (lvg) *is a directed graph each of whose nodes is*

- *a function symbol,*
- *a variable identifier, or*
- *a constant*

There are two kinds of edges between the nodes of an lvg. An edge of the first kind connects a node which produces a value to a node which uses that value; it is called a value *edges. An edge of the second kind connects a node which produces a value to a node which stores that value; it is called a* store *edge.*

For a given node N, $\boldsymbol{\delta}(N)$ represents the list of memory locations in which the results of applying $\boldsymbol{\phi}(N)$ to $\boldsymbol{\iota}(N)$ are to be stored, and the numbering of $\boldsymbol{\iota}(N)$ determines the match between the arguments and parameters of $\boldsymbol{\phi}(N)$.

11.3 SSA-PDG

Some of the analyses and transformations to be performed on a IPC++ program require more information about the program than is readily available in a cfg. the SSA-PDG is used in these cases. This representation is a combination of the graphical representations proposed in [27] and [21, 22]. PDG stands for Program Dependence Graph and SSA stands for Static Single Assignment. The algorithms used to construct the SSA-PDG are derived from the algorithms given in [21, 22].

Description 4 *In the SSA-PDG there is a node for each expression in the program. There are also additional* join *nodes which serve as summary nodes for reaching definitions as in SSA representation (see [21, 22]). There is an edge in the SSA-PDG from each definition to each use (include join) node. There is also an edge from each use to the node in which the use occured (these are data flow, or operand-operator nodes). Finally there is a node from each controlling expression to the nodes which are control dependent (see [27]) on it.*

11.4 Interval Graphs

In addition to the SSA-PDG, interval graphs are used to identify loops and to estimate cost functions for code motion (see chapter 13). Interval graphs are constructed by an algorithm based on the one given by Tarjan in [60]. The algorithm given by Tarjan identifies intervals by construction. The IPC++ compiler uses the same algorithm, except it records the intervals as they are recorded. The interval graph is simply the tree of nested intervals constructed by this algorithm. The interval depth of a program construct (an expression or statement) is its level in this tree of nested interval. Thus the interval depth of a construct is a measure of how many level of loops surround the construct.

Chapter 12

Uniform Expressions

Uniform expressions are expressions which carry no more information than scalars—*i.e.* all of their elements have the same value. Knowing which expressions are uniform allows a compiler to avoid unnecessary communication because elements of uniform $\mathbb{P}$ arrays need not be communicated. Such knowledge also allows a compiler to spot simpler patterns of memory reference and avoid expensive, general communication and synchronization protocols.

Definition 1 *An expression e is* uniform *at a point p of a cfg if e reaches p and e's value is a $\mathbb{P}$ array all of whose elements have the same value at p.*

I will present two ways of computing uniformity information. One is a dataflow framework which uses the cfg and the postdominance relation between nodes of the cfg. The other is a worklist-based reachability algorithm which uses the SSA-PDG. Both techniques have been implemented and give correct results for all of the $\mathbb{P}$C++ programs presented in this dissertation.

12.1 A Data Flow Framework

The framework which computes uniformity information is based on a set of mutually recursive equations.

Description 1 *If functions* $N\colon \boldsymbol{P}(g) \rightarrow \mathcal{A}(g)$ *and* $C\colon \boldsymbol{P}(g) \rightarrow \mathcal{N}(g)$ *satisfy the following recurrence equations, then for every point* p *of* g*, every expression which may be nonuniform at* p *is in* $N(p)$*.*

$$
\begin{aligned}
C(n_b) &= \bigcup_{p \in \mathbf{P}(n)} C(p_a) \\
C(n_a) &= (C(n) \cup C(n_b)) \setminus \mathbf{f}'(n), \\
N(n_b) &= \bigcup_{p \in \mathbf{P}(n)} N(p_a), \\
N(n_a) &= N(n) \cup N(n_b),
\end{aligned}
$$

where

$$
\begin{aligned}
C(n) &:= \begin{cases} \{n\}, & \textit{if } |\,\mathbf{c}(n)| > 1 \textit{ and } \mathbf{u}(n) \cap N(n_b) \neq \emptyset, \\ \emptyset, & \textit{otherwise}, \end{cases} \\
N(n) &:= \begin{cases} \mathbf{d}(n), & \textit{if } C(n_b) \neq \emptyset. \\ \emptyset, & \textit{otherwise.} \end{cases}
\end{aligned}
$$

12.2 A Worklist Algorithm

The worklist algorithm given in figure 12.1 is taken directly from the current implementation. The algorithms simply maintains a worklist of PDG form SSA nodes which have beed found to be reachable from `thisnode`. The algorithm is simple enough to be given here as a C++ program fragment taken directly from the compiler. This is meant in part to illustrate some of the internal aspects of the IPC++ compiler. Within the compiler, a node of the SSA-PDG is encapsulated in the class `SsaNode`. TThe class `SsaNodes` represents a collection of `SsaNodes`.

```
// first mark nodes which are nonuniform by fiat
// ("thisnode" and results of function calls in the absence of IP analysis)
SsaNodes work(n);
Ints marked(n, 0);
for (int i=0; i<n; ++i) {
  SsaNode* s = (*nodes)[i];
  int sn = s->number();
  if (s->is_nonuniform() && !marked[sn]) {
    work.add(s);
    marked[sn] = 1;
  }
}

// propagate marks to outputs of marked nodes:
int wfirst = 0;
while (wfirst != work.size()) {
  SsaNode* s = work[wfirst++];
  for (int j=0; j<s->outputs().size(); ++j) {
    SsaNode* c = s->outputs()[j];
    int cn = c->number();
    if (!c->is_const() && !marked[cn]) {
      work.add(c);
      marked[cn] = 1;
    }
  }
}
```

Fig. 12.1. Worklist algorithm for finding nonuniform IP arrays

Chapter 13

Code Motion

Consider the problem of generating communication primitives for nonlocal accesses in a IPC++ program. The simplest way to carry out such communication is to insert appropriate calls to communication primitives at every non-local reference. This naive approach leads to unacceptable performance in many important cases. We can do better by placing communication and synchronization primitives for a non-local reference in places other than its original site. The new site can be preferable because it is less frequently executed or because the operations performed there can be combined to lower overhead, or because its overhead can be masked with simultaneous computation.

This chapter presents a theory of code motion which can be applied to the operations found in IP programs. These operations include communication of offprocessor elements of IP arrays on distributed memory machines, synchronizations necessary for implementing IP semantics on shared memory hardware, deadlock detection, and others.

Let g be a fixed cfg. All of the theory in this chapter is developed with respect to g, so that g does not appear directly as a parameter in the definitions, theorems and algorithms.

13.1 The Problem of Optimal Code Motion

A code motion problem is formulated in terms of a set $\mathcal{C}$ of *codes*. The details of this set are very much dependent on the problem to which we intend to apply the theory developed here. Hence these details are exposed elsewhere (see section 10.4). The formulation of a code motion problem will require various functions to and from $\mathcal{C}$. One set of required functions is special and deserves a name. The set of code placements $\Psi(\mathcal{C}, g)$ is defined to be the set of functions $\boldsymbol{P}(g) \to \mathcal{P}(\mathcal{C})$. $\mathcal{C}$ and g will be omitted when they are obvious from the context.

Definition 1 *An optimal code motion problem is: given a set of codes $\mathcal{C}$ and functions $\kappa \in \Psi$, $\$\colon \Psi \to \mathbb{R}$, $\omega\colon \mathcal{C} \to \boldsymbol{P}(g)$, find $\psi \in \Psi$ which satisfies*

1. $\$(\psi) = \min_{\xi \in \Psi} \$(\xi)$, *and, for each point p,*
2. $\psi(p) \subseteq \kappa(p)$,
3. $C_p(\bigcup_{q \in \boldsymbol{P}(Q)} \psi(q)) = C_p(\mathcal{C})$ *for any path Q which goes through p, and*

where $C_p(X) := \{c \in X \mid \omega(c) = p\}$ for $X \subseteq \mathcal{C}$.

The function κ is a semantic constraint function which maps each point of g to the set of codes which can legally be placed there. Thus constraint 3 says that all code placement requirements at p must be met on any path P through p. Constraint 2 says that code must be placed at semantically correct points. And constraint 1 says that any such placement must be optimal with respect to the cost function \$.

13.2 Cost Functions

Both the the speed with which a code placement problem can be solved and the accuracy of its solution depend on the nature of the given cost function. The simplest cost functions are those which are determined by purely local information.

Definition 2 *A cost function* $\$$ *is* local *iff there is a set of functions* $\{\$_p\colon \mathcal{P}(\mathcal{C}) \to \mathbb{R} \mid p \in \boldsymbol{P}(g)\}$ *such that for any placement* $\psi \in \Psi$,

$$\$(\psi) = \max_{p \in \boldsymbol{P}(g)} \$_p(\psi(p)).$$

A cost function $\$$ *is* pointwise *iff there is a set of functions* $\{\$_p\colon \mathcal{C} \to \mathbb{R} \mid p \in \boldsymbol{P}(g)\}$ *such that for any placement* $\psi \in \Psi$,

$$\$(\psi) = \max_{p \in \boldsymbol{P}(g)} \max_{c \in \psi(p)} \$_p(c).$$

An additional simplifying constraint on cost functions is monotonicity. Monotone cost function are analogous to linear objective functions in mathematical programming. This means, essentially, that their direction of change is constant. First, we need to extend the path orderings $\prec$ and $\succ$ of a cfg g to placements.

Definition 3 *The order relations* $\prec$ *and* $\succ$ *on* $\Psi(g, \mathcal{C})$ *are defined as follows. Let* ψ *and* ϕ *be in* $\Psi(g, \mathcal{C})$. *Define* $\psi \prec \phi$ *iff for every* $p \prec\!\succ q \in \boldsymbol{P}(g)$, *if* $\psi(p) \cap \phi(q) \neq \emptyset$ *then* $p \prec q$. *Define* $\psi \succ \phi$ *iff for every* $p \prec\!\succ q \in \boldsymbol{P}(g)$, *if* $\psi(p) \cap \phi(q) \neq \emptyset$ *then* $p \succ q$.

Definition 4 *A cost function* $\$$ *is* monotonically increasing *iff* $\psi \prec \phi \Longrightarrow \$(\psi) < \$(\phi)$ *and* monotonically decreasing *iff* $\psi \succ q \Longrightarrow \$(\psi) < \$(\phi)$. *A cost function is called* monotone *if it is monotonically increasing or monotonically decreasing.*

The cost function used in the current implementation is the pointwise function:

$$\$_p(c) := depth(p) + distance(p, \varpi(c)),$$

where $depth(p)$ is the interval depth (see 11.4) of the program point p and $distance(p, q)$ is the length of the shortest path in the control flow graph between points p and q.

13.3 A Greedy Solution

The simplest way to solve a code placement problem with a monotone, pointwise cost function is to use a greedy approach: repeatedly move code from points with higher cost to (semantically correct) points with lower cost. The following description makes precise the idea of a greedy algorithm. It says that we can reduce the global cost and path constraints of definition 1 to local constraints at each point of g.

Description 1 *A placement $\psi \in \Psi(g, \mathcal{C})$ solves a code placement problem $[\mathcal{C}, \kappa, \$]$ iff*

$$\psi(p) \subseteq \kappa(p)$$

and

$$\psi(p) = (\psi(p) \cup \bigcup_{n \in N(p)} b(p,n) \cap \psi(n)) \setminus \bigcup_{n \in N(p)} b(n,p) \cap \psi(p) \qquad (13.1)$$

where $b : \mathcal{N}(g) \times \mathcal{N}(g) \to \mathcal{C}$ is defined by

$$b(p,q) := \{c \in \kappa(p) \mid \$_p(c) < \$_q(c)\}.$$

for all $p, q \in \boldsymbol{P}(g)$.

Equation 13.1 can be solved by simple iterative algorithms which update a provisional solution according to equation 13.1 for each point of g in some order. One such greedy algorithm is shown in figure 13.1.

13.4 Semantic Constraints

To pose a code placement problem $[\mathcal{C}, \kappa, \$]$, one needs to form the semantic constraint function κ. This section recalls (see [50]) two fundamental semantic constraint functions: δ and υ. These functions encode how far down and how far up in the cfg the code required at each node of the cfg can be moved.

The function υ characterizes points where the values necessary for correctly computing the calls required at a point p can be anticipated and which are on paths leading

Algoritm: Greedy
Input: A code placement problem $[\mathcal{C}, \kappa, \$]$
Output: $\psi \in \Psi(g, \mathcal{C})$

for every edge $[p, q] \in \mathcal{E}(g)$
  for every $c \in \mathcal{C}$
    if $\$_p(c) < \$_q(c)$
      add c to $b(p, q)$
    if $\$_p(c) > \$_q(c)$
      add c to $b(q, p)$
$\psi_o \leftarrow \rho$
$\psi_n \leftarrow \emptyset$
while $\psi_n \neq \psi_o$
  for every point $p \in \boldsymbol{P}(g)$
    $A \leftarrow \bigcup_{n \in N(p)} b(p, n) \cap \kappa(p) \cap \psi_o(n)$
    $D \leftarrow \bigcup_{n \in N(p)} b(n, p) \cap \kappa(n) \cap \psi_o(p)$
    $\psi_n(p) \leftarrow (\psi_o(p) \cup A) \setminus D$

Fig. 13.1. Greedy Algorithm for Solving a Code Placement Problem

into p. The recurrence equations which make possible the computation of its inverse υ are

$$\begin{aligned} \upsilon(B_a) &= \bigcap_{C \in \mathbf{c}(B)} \upsilon(C_b) \\ \upsilon(B_b) &= \mathbf{u}_f(B) \cup (\upsilon(B_a) \setminus \mathbf{d}(B)) \end{aligned} \tag{13.2}$$

Similarly, the function δ characterizes points where the values necessary for making the calls required at a point p are still available and which are on paths leading out of p:

$$\begin{aligned} \delta(B_b) &= \bigcap_{P \in \mathbf{p}(B)} \delta(P_a) \\ \delta(B_a) &= \mathbf{d}_l(B) \cup (\delta(B_b) \setminus \mathbf{a}(B)) \end{aligned} \tag{13.3}$$

Equations 13.2 and 13.3 can be solved by simple iterative techniques [50].

Appendix A

IPFortran

This chapter presents a core IP extension of Fortran. This language has been in use for a number of years. The experience gained from its use and implementation has lead directly to the design of general IP extensions and IPC++.

A.1 IPfortran Constants

IPfortran defines the following constants for writing the node program. These will be used throughout the text.

- `thisnode` is the logical processor number of the processor executing the program. (Also referred to as TN).
- `numnode` is the total number of processors.

A.2 The Operator @

Accessing variables at other nodes can be done in IPfortran using a new binary operator, @, which indicates that a variable is *at* some node. For example,

$$x = y@0$$

assigns the value of variable y of node 0 to x at every node (*broadcast*). Similarly

$$x = y@(n + 1)$$

broadcasts the value of variable y of node $n + 1$ to x (at every node). Mixed expressions such as

$$x = y@p + z$$

assigns to x (at every node) the sum of z at that same node and y from node p.

Multiple uses of @ in a statement are permitted, but only in assignments (i.e. expressions involving =) or in reduction operations (Section 3). (@, in principle, makes sense in any expression, e.g. predicates, but has not been implemented there.)

The operator precedence in IPfortran, from highest to lowest precedence, is

$$@ \qquad ** \qquad (*, /) \qquad (+, -).$$

So, for example, $a = b@p + 1$ is not equivalent to $a = b@(p + 1)$ and $a@p + 1 = b@q$ is meaningless where $a@(p + 1) = b@q$ is not. @ is non-associative, therefore, $x = y@i@p$ is illegal and $x = y@(i@p)$ or $x = (y@i)@p$ are legal.

We can now give a simple program segment to compute a sum of values at each node, say in the variable y. First, all nodes broadcast their own value and then each sums the values independently.

```
      DO 10 n=0, numnode-1
10       x(n) = y @ n
      sum=0.0
      DO 20 n=0, numnode-1
20       sum = sum + x(n)
```

This technique, while not efficient, serves to illustrate an important point concerning our assumption about communication in IPfortran. In the first loop in the code above, the node from which we receive data is a variable. However, at each step in the iteration the value of this variable is the same at each node. Thus n denotes a value universally agreed upon at each iteration, a value *global* to all nodes, P. If the value of n were different in different nodes, then a *local* value results and a more complicated implementation would be needed.

If values refering to node numbers are *global*, we can predict the communication pattern required for the data exchange. Otherwise, for node numbers given by *local* values, the communication pattern must be determined at run time. This is not difficult to do, but it requires a significant amount of overhead.

In a future version of IPfortran we plan to do compile-time analysis to determine which variables referring to node numbers are *local* or *global*. At the moment, the programmer should limit the use of variable node numbers to *global* values. Our experience has shown that this is not a significant hindrance and results in a distinctive programming style.

Ranges of nodes

There are many instances in which sets of nodes or arrays of values are useful objects. We have implemented one class of such sets, namely *ranges*. The construct

$$<expr> \ @ \ (<expr>_1 \ : \ <expr>_2 \ )$$

refers to the value of <*expr*> in the range of nodes from $<expr>_1$ to $<expr>_2$. It can appear on the left-hand side of an assignment statement. For example,

$$x@(1:5) = y@0$$

means: "Assign the value of variable y at node 0 to variable x at nodes 1 through 5." On the right-hand side, the range logically returns multiple values. For expression

$$x@(i:j) = y@(r:s),$$

the range $(r:s)$ will map 1:1 onto the range $(i:j)$ on the left-hand side; if $j - i \neq s - r$, then the statement is not defined with unpredictable results. In addition to a single range or scalar node identifier, lists of scalars and ranges are allowed, but also with a 1:1 correspondence if more than one node is specified on the right-hand side. Some examples follow:

$$a@(i,j,k) = b@(m,n,o)$$

$$a@(1,2,3,4) = b@0$$

$$a@(k+m:n) = b@0$$

Functions of `numnode`

It is tempting to write $x@\texttt{numnode} = y@(\texttt{numnode} - 1)$, however, this is incorrect in the current version of IPfortran and will result in deadlock. It is not unreasonable to expect the translator to detect the presence of `numnode`and generate correct code. The remainder of this section talks about the first case, detecting the direct presence of numnodein a communication statement. This has been implemented in IPfortran's cousin, PC [10].

Using standard compiler algorithms, analysis can detect the dependence of a variable's value on numnode. The expression

$$x = y@mod((\mathtt{numnode}+1), \mathtt{numnode})$$

means: assign the value of variable y of node *(*numnode*+1)* (modulo numnode) to x in numnode. We can have any expression which mentions numnode explicitly (including functions which return integers, arithmetic operators, etc.).

An example using functions of numnodecan easily be constructed to give a slightly more efficient summation algorithm. Think of our nodes as making up a simple ring. Then we can sum by accumulating and passing partial sums around this ring:

```
     sum=y
     DO 10 n=1, numnode - 1
       y = sum@mod((numnode+1), numnode)
  10   sum = sum + y
```

This has the effect of computing the nth partial sum

$$(\ldots(y@me + y@(me+1)) + \cdots + y@(me+n+1))$$

at the nth iteration, where node indices in the latter representation must be identified modulo numnode.

A.3 Reductions

To allow reduction operations, $\mathbb{P}$fortran introduces the following constructs:

$$<id> \; = <id> \; \{ \; <expr> \; [\text{,opt args}]\}$$

$$\text{call} <id> \; \{ \; <expr> \; , <id> \; [\text{,opt args}]\}$$

where $<expr>$ can reduce to a variable or constant. Functions and subroutines used along with {} are called *reducing functions* and *reducing subroutines*, respectively. Some examples of reduction operations that one may reducing include: SUM, MIN, MAX, OR, and AND. $\mathbb{P}$fortran provides the reduction operation SUM described in Section A.3. Other binary operators can be cast as reduction operations as described below.

The distributed value is represented by the symbolic name closest to the left curly brace }. The *set notation* emphasizes the notion of a collection of data objects, distributed in some manner across a collection of nodes. *function_name* (respectively, *subroutine_name*) must be a function (respectively, subroutine) of at least two (respectively, three) parameters. Functions must return a value of the same type as $<expr>$. The target variable is given by the left-hand side when using the function syntax, or as the second parameter, with procedure syntax (following the familiar mnemonic of *in*,*out* for parameter order). The designated operator, encapsulated in a function or subroutine, is then applied to the set of distributed values. As with other $\mathbb{P}$fortran statements, Fortran90 array syntax may be used.

We symbolically write the reduction operation for a p-process model

$$A = \oplus\{B\}$$

or

$$\text{call} \quad \oplus\{B, A\}$$

as
$$A = B_1 \quad \oplus \quad B_2 \quad \oplus \quad \cdots \quad \oplus \quad B_p.$$

Since the order of application of $\oplus$ to the set of values is not specified, $\oplus$ must be associative and commutative. (More precisely, we are assuming that the set of values of the type of B forms an Abelian semi-group with respect to the binary operator $\oplus$.)

These concepts are illustrated with a concrete example. Consider a user defined reduction operation implementing a summation of a set of scalars distributed across p nodes into a single scalar. Expressed mathematically,

$$\forall j, \quad result@j = \sum_{i=1}^{p} scalar@i.$$

By defining a function, `mysum`,

```
REAL*8 FUNCTION mysum(in1,in2)

mysum = in1 + in2

RETURN

END
```

now we can write

```
result = mysum{scalar}.
```

Reduction operation procedural interface and parameter typing

A reduction operation can be associated with either intrinsic functions or user defined functions; we consider examples of each. Using the intrinsic Fortran functions, AMIN1, the statement y = AMIN1 { x }

computes the minimum of all values of `x` in all nodes and assigns this value to the variable `y` (at all nodes).

IPfortran generates the correct procedural interface, AMIN1(a,b), for $\lfloor log_2 k \rfloor + 1$ invocations, where $k = \mid x \mid$.

The general structure of the reduction procedure invocations are:

function out = f1in

subroutine call s1in,out

where *in* is the distributed operand. *f1* and *s1* implement some binary operation with the results assigned to the distributed variable *out*. The corresponding function and subroutine definitions are of the form:

function TYPE FUNCTION f1(in1,in2)

subroutine SUBROUTINE s1(in1,in2,out).

With the function, the returned value if passed through the function return mechanism whereas in using the subroutine, the optional parameter *out* is provided. In both cases, the parameters *in1* and *in2* are not used to return results.

The user-defined reduction operations may include optional parameters. These parameters are local parameters to a node, i.e. they are not communicated, but are included in the calling sequence as shown above. These parameters can carry type information for the parameters `in` and `out`, or can be used in generating side-effects with the reduction operation.

Optional parameters are often useful. For example in FORTRAN77, without user-defined types, these parameters can carry type information. The contrived statement:

CALL opuratur { in,out,local1,local2 }

illustrates the use of optional parameters. The corresponding subroutine definition is:

SUBROUTINE opuratur(in1,in2,out,local1,local2)

where *local1* and *local2* are optional parameters and *out* is the output parameter. We notice that the parameters supplied to the function or procedure implementing the binary operator may be used non-destructively, that is, read-only.

The reduction of a function, e.g. `out=opratr{in,local1,local2}`, requires a function with the definition

REAL FUNCTION opratr{in1,in2,local1,local2}.

Parameter conformance

The actual parameters *in* and *out* must conform in shape, and if *out* is an array, `in` must also be an array. The formal parameters *in1* and *in2* correspond to some pair of operands for an operation $B_i \oplus B_j$, using with the notation of Section A.3.

The memory for at least one of the two input parameters, perhaps both, comes from a IPfortran scratch array; if possible, the output parameter `out` is used as a working memory through the duration of the reduction operation. This implementation detail is of practical concern only in assuring that the IPfortran temporary array is of sufficient size to accommodate intermediate results (see 8. User's Guide).

The array-section actual parameter will be reshaped to *rank 1* arrays, where the reshaping will construct the *rank 1* array in array order from the array subsection `in`, provided `in` is not already *rank 1*. The array section syntax denotes the description of a subarray with a shape indicated by the subscripts.

The nondestructive subroutine interface

A slightly more efficient reduction operation with a subroutine interface can be defined for arrays by using IPfortran function syntax for the global operation. IPfortran does not support anything but scalar-typed functions for global operations using the syntax $a = \oplus\{b\}$ since FORTRAN77 only supports this type of function. However, it is more convenient to write

$$A = \oplus A$$

than

$$\text{call } oplus\{a_{in}, A_{out}\}$$

Therefore, using the non-destructive procedural interface, the user is forced to supply the additional array A_{in} even though the destructive use (i.e. read–write), would have been acceptable.

To allow a destructive subroutine interface, with the spinoff of a streamlined syntax, IPfortran supports the statement such as `In = Globul{In}` where the user has defined the subroutine *Globul*:

SUBROUTINE Globul(In,InOut)

Dimension In(1),InOut(1)

InOut = In $\oplus$ InOut

RETURN

END

Length one arrays

The IPfortran translator generates a Fortran function interface and a subroutine interface for scalar or array parameters, respectively. Some care must be taken with regard to the interpretation of arrays of length one. In particular, note that `in(1)` denotes a scalar, whereas `in(1:1)` denotes an array (of length one). Similarly, if there is a statement such as `Dimension In(1)` then the parameter `in` (no parentheses) denotes an array. Thus it would not be correct usage to write `mysumIn` in this case.

Intrinsic reduction operation

The accumulation of the values of the elements of an array into a target array of the same shape is a common operation worthy of an intrinsic function. We wish to write code for the following sum:

$$\forall i, j, \quad A(i)@j = \sum_{k=1}^{p} A(i)@k.$$

Using IPfortran we can write

```
do i = 1, P
  B = A@i
  do j = 1, n
    A(j) = B(j) + A(j)
```

```
    enddo
  enddo
```

Alternatively, we write the more concise expression using the IPfortran intrinsic reduction operator, $+\{\}$:

$$A = +\{A\}$$

Note that the result of the user-defined reduction

$$\text{result} = \text{mysum}\{\text{scalar}\}$$

described earlier can be achieved using the intrinsic `result = +{scalar}` just as easily.

Reduction operation applications

The reduction of a set of distributed data by a binary operator is a common occurrence in scientific and engineering computations. Having a concise and intuitive notation streamlines programs, resulting in enhanced readability. Furthermore, by flagging the operation in a single statement, efficient algorithms can be implemented by the translator: these operations are efficiently implemented in at most $log_2 p$ steps on a hypercube or mesh network, where p nodes are involved. We now conclude this section with a few examples.

Consider a user defined reduction operation implementing the global `minmod` function. In solving discretized hyperbolic partial-differential equations, the flux of a quantity at a point is characterized by the smaller of finite-differences of that quantity in some vicinity of the point because near a shock the larger values may be unreliable. The binary function `minmod` is used to find the smaller value. It is defined as follows. Given two input values of the same sign, it returns the value with the minimum modulus (absolute value). When there is a sign disagreement between two values, zero is returned. (This corresponds to the case of a variable that cannot decide whether it is coming or going.) We define the scalar function `minmod` as

```
      REAL FUNCTION minmod(s1,s2)
      if (s1*s2 .gt. 0.0 .and. s1.lt.0.0) then
C       s1 and s2 are < 0
        minmod =  AMAX1(s1,s2)
      else if (s1*s2 .gt. 0.0)  then
C       s1 and s2 are > 0
        minmod = AMIN1(s1,s2)
      else
C      If s1 and s2 are not of the same sign or one is 0
        minmod = 0.0
      endif
      return
      end
```

To compute the `minmod` of a set of values `myslope` over all nodes, one simply writes `slope = minmod`{`myslope`}.

A slightly more complicated application of a reduction operation arises in Gaussian elimination with row pivoting. Suppose an $m \times n$ matrix is stored by rows across

P nodes where row_i is assigned to node $q = mod(i-1, P)$ and that nodes are numbered $0, \ldots, P-1$. Define the space $\mathcal{X} \equiv \mathcal{Z}^+ \times \mathcal{R}^+$, where $\mathcal{Z}^+ \equiv$ *positive integers* and $\mathcal{R}^+ \equiv$ *non-negative real numbers.* This space represents the Cartesian product of *row numbers* with absolute values of *pivot entries* from the matrix. We now define a function $\mathcal{C} : \mathcal{X} \times \mathcal{X} \rightarrow \mathcal{X}$ where

$$\mathcal{C}((i,x),(j,y)) = \begin{cases} \text{(i,x)}, & \text{if } x > y \\ \text{(j,y)}, & \text{if } y > x \\ \text{(i,x)}, & \text{if } i < j,\ y = x \\ \text{(j,y)}, & \text{if } j < i,\ y = x \\ \text{(i,x)} = \text{(j,y)}, & \text{if } i = j,\ y = x. \end{cases}$$

Thus, $\mathcal{C}$ is commutative and associative and can be utilized in a reduction operation. The following subroutine illustrates the use and definition of $\mathcal{C}$ to calculate a row-pivot element among P nodes.

```
      SUBROUTINE pivot(B,m,n)
      IMPLICIT NONE
      PARAMETER (size=64,minp=1)
c     the rows for this node
      REAL*8 B(size/minp,size)
c     matrix dimensions
      INTEGER m,n
```

```
   INTEGER row,i
c  cycle in Gaussian Elimination
   INTEGER k
c  pivot(1) has the value, pivot(2) the row number as a real
   REAL*8 pivot(2),pivot2(2)


c   LOOP OVER COLUMNS OF MATRIX
    DO 500 k = 1, n
c  basis: determine pivot row from local rows
      pivot(1) = B(1,k)
      pivot(2) = thisnode+1
      do 20 irow = 2, m/numnode
        row = thisnode + 1 + (irow-1)*numnode
        pivot2(1) = B(irow,k)
        pivot2(2) = row
        call Cpivot(pivot2,pivot)
 20   continue


 c determine pivot row globally
      call Cpivot {pivot(1:2),pivot(1:2)}


 c output results
      print* ,thisnode, k, INT(pivot(2)), pivot(1)
```

```
500 continue
    END

          SUBROUTINE Cpivot(p1,p2)
          REAL* 8 p1(2),p2(2)
  c  calculate a pivot where p1 and p2 contain
  c  row(i,k) element and  value k
  c  output tuple goes to p2
          INTEGER* 4 i,j
          REAL* 4 v,w
          v = p1(1)
          i = p1(2)
          w = p2(1)
          j = p2(2)
          if ( v .gt. w ) then
            p2(1) = p1(1)
            p2(2) = p1(2)
          else if ( v .eq. w ) then
           if ( i.eq.j ) then
      print* , Error: Cpivot() has same row p1 and p2
            else if ( i .lt. j ) then
              p2(1) = p1(1)
               P2(2) = p1(2)
```

```
      endif
      endif
      RETURN
      END
```

Quite complex commutative operators can be devised. Consider the following approach to computing a matrix times a vector *matvec* that uses the reducing version of *addvec*, a subroutine that adds vectors. For this new data type, we define *addition* by introducing a subroutine that adds two vectors and returns their sum:

```
      SUBROUTINE addvec(a,b,c,D)
      dimension a(1),b(1),c(1)
      DO 1 i=1,D
1            c(i) = a(i) + b(i)
      RETURN
      END
```

We now define matrix multiplication using the reducing version of *addvec*. Note that D is the dimension of the matrix, and $N << D$ is the number of columns stored in each node. The index $global(i)$, $i = 1, \ldots, N$, indicates how the columns are distributed across nodes, but the details of a particular choice of such a distribution is left unspecified. One such choice is defined in the following:

$$\text{global(j)=THISNODE} + \text{j} * \text{NN}$$

where $NN = D/N$. The matrix is assumed to be distributed in a similar way.

```
      SUBROUTINE matvec(res, mat, vec, D, N)
      dimension res(N), mat(D,N),vec(N), s(1024),t(N)
      DO 1 i=1, D
          s(i) = 0.0
          DO 1 j=1, N
              s(i) = s(i) + mat(i,j) * vec(j)
1       continue
       addvec{s,t,D}
       DO 2 i=1,N
2        res(i) = t(global(i))
```

Note that the function *addvec* is clearly associative and commutative; its global version adds all the components among all nodes.

Appendix B

Review of Some Numerical Methods

B.1 Finite Differences

Finite difference discretization methods are based on the approximation of differential operators by simple arithmetic expressions at each point of a domain.

Let us consider Poisson's equation.

$$-\Delta u = f, \quad u = 0 \quad \text{on} \quad \partial\Omega$$

From the definition of partial derivatives, for each point (x, y) and small enough h, we have

$$\frac{\partial^2 u}{\partial x \partial x}(x, y) \approx \frac{u(x+h, y) - 2u(x, y) + u(x-h, y)}{h^2}$$

and

$$\frac{\partial^2 u}{\partial y \partial y}(x, y) \approx \frac{u(x, y+h) - 2u(x, y) + u(x, y-h)}{h^2}.$$

Hence Laplace's operator Δ is approximately

$$\Delta_h u = 4u(x, y) - u(x+h, y) - u(x-h, y) - u(x, y+h) - u(x, y-h)$$

If we simply replace Laplace's operator by this approximation in Poisson's equation, we get

$$\Delta_h u = 0$$

for each point in the interior of the unit square and $u = 0$ for each point on the boundary of the unit square.

Now if we subdivide the unit square by parallels to the axes spaced $\Delta x = \Delta y = h = 1/n$ units apart and let $u_{i,j} = u(ih, jh)$, then the discretization of the Laplace operator is

$$(\Delta_h u)_{i,j} = 4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1},$$

and the discretized version of Poisson's equation becomes

$$(\Delta_h u)_{i,j} = 0$$

for i and j in $\{1, \ldots, n-1\}$ and

$$u_{i,j} = 0$$

for each i and j in $[0,1] \times \{0,n\} \cup \{0,n\} \times [0,1]$ (according to our simple boundary conditions).

We can solve these equations directly (using Gaussian elimination say). Typically though, we will have a very large number of mesh points, and direct methods will be prohibitively expensive. Alternatively, we can use an iterative method. One such is the following method where we write the discretized Poisson's equation as

$$u_{i,j} = \frac{1}{4}\left[u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right],$$

make an initial guess for each $u_{i,j}$ and use the above formula to update all $u_{i,j}$'s in some order. These methods are sometimes called relaxation methods. Such pointwise computations of elements of a collection from the values of the collection at a few neighboring points are typical of finite difference methods. They arise from the very nature of finite difference approximations of differential operators.

B.2 Finite Elements

The finite element method involves three essential steps: the formulation of a variational equation, choosing approximation spaces for the variational formulation, and translating the resulting discretization into a computer program. The road from differential to discrete variational equations is a windy and difficult one. There are many places where mathematical expertise must be brought to bear. Thus it is unlikely that this process can be completely automated. However, once we have a variational formulation, a specification of the mesh and a reference finite element, the production of computer programs to solve the resulting discrete variational equations can be automated. By automated we mean that a user can express algorithms in variational form without bothering about low-level details.

We begin with Poisson's equation and define a suitable space of functions V on Ω [16]. Fiven $v \in V$, multiply both sides of Poisson's equation by v and integrate both sides over Ω. After integration by parts, we will have

$$\forall v \in V, \quad \int_\Omega \nabla u \cdot \nabla v = \int_\Omega f v.$$

This equation is called the weak or variational (or weak) formulation of the original differential equation.

One can show that the solutions of the variational equation correspond to solutions of the original differential equation [16]. We can give names to the two sides of the variational equation:

$$a(u,v) := \int_\Omega \nabla u \cdot \nabla v \quad \text{and} \quad (f,v) := \int_\Omega fv,$$

These functions are in fact (bilinear) forms on $V \times V$ and $F \times V$ respectively where F is a suitable space to which f belongs.

The finite element method is a method for discretizing such forms and thus discretizing differential equations. The idea is to approximate elements of V by members of a finite-dimensional subspace of V in a systematic way. To do so, one divides Ω into small, polygonal pieces, defines finite-dimensional approximation spaces on each piece and uses these spaces to compute the integrals appearing in multilinear forms. Such subdivisions of a region are called meshes. Each piece of a mesh is sometimes called a general element.

The systematic way to define finite-dimensional approximation function spaces on a mesh is to choose a "reference element" which can be mapped affinely onto each general element and on which we can compute finite-dimensional approximations easily. Thus a (reference) finite element is often defined [16] as a triple: a domain $\mathcal{K}$ with piecewise smooth boundary, a finite-dimensional space $\mathcal{P}$ of functions from $\mathcal{K}$ to $\mathbb{R}$, and a basis $\mathcal{N}$ for $\mathcal{P}'$ the dual of $\mathcal{P}$. The basis $\mathcal{N}$, sometimes called the nodal basis or the set of nodal variables, allows us to interpolate functions on $\mathcal{K}$ by members of $\mathcal{P}$. Members of a basis for $\mathcal{P}$ are often called shape functions. One can then expand multilinear forms in terms of "local" forms defined on the reference element and shape functions using the fact that the operation is linear.

For example, for the form a, a mesh $\mathcal{M}$ on Ω and a reference element $\mathcal{T}$, we have

$$a(v, w) = \sum_{e \in \mathcal{M}} a_e(v, w)$$

where

$$\begin{aligned}
a_e(v, w) &:= \int_e \nabla v(x) \cdot \nabla w(x)\, dx \\
&= \int_{\mathcal{T}} \sum_{j=1}^{2} \frac{\partial}{\partial x_j} v(J_e \xi + x_e) \frac{\partial}{\partial x_j} w(J_e \xi + x_e) \det(J_e)\, d\xi \\
&= \int_{\mathcal{T}} \sum_{j,m,m'=1}^{2} \frac{\partial \xi_m}{\partial x_j} \frac{\partial}{\partial \xi_m} \left(\sum_{\lambda \in \mathcal{L}} v^{g(e,\lambda)} \phi_\lambda(\xi) \right) \times \\
&\quad \frac{\partial \xi_{m'}}{\partial x_j} \frac{\partial}{\partial \xi_{m'}} \left(\sum_{\mu \in \mathcal{L}} w^{g(e,\mu)} \phi_\mu(\xi) \right) \det(J_e)\, d\xi \\
&= \begin{pmatrix} v^{g(e,1)} \\ \cdot \\ \cdot \\ v^{g(e,|\mathcal{L}|)} \end{pmatrix}^t \mathbf{K}^e \begin{pmatrix} w^{g(e,1)} \\ \cdot \\ \cdot \\ w^{g(e,|\mathcal{L}|)} \end{pmatrix}.
\end{aligned}$$

Here, the *local stiffness matrix*, $\mathbf{K}^e$, is given by

$$K^e_{\lambda,\mu} := \sum_{j,m,m'=1}^{2} \frac{\partial \xi_m}{\partial x_j} \frac{\partial \xi_{m'}}{\partial x_j} \det(J_e) \int_{\mathcal{T}} \frac{\partial}{\partial \xi_m} \phi_\lambda(\xi) \frac{\partial}{\partial \xi_{m'}} \phi_\mu(\xi)\, d\xi$$

$$= \sum_{j,m,m'=1}^{2} \frac{\partial \xi_m}{\partial x_j} \frac{\partial \xi_{m'}}{\partial x_j} \det(J_e) K_{\lambda,\mu,m,m'}$$

for $\lambda, \mu \in \mathcal{L}$. We have written out in full the affine maps from the reference element $\mathcal{T}$ to the general element e, and $v^{g(e,j)}$ stands for $N_{g(e,j)}(v)$. As we will see, this kind of element-by-element expansion (or "assembly") of a multilinear form translates directly into code using the classes we have defined.

One can recover the spirit of the original "operator" form of the problem by introducing the notion of the "action" of the bilinear form, $a(\cdot,\cdot)$. Given $v \in V$, the action of $a(\cdot,\cdot)$ on v, denoted by Av is given by $\sum_i (v, \phi_i)\phi_i$, where $\{\phi_i\}$ is a basis for the finite elment subspace of V. Then the variational formulation becomes

$$Au = g$$

where $g = \sum_i (f, \phi_i)\phi_i$.

Bibliography

[1] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9), September 1990.

[2] V. Aho, A, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[3] J. R. Allen and K. W. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, pages 491–542, October 1987.

[4] F. André, J.-L. Pazat, and H. Thomas. Pandore: A system to manage data distributions. In *Proceedings of the International Conference on Supercomputing*, pages 380–388, 1990.

[5] R. K. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, New York, New York, 1991.

[6] B. Bagheri, M. Draghicescu, and L. R. Scott. Object-oriented finite elements. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, Sunriver, Oregon, April 1994.

[7] B. Bagheri, A. Ilin, and L. R. Scott. Parallelizing UHBD. Technical Report 167, Department of Mathematics, the University of Houston, 1993.

[8] B. Bagheri, A. Ilin, and L. R. Scott. A comparison of shared and distributed memory scalable parallel processors : 1. KSR shared memory. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 9–16, Knoxville, Tennessee, May 1994. IEEE.

[9] B. Bagheri, A. Ilin, and L. R. Scott. Parallel 3-D MOSFET simulation. In *Proceedings of the* 27^{th} *Annual Hawaii International Conference on System Sciences*, volume 1, pages 46–54, 1994.

[10] B. Bagheri, B. Raghavachari, and L. R. Scott. IPC: a parallel extension of C (the IPC reference manual). Technical report, Department of Mathematics, the University of Houston, 1990.

[11] B. Bagheri and L. R. Scott. Software environements for the parallel solution of partial differential equations. In *Computing Methods in Applied Sciences and Mathematics*, pages 378–392, Philadelphia, PA, 1990. Society of Industrial and Applied Mathematics.

[12] B. Bagheri, L. R. Scott, and S. Zhang. Implementing and using high-order finite elements. In *Proceedings of the Second International Conference on Spectral and High Order Methods*, Montpelier, France, 1992. also in Computational Methods in Applied Mechanics and Engineering, SIAM.

[13] François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby Yang. Distributed pC++: Basic ideas for an object parallel language. In *Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 1–24, Sunriver, Oregon, April 1994. Society of Industrial and Applied Mathematics.

[14] D. Callahan and K. Kennedy. Compiling programs for distributed-memory machines. *Journal of Supercomputing*, October 1988.

[15] Andrew A. Chien. Concurrent aggregates: Using multiple-access data abstractions to manage complexity in concurrent programs. *OOPS messenger*, 2(2), April 1991.

[16] P. G. Ciarlet. *The finite element method for elliptic problems.* North-Holland Publishing Co., Amsterdam, 1978.

[17] Andrew A. Cien and William J. Dally. Concurrent aggregates (CA). In *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 187–196. Association for Computing Machinery, March 1990.

[18] Terry W. Clark, Reinhard v. Hanxleden, J. A. McCammon, and L. Ridgway Scott. Parallelizing molecular dynamics using spatial decomposition. In *Proceedings of the Scalable High-Performance Computing Conference*, Knoxville, Tennessee, May 1994.

[19] M. Conway. A multiprocessor system design. In *AFIPS Fall Joint Computer Conference*, pages 139–146, 1963.

[20] David Culler. Efficient communication in MPPs and NOWs. In *Proceedings of the Scalable High-Performance Computing Conference*, May 1994. Invited presentation, text available through gopher.

[21] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1989.

[22] Ron Cytron, Jeeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 1991.

[23] M. E. Davis, J. D. Luty, B. A. Allision, and J. A. McCammon. Electrostatics and diffusion of molecules in solution: Simulations with the University of Houston Brownian Dynamics program. *Computer Physics Communications*, 62:187–197, 1990.

[24] M. E. Davis and J. A. McCammon. Solving the finite difference linearized Poisson-Boltzmann equation: A comparison of relaxation and conjugate gradient methods. *Journal of Computational Chemistry*, 10:386–391, 1989.

[25] Edsger W. Dijkstra. Cooperating sequential processes. Technical report, Technical University, Eindhoven, Eindhoven, The Netherlands, 1965.

[26] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, Massachusetts, 1986.

[27] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependance graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987 1987.

[28] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Department of Computer Science, Rice University, December 1990.

[29] G. A. Geist and C. H. Romine. LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM J. Sci. Stat. Comput.*, 9:639–649, 1988.

[30] D. H. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[31] Roland Glowinski. *Numerical methods for nonlinear variational problems.* Springer-Verlag, New York, 1984.

[32] H. Golub, G and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, Maryland, 1983.

[33] Rajiv Gupta. SPMD execution of programs with pointer-based data structures on distributed memory machines. *Journal of Parallel and Distributed Computing*, 16:92–107, 1992.

[34] B. Hansen. The nucleous of a multiprogramming system. *Communications of the ACM*, 13(4), April 1970.

[35] Philip J. Hatcher and Michael J. Quinn. *Data-parallel programming on MIMD computers.* MIT Press, Cambridge, Mass., 1991.

[36] M. C. B. Henessy and G. D. Plotkin. Full abstraction for a simple programming language. *Lecture Notes in Computer Science*, 74, 1979.

[37] High Performance Fortran Forum. *Proceedings of the High Performance Fortran Forum.* Houston, TX, January 1992. Available as technical report CRPC-TR 92225 from the Center for Research on Parallel Computation, Rice University.

[38] C. A. R Hoare. Communicating sequential processes. *Communications of the ACM*, August 1978.

[39] C. A. R Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, 1985.

[40] M. Holst and F. Saied. Multigrid solution of the Poisson-Boltzmann equation. *Journal of Computational Chemistry*, 14:105–113, 1993.

[41] Susan Horwitz, Thomas Reps, and David Binkley. Interpocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[42] V. P. Il'in. *Iterative Incomplete Factorization Methods.* World Scientific, Singapore, 1992.

[43] Inmos Ltd. *Occam Programming Manual.* Prentice-Hall, Hertfordshire, UK, 1988.

[44] Intel Corporation. *iPSC/2 and iPSC/860 Programmer's Reference Manual.* Intel, 1991.

[45] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, EnglewoodCliffs, N.J., 1978.

[46] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic process partitioning for parallel computation. *International Journal of Parallel Programming*, 16(5):365–382, 1987.

[47] M. S. Lam and M. Rinard. Coarse grained parallelism in jade. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, April 1991.

[48] J. K. Lee and Dennis Gannon. Object oriented parallel programming: Experiments and results. In *Proceedings of Supercomputing '91*, pages 273–282, Albuquerque, New Mexico, November 1991.

[49] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using kali. In *Advances in Languages and Compilers for Parallel Processing*, pages 364–384, 1991.

[50] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2), February 1979.

[51] E. Morenoff and J. B. McLean. Interprogram communications, program string structures and buffer files. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 175–183, 1967.

[52] M. H. A. Newman. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics*, 43(2):223–243, 1942.

[53] G. D. Plotkin. A powerdomain consruction. *SIAM Journal on Computing*, 5:452–487, 1976.

[54] G. D. Plotkin. An operational semantics for CSP. In *Formal Description of Programming Concepts–II*, pages 199–223. North Holland, 1983.

[55] D. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[56] L. R. Scott, J. M. Boyle, and B. Bagheri. Distributed data structures for scientific computation. In *Hypercube Multiprocessors 1987*, pages 55–66, Philadelphia, 1987. Society of Industrial and Applied Mathematics.

[57] J. E. Stoy. *Denotational Semantics: The Scott Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Mass., 1977.

[58] C. Strachey and R. Milne. *A theory of programming language semantics.* Wiley & Sons, New York, New York, 1976.

[59] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[60] R. Andre Tarjan. Testing for graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[61] Robert D. Tennent. *Semantics of Programming Languages.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[62] F. Thomasset. *Implementation of finite element methods for Navier-Stokes equations.* Springer-Verlag, New York, 1981.

[63] A. Tucker, L.W. Mainwaring. Cmmd: Active messages on the cm-5. *Parallel computing*, 20(4), April 1994.

[64] R. von Hanxleden and L. Ridgway Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13:312–324, 1991.

[65] R. von Hanxleden and L. Ridgway Scott. Correctness and determinism of parallel monte carlo processes. *Parallel Computing*, 18:121–132, 1992.

[66] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE–10(4), July 1984.

[67] Hans Zima, H. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.