Integrating Task and Data Parallelism

Thesis by
Berna Massingill

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

1993
(submitted 3 May 1993)

Caltech CS-TR-93-01

Abstract

Many models of concurrency and concurrent programming have been proposed; most can be categorized as either task-parallel (based on functional decomposition) or data-parallel (based on data decomposition). Task-parallel models are most effective for expressing irregular computations; data-parallel models are most effective for expressing regular computations. Some computations, however, exhibit both regular and irregular aspects. For such computations, a better programming model is one that integrates task and data parallelism. This report describes one model of integrating task and data parallelism, some problem classes for which it is effective, and a prototype implementation.

Acknowledgments

I want first to thank my academic and research advisor, K. Mani Chandy, without whose guidance, support, and endless patience the work presented in this report would not have been possible.

I also want to thank the other members of my research group—Carl Kesselman, John Thornley, Paul Sivilotti, Ulla Binau, Adam Rifkin, Peter Carlin, Mei Su, Tal Lancaster, and Marc Pomerantz—and Peter Hofstee; they listened to my ideas, read my prose, and provided constructive criticism.

Finally, I want to thank Eric van de Velde, who provided a library of SPMD routines with which to test my implementation and answered many questions; Ian Foster, Steve Tuecke, Sharon Brunett, and Bob Olson, who explained the internal workings of the PCN implementation and helped me track down errors; and Chau-Wen Tseng, Seema Hiranandani, and Charles Koelbel, who explained the implementation of Fortran 77 D.

The research described in this report was supported in part by an Air Force Laboratory Graduate Fellowship (under the sponsorship of Phillips Laboratory at Kirtland AFB, Albuquerque, NM), in part by a Milton E. Mohr Graduate Fellowship (administered by Caltech), in part by the Air Force Office of Scientific Research under grant number AFOSR-91-0070, and in part by the NSF under Cooperative Agreement No. CCR-9120008. The government has certain rights in this material.

Contents

List of Figures

Preface

Many different models of concurrency and of concurrent programming have been proposed. Although models may differ in many respects, most can be categorized as either task-parallel (based on functional decomposition) or data-parallel (based on data decomposition). Task-parallel models are general and flexible, and hence effective for expressing irregular computations, but they can be difficult to reason about and cumbersome for expressing regular computations. Data-parallel models are more restricted, but they are very effective for expressing regular computations and are easier to reason about. Some computations, however, exhibit both regular and irregular aspects. For such computations, we propose that a better programming model is one that integrates task and data parallelism, retaining the flexibility of the former while exploiting the advantages of the latter. This report describes one model of integrating task and data parallelism, some problem classes for which it is effective, and a prototype implementation.

Organization of this report

The chapters of this report are organized as follows:

- §1 presents an overview of task parallelism and data parallelism.
- §2 describes the proposed programming model for integrating task parallelism and data parallelism and describes some problem classes for which the model is effective.
- §3 describes the overall design of a prototype implementation.
- §4 presents detailed specifications for the prototype implementation's library of procedures.
- §5 discusses internal details of the prototype implementation.
- §6 presents two example programs in detail.
- §7 summarizes the work described in this report, presents conclusions, and suggests directions for further research.

The report also includes the following appendices:

- §A gives an overview of PCN syntax and terminology. (PCN [5], or Program Composition Notation, is the task-parallel notation used for the prototype implementation.)
- §B describes how to compile, link, and execute programs written using the prototype implementation.
- §C describes additional library procedures included in the prototype implementation.
- §D presents a case study of adapting an existing library of data-parallel programs for use with the prototype implementation.
- §E describes the required files and installation procedure for the prototype implementation.
- §F presents additional internal details of the prototype implementation.

Terminology and conventions

Concurrency and parallelism

Throughout this report, *concurrency* and *parallelism* have the same meaning. Processes A and B are said to execute concurrently if the interleaving of their atomic actions is arbitrary.

Processes, processors, and virtual processors

In the mapping of a program to a machine, processes are assigned to physical processors. If the machine does not have a single address space, data elements are also assigned to physical processors. It is sometimes useful, however, to introduce an intermediate layer of abstraction, the *virtual processor* (or *abstract processor*) and to think of assigning processes and data elements to virtual processors, which are then in turn mapped (not necessarily one-to-one) to physical processors. Virtual processors are considered to be entities that persist throughout the execution of a program; if processors have distinct address spaces, the virtual processors are considered to have distinct address spaces as well.

By mapping processes to different virtual processors, the programmer can capture the idea that the processes could be executed simultaneously; at the same time, the use of virtual rather than physical processors achieves a degree of independence from the physical machine and hence a degree of portability.

In the following chapters, for the sake of brevity, the term *processor* will be taken to mean "virtual processor".

In addition, it is assumed that each (virtual) processor can be identified by a unique number, referred to as its *processor number*.

Typesetting conventions

The following typesetting conventions are used in subsequent chapters:

- `Typewriter font` is used for literals—words that are to appear exactly as shown in commands and programs, including program names, file names, and variable names.
- *Italics* are used for variables—parts of commands and program statements that are to be replaced by values of the user's choice.

For example, the following gives the syntax for a program statement to assign a value to a variable named `X`:

`X :=` *Value*

The characters "`X :=`" must appear exactly as shown; *Value* is to be replaced by a value of the user's choice.

Chapter 1

Overview of task and data parallelism

As noted in the preface, although there are many different models of concurrent programming, most can be categorized as either task-parallel or data-parallel. The most important differences between task-parallel models and data-parallel models are as follows:

- Task-parallel models emphasize functional decomposition. Data-parallel models emphasize data decomposition.
- A task-parallel program may contain multiple concurrently-executing threads of control, and these threads of control may interact. A data-parallel program initially has a single thread of control and can contain multiple threads of control only as part of a parallel `FOR` or `DO` loop (described in §1.2.1) in which the concurrently-executing threads of control do not interact.
- Task-parallel models are more general and hence have greater expressive power for irregular computations. Data-parallel models are more restrictive and hence more suitable for regular computations.
- Reasoning about task-parallel programs requires techniques other than those used to reason about sequential programs and can be very difficult. Reasoning about data-parallel programs can be done with the same techniques used for sequential programs.

In the following sections, we describe task-parallel and data-parallel models in more detail.

1.1 Task parallelism

1.1.1 Programming models

In a task-parallel programming model, the focus is on decomposing the problem by dividing it into (possibly different) tasks, i.e., on functional decomposition. There is a wide range of programming models that can be categorized as task-parallel. Some aspects in which they differ are the following:

- The unit of concurrency, which can range from coarse-grained processes to individual expressions or subexpressions.
- Whether concurrency is explicit or implicit.
- What constitutes an atomic action, i.e., an action with which another processor cannot interfere.
- Whether the number of processes or other units of concurrency is fixed (static) or can change during program execution (dynamic).
- Whether all processes share a single address space or some processes occupy distinct address spaces.
- The mechanisms for communication and synchronization among concurrently-executing units of computation. Communication and synchronization mechanisms can be based on either message-passing (synchronous or asynchronous) or shared data (through a variety of constructs including monitors, semaphores, and single-assignment variables).

1.1.2 Example programming notations

Given the range of ways in which task-parallel models can differ, it is not surprising that the range of task-parallel programming notations is broad. Examples include the following:

- Imperative notations with support for communicating sequential processes. In such notations, the basic unit of concurrency is the process, and concurrency is explicit. Examples include the following:
  - Fortran M [8], which adds parallel blocks and asynchronous channel communication to Fortran 77.
  - Cosmic C [20], which adds process creation and asynchronous point-to-point message-passing to C.
  - Ada [2], in which programs may define tasks that execute concurrently, and in which communication and synchronization are accomplished via the rendezvous construct.

In the first two examples, each process has a distinct address space; in the last example, concurrently-executing processes share an address space.

- Declarative notations, in which concurrency is implicit, there is a single address space, the unit of concurrency is the (sub)expression, and synchronization is based on data flow. An example is the following:
  - Declarative Ada [22], in which programs are written in a declarative style using only single-assignment variables. Synchronization is controlled by the semantics of the single-assignment variable: Such a variable can have a value assigned to it at most once, and attempts to read its value suspend until a value has been assigned.
- Notations that combine imperative and declarative styles. Examples include the following:
  - PCN (Program Composition Notation) [5, 9], which combines features of imperative programming (sequential composition and multiple-assignment variables) with features of declarative programming (parallel composition and single-assignment variables).
  - CC++ (Compositional C++) [4], which adds parallel loops and blocks, dynamic process creation, explicitly-defined atomic actions, and single-assignment variables to C++.
- Notations based on the Actors model [1]. An example is the following:
  - Cantor [3]: The basic unit of concurrency is the object; concurrently-executing objects communicate via asynchronous message-passing. The computation is message-driven: An object is inactive until it receives a message. The object's response to the message may include sending messages to other objects, creating new objects, or modifying its state; it performs a finite number of such actions and then either waits for another message or terminates.

What these diverse notations have in common is the potential for multiple interacting threads of control.

1.1.3 Reasoning about programs

Reasoning about programs based on a task-parallel model can be very difficult. Methods of proof based on Hoare triples [17] work well for sequential programs because the behavior of any program depends only on its preconditions and the semantics of its statements. In a program containing multiple concurrent processes, however, the behavior of each process depends not only on its preconditions and the semantics of its statements, but also on its environment (the actions of other concurrently-executing processes). Proofs of concurrent programs depend not only on showing that each process considered in isolation behaves correctly, but also on showing that the processes interact correctly.

1.1.4 Application classes

Task-parallel models, especially those that allow dynamic process creation, are particularly effective for expressing irregular or dynamic computations, such as those found in operating systems and in discrete-event simulations.

1.2 Data parallelism

1.2.1 Programming models

In a data-parallel programming model, the focus is on decomposing the problem by decomposing the data: Large data structures are decomposed and distributed among processes or processors; the computation is viewed as a sequence of operations on these distributed data structures; and concurrency is obtained by acting concurrently on different parts of the data [11, 16].

In the simplest view, then, a data-parallel computation is a sequence of *multiple-assignment statements*, where a multiple-assignment statement is an assignment statement with multiple expressions on the right-hand side and multiple variables on the left-hand side. A multiple-assignment statement has the following operational semantics: First evaluate all right-hand-side expressions; then assign the values of the right-hand-side expressions to the left-hand-side variables. Multiple-assignment statements may be implicit, as in whole-array operations, or explicit, as in the `FORALL` statement of CM Fortran [21].

Observe, however, that the idea of the multiple-assignment statement can be generalized to include parallel `FOR` or `DO` loops in which each iteration of the loop is independent of all other iterations. Each loop iteration constitutes a thread of control; the loop as a whole consists of a collection of threads of control that begin together (at loop initiation) and terminate together (at loop termination) but do not otherwise interact. Execution of such a loop is the only situation in which a data-parallel program may have multiple threads of control.

Thus, concurrency in a data-parallel computation can be:

- Implicitly specified via an operation on a distributed data structure.
- Explicitly specified via a multiple-assignment statement or a parallel loop.

Observe further that implicit concurrency and some forms of explicit concurrency imply a single address space, often referred to as a *global name space*.

1.2.2 Example programming notations

Given that data-parallel models are all based on viewing a computation as a sequence of multiple-assignment statements, it is not surprising that the range of data-parallel programming notations is much narrower than the range of task-parallel notations. Examples of data-parallel notations include:

- Connection Machine Fortran [21]. This set of extensions to Fortran 77 provides a global name space and allows both implicit concurrency, in the form of operations on distributed data structures, and explicit concurrency, in the form of multiple-assignment statements (the `FORALL` statement).
- Fortran 77 D [12]. This set of extensions to Fortran 77 provides a global name space and allows only implicit concurrency, inferred by the compiler from `DO` loops on elements of distributed data structures.
- High Performance Fortran [15]. This set of extensions to Fortran 90 [18] provides a global name space and allows both implicit concurrency, in the form of operations on distributed data structures, and explicit concurrency, in the form of multiple-assignment statements and parallel loops (the `FORALL` construct).
- Dataparallel C [14, 13]. This set of extensions to C allows explicit concurrency, in the form of parallel loops (the *domain select* statement). A global name space is provided; in addition, each iteration of a parallel loop has a local name space.

1.2.3 Reasoning about programs

One of the biggest advantages of the data-parallel model is that it leads to concurrent programs that we believe are no more difficult to reason about than sequential programs. To the extent that a data-parallel program follows the model described above, it is possible to reason about it in the same way that one reasons about sequential programs. Even when there are multiple concurrently-executing threads of control, each thread of control executes independently of the others. Thus, we believe that it is possible to reason about each thread of control in isolation, in the same way that one reasons about a sequential program.

Observe, however, that departures from the model—e.g., parallel loops in which the concurrently-executing iterations of the loop are not independent—introduce the same difficulties involved in reasoning about task-parallel programs.

1.2.4 Application classes

Data-parallel models are particularly effective for expressing regular computations, such as the large array-based computations found in many scientific programs.

1.2.5 Implementation issues

It may be noted that the preceding discussion of data-parallel models excludes some widely-used methods of writing concurrent programs based on data decomposition, most notably the SPMD (single program, multiple data) method (described later in this section). It is our contention, however, that these methods are more properly viewed as ways of implementing the programming model discussed above (§1.2.1).

There are a number of methods for implementing the data-parallel programming model described in §1.2.1. The model maps readily to some architectures; for example, it is easy to see how a sequence of multiple-assignment statements can be performed on an SIMD (single instruction stream, multiple data stream) architecture. In implementing a data-parallel model on an MIMD (multiple instruction stream, multiple data stream) architecture, however, some care must be taken to ensure that the implementation preserves the semantics of the programming model. The following two points must be considered in developing any implementation of the data-parallel model:

- The relationship between data elements and processes.
- The degree of synchronization between processes.

A distributed data structure, as briefly defined in §1.2.1, is a data structure in which the data is partitioned into sections and distributed among processes. This can be done in either of the following two ways:

- One data element per process. Multiple assignment can then easily be implemented by having all processes simultaneously execute an assignment statement.
- Multiple data elements per process. Multiple assignment must then be simulated, with each process executing a sequence of assignment statements, one for each of its elements. Care must be taken in this case that the implementation preserves the semantics of the programming model—i.e., that evaluations of expressions on the right-hand side of a multiple-assignment statement are based on the values of variables before execution of the statement.

Inter-process synchronization can also be handled in different ways, including the following:

- Tight synchronization, i.e., synchronization between each pair of consecutive operations. With this method, suitable for SIMD architectures, individual processing elements execute a sequence of operations in lockstep. This method clearly preserves the semantics of the programming model.
- Loose synchronization with a master process. With this method, suitable for MIMD architectures, there is a master process that executes the initial single thread of control. When control reaches an operation involving concurrency—e.g., a multiple assignment—the master process creates a number of concurrently-executing slave processes that perform the operation and then terminate, returning control to the master process. Synchronization thus consists of a series of fork/join operations. A sequence

of multiple-assignment statements can be implemented using a single fork/join pair, provided such an implementation maintains the semantics of the programming model.

- Loose synchronization without a master process. With this method, suitable for MIMD architectures and usually referred to as SPMD (single program, multiple data), all processes execute the same program, each on its own data. This method, like the preceding method, is loosely synchronized in that at a given instant, different processes can be executing different parts of the program. As with the preceding method, however, synchronization must be performed with sufficient frequency to maintain the semantics of the programming model. Synchronization among processes can be enforced in a number of ways; a common method is the barrier construct, which requires all processes to arrive at a particular point in the computation before any of them can proceed beyond it.

  SPMD implementations are often based on multiple address spaces, which is particularly appropriate for multiple-address-space architectures but can be implemented on single-address-space architectures as well. In a multiple-address-space SPMD implementation, each process has direct access only to its own portion of a distributed data structure (referred to as a *local section*); to obtain data from another local section, it must communicate with the process that owns the local section.

Chapter 2

Integrating task and data parallelism

As described in §1, both task parallelism and data parallelism have their strengths and weaknesses. Task parallelism is extremely general and thus effective for expressing a wide range of computations, including irregular and dynamic computations. However, this generality has a price; task-parallel programs can be difficult to reason about. Data parallelism, on the other hand, is fairly restrictive and hence most suitable for regular computations. The restrictiveness of the model, however, means that data-parallel programs are usually easier to reason about and understand than task-parallel programs. Further, the amount of concurrency in a data-parallel program scales with problem size in a way that is not generally true for a task-parallel program.

Thus, it is natural to ask whether we can integrate task and data parallelism into a single programming model that combines the strengths of both. Such a programming model would be particularly effective for problems that exhibit both task and data parallelism.

This chapter presents one such programming model and describes some classes of problems for which it is appropriate. While our programming model is simple and restricted, it is appropriate for a number of interesting problem classes, and its simplicity makes it easier to define and implement than a more general model.

2.1 A programming model for integrating task and data parallelism

There are a number of classes of problems (described in more detail in §2.3) in which the problems can, at the top level, be functionally decomposed into subproblems, some or all of which are suitable for data decomposition. Such a problem can be readily expressed as a task-parallel program in which one or more tasks are data-parallel programs.

Thus, we propose a simple programming model based on allowing data-parallel programs to serve as subprograms within a task-parallel program. We define the interaction as follows:

- The task-parallel program can call a data-parallel program in the same way that it calls a sequential subprogram. The calling task-parallel program and the called data-parallel program interact and share data only via call/return and parameter passing, but parameters can include distributed data structures.
- The task-parallel program can create distributed data structures and perform simple manipulations on them. This allows a more natural interaction with data-parallel programs, much of whose data may be in the form of distributed data structures. In particular, it allows the task-parallel program to transfer data from one distributed data structure to another, as might be appropriate when the task-parallel program calls several different data-parallel programs.

For example, consider a task-parallel programming model in which a program consists of a set of concurrently-executing sequential processes. Each process performs a sequence of operations drawn from a repertoire of possible operations, typically including the following:

- Primitive operations (e.g., expression evaluation or assignment).
- Calls to subprograms.
- Process creation.
- Communication and synchronization with other processes.

The key idea of our programming model is to add to this repertoire the following two kinds of operations:

- Creation and manipulation of distributed data structures.
- Calls to data-parallel programs.

Exactly what a call to a data-parallel program involves depends on the implementation of the data-parallel notation, but in any case a call to a data-parallel program is to be semantically equivalent to a call to a sequential subprogram.

2.2 Alternative models

We believe that the model proposed in §2.1 combines simplicity with sufficient expressive power to handle a variety of interesting problems. However, other models of integrating task and data parallelism are possible and may be more appropriate for other classes of problems.

For example, another simple model is based on allowing task-parallel programs to serve as subprograms in a data-parallel program. In this model, one of the sequence of actions performed by the data-parallel program can be a call to a task-parallel program. Parameters to the called task-parallel program can include distributed data structures; calling a task-parallel program on a distributed data structure is equivalent to calling it concurrently once for each element of the distributed data structure, and each copy of the task-parallel program can consist of multiple processes.

2.3 Problem classes for our model

In this section, we describe several classes of problems for which our programming model is suitable.

2.3.1 Coupled simulations

A problem in this class consists of two or more coupled (interdependent) subproblems. Each subproblem can be expressed as a data-parallel program; solution of the whole problem requires communication among the programs solving the subproblems, which can be accomplished via a task-parallel top-level program.

Observe that if solutions to the different subproblems exist in the form of different data-parallel programs optimized for different architectures, it might be effective to solve the coupled problem by coupling these subproblem solutions using a task-parallel program executing on a heterogeneous system.

Example: climate simulation (heterogeneous domain decomposition)

One type of coupled simulation problem is the heterogeneous domain decomposition, in which the subproblems arise from dissimilar subdomains of the problem domain. An example is a climate simulation, as illustrated by figure 2.1. The simulation consists of an ocean simulation and an atmosphere simulation. Each simulation is a data-parallel program that performs a time-stepped simulation; at each time step, the two simulations exchange boundary data. This exchange of boundary data is performed by a task-parallel top layer.

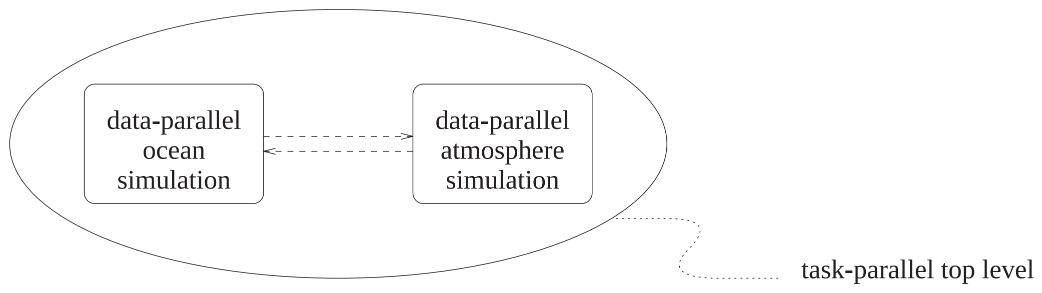
data-parallel ocean simulation
data-parallel atmosphere simulation
task-parallel top level

Figure 2.1: Climate simulation

Example: aeroelasticity simulation (multidisciplinary design and optimization)

Another type of coupled simulation problem is the multidisciplinary design and optimization problem [6], in which the subproblems arise from applying different analysis disciplines to different aspects of the problem. An example is an aeroelasticity simulation of a flexible wing in steady flight. Airflow over the wing imposes pressures that affect the shape of the wing; at the same time, changes in the wing's shape affect the aerodynamic pressures. Thus, the problem consists of two interdependent subproblems, one aerodynamic and one structural. As in the climate-simulation example, each subproblem can be solved by a data-parallel program, with the interaction between them performed by a task-parallel top-level program.

2.3.2 Pipelined computations

A problem in this class can be decomposed into subproblems that form the stages of a pipeline; the stages execute concurrently as tasks in a task-parallel program, and each stage of the pipeline is represented by a data-parallel program.

Example: iterated Fourier-transform computation

An example is a Fourier-transform computation on multiple sets of data, in which the computation has the following form for each set of data:

1. Perform a discrete Fourier transform (DFT) on a set of data.
2. Manipulate the result of the transform elementwise.
3. Perform an inverse DFT on the result of the manipulation.

Examples of such computations include signal-processing operations like convolution, correlation, and filtering, as discussed in [19], and polynomial multiplication, as described in §6.2.

Each of these steps can be performed by a data-parallel program; they can be linked to form a 3-stage pipeline, with the stages of the pipeline executing concurrently, as illustrated by figure 2.2. Each stage of the pipeline processes one set of data at a time. However, except during the initial "filling" of the pipeline, all stages of the pipeline can operate concurrently; while the first stage is processing the N-th set of data, the second stage is processing the $(N-1)$-th set of data and the third stage is processing the $(N-2)$-th set of data.

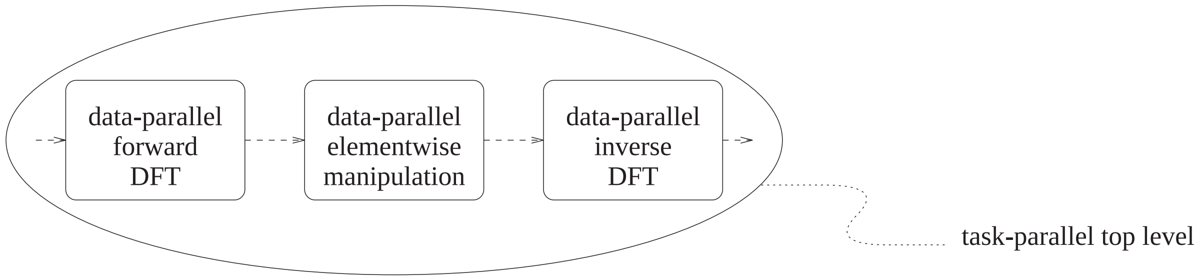
data-parallel forward DFT
data-parallel elementwise manipulation
data-parallel inverse DFT
task-parallel top level

Figure 2.2: Fourier-transform pipeline

2.3.3 Reactive computations

This problem class is a more general form of the pipeline class described in §2.3.2. A problem in this class can be readily expressed as a reactive program—i.e., a not-necessarily-regular graph of communicating processes operating asynchronously—in which each process is a data-parallel computation, and in which communication among neighboring processes is performed by a task-parallel top-level program. If the task-parallel notation allows dynamic process creation, the graph can change as the computation proceeds, adding and deleting processes as needed.

Example: discrete-event simulation

One type of problem that can be formulated as a reactive computation is discrete-event simulation; each process in the graph represents a component of the system being simulated, and communication among processes represents the interaction of the system's components. If the events being simulated for some or all of the components of the system are sufficiently computationally intensive, each component can be represented by a data-parallel program, with the interaction between components handled by a task-parallel top-level program.

An example of such a system is a nuclear reactor, as illustrated by figure 2.3. Components of the system include pumps, valves, and the reactor itself. Depending on the degree of realism desired, the behavior of each component may require a fairly complicated mathematical model best expressed by a data-parallel program. The data-parallel programs representing the individual components execute concurrently, with communication among them performed by a task-parallel top-level program.

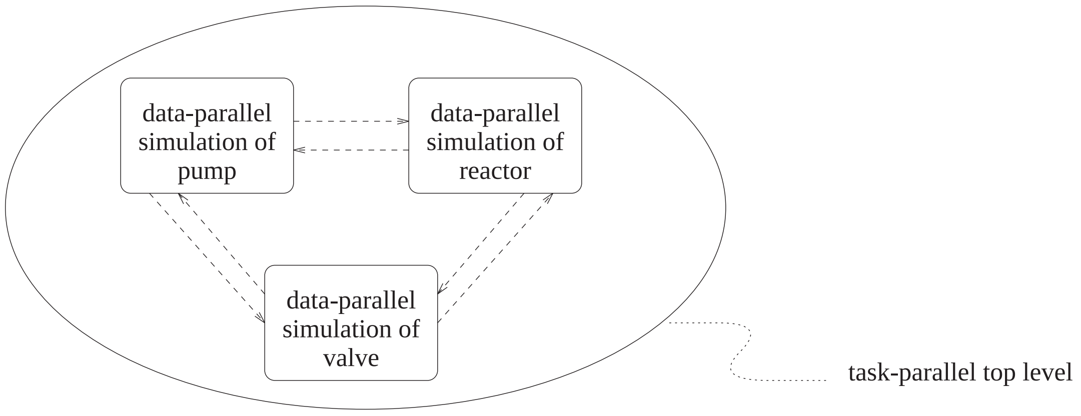
data-parallel simulation of pump
data-parallel simulation of reactor
data-parallel simulation of valve
task-parallel top level

Figure 2.3: Discrete-event simulation of reactor system

2.3.4 Inherently parallel computations

A problem in this class can be decomposed into several independent subproblems, each of which can be solved by a data-parallel program, with minimal or no communication among them.

Example: animation

One such problem is the generation of frames for a computer animation; two or more frames can be generated independently and concurrently, each by a different data-parallel program, as illustrated by figure 2.4.

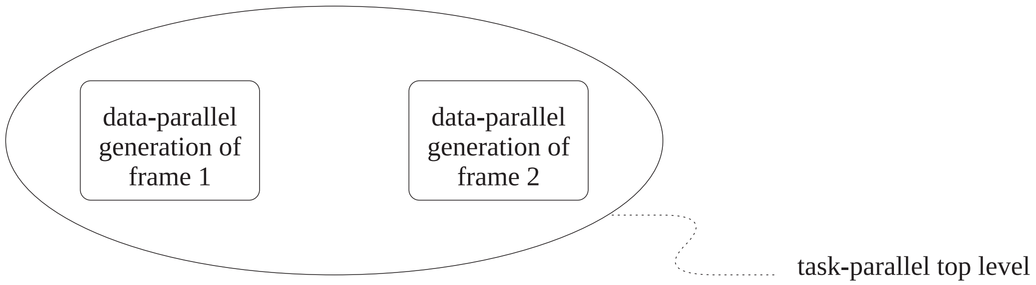
data-parallel generation of frame 1
data-parallel generation of frame 2
task-parallel top level

Figure 2.4: Generation of animation frames

Chapter 3

Implementation overview

This chapter presents an overview of a prototype implementation of the model of integrating task and data parallelism described in §2.1. We first describe general specifications for the prototype implementation. We then present a more detailed programming model and an overview of how it is supported in the prototype implementation.

3.1 General specifications

As defined in §2.1, our integration model has two key ideas: access to distributed data structures from task-parallel programs, and calls to data-parallel programs from task-parallel programs. Thus, an implementation consists of the following components:

- In terms of the task-parallel notation, implementation requires providing support for distributed data structures and for calls to data-parallel programs, including concurrent calls to different data-parallel programs. We provide this support through additions to a particular task-parallel notation.
- In terms of the data-parallel notation, implementation requires providing support for execution under control of a task-parallel program. We provide this support not by modifying a particular data-parallel notation, but by defining requirements that must be satisfied by a data-parallel program if it is to be called from the task-parallel notation. Our goal is to allow existing data-parallel programs to be called from the task-parallel notation with at most minor modifications.

In the remainder of this section, we describe the task-parallel notation chosen for the prototype implementation, introduce an initial restriction on the data-parallel programs, and present an overall design.

3.1.1 The task-parallel notation

The prototype implementation uses as its task-parallel notation PCN (Program Composition Notation) [5, 9]. PCN was chosen because it is simple, flexible, portable, and relatively easy to reason about.

It is important to note that, insofar as possible, design of the required extensions to the task-parallel notation is intended to be independent of the particular notation used and can be discussed without reference to the details of PCN syntax. It is hoped that many aspects of our implementation design are sufficiently general to be useful in developing implementations based on other task-parallel notations.

However, since the design involves extensions to a task-parallel notation, it is useful to begin by summarizing the features of the particular task-parallel notation chosen for implementation.

3.1.1.1 Program construction and concurrency

The basic idea of PCN, as its name suggests, is that programs are created by composing other programs. That is, a program is a composition of statements, each of which can be a subprogram call or a primitive operation such as assignment; the statements can be composed in sequence or in parallel. Execution of a parallel composition is equivalent to creating a number of concurrently-executing processes, one for each statement in the composition, and waiting for them to terminate.

With regard to the prototype implementation, the important point is that PCN allows both sequential and concurrent execution and supports dynamic process creation.

3.1.1.2 Communication and synchronization

Communication and synchronization among concurrently-executing processes are accomplished using single-assignment variables. Single-assignment variables can be written at most once; their initial value is a special "undefined" value, and a process that requires the value of an undefined variable is suspended until the variable has been defined (assigned a value).

With regard to the prototype implementation, the important point is that PCN provides a flexible mechanism for communicating and synchronizing among processes.

3.1.1.3 Variables

In addition to the single-assignment variables described in §3.1.1.2, PCN supports multiple-assignment variables. Multiple-assignment variables are like the variables of traditional imperative programming languages; their initial value is unspecified, and they can be assigned a new value arbitrarily many times.

With regard to the prototype implementation, the important point is that PCN supports multiple-assignment variables, since the elements of a distributed data structure are variables of this type.

3.1.1.4 Avoiding conflicting access to shared variables

Many errors in concurrent programs arise from conflicting access to shared variables. PCN is designed to prevent such errors by avoiding such conflicts. Conflicting accesses to a shared single-assignment variable are impossible by the nature of such variables. A single-assignment variable can change state at most once, from undefined to defined, after which its value does not change. Since programs that require the variable's value suspend until the variable has been defined, all programs that read the variable's value obtain the same value. Conflicting accesses to shared multiple-access variables are prevented by the following restriction: If two concurrently-executing processes share a multiple-assignment variable, neither is allowed to modify its value.

With regard to the prototype implementation, the important point is that PCN provides a simple way of avoiding conflicting access to shared variables, and thus provides a foundation for avoiding conflicting access in programs that combine PCN and a data-parallel notation, as discussed in §3.4.2.

3.1.1.5 Interface to sequential languages

An integral part of PCN is support for calls to programs written in traditional imperative languages, particularly C and Fortran. Single-assignment variables may be passed to the called programs only as input; multiple-assignment variables may be passed as either input or output. This support is based on an interface to the object code created by a C or Fortran compiler, so it readily extends to support for data-parallel notations based on C or Fortran.

3.1.1.6 Source-to-source transformations and higher-order functions

Associated with PCN is a notation for performing source-to-source transformations (Program Transformation Notation, or PTN [7]). PCN also provides some support for higher-order program calls; it allows making a call to a program whose name is supplied at runtime by the value of a character-string variable. While both of these features proved useful in the prototype implementation, neither is essential.

3.1.2 The data-parallel notation

As noted at the beginning of §3.1, the prototype implementation does not involve modifications to a particular data-parallel notation; rather, it defines notation-independent requirements for called data-parallel programs. A complete description of the requirements is deferred until §3.5. At this point, we impose a single initial requirement: We restrict attention to multiple-address-space SPMD implementations of data-parallel notations, as described in §1.2.5.

3.1.3 Distributed data structures

As described in §1.2.1, a distributed data structure is one that is partitioned and distributed among processes. In a multiple-address-space SPMD implementation, then, the data in a distributed data structure is partitioned into local sections, each containing one or more individual data elements, which are then distributed one per processor, as in figure 3.1. Each individual data element may be thought of as having both a global position (with respect to the whole data structure) and a local position (with respect to the local section in which it appears).

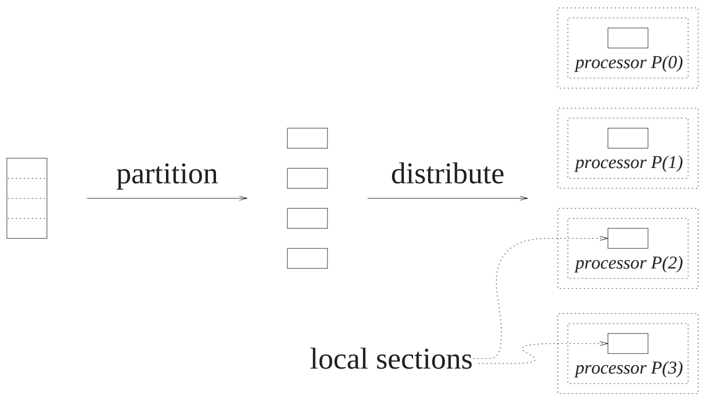

Figure 3.1: A distributed data structure

In order to support concurrent calls to different data-parallel programs, we extend this model to allow the distribution to be restricted to a subset of the available processors.

In addition, we require that programs in the task-parallel notation be able to access the distributed data structure as a global construct; i.e., we require that a single program be able to create a data structure distributed over many processors, and that a program be able to access any element of the data structure in terms of its global position.

3.1.4 Calls to data-parallel programs (distributed calls)

With a multiple-address-space SPMD implementation, a data-parallel program consists of a fixed set of processes, one per processor. Since the local sections of a distributed data structure are assigned to processors in the same way, there is a one-to-one correspondence between processes and local sections of a particular distributed data structure, and each process is said to *own* the local section on the processor on which it executes. All processes execute the same program, each on its own data; within the program, a distributed data structure is represented by the local section. A process accesses data elements outside its local section by communicating with the processes that own them. We refer to such a program as an *SPMD program.*

Given the requirement for support for concurrent calls to different data-parallel programs, then, a call to an SPMD program from a task-parallel program implies concurrent execution of the SPMD program on each of a subset of the available processors, with distributed-data parameters represented by local sections. We term such a call a *distributed call.*

3.1.4.1 Control flow

Figure 3.2 illustrates the flow of control when a task-parallel program calls a data-parallel program. When task-parallel program `TPA` makes a distributed call to data-parallel program `DPA` on a group of processors P, one copy of `DPA` is initiated on each processor in P, and caller `TPA` suspends execution while the copies of `DPA` execute. When all copies of `DPA` terminate, control returns to `TPA`. Each copy of `DPA` constitutes a process, created when `DPA` is called and destroyed when `DPA` terminates.

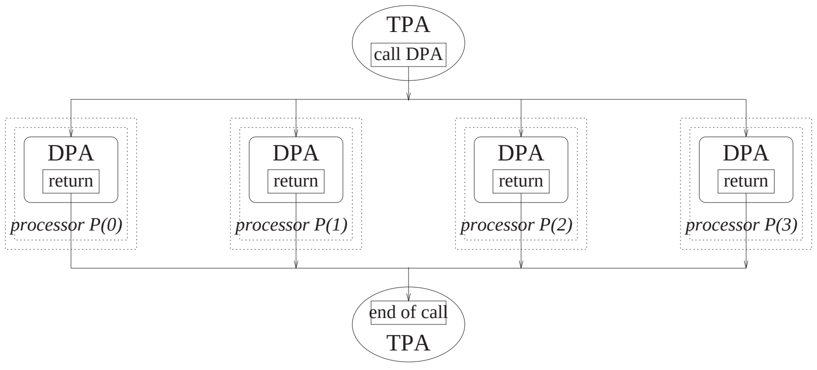

Figure 3.2: Control flow in a distributed call

3.1.4.2 Data flow

Figure 3.3 illustrates the flow of data when a task-parallel program calls a data-parallel program. When task-parallel program `TPA` makes a distributed call to data-parallel program `DPA` on a group of processors P with distributed data structure `DataA` as a parameter, one copy of `DPA` is initiated on each processor in P, with a parameter that references the local section of `DataA`. As indicated by the dotted lines, task-parallel program `TPA` has access to `DataA` as a global construct, while each copy of data-parallel program `DPA` has access to its local section of `DataA`. In addition, as indicated by the dashed line, the copies of data-parallel program `DPA` can communicate, as described in §3.1.4.

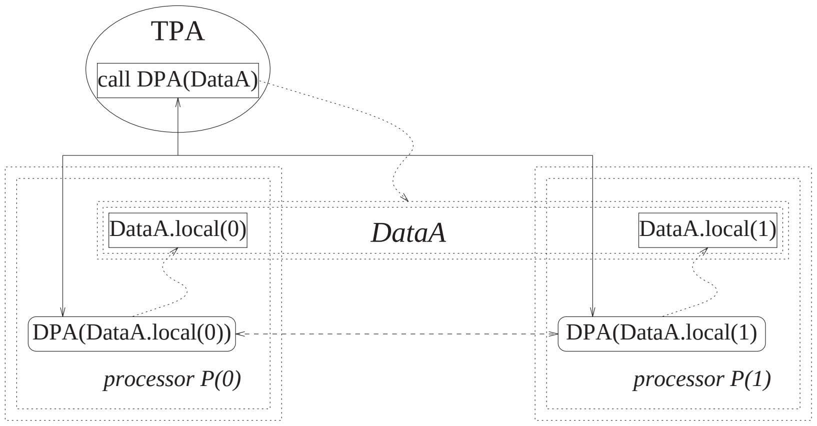

Figure 3.3: Data flow in a distributed call

3.1.4.3 Concurrent distributed calls

Figure 3.4 illustrates a more complex situation, in which two processes in the task-parallel program call data-parallel programs concurrently. Task-parallel program `TPA` calls data-parallel program `DPA`, passing distributed data structure `DataA`, while concurrently `TPB` calls `DPB` with `DataB`. Solid lines indicate program calls, dashed lines indicate communication, and dotted lines indicate data access. Observe that the two copies of `DPA` can communicate (as indicated by the dashed line in the figure), and that the two copies of `DPB` can communicate, but that no direct communication between `DPA` and `DPB` is allowed. Any transfer of data between `DataA` and `DataB` must be done through the task-parallel program.

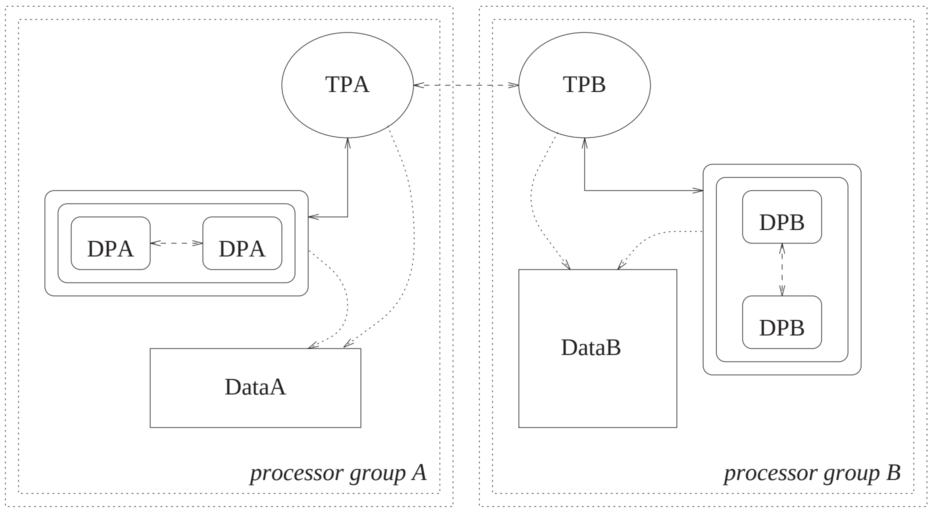

Figure 3.4: Concurrent distributed calls

3.2 Support for distributed data structures in the task-parallel notation

The prototype implementation supports only one kind of distributed data structure, the distributed array. This section describes in detail our model of distributed arrays and gives an overview of how they are supported in the task-parallel notation.

Although this model of distributed arrays is intended to be sufficiently general to be compatible with a variety of data-parallel notations, several aspects are based on the Fortran D [12] model and are specifically intended to support aspects of the Fortran D implementation.

3.2.1 The programming model

3.2.1.1 Introduction

The basic idea of a distributed array is this: An N-dimensional array is partitioned into one or more N-dimensional subarrays called *local sections*; these local sections are distributed among processors. There are a variety of methods for doing this; in the simplest method, called *block distribution*, each local section is a contiguous subarray of the whole array, and the processors are visualized as forming an N-dimensional array (referred to as a *processor grid*) as well.

Figure 3.5 illustrates a simple example. A 16 by 16 array is partitioned into eight 2 by 4 local sections and distributed among eight processors, which are conceptually arranged as a 4 by 2 array. One element is blacked in to illustrate the relationship among elements of the global array (on the left), elements of the partitioned array (in the middle), and elements of the local sections (on the right).

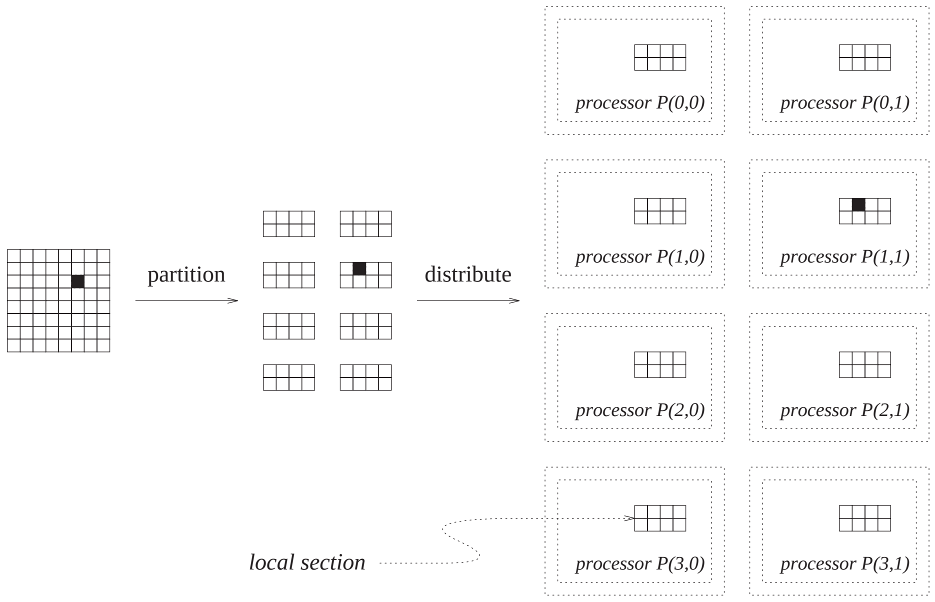

Figure 3.5: Partitioning and distributing an array

Observe that the dimensions of the local sections are related to the dimensions of the processor grid. In general, an N-dimensional array can be distributed among P processors using any N-dimensional process grid the product of whose dimensions is less than or equal to P. (In the example above, a 2 by 4 process grid would be acceptable, but a 3 by 3 process grid would not.) If the dimensions of the process grid are known, the dimensions of each local section can be readily computed; the i-th dimension of a local section is just the i-th dimension of the global array divided by the i-th dimension of the processor grid. (We assume for convenience that each dimension of the processor grid divides the corresponding array dimension.)

In the simplest scheme, local sections do not overlap, so each element of the original array corresponds to exactly one element of one local section. Thus, each N-tuple of global indices (indices into the original array) corresponds to a pair {*processor-reference-tuple, local-indices-tuple*}, where *processor-reference-tuple* is an N-tuple reference to a processor, and *local-indices-tuple* is an N-tuple of local indices (indices into the local section). Conversely, each {*processor-reference-tuple, local-indices-tuple*} pair corresponds to exactly one element of the original array. For example, in figure 3.5, the blacked-in array element has global indices (2,5); in the distributed array, this element is represented by local indices (0,1) on processor (1,1). We assume that indices start with 0.

Further, we assume that each processor is identified by a single number, referred to as a processor number, and we assume that a multidimensional array is represented by a contiguous block of storage, equivalent to a 1-dimensional array. Thus, each N-tuple reference to a processor ultimately corresponds to a single processor number, and each N-tuple of local indices ultimately corresponds to a single index. (This mapping from multiple dimensions to a single dimension can be either row-major or column-major, as described below in §3.2.1.3.) To continue the example of the preceding paragraph, if we assume a row-major ordering, the blacked-in array element is represented by element 1 of the local section on processor 3.

3.2.1.2 Decomposing the array

Only block decompositions (as described in §3.2.1.1) are allowed, but the user can control the dimensions of the process grid and hence of the local sections.

By default, an N-dimensional array is distributed among P processors using a "square" processor grid—an N-dimensional array each of whose dimensions is $P^{\frac{1}{N}}$. For example, a 2-dimensional array ($N = 2$) is by default distributed among 16 processors ($P = 16$) using a 4 by 4 processor grid ($16^{\frac{1}{2}} = 4$).

The user can specify some or all of the dimensions of the process grid, however, as described below. Any dimensions that are not specified are computed in a way consistent with the default: If there are M specified dimensions, and their product is Q, each unspecified dimension is set to $(\frac{P}{Q})^{\frac{1}{N-M}}$. For example, if a 3-dimensional array ($N = 3$) is to be distributed among 32 processors ($P = 32$) with the second dimension of the processor grid specified as 2 ($M = 1$, $Q = 2$), the resulting processor grid has dimensions 4 by 2 by 4 ($(\frac{32}{2})^{\frac{1}{3-1}} = 4$).

The syntax for controlling the dimensions of the processor grid is based on Fortran D; for each dimension of the array, the user can specify one of the following decompositions:

- `block`, to allow the corresponding dimension of the processor grid to assume the default value.
- `block(`N`)`, to specify the corresponding dimension of the processor grid as N.
- `*`, to specify the corresponding dimension of the processor grid as 1, i.e., to specify that the array is not to be decomposed along this dimension.

Figure 3.6 illustrates the effect of applying the following different decomposition specifications to a 400-by-200 array and 16 processors:

- The decomposition `(block, block)` implies a square 4-by-4 processor grid and local sections of size 100 by 50.

- The decomposition `(block(2), block(8))` implies a 2-by-8 processor grid and local sections of size 200 by 25. `(block(2), block)` is equivalent, as is `(block, block (8))`.
- The decomposition `(block, *)` implies a 16-by-1 processor grid and local sections of size 25 by 200. (I.e., this is a decomposition by row only.)

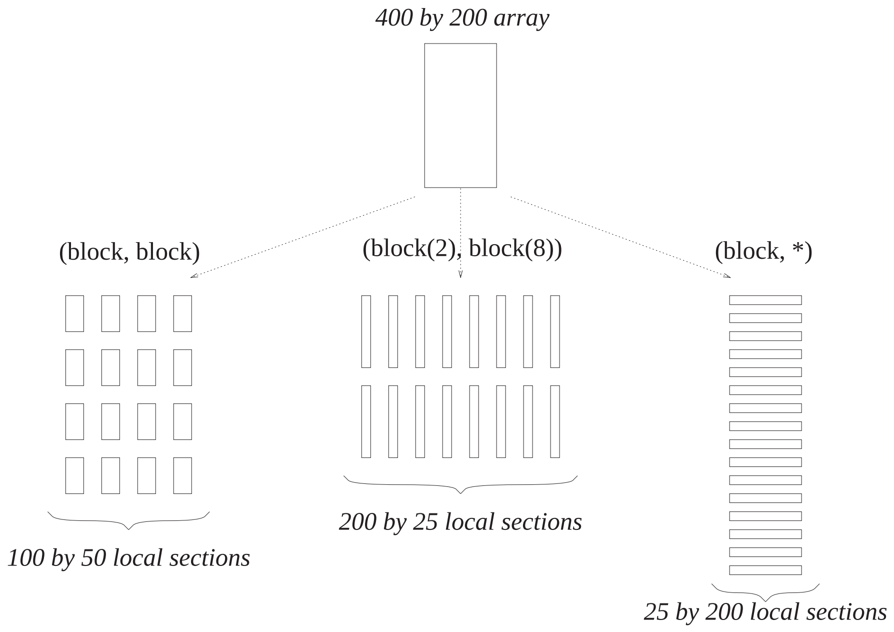

Figure 3.6: Decomposing an array

3.2.1.3 Local sections

Each local section is a "flat" piece of contiguous storage; its size is computed as the product of the local section dimensions, which are in turn computed as described in §3.2.1.1. The user specifies whether indexing of a multidimensional array, and hence of its local sections, is row-major (C-style) or column-major (Fortran-style). This allows support for calls to data-parallel programs using either type of indexing.

Some data-parallel notations add to each local section borders to be used internally by the data-parallel program. For example, Fortran D adds borders called *overlap areas*, which it uses as communication buffers. Figure 3.7 shows a local section with borders. The local section has dimensions 4 by 2 and is surrounded by borders consisting of 1 additional element at the beginning and end of each row and 2 additional elements at the beginning and end of each column.

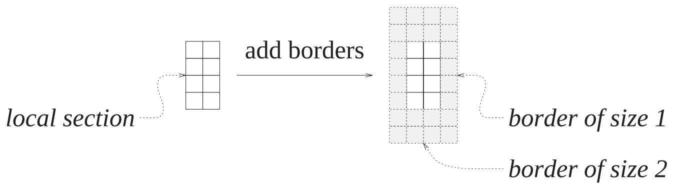

Figure 3.7: Local section with borders

If a notation makes use of such borders, we assume that local sections passed to it as parameters must include borders. Thus, when an array to be passed to such a notation is created, extra space must be allocated for borders. The simplest way to do this is for the task-parallel program creating the array to explicitly provide the sizes of the required borders. Alternatively, the user can indicate that the size of the borders is to be controlled by the data-parallel program at runtime. With this approach, the user specifies as part of the array-creation request that the array's borders are to be consistent with its being passed as parameter number *Parm_number* to data-parallel program *Program_name*. The code for *Program_name* must then include a procedure *Program_name_* (note the trailing underscore) that, given *Parm_num* as input, supplies the proper border sizes as output. This approach allows the data-parallel notation to supply different border sizes for each parameter of each program.

Observe that once border sizes for an array have been established, they cannot be changed without reallocating and copying all local sections of the array. Although this is an expensive operation, it might be unavoidable if a single distributed array is to be used with two different data-parallel programs. Thus, support is also provided for "verifying" a distributed array's border sizes—i.e., comparing them to a set of expected border sizes and reallocating and copying the array if a mismatch is found. The expected borders are specified in the same way as the original borders—either explicitly by the task-parallel program or under control of the data-parallel program.

Finally, observe that this support for local section borders is limited to that required for compatibility with data-parallel notations that require it. Locations in the border areas can be accessed only by the data-parallel program; programs in the task-parallel notation have access only to the interior (non-border) data in the local sections.

3.2.1.4 Distributing the array among processors

The user specifies the processors among which the array is to be distributed by providing a 1-dimensional array of processor numbers. (The array is 1-dimensional so that the task-parallel notation need not directly support multidimensional arrays.) These processors are conceptually arranged into a processor grid as described in §3.2.1.1, with the mapping from N-dimensional processor grid into 1-dimensional array either row-major or column-major, depending on the type of indexing the user selects for the array. One local section of the array is then assigned to each processor.

Figure 3.8 illustrates distributing a 4-by-4 array `X` among 4 processors numbered 0, 2, 4, and 6. Observe that when indexing is row-major, `X(0,1)` is assigned to processor 2, but when indexing is column-major, the same element is assigned to processor 4.

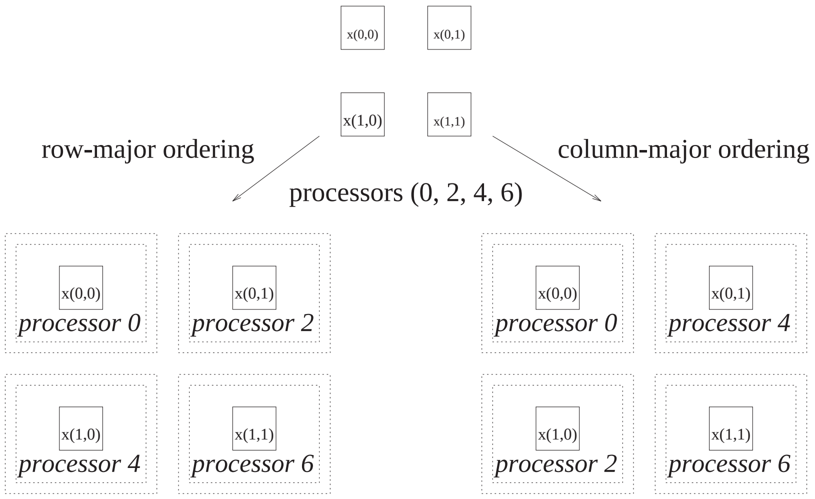

Figure 3.8: Distributing an array

3.2.1.5 Operations on distributed arrays

The following operations on distributed arrays are supported in the task-parallel notation:

- Creating a distributed array. A single array-creation operation creates the entire array; it is not necessary for the user program to explicitly perform the operation separately on each processor.
- Deleting (freeing) a distributed array. As with array creation, a single array-deletion operation deletes the entire array.
- Reading an individual element, using global indices.
- Writing an individual element, using global indices.
- Obtaining a reference to the local section of a distributed array in a form suitable for passing to a data-parallel program—i.e., a direct reference to the storage for the local section.
- Obtaining additional information about a distributed array—for example, its dimensions.

- Verifying that the local sections of a distributed array have the expected borders and reallocating and copying them if not, as described in §3.2.1.3. As with array creation and deletion, a single array-verification operation verifies borders on the entire array.

With the exception of obtaining a reference to a local section, these operations are based on viewing a distributed array as a global construct, as discussed in §3.1.3. Array creation can be performed on any processor; with the exception of obtaining a reference to a local section, any of the remaining operations can be performed on any processor that includes a local section of the array or on the processor on which the array-creation request was made, with identical results. (For example, a request to read the first element of a distributed array returns the same value no matter where it is executed.) The exception, obtaining a reference to a local section, clearly requires a local rather than a global view of the array.

3.2.2 Implementation

3.2.2.1 Syntax

Full syntactic support for distributed arrays would allow the user to create and manipulate distributed arrays like non-distributed arrays. A distributed array would be created when the procedure that declares it begins and destroyed when that procedure ends, and single elements would be referenced, for reading or writing, in the same way as single elements of non-distributed arrays.

Such full syntactic support, however, is beyond the scope of a prototype implementation. A simpler approach is to implement the required operations using a library of procedures, one for each of the operations described in §3.2.1.5. Detailed specifications for such procedures are given in §4.2.

With this approach, each distributed array is identified by a globally-unique *array ID*; this ID is created by the procedure that creates the array and is used for all subsequent references to the array. This array ID is analogous to a file pointer in C: It is a unique identifier by which other programs can reference the object (file or array). Use of array IDs to reference arrays is described further, with an example, in §4.1.3.

3.2.2.2 Support

Since array creation and manipulation is performed by library procedures, no compile-time support is needed for the operations described in §3.2.1.5. However, in the prototype implementation, support for allowing the called data-parallel program to control border sizes, as described in §3.2.1.3, is provided by a source-to-source transformation.

Runtime support for distributed arrays is provided by an entity called the *array manager*. The array manager consists of one array-manager process on each processor; all requests by procedures in the task-parallel notation to create or manipulate distributed arrays are handled by the local array-manager process, which communicates with other array-manager processes as needed to fulfill the requests. For example, if the request is for creating an array, the local array-manager process must communicate with the array-manager processes on all processors over which the array is to be distributed, while if the request is for reading an element, the local array-manager process must communicate with the array-manager process on the processor containing the requested element.

Figure 3.9 illustrates this design. There is one array-manager process on each processor, and there is a communication path between every pair of array-manager processes. The array manager maintains references to storage for local sections; it also stores dimensions and other information about each distributed array. User processes in the task-parallel notation perform operations on distributed arrays only by communicating with the local array-manager process. Observe, however, that if a process in the task-parallel notation passes a local section to a data-parallel program, the data-parallel program may access the local section directly.

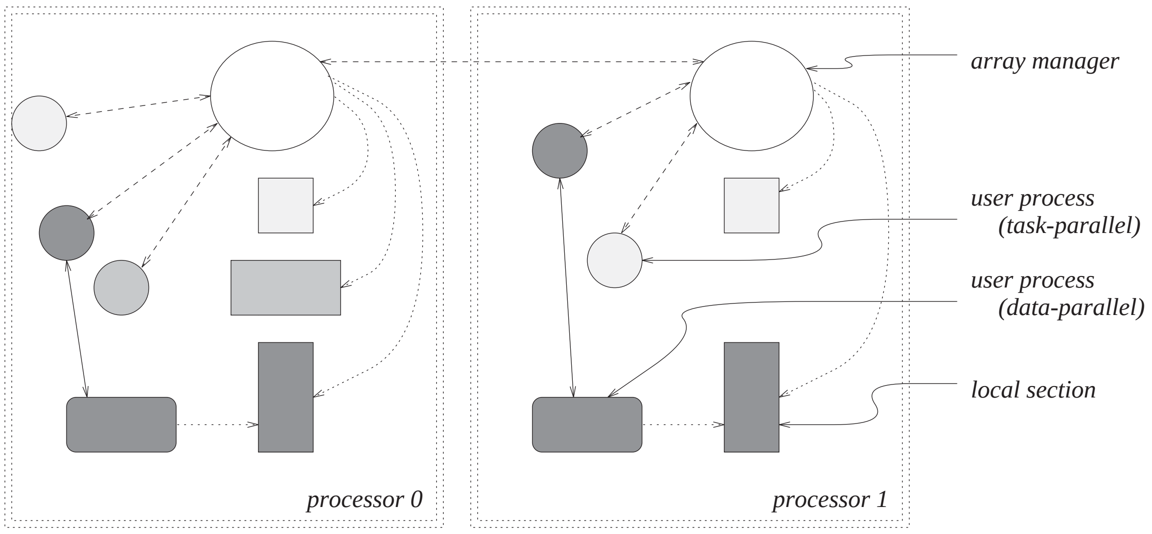

Figure 3.9: Runtime support for distributed arrays

3.3 Support for distributed calls in the task-parallel notation

This section describes in detail our model of distributed calls and gives an overview of how they are supported in the task-parallel notation.

3.3.1 The programming model

The programming model for distributed calls is based on the description of SPMD data-parallel programs in §3.1.4. Executing a distributed call to an SPMD data-parallel program is equivalent to calling the SPMD program concurrently on each of a group of processors (not necessarily all the available processors) and waiting for all copies to complete execution. Parameters to the SPMD program can include local sections of distributed arrays, and the concurrently-executing copies of the SPMD program can communicate with each other just as they normally would (i.e., if not executing under control of a task-parallel program). Execution of the distributed call terminates when all copies of the called program have terminated.

3.3.1.1 Parameters for the distributed call

A task-parallel program making a distributed call specifies the following:

- the data-parallel (SPMD) program to be called.
- the processors on which the data-parallel program is to be executed. As when creating a distributed array (§3.2.1.4), these processors are specified by means of a 1-dimensional array of processor numbers.
- the parameters to be passed to the data-parallel program.

3.3.1.2 Parameters for the called data-parallel program

To provide flexibility, the programming model defines several different kinds of parameters that can be passed from the task-parallel-notation caller to the called data-parallel program in a distributed call. A parameter to be passed from the task-parallel caller to the called data-parallel program can be any of the following:

- (input) A *global constant*—either a true constant or a variable whose value does not change during execution of the called program. Each copy of the called program receives the same value; it can be used as input only.
- (input/output) A *local section* of a distributed array. The calling program specifies the array as a global construct, i.e., using its array ID. Each copy of the called program, however, gets its own distinct local section, which can be used as input or output or both.
- (input) An integer *index*. Each copy of the called program receives a unique value for this parameter; the value is an index into the array of processors over which the call is distributed, and the variable can be used as input only.

- (output) An integer *status variable* to be used to pass information back to the caller. Each copy of the called program has its own local status variable, which is to be used as output; at termination of the called program, the values of these local status variables are merged (using any binary associative operator—by default `max`, but the user may provide a different operator) into a single value that is then returned to the caller. A distributed call can have at most one status variable.
- (output) A *reduction variable*. Such a variable is handled like the status variable—i.e., each copy of the called program sets a local value, and at termination the local values are merged into a single value that is returned to the caller. Unlike the status variable, however, a reduction variable can be of any type, including arrays; also, a distributed call can have any number of reduction variables. This option is provided to allow more flexibility in returning global information to the caller.

3.3.2 Implementation

3.3.2.1 Syntax

The task-parallel program makes a distributed call by calling a library procedure `distributed_call`, whose parameters allow the user to specify:

- the data-parallel (SPMD) program to be called.
- the processors on which the data-parallel program is to be executed, specified via a 1-dimensional array of processor numbers.
- the parameters to be passed to the data-parallel program. The different types of parameters described above in §3.3.1.2 are distinguished syntactically. Details of syntax for all the types of parameters are given, with examples of their use, in §4.3.1; for example, the syntax {`"local", A`} specifies the local section of distributed array `A`.

3.3.2.2 Support

Execution of a distributed call on processors *Procs* consists of the following sequence of actions:

1. Creating a process for the called program on each processor in *Procs*.
2. Passing parameters from the caller to each copy of the called program. Some of the types of parameters require special handling—for example, a local-section parameter requires that a reference to the local section of the specified distributed array be obtained.
3. Transferring control to the called program and waiting for all copies to complete.
4. Returning information to the caller as specified by the call's parameters—i.e., merging local values for any status or reduction variables.

The distributed call is considered to have terminated when all of these actions have been completed.

As illustrated in figure 3.10, one way to accomplish these actions is via a "wrapper" program. One copy of the wrapper program is executed on each processor in *Procs*; each copy does the following:

1. Obtains references to local sections of distributed arrays, using their array IDs.
2. Declares local variables for status and reduction parameters.
3. Calls the data-parallel (SPMD) program, passing it the proper parameters, including the local sections and local variables from the preceding steps.
4. Communicates with other copies of the wrapper program to perform the required merges for status and reduction variables.

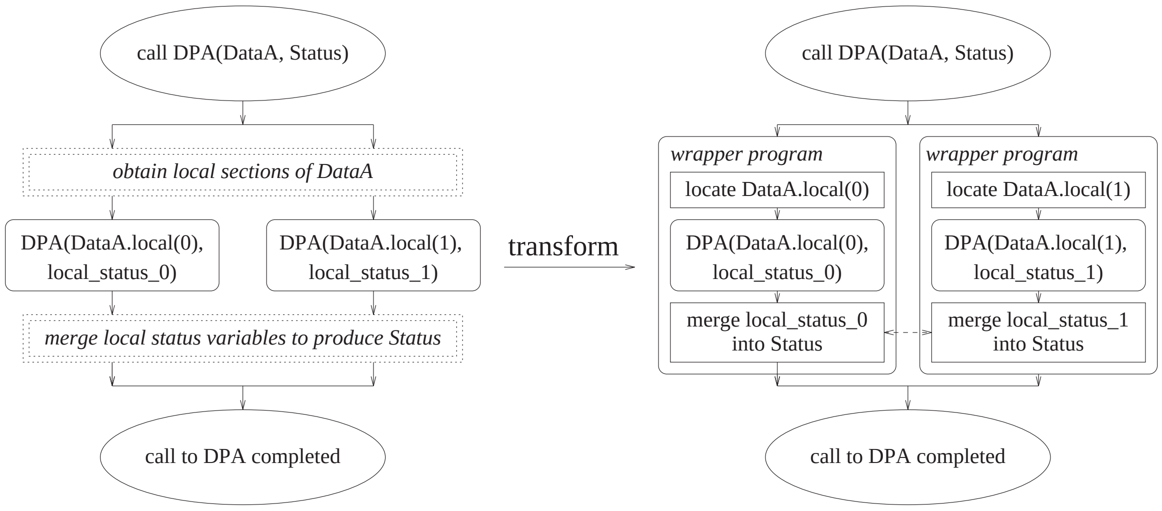
call DPA(DataA, Status)
obtain local sections of DataA
DPA(DataA.local(0), local_status_0)
DPA(DataA.local(1), local_status_1)
merge local status variables to produce Status
call to DPA completed
transform
call DPA(DataA, Status)
wrapper program
locate DataA.local(0)
DPA(DataA.local(0), local_status_0)
merge local_status_0 into Status
wrapper program
locate DataA.local(1)
DPA(DataA.local(1), local_status_1)
merge local_status_1 into Status
call to DPA completed

Figure 3.10: Support for distributed calls—the wrapper program

Since PCN provides a mechanism for source-to-source transformations, the prototype implementation implements distributed calls primarily via compile-time transformations. Each source call to the procedure `distributed_call` is used to generate a wrapper program with the behavior described above and then is transformed into code to execute this wrapper program concurrently on all processors in *Procs*.

3.4 Synchronization issues

Since it is possible to have a situation in which both task-parallel and called data-parallel (SPMD) programs are concurrently executing, it is necessary to consider whether this introduces synchronization or communication problems in addition to those associated with either program alone. What additional conflicts are possible and how they can be prevented depends to some extent on the underlying communication mechanisms used by the task-parallel implementation and the data-parallel implementation. Since communication mechanisms can vary widely, we do not attempt a completely general discussion. Rather, we focus on the situation where both implementations communicate using point-to-point message-passing. In such an environment, there are two sources of possible problems: message conflicts and conflicting access to shared variables.

3.4.1 Message conflicts

First, we consider whether the task-parallel program could intercept a message intended for the called data-parallel program, or vice versa. Any such conflict can be avoided by requiring that both the task-parallel notation and the called data-parallel program use communication primitives based on typed messages and selective receives, and ensuring that the set of types used by the task-parallel notation and the set of types used by the called data-parallel program are disjoint.

3.4.2 Conflicting access to shared variables

Next, we consider whether the task-parallel program and the called data-parallel program could make conflicting accesses to a shared variable. Whether this happens depends in part on the restrictions placed on simultaneous accesses to shared variables in the task-parallel notation; we consider whether it is possible when the task-parallel notation is PCN.

As mentioned in §3.1.1.4, PCN avoids conflicting accesses to shared variables by requiring that if concurrently-executing processes share a variable, either it is a single-assignment variable, or none of the processes modifies its value. In the former case, the variable can be written by at most one process, and every process that reads it obtains (after possible suspension) the same value. In the latter case, no conflicts can occur since none of the processes changes the variable's value.

Conflicts in a program based on the prototype implementation could take one of the following forms: conflicts between PCN processes, conflicts between PCN and data-parallel processes, and conflicts between data-parallel processes. If the called data-parallel programs are correct, then there are no conflicts between the data-parallel processes that comprise a single distributed call. By virtue of the restriction described above, conflicts between PCN processes do not occur; neither are there conflicts between subprocesses of concurrently-executing PCN processes. Since the data-parallel processes comprising a distributed call are implemented as subprocesses of the calling PCN process, this restriction also rules out conflicts between data-parallel processes in different distributed calls and between a data-parallel process and a PCN process other than its caller. Finally, conflicts between a data-parallel process and its caller do not occur because the caller and the called program do not execute concurrently. Thus, the program as a whole is free of conflicting accesses to shared variables.

3.5 Requirements for called data-parallel programs

This section defines the requirements that must be met by a data-parallel program if it is to be called from the task-parallel notation.

SPMD implementation

As mentioned in §3.1.2. the called data-parallel program must be based on a multiple-address-space SPMD implementation, as described in §1.2.5.

Relocatability

A called data-parallel program must be "relocatable"; i.e., it must be capable of being executed on any subset of the available processors. This restriction has the following important consequences:

- If the program makes use of processor numbers for communicating between its concurrently-executing copies, it must obtain them from the array of processor numbers used to specify the processors on which the distributed call is being performed. This array can be passed as a parameter to the data-parallel program.
- The program must not make use of global-communication routines, unless it is possible to restrict those routines to a particular subset of the available processors.

Compatibility of parameters

The formal parameters of a called data-parallel program must be compatible with the model described in this chapter, particularly in §3.2.1.3 and §3.3.1.2. For example, a local section is simply a contiguous block of storage, not a more complex representation of a multidimensional array like the arrays of arrays sometimes found in C-based programs.

Compatibility of communication mechanisms

If the concurrently-executing copies of a called data-parallel program communicate, they must do so in a way that does not interfere with the way processes in the task-parallel notation communicate, as discussed in §3.4.1.

Language compatibility

A called data-parallel program must be written in a notation to which the task-parallel notation supports an interface. However, observe that if the task-parallel notation supports an interface to a particular sequential notation, this interface should also support calls to data-parallel notations based on the sequential notation. In the case of PCN, the interface to C and Fortran also supports calls to data-parallel notations that are implemented using C or Fortran and a library of communication routines.

Chapter 4

Implementation details: library procedure specifications

As described in §3.2.2.1 and §3.3.2.1, syntactic support for creating and manipulating distributed arrays and for making distributed calls is provided in the prototype implementation via a set of library procedures. This chapter presents complete specifications for the library procedures. Although the procedures are written in and callable only from PCN, the specifications are as far as possible language-independent.

Additional information about using the prototype implementation appears in §B, which explains how to compile, link, and execute programs; and in §C, which describes additional library procedures that, while useful, are not central to the implementation. For readers completely unfamiliar with PCN, §A provides an overview of PCN syntax and terminology; complete information about PCN may be found in [5] and [9].

4.1 General information about the library procedures

4.1.1 Documentation conventions

Documentation for each procedure discussed in this chapter consists of the following:

- A short description of its function.
- A procedure heading giving its full name in the form *module_name*:*program_name* and listing its formal parameters. Parameters are definitional (single-assignment) unless otherwise stated. Annotation for each parameter indicates its expected type and whether it is to be used as input or output. Parameters annotated as `in` are used as input only and must have values assigned to them elsewhere; parameters annotated as `out` are used as output only and will be assigned values by the procedure.

- A description of the procedure's precondition. The precondition describes what must be true before the procedure is called, including restrictions on the actual parameters beyond those imposed by the parameter annotations. The precondition may also include comments on the meaning and usage of the parameters. A precondition of `TRUE` indicates that there are no requirements aside from those given by the parameter annotations.
- A description of the procedure's postcondition. The postcondition describes what is guaranteed to be true after execution of the procedure.

Together, the procedure heading, parameter annotations, precondition, and postcondition constitute a specification for the procedure.

For example, consider the procedure described by the following specification:

```
double_it(
        /*in int*/              In,
        /*out int*/             Out
        )
```

Precondition:

- `TRUE`.

Postcondition:

- `Out` = 2 * `In`

This specification indicates that procedure `double_it` has one input parameter (`In`) and one output parameter (`Out`), both definitional integers; at exit, the value of `Out` is twice the value of `In`.

Observe that for a definitional variable `X` it is possible to speak of "the value of `X`" without ambiguity, since a value can be assigned to `X` only once. Thus, in preconditions and postconditions, for definitional variable `X` we use "`X`" to denote the value of `X`.

4.1.2 Status information

All of the procedures described in this chapter have an integer output parameter `Status` whose value indicates the success or failure of the operation being performed. The possible values of this parameter, with their meanings and the symbolic names used to reference them in this chapter, are as follows:

symbolic name	*value*	*meaning*
`STATUS_OK`	0	no errors
`STATUS_INVALID`	1	invalid parameter
`STATUS_NOT_FOUND`	2	array not found
`STATUS_ERROR`	99	system error

The `Status` parameter is a definitional variable that becomes defined only after the operation has been completed, so callers can use it for synchronization purposes if necessary.

4.1.3 Referencing arrays—array IDs

As described in §3.2.2.1, a distributed array is referenced using a globally-unique array ID. This array ID is a 2-tuple of integers (the processor number on which the original array-creation request was made, plus an integer that distinguishes this array from others created on the same processor). It is set by the array-creation procedure and used for all subsequent operations on the array. For example, the following program block creates and then frees a distributed array with array ID `A1`:

```
{;
        am_user:create_array(/*out*/ A1,
                /*in*/ "double", Dims, Procs, Distrib, Borders, "row",
                /*out*/ Stat1) ,
        am_user:free_array(/*in*/ A1, /*out*/ Stat2)
}
```

4.2 Distributed-array procedures

The procedures described in this section provide support for the programming model of distributed arrays defined in §3.2.1.

4.2.1 Creating an array

The following procedure creates a distributed array.

```
am_user:create_array(
        /*out 2-tuple of int*/          Array_ID,
        /*in string*/                   Type,
        /*in array of int*/             Dimensions,
        /*in array of int*/             Processors,
        /*in tuple*/                    Distrib_info,
        /*in*/                          Border_info,
        /*in string*/                   Indexing_type,
        /*out int*/                     Status
        )
```

Precondition:

- `Type` is `"int"` or `"double"`.

- `Dimensions` are the array dimensions; length(`Dimensions`) is the number of dimensions.
- `Processors` are the processors over which the array is to be distributed. The array is distributed as described in §3.2.1.4.
- `Distrib_info` describes how the array is to be decomposed. Length(`Distrib_info`) = length(`Dimensions`), and each `Distrib_info[i]` is one of the following:
  - `"block"`.
  - {`"block"`, N}, where N is an integer or an integer-valued variable.
  - `"*"`.

  where these decomposition options are as described in §3.2.1.2.
- `Border_info` defines borders for each local section of the array, as described in §3.2.1.3. (Recall that this feature is included for compatibility with some data-parallel notations, such as Fortran D.) Its value is one of the following:
  - `[]`, to indicate that local sections do not include borders.
  - An array of integers, to directly specify the sizes of the borders. Length(`Border_info`) is twice length(`Dimensions`), and elements $2i$ and $2i+1$ specify the border on either side of dimension i. For example, for a 2-dimensional array, a value of (2, 2, 1, 1) for `Border_info` indicates that each local section has a border consisting of two extra rows above, two extra rows below, and one extra column on either side of the local section, as illustrated in §3.2.1.3.
  - {`"foreign_borders"`, *Program*, *Parm_num*}, where *Program* is a string and *Parm_num* is an integer, to allow a foreign-code (C or Fortran) program to specify the sizes of the borders at runtime, as described in §3.2.1.3. This option is designed to be used when the array being created is to be passed to program *Program* as parameter *Parm_num*. The foreign-code module containing *Program* must contain a routine *Program_* (note the trailing underscore in the name) of the following form:

    ```
    Program_(
            /*in int*/                     Parm_num,
            /*out mutable array of int*/   temp_borders
            )
            ...
    ```

    where length(`temp_borders`) is twice length(`Dimensions`), and, as above, elements $2i$ and $2i+1$ of `temp_borders` specify the border on either side of dimension i.

    For example, suppose a 1-dimensional array is being created and is to be passed to Fortran program `fpgm` as its first parameter. The call to `create_array` would specify `Border_info` as follows:

    ```
    {"foreign_borders", "fpgm_", 1}
    ```

(in calling Fortran from PCN, an underscore is always appended to the name of the program to be called) and the source for `fpgm` would include a routine like the following:

```
subroutine fpgm_(iarg, isizes)
integer iarg
integer isizes(*)
if (iarg .eq. 1) then
        isizes(1) = 2
        isizes(2) = 2
else
        ...
endif
return
end
```

– {`"borders"`, *Module*, *Program*, *Parm_num*}, where *Module* and *Program* are strings and *Parm_num* is an integer, to allow a PCN program to specify the sizes of the borders at runtime. Users should not need to make use of this option directly, but it is required to support the `foreign_borders` option described above, as explained in §5.1.7, and so is included here for completeness. There must be a PCN program of the following form:

```
Module:Program(
        /*in int*/                                Parm_num,
        /*in int*/                                N_borders,
        /*out array of int of length N_borders*/
                                                  Borders
        )
        ...
```

where the elements of `Borders` are as described for `temp_borders` above.

- `Indexing_type` is one of the following:
  - `"row"` or `"C"`, to indicate row-major indexing of multidimensional arrays.
  - `"column"` or `"Fortran"`, to indicate column-major indexing of multidimensional arrays.

  This parameter determines the indexing of both the array and the processor grid, as described in §3.2.1.4.

Postcondition:

- `Status` reflects the success of the operation; §4.1.2 lists the possible values.
- If `Status` = `STATUS_OK`, `Array_ID` is a globally-unique identifier that references an array with the parameters given by the input parameters.

4.2.2 Deleting a distributed array

The following procedure deletes a distributed array and frees the associated storage.

```
am_user:free_array(
        /*in 2-tuple of int*/           Array_ID,
        /*out int*/                     Status
        )
```

Precondition:

- TRUE.

Postcondition:

- Status reflects the success of the operation; §4.1.2 lists the possible values.
- If Status = STATUS_OK, the array has been deleted; i.e., subsequent references to distributed array Array_ID will fail, and its associated storage (for local sections) has been freed.

4.2.3 Reading an element

The following procedure reads a single element of a distributed array, given its global indices.

```
am_user:read_element(
        /*in 2-tuple of int*/           Array_ID,
        /*in tuple of int*/             Indices,
        /*out*/                         Element,
        /*out int*/                     Status
        )
```

Precondition:

- Length(Indices) is the number of dimensions of the array referenced by Array_ID.
- Each element of Indices is within range; i.e., each index is non-negative and less than the corresponding dimension.

Postcondition:

- Status reflects the success of the operation; §4.1.2 lists the possible values.
- If Status = STATUS_OK, Element is the element of the array corresponding to Indices, with type double or int, depending on the type of the array; Element is [] otherwise.

4.2.4 Writing an element

The following procedure writes an element of a distributed array, given its global indices.

```
am_user:write_element(
        /*in 2-tuple of int*/           Array_ID,
        /*in tuple of int*/             Indices,
        /*in*/                          Element,
        /*out int*/                     Status
        )
```

Precondition:

- Length(`Indices`) is the number of dimensions of the array referenced by `Array_ID`.
- Each element of `Indices` is within range; i.e., each index is non-negative and less than the corresponding dimension.
- `Element` is numeric.

Postcondition:

- `Status` reflects the success of the operation; §4.1.2 lists the possible values.
- If `Status` = `STATUS_OK`, `Element` has been written to the element of the array corresponding to `Indices`.

4.2.5 Obtaining a local section

The following procedure obtains the local section of a distributed array—i.e., the local section of the array on the processor on which the procedure executes. Users should never need to call this procedure directly; its most common use is during a distributed call, where it is automatically invoked by the distributed-call implementation, as described briefly in §3.3.2.2 and in more detail in §5.2. Its specification is given here, however, for the sake of completeness.

```
am_user:find_local(
        /*in 2-tuple of int*/           Array_ID,
        /*out array*/                   Local_section,
        /*out int*/                     Status
        )
```

Precondition:

- `TRUE`.

Postcondition:

- `Status` reflects the success of the operation; §4.1.2 lists the possible values.
- If `Status` = `STATUS_OK`, `Local_section` is the local section of the array referenced by `Array_ID`; `Local_section` is `[]` otherwise.

Because of the way the implementation handles local sections (described in detail in §5.1.5), the local section is returned in the form of a definitional array, but it may be passed to another program as a mutable array, as is done automatically during execution of a distributed call. The rationale for this unorthodox approach is explained in some detail in §5.1.5; briefly, it was developed as a way to circumvent certain limitations of PCN.

4.2.6 Obtaining information about a distributed array

The following procedure obtains information about a distributed array—for example, its dimensions.

```
am_user:find_info(
        /*in 2-tuple of int*/           Array_ID,
        /*in string*/                   Which,
        /*out*/                         Out,
        /*out int*/                     Status
        )
```

Precondition:

- `Which` is one of the following:
  - `"type"`.
  - `"dimensions"`.
  - `"processors"`.
  - `"grid_dimensions"`.
  - `"local_dimensions"`.
  - `"borders"`.
  - `"local_dimensions_plus"`.
  - `"indexing_type"`.
  - `"grid_indexing_type"`.

Postcondition:

- `Status` reflects the success of the operation; §4.1.2 lists the possible values.
- If `Status` = `STATUS_OK`, `Out` is information about the array referenced by `Array_ID`, as follows:

- If `Which` = `"type"`, `Out` is the type of the array elements—`"int"` or `"double"`.
- If `Which` = `"dimensions"`, `Out` is an array containing the global array dimensions.
- If `Which` = `"processors"`, `Out` is an array containing the processors over which the array is distributed.
- If `Which` = `"grid_dimensions"`, `Out` is an array containing the dimensions of the processor grid used to decompose the array.
- If `Which` = `"local_dimensions"`, `Out` is an array containing the dimensions of a local section of the array, excluding borders.
- If `Which` = `"borders"`, `Out` is an array of size twice the number of dimensions containing the sizes of the borders of a local section of the array, in the form described in §4.2.1.
- If `Which` = `"local_dimensions_plus"`, `Out` is an array containing the dimensions of a local section of the array, including borders.
- If `Which` = `"indexing_type"`, `Out` is the indexing type of the array—`"row"` or `"column"`.
- If `Which` = `"grid_indexing_type"`, `Out` is the indexing type of the processor grid—`"row"` or `"column"`.

`Out` is `[]` otherwise.

4.2.7 Verifying a distributed array's borders

The following procedure verifies that the local-section borders of a distributed array are as expected and, if not, corrects them if possible. As described in §3.2.1.3, local-section borders are provided for compatibility with notations that require them. Since an array could be created (and its borders specified) in one PCN procedure and then passed to a data-parallel program in another PCN procedure, it is useful to have some way of verifying that the array's borders are those expected by the data-parallel program. This procedure performs that function: It verifies that the referenced array has the correct indexing type—row-major or column-major—and then compares its borders with the expected ones. If they differ, it reallocates the local sections of the array with the expected borders and copies all interior (i.e., non-border) data.

```
am_user:verify_array(
        /*in 2-tuple of int*/           Array_ID,
        /*in int*/                      N_dims,
        /*in tuple*/                    Border_info,
        /*in string*/                   Indexing_type,
        /*out int*/                     Status
        )
```

Precondition:

- `N_dims` is the number of dimensions of the array referenced by `Array_ID`.
- `Border_info`, `Indexing_type` are as described for `create_array()` in §4.2.1.

Postcondition:

- `Status` reflects the success of the operation; §4.1.2 lists the possible values.
- If `Status` = `STATUS_OK`, then the array referenced by `Array_ID` has the specified indexing type and borders and unchanged interior (non-border) data. The local sections may have been reallocated and their data copied.

Examples

For example, suppose that 2-dimensional array `A` has been created with row-major indexing and borders of size 2. Suppose also that foreign-code program `pgmA` expects its first parameter to be an array with borders of size 2, foreign-code program `pgmB` expects its first parameter to be an array with borders of size 1, and both programs contain routines to specify borders, as described in §4.2.1.

The following call:

```
am_user:verify_array(/*in*/ A, 2,
                 {"foreign_borders", "pgmA", 1}, "row",
        /*out*/ Status)
```

will set `Status` to `STATUS_OK`.

The following call:

```
am_user:verify_array(/*in*/ A, 2,
                 {"foreign_borders", "pgmB", 1}, "row",
        /*out*/ Status)
```

will change the borders of `A`, reallocate all local sections and copy their interior data, and set `Status` to `STATUS_OK`.

The following call:

```
am_user:verify_array(/*in*/ A, 2,
                 {"foreign_borders", "pgmA", 1}, "column",
        /*out*/ Status)
```

will set `Status` to `STATUS_INVALID`, since the indexing type in the procedure call does not match that of `A`.

4.3 Distributed-call procedures

The procedures described in this section provide support for the programming model of distributed calls defined in §3.3.1.

4.3.1 Making a distributed call

The following procedure calls a program concurrently on one or more processors, with parameters as described in §3.3.1.2.

```
am_user:distributed_call(
        /*in array of int*/             Processors,
        /*in string*/                   Module,
        /*in string*/                   Program,
        /*in tuple*/                    Parameters,
        /*in string*/                   Combine_module,
        /*in string*/                   Combine_program,
        /*out int*/                     Status
        )
```

Precondition:

- Actual parameters `Module`, `Program`, `Combine_module`, and `Combine_program` are literals (`[]` or strings, not variables).
- There is a program `Module:Program`. (If `Module` = `[]`, program `Program` may be a foreign—i.e., non-PCN—program.) The formal parameters of `Module:Program` are compatible with the actual parameters derived from `Parameters` as described in the postcondition.
- If `Combine_module` $\neq$ `[]`, there is a program of the following form:

  ```
  Combine_module:Combine_program(
          /*in int*/                      S_in1,
          /*in int*/                      S_in2,
          /*out int*/                     S_out
          )
          ...
  ```

  that combines `S_in1` and `S_in2` to yield `S_out`. Note that since this program assigns a value to a definition variable, it must be a PCN program. `Combine_module` $\neq$ `[]` is only meaningful if `Parameters[i]` = `"status"` for some i.
- `Parameters[i]` = `"status"` for at most one i.
- If `Parameters[i]` = {`"local"`, *Array_ID*}, then the array corresponding to *Array_ID* is distributed over `Processors`.

- If `Parameters[i]` = {`"reduce"`, *Type*, *Length*, *Reduce_combine_mod*, *Reduce_combine_pgm*, *Variable*}, then *Type* is `"int"`, `"char"`, or `"double"`; *Combine_mod* and *Combine_pgm* are literals (`[]` or strings, not variables); and there is a program of the following form:

```
Reduce_combine_module : Reduce_combine_program(
        /*in array of Type of length Length*/           S_in1,
        /*in array of Type of length Length*/           S_in2,
        /*out array of Type of length Length*/          S_out
        )
        ...
```

  that combines `S_in1` and `S_in2` to yield `S_out`. If *Length* = 1, the three parameters are scalars (identical with arrays of length 1). Note that since this program assigns a value to a definition variable, it must be a PCN program.

Postcondition:

- `Module:Program` has been executed on each processor in `Processors`, with actual parameters `New_parameters` passed by reference and defined as follows:
  - Length(`New_parameters`) = length(`Parameters`).
  - For each i, `New_parameters[i]` is defined as follows:
    * If `Parameters[i]` = {`"local"`, *Array_ID*}:
      · `New_parameters[i]` is the local section of the distributed array represented by *Array_ID*. The local section is passed to the called program as a mutable array of type `"double"` or `"int"`, depending on the type of the distributed array represented by *Array_ID*, and it may be used as input or output or both.
    * else if `Parameters[i]` = `"index"`:
      · `New_parameters[i]` is an index into `Processors`, with value j on processor `Processors[j]`. This parameter is an integer and may be used as input only.
    * else if `Parameters[i]` = `"status"`:
      · `New_parameters[i]` is a local variable `status_local`. This parameter is an integer and may be used as output only.
    * else if `Parameters[i]` = {`"reduce"`, *Type*, *Length*, *Combine_mod*, *Combine_pgm*, *Variable*}:
      · `New_parameters[i]` is a local variable *Variable_local*. This parameter has type *Type* and length *Length* and may be used as output only.
    * else:
      · `New_parameters[i]` = `Parameters[i]`. Such a parameter is a global "constant", as described in §3.3.1.2; it may be of any type, but it may be used as input only.

- If `Parameters[i]` = `"status"` for some i:
  - If `Combine_module` $\neq$ `[]`, `Status` is the result of applying `Combine_module:Combine_program` pairwise to the set of `status_local` values, one for every copy of the called program.
  - else `Status` is the maximum of `status_local` over all copies of the called program.

  Otherwise (i.e., if no parameter is `"status"`), `Status` = `STATUS_OK`.

  In any case, `Status` is assigned a value only on completion of all copies of `Module:Program`.

- If `Parameters[i]` = {`"reduce"`, *Type*, *Length*, *Reduce_combine_mod*, *Reduce_combine_pgm*, *Variable*}, then the value of *Variable* is the result of applying *Reduce_combine_module*:*Reduce_combine_program* pairwise to the set of *Variable_local* values, one for every copy of the called program.

Examples

The following examples illustrate distributed calls to C and Fortran programs with various types of parameters (constants, local sections, index variables, status variables, and reduction variables). In the examples, `A` references a 1-dimensional distributed array of type `"double"`. Observe that, as noted in the postcondition for `distributed_call`, all parameters are passed by reference.

Distributed call with index and local-section parameters

Consider the following distributed call to C program `cpgm1`:

```
am_user:distributed_call(/*in*/ Procs, [], "cpgm1",
        {Procs, Num_procs, "index", {"local", A}},
        [], [],
        /*out*/ Status)
```

Called program `cpgm1` has the following form:

```
cpgm1(Procs, Num_procs, Index, local_section)
int     Procs[] ;               /* in */
int     *Num_procs ;            /* in */
int     *Index ;                /* in */
double  local_section[] ;       /* in/out -- local section of A */
...
```

Execution of the distributed call causes `cpgm1` to be executed once on each processor in `Procs`. On processor `Procs[j]`, the value of parameter `*Index` is j, and the value of parameter `local_section` is the local section of array `A`—a standard C array of type `double`. When all copies of `cpgm1` have completed execution, variable `Status` in the calling PCN program is set to `STATUS_OK`.

Distributed call with index, status, and local-section parameters

Now consider the following distributed call to Fortran program `fpgm1`:

```
am_user:distributed_call(/*in*/ Procs, [], "fpgm1_",
        {Procs, Num_procs, "index", {"local", A}, "status"},
        [], [],
        /*out*/ Status) ,
```

(Observe that in calling Fortran from PCN, an underscore is appended to the name of the program to be called.)

Called program `fpgm1` has the following form:

```
        subroutine fpgm1(procs, num, index, local, status)
c               in:  procs, num, index
c               in/out:  local
c               out:  status
        integer procs(*)
        integer num
        integer index
        double precision local(*)
        integer status
...
```

Execution of the distributed call causes `fpgm1` to be executed once on each processor in `Procs`. On processor `procs(j)`, the value of parameter `index` is $j-1$. (Its value is $j-1$ and not j because Fortran array indices start with 1, while PCN indices start with 0.) The value of parameter `local` is the local section of array `A`—a standard Fortran array of type `double precision`. Before completing execution, program `fpgm1` must assign a value to variable `status`; when all copies of `fpgm1` have completed execution, variable `Status` in the calling PCN program is set to the maximum value, over all copies of `fpgm1`, of local variable `status`.

Distributed call with status, reduction, and local-section parameters

Finally, consider the following distributed call to C program `cpgm2`:

```
am_user:distributed_call(/*in*/ Procs, [], "cpgm2",
        {Procs, Num_procs, {"local", A}, "status",
                {"reduce", "double", 2, "thismod", "combine", RR}},
        "thismod", "min",
        /*out*/ Status) ,
```

Called program cpgm2 has the following form:

```
cpgm2(Procs, Num_procs, local_section, local_status, other_output)
int     Procs[] ;               /* in */
int     *Num_procs ;            /* in */
double  local_section[] ;       /* in/out -- local section of A */
int     *local_status ;         /* out */
double  other_output[2] ;       /* out */
...
```

Program thismod:min, used to combine local status variables, is a PCN program of the following form:

```
thismod:min(
        /*in int*/                      In1,
        /*in int*/                      In2,
        /*out int*/                     Out
        )
...
```

Program thismod:combine, used to combine local variables for reduction variable RR, is a PCN program of the following form:

```
thismod:combine(
        /*in array of double of length 2*/              In1,
        /*in array of double of length 2*/              In2,
        /*out array of double of length 2*/             Out
        )
...
```

Execution of the distributed call causes cpgm2 to be executed once on each processor in Procs. On processor Procs[j], the value of parameter local_section is the local section of array A—a standard C array of type double. Before completing execution, program cpgm2 must assign values to variables *local_status (an integer) and *other_output (an array of two double-precision reals). When all copies of cpgm2 have completed execution, the values of these local variables are used to set Status and RR in the calling PCN program as follows: Status is set to the result of combining the local values of *local_status, one per copy of cpgm2, pairwise using program thismod:min. RR is set to the result of combining the local values of *other_output pairwise using program thismod:combine.

Chapter 5

Implementation details: internals

This chapter describes the internal design and operation of our prototype implementation; that is, it describes the modifications and additions made to PCN to support the programming model described in §3.2.1 and §3.3.1 and the library procedures described in §4. The reader is assumed to be familiar with PCN syntax and terminology [5, 9] and to have some knowledge of how PCN is implemented [9, 10].

Note that our implementation is based on PCN Version 1.2; the modifications and additions described in this chapter may not apply to later versions.

5.1 Support for distributed arrays

An overview of how the prototype implementation supports distributed arrays was presented in §3.2.2; in this section, we present a more detailed description of that support.

5.1.1 The array-manager server

The array manager, as described in §3.2.2.2, consists of one array-manager process on each processor. Every application program that uses distributed arrays must be able to communicate with the local array-manager process; in addition, every array-manager process must be able to communicate with every other array-manager process. The most obvious approach to providing this communication is to establish explicit communication channels among all the array-manager processes and between every application program that uses distributed arrays and the local array-manager process.

This approach is somewhat cumbersome, however, so our implementation instead makes use of the PCN server mechanism provided in PCN Version 1.2. Like the array manager, the PCN server consists of one server process per processor. Any program can communicate with the local PCN server process via a *server request*, which has the following syntax:

```
! request_type (request_parameters)
```

The ability to service a particular request type is referred to as a *capability*. New capabilities may be added to the server by loading (via the PCN runtime command `load`) a module that contains a capabilities directive and a server program. The capabilities directive lists the new request types to be supported; the server program processes server requests of those types. After such a module has been loaded, the PCN server passes all server requests of the types listed in the module's capabilities directive to the module's server program, in the form of a tuple whose elements are the request type (a character string) and the request parameters. For example, after loading the array manager, which adds a `free_array` capability, the following server request:

```
!  free_array(A1, Status)
```

will be routed by the PCN server to the array-manager server program, as a tuple of the following form:

```
{"free_array", A1, Status}
```

Observe that this routing of a server request from the application program through the PCN server to the array-manager server program is all local to a processor. Routing the request to the array-manager server program on another processor can be accomplished, however, by executing the server request on the desired processor via the `@`*Processor_number* annotation.

Thus, application programs can communicate with server programs without explicit communication channels. Further, the communication can be bidirectional. Bidirectional communication occurs when one of the parameters of the server request is an undefined definitional variable that is set by the server program; in the `free_array` request above, `Status` is such a variable.

By implementing the local array-manager processes as server programs using this feature, we avoid the need for defining explicit communication channels for the array manager; communication between array-manager processes and between application processes and array-manager processes is automatically provided by the PCN server mechanism.

In order to support the operations on distributed arrays described in §3.2.1.5 and the library procedures described in §4.2, the array-manager server program processes the following types of server requests:

- `create_local`, which creates a local section for a distributed array.
- `create_array`, which creates a distributed array by making a `create_local` request on every processor in the array's distribution.

- `free_local`, which frees a local section of a distributed array.
- `free_array`, which frees a distributed array by making a `free_local` request on every processor in the array's distribution.
- `read_element_local`, which reads an element of the local section of an array.
- `read_element`, which translates global indices into a processor reference and local indices and makes a `read_element_local` request on the appropriate processor.
- `write_element_local`, which writes an element of the local section of an array.
- `write_element`, which translates global indices into a processor reference and local indices and makes a `write_element_local` request on the appropriate processor.
- `find_local`, which obtains a reference to the local section of a distributed array.
- `copy_local`, which reallocates the local section of a distributed array with different borders and copies its data.
- `verify_array`, which compares a distributed array's actual borders to the specified expected borders and, if they do not match, makes a `copy_local` request on every processor in the array's distribution.
- `find_info`, which obtains information—dimensions, for example—about a distributed array.

Observe that the requests fall into two categories—requests corresponding to one of the array-manipulation operations described in §3.2.1.5, and *operation*`_local` requests that are used internally by the array manager as described in the preceding list.

Processing of all of these request types is straightforward given the internal representation of arrays described in §5.1.3.

5.1.2 Library procedures

One somewhat inconvenient aspect of the server mechanism described in §5.1.1 is that, considered as a program statement, a server request completes immediately. Thus, in order for an application program to guarantee that a particular server request has been serviced, the request must contain a variable that will be set by the server when the request has been serviced, and the application program must explicitly test whether that variable has been set. For example, to verify that the `free_array` request shown in §5.1.1 has been serviced, the application program in which the request appears must explicitly test variable `Status`.

Since it can be cumbersome to thus explicitly wait for server requests to be serviced, our implementation provides the library procedures described in §4.2. The function of each library procedure is to issue the appropriate server request and wait for it to be serviced. The library procedure terminates only after the request has been serviced, so an application program can, for example, perform a sequence of distributed-array manipulations by sequentially composing calls to the appropriate library procedures, without explicit checks for termination of the array-manager server requests. Since the syntactic requirements for some parameters, particularly those for array creation, can be involved, some of the library procedures also perform syntax checking on their input parameters before making the appropriate array-manager server request.

5.1.3 Internal representation of distributed arrays

Each array-manager process keeps a list of tuples; each tuple in the list represents a distributed array. When an array is created, an entry is added to the array manager's list on every processor over which the array is distributed, as well as on the creating processor; when an array is freed, the corresponding entry is invalidated to prevent further access. Each array-representation tuple consists of the following elements:

- A globally-unique ID (a tuple consisting of the creating processor's processor number and an integer that distinguishes the array from others created on the same processor).
- The type of the array elements (`double` or `int`).
- The dimensions of the global array, stored as a 1-dimensional array.
- The processor numbers of the processors over which the array is distributed, stored as a 1-dimensional array.
- The dimensions of the processor grid over which the array is distributed, stored as a 1-dimensional array.
- The dimensions of a local section of the array, exclusive of borders, stored as a 1-dimensional array.
- The sizes of the borders around a local section, stored as a 1-dimensional array of length twice the number of dimensions, with values as described in §4.2.1.
- The dimensions of a local section of the array, including borders, stored as a 1-dimensional array.
- The indexing type of the array (row-major or column-major).
- The indexing type of the array's processor grid (row-major or column-major).
- A reference to the storage for the local section. §5.1.5 discusses our implementation's representation of local sections in more detail.

This representation contains some redundant information—for example, the dimensions of a local section can be computed from the dimensions of the global array and the dimensions of the processor grid—but we choose to compute the information once and store it rather than computing it repeatedly as needed.

5.1.4 References to distributed arrays

As described in §4.1.3, a distributed array is referenced in a PCN program by its unique ID; this ID is created by a `create_array` request and must be supplied for all other array-manager requests. During processing of an array-manager request, the array manager uses the array ID to locate the array's internal representation (the tuple described in §5.1.3). As noted in §5.1.3, such a representation occurs in the array manager's list on each processor that contains a local section of the array and on the processor on which the initial array-creation request was made. Thus, our implementation supports the model described in §3.2.1.5: A `create_array` request for a distributed array can be made on any processor, even one that will not contain a local section of the array. With the exception of `local_section` requests, subsequent requests involving the array can be made on the creating processor or on any of the processors over which the array is distributed. A `local_section` request, however, clearly requires a local rather than a global view of the array, and thus can be made only on a processor that contains a local section of the array, i.e., on one of the processors over which the array is distributed.

5.1.5 Internal representation of local sections

The approach taken by our implementation to representing local sections of distributed arrays is somewhat unorthodox. Its design is based on the following observations: The actual storage for a local section cannot be a true mutable, since we want to represent each array by a tuple as described in §5.1.3, and a mutable cannot be included as an element of a tuple. Neither can it be a true definition variable, however, because definition variables are single-assignment, and a local section must be multiple-assignment. Further, for efficiency reasons it is desirable to explicitly allocate and deallocate storage for local sections using the C memory-management routines rather than relying on PCN's normal mechanism for allocating storage for variables. PCN's normal method is to allocate space for both definitional and mutable variables on its internal heap, which is copied during periodic garbage collection; for large arrays, this copying is expensive. Our implementation therefore provides support for a third category of array variables that are explicitly allocated and deallocated with the C memory-management routines. We refer to an array in this category as a *pseudo-definitional* array: "definitional" because it is created without a declaration and thus syntactically resembles a definition variable, and "pseudo-" because it is intended to be used as a multiple-assignment variable and thus semantically resembles a mutable variable.

Local sections are implemented as pseudo-definitional arrays, which allows them to be manipulated in the desired way. They can be incorporated into and retrieved from tuples as if they were definitional arrays, and they can also be passed to PCN or foreign-code programs as mutable arrays, with the following stipulations:

- Passing a pseudo-definitional array as a mutable gives rise to compiler warning messages, as described in §B.1.

- Any use of a pseudo-definitional array as a mutable must be preceded by a `data()` guard, to ensure that it has been defined and to provide proper dereferencing. This requirement must be and is met in all library procedures and in procedures generated by the source-to-source transformation described in §5.2. Application programs, however, normally need not be aware of the restriction, since there is no need for explicit use of pseudo-definitional arrays except in the library procedures and the generated procedures.
- Like a mutable array, a pseudo-definitional array may only be shared by two concurrently-executing processes if neither process writes to it.

5.1.6 Explicit allocation and deallocation of storage for local sections

Source-level support for the explicitly-allocated pseudo-definitional variables used to implement local sections (§5.1.5) is provided by the following new PCN primitive operations:

- `build(`*Type*, *Size*, *Def*`)`. This primitive operation creates an array of type *Type* and size *Size* and defines definition variable *Def* to be that array. *Type* must be a literal `"int"` or `"double"`; *Size* can be a constant or a variable. Storage is allocated using the C function `malloc()`.
- `free(`*Variable*`)`. This primitive operation deallocates (frees) the storage associated with *Variable* using the C function `free()`.

These primitive operations are used in the array-manager implementation to allocate and deallocate storage for local sections.

Runtime support for pseudo-definitional variables requires minor modifications to the PCN emulator. The PCN emulator [10] represents both definitional and mutable variables using data structures that combine header information (type and length) with actual data and/or pointers to data. All data is allocated on the emulator's internal heap using the emulator instructions `build_static`, `build_dynamic`, and `build_def`, each of which reserves space and builds a particular type of data structure in the reserved space. Data is never explicitly deallocated; rather, periodic garbage collection removes data that is no longer in use.

Since the internal representation of data includes support for pointers to other data structures, no modification to the representation is required to support references to data allocated outside the heap. Support for explicitly allocating and deallocating storage outside the heap is provided using `malloc` and `free` emulator instructions, and support for building data structures in the explicitly-allocated storage is provided using modified versions of the `build_static` and `build_dynamic` emulator instructions. (Since support for pseudo-definitional variables is limited to arrays, no modified version of the `build_def` emulator instruction is needed.) Data allocated outside the heap is manipulated in the same way as data on the heap, except during garbage collection, when it is not moved or copied.

Thus, runtime support for explicit allocation and deallocation of pseudo-definitional arrays requires minor modifications to the garbage collection procedure, plus support for the following new emulator instructions:

- `malloc(`*Tag*, *Num*, *Reg*`)`. To execute this instruction, the emulator allocates space, using the C `malloc()` function, for *Num* objects of type *Tag* with a total combined length given by the cell pointed to by *Reg*. It puts the pointer to the newly allocated space in the emulator's `SP` register. (This instruction was originally designed as part of more general support in PCN for explicitly-allocated variables and is thus somewhat more general than is necessary for this use.)
- `m_build_static(`*Reg*, *Tag*, *Size*`)` and `m_build_dynamic(`*Tag*, *Reg1*, *Reg2*`)`. To execute these instructions, the emulator proceeds as for the current `build_static` and `build_dynamic` instructions, except that it uses the `SP` register (assumed to have been set by a preceding `malloc` instruction) to point to the next available space instead of the `HP` (heap pointer) register.
- `free(`*Reg*`)`. To execute this instruction, the emulator frees the space pointed to by register *Reg*, using the C `free()` function.

The new primitive operations `build` and `free` are compiled into the new emulator instructions as follows:

- A `free` primitive operation is compiled directly into a `free` emulator instruction.
- A `build` primitive operation is compiled into a sequence of emulator instructions. The following primitive operation:

  `build(`*Type*, *Size*, *Def*`)`

  is compiled into the following sequence of emulator instructions:

  `malloc(`*Type*, 1, *Rm*`)`
  `m_build_dynamic(`*Type*, *Rm*, *Rj*`)`
  `define(`*Rn*, *Rj*`)`

  where *Rm* is the emulator register representing *Size*, *Rn* is the emulator register representing *Def*, and *Rj* is a work register.

Note that every use of a definitional variable set by a `build` operation, including a `free` operation, must be preceded by a `data()` guard. This has two purposes:

- To ensure that the storage has in fact been allocated before attempting to use it—particularly important if the storage is to be used as a mutable or freed with a `free` operation.
- To provide proper dereferencing.

5.1.7 Support for `foreign_borders` option

As described in §3.2.1.3 and §4.2.1, our implementation provides a way for a data-parallel program to specify the borders for local sections of a distributed array. This is done by calling `am_user:create_array` or `am_user:verify_array` with a `Border_info` parameter of the following form:

{`"foreign_borders"`, *Program*, *Parm_num*}

Program must in turn contain a routine *Program_* that supplies the correct borders.

At runtime, then, program *Program_* must be executed. PCN supports higher-order program calls—calls to a program whose name is supplied at runtime by the value of a character-string variable—but this support does not extend to calls to foreign-code (non-PCN) programs, since the names of foreign-code programs to be called must be fixed at compile time. Our implementation resolves this difficulty by using a source-to-source transformation to first generate a PCN wrapper program that calls the desired foreign-code program *Program_* and then replace the `Border_info` parameter with one that will result in a call to the wrapper program when the array is created or verified.

The required source-to-source transformation replaces every tuple in the input module of the following form:

{`"foreign_borders"`, *Pgm_name*, *Parm_num*}

with a tuple of the following form:

{`"borders"`, *Current_module*, *New_program*, *Parm_num*}

where *Current_module*:*New_program* is an inserted program of the following form (using the documentation conventions of §4.1.1):

```
New_program(
        /*in int*/                                  Parm_num,
        /*in int*/                                  N_borders,
        /*out array of int of length N_borders*/    Borders
        )
int temp_borders[N_borders] ;
{;
        Pgm_name_(/*in*/ Parm_num, /*out*/ temp_borders)
        Borders = temp_borders
}
```

When procedure `am_user:create_array` or `am_user:verify_array` is called with a `Border_info` parameter of the form {`"borders"`, *Module*, *Program*, *Parm_num*}, it executes a program call of the following form:

Module : *Program* (*Parm_num*, 2 ∗ *N_dims*, *Borders*)

where *N_dims* is the number of dimensions of the array being created or verified. The `create_array` or `verify_array` procedure then uses the resulting array *Borders* to define the borders for the array.

Example

Suppose input module `xform_ex1` contains the following tuple:

```
Borders = {"foreign_borders", "cpgm", 1}
```

In the transformed module, this statement is replaced with the following tuple:

```
Borders={"borders","xform_ex1","make_borders_0",1}
```

and the transformed module contains the following program:

```
make_borders_0(Parm_num,N_borders,Borders)
int temp_borders[N_borders];
{ ; cpgm_(Parm_num,temp_borders),
    Borders=temp_borders
}
```

5.2 Support for distributed calls

An overview of how the prototype implementation supports distributed calls was presented in §3.3.2; in this section, we present a more detailed description of that support.

5.2.1 The `do_all` program

As described previously in §3.3.2.2 and §4.3, a call to `am_user:distributed_call` must result in executing the desired program concurrently on many processors. It is relatively straightforward to write a PCN procedure that performs such an operation; its specification (using the conventions of §4.1.1) is as follows:

```
am_util:do_all(
        /*in array of int*/             Processors,
        /*in string*/                   Module,
        /*in string*/                   Program,
        /*in tuple*/                    Parms,
        /*in string*/                   Combine_module,
      /*in string*/                   Combine_program,
      /*out*/                         Status
      )
```

Precondition:

- There is a program of the following form:

  ```
  Module:Program(
          /*in int*/                      Index,
          /*in tuple*/                    Parms,
          /*out*/                         Status
          )
          ...
  ```

- There is a program of the following form:

  ```
  Combine_module:Combine_module(
          /*in Type*/                     S_in1,
          /*in Type*/                     S_in2,
          /*out Type*/                    S_out
          )
          ...
  ```

  where all parameters have the same type *Type* and length *Length*.

Postcondition:

- For each i with $0 \leq i <$ length(`Processors`), `Module:Program` has been executed on processor `Processors[i]` as follows:

  ```
  Module:Program(/*in*/ i, Parms, /*out*/ S_i)
  ```

  where each `S_i` is a distinct variable of the same type *Type* and length *Length* as the parameters of `Combine_module:Combine_program`.

- `Status` has the same type *Type* and length *Length* as the parameters of `Combine_module:Combine_program` and a value that is the result of pairwise combining the values of the `S_i`'s using `Combine_module:Combine_program`.

Since there is a one-to-one correspondence between the parameters of `do_all` and the parameters of `am_user:distributed_call`, `do_all` could be used to perform distributed calls were it not for two difficulties: First, as noted earlier (§5.1.7), support in PCN for higher-order program calls does not include calls to foreign-code programs. Second, as noted in §3.3.2.2, processing of some types of parameters (local sections, for example) requires a wrapper program.

Thus, our implementation provides support for distributed calls via a source-to-source transformation that first generates an appropriate wrapper program and then replaces the call to `am_user:distributed_call` with a call to `do_all` that results in the wrapper program being executed concurrently on the specified processors.

5.2.2 The wrapper program

Our programming model for distributed calls, as described in §3.3.1.2, includes support for passing the following kinds of parameters to the called data-parallel program:

- Global constants.
- Index variables.
- Local-section variables.
- Status variables.
- Reduction variables.

Support for many of these options must be provided by the wrapper program that is to be called by `do_all`. From the specification of `do_all`, the wrapper program will be called with the following parameters:

- An index parameter `Index`, used as input. The value of this parameter is different for each copy of the wrapper program.
- An input parameter `Parms`. The value of this parameter is the same for each copy of the wrapper program. Like the corresponding parameter of `am_user:distributed_call`, however, this parameter can be a tuple containing array IDs and other data.
- An output parameter `Status`. Each copy of the wrapper program assigns a value to its status parameter; `do_all` combines these values pairwise using the combine program specified by its parameters and then assigns the result to the `do_all` status parameter. Observe, however, that this parameter can be a tuple, given an appropriate combine program.

The wrapper program must transform these parameters into the proper parameters for the called data-parallel program. This is done as follows:

- Global constants: These are included in the tuple passed to `do_all` and then to the wrapper program as `Parms`; the wrapper program extracts them from the tuple.
- Index variables: The wrapper program obtains the required value from its `Index` parameter.
- Local-section variables: Array IDs for local-section variables are included in the tuple passed via the `Parms` parameter; the wrapper program obtains the required reference to each local section by calling `am_user:find_local` with the appropriate array ID.
- Status and reduction variables: The wrapper program declares a local mutable variable for each such variable. On completion of the called program, the wrapper program combines all such variables into a single tuple of the form {*local_status*, *local_reduction_1*, *local_reduction_2*, ...}. A new combine program, generated along with the wrapper program and specified by `do_all` as its combine program, merges these tuples pairwise into an output tuple of the same form. The calling program (i.e., the program making the distributed call) then extracts the elements of this output tuple and assigns them to its status and reduction variables.

5.2.3 The transformation

Thus, the transformation used by our implementation to support distributed calls must accomplish the following:

- Transform the original call into a block containing a call to `do_all` and statements to assign values to status and reduction variables.
- Generate a wrapper program.
- Generate a combine program.

The generated wrapper program must do the following:

- "Unbundle" the tuple passed as `Parms`.
- Declare local variables for status and reduction variables.
- Obtain local sections of distributed arrays.
- Call the specified data-parallel program.
- Combine local status and reduction variables into a single tuple, where the first element of the tuple represents status and remaining elements represent reduction variables.

Observe that the size of local reduction variables can depend on a global-constant parameter provided as an element of `Parms`. Since the value of such a parameter cannot be extracted before declarations of local variables are processed, the wrapper program must actually consists of two nested wrapper programs: an outer or first-level program to extract from the `Parms` tuple any parameters needed to declare local variables, and an inner or second-level program to perform the actions described above.

The generated combine program must do the following:

- Pairwise combine tuples of status and reduction variables, applying to each element of the tuple the appropriate combining procedure. For a reduction variable, the appropriate combining procedure is the one given in the parameter specification for the reduction variable. For a status variable, the appropriate combining procedure is either the one specified in the distributed call or the default, `am_util:max`.

A detailed specification of such a transformation is presented in §F.

5.2.4 Examples of applying the transformation

The examples in this section illustrate how the transformation supports passing various kinds of parameters to a called program.

Distributed call with index and local-section parameters

Suppose input module xform_ex2 contains the following statement:

```
am_user:distributed_call(/*in*/ Processors, [], "cpgm",
        {Processors, P, "index", {"local", AA}},
        [], [], /*out*/ Status)
```

In the transformed module, this statement is replaced with the following block:

```
{|| am_util:do_all(Processors,"xform_ex2","wrapper_1",
                {{Processors,P,_,AA}},
                "xform_ex2","combine_2",_l1),
    Status=_l1[0]
}
```

Since there are no reduction variables, the status variable returned by the do_all call (_l1) is a tuple with a single element whose value is used to set variable Status.

The transformed module contains the following two wrapper programs:

```
wrapper_1(Index,Parms,_l1)
{ ? Parms?={_l7} ->
        wrapper2_0(Index,_l7,_l1),
    default ->
        _l1=1
}

wrapper2_0(Index,Parms,_l1)
{ ? Parms?={_l2,_l3,_,_l4} ->
        {|| am_user:find_local(_l4,_l5,_l6),
            { ? _l6==0,data(_l5) ->
                    { ; cpgm(_l2,_l3,Index,_l5),
                        make_tuple(1,_l1),
                        _l1[0]=0
                    },
                default ->
                    _l1=1
            }
        },
    default ->
        _l1=1
}
```

Data-parallel program cpgm is called from the second-level wrapper program with four parameters corresponding to the four parameters specified in the original distributed call's parameters tuple: two global constants whose values are the values of Processors and P, an index variable whose value is supplied by do_all's Index parameter, and the local section of the distributed array referenced by AA. This local section is obtained by the wrapper program by calling am_user:find_local with array ID equal to AA. Since the original distributed call contains no status or reduction variables, the wrapper program returns as its status a tuple with a single element whose value is determined by the success of the am_user:find_local call. (If the distributed call meets the precondition defined in §4.3.1 for am_user:distributed_call, the am_user:find_local call will return a status of STATUS_OK, i.e., 0.)

The transformed module also contains the following combine program:

```
combine_2(C_in1,C_in2,C_out)
{ ? data(C_in1),tuple(C_in2),length(C_in1)==1,length(C_in2)==1 ->
        {|| make_tuple(1,C_out),
            data(C_out) ->
                am_util:max(C_in1[0],C_in2[0],C_out[0])
        },
    default ->
        C_out=1
}
```

Since the original distributed call contains no status or reduction variables, the combine program combines the single-element tuples returned by the wrapper programs using the default status-combining program am_util:max.

Distributed call with status and local-section parameters

Suppose input module xform_ex3 contains the following statement:

```
am_user:distributed_call(/*in*/ Processors, [], "cpgm",
        {Processors, P, {"local", AA}, "status"},
        [], [], /*out*/ Status)
```

In the transformed module, this statement is replaced with the following block:

```
{|| am_util:do_all(Processors,"xform_ex3","wrapper_1",
                {{Processors,P,AA,_}},
                "xform_ex3","combine_2",_l1),
    Status=_l1[0]
}
```

Again, since there are no reduction variables, the status variable returned by the do_all call (_l1) is a tuple with a single element whose value is used to set variable Status.

The transformed module contains the following two wrapper programs:

```
wrapper_1(Index,Parms,_l1)
{ ? Parms?={_l7} ->
        wrapper2_0(Index,_l7,_l1),
    default ->
        _l1=1
}

wrapper2_0(Index,Parms,_l1)
int local_status;
{ ? Parms?={_l2,_l3,_l4,_} ->
        {|| am_user:find_local(_l4,_l5,_l6),
            { ? _l6==0,data(_l5) ->
                    { ; cpgm(_l2,_l3,_l5,local_status),
                        make_tuple(1,_l1),
                        _l1[0]=local_status
                    },
                default ->
                    _l1=1
            }
        },
    default ->
        _l1=1
}
```

Data-parallel program cpgm is called from the second-level wrapper program with four parameters corresponding to the four parameters specified in the original distributed call's parameters tuple: two global constants whose values are the values of Processors and P, the local section of the distributed array referenced by AA, and a local status variable. The local section of AA is obtained by the wrapper program by calling am_user:find_local with array ID equal to AA. Since the original distributed call contains a status variable but no reduction variables, the wrapper program returns as its status a tuple with a single element whose value is that returned by cpgm in local_status.

The transformed module also contains the following combine program:

```
combine_2(C_in1,C_in2,C_out)
{ ? data(C_in1),tuple(C_in2),length(C_in1)==1,length(C_in2)==1 ->
        {|| make_tuple(1,C_out),
            data(C_out) ->
                am_util:max(C_in1[0],C_in2[0],C_out[0])
        },
    default ->
        C_out=1
}
```

Since the original distributed call contains a status variable but no reduction variables and does not specify a combine program for the status variable, the generated combine program combines the single-element tuples returned by the wrapper programs using the default status-combining program am_util:max.

Distributed call with status, reduction, and local-section parameters

Suppose input module xform_ex4 contains the following statement:

```
am_user:distributed_call(/*in*/ Processors, [], "cpgm",
        {Processors, P, {"local", AA}, "status",
                {"reduce", "double", 10, "xform_ex4", "combine_it", RR}},
        "am_util", "max", /*out*/ Status)
```

In the transformed module, this statement is replaced with the following block:

```
{|| am_util:do_all(Processors,"xform_ex4","wrapper_1",
               {{Processors,P,AA,_,_},10},
               "xform_ex4","combine_2",_l1),
    Status=_l1[0],
    RR=_l1[1]
}
```

Since the original distributed call contains one reduction variable, the status variable returned by the do_all call (_l1) is a tuple with two elements. Its first element is used to set variable Status, and its second element is used to set reduction variable RR.

The transformed module contains the following two wrapper programs:

```
wrapper_1(Index,Parms,_l1)
{ ? Parms?={_l9,_l8} ->
        wrapper2_0(Index,_l9,_l1,_l8),
    default ->
        _l1=1
}

wrapper2_0(Index,Parms,_l1,_l8)
int local_status;
double _l7[_l8];
{ ? Parms?={_l2,_l3,_l4,_,_} ->
        {|| am_user:find_local(_l4,_l5,_l6),
            { ? _l6==0,data(_l5) ->
                    { ; cpgm(_l2,_l3,_l5,local_status,_l7),
                        make_tuple(2,_l1),
                        _l1[0]=local_status,
                        _l1[1]=_l7
                    },
                    default ->
                        _l1=1
                }
            },
        default ->
            _l1=1
    }
```

Data-parallel program cpgm is called from the second-level wrapper program with five parameters corresponding to the five parameters specified in the original distributed call's parameters tuple: two global constants whose values are the values of Processors and P, the local section of the distributed array referenced by AA, a local status variable, and a local reduction variable of type double. The local section of AA is obtained by the wrapper program by calling am_user:find_local with array ID equal to AA. The length of the local reduction variable is obtained from the length specified in the original distributed call: The correct value, 10, is passed from the do_all call to the first-level wrapper program as part of the parameters tuple. The first-level wrapper program extracts it from the parameters tuple and passes it directly to the second-level wrapper program, where it is used in the declaration of the local reduction variable _l7. Since the original distributed call contains a status variable and one reduction variable, the wrapper program returns as its status a tuple with two elements, the first containing the value returned by cpgm in local_status and the second containing the value returned by cpgm in the local reduction variable _l7.

The transformed module also contains the following combine program:

```
    combine_2(C_in1,C_in2,C_out)
    { ? data(C_in1),tuple(C_in2),length(C_in1)==2,length(C_in2)==2 ->
            {|| make_tuple(2,C_out),
                data(C_out) ->
                    am_util:max(C_in1[0],C_in2[0],C_out[0]),
                data(C_out) ->
                    combine_it(C_in1[1],C_in2[1],C_out[1])
            },
        default ->
            C_out=1
    }
```

Since the original distributed call contains a status variable and one reduction variable and specifies combine programs for both the status variable and the reduction variable, the generated combine program is somewhat more complicated than in the preceding examples. Recall that the wrapper program returns a tuple whose first element represents a status variable and whose second element represents a reduction variable. The generated combine program combines two tuples of that form to produce an output tuple of the same form. Thus, the first element of the output tuple produced by the generated combine program is the result of combining the first elements of the input tuples using the combine program specified for the status variable, `am_util:max`, and the second element of the output tuple is the result of combining the second elements of the input tuples using the combine program specified for the reduction variable, `xform_ex4:combine_it`. Program `xform_ex4:combine_it`, supplied by the user, must perform a similar operation, combining two input arrays of type `double` and length 10 to produce an output array of type `double` and length 10.

5.3 Architecture-specific issues

As described in §3.4.1, care must be taken to ensure that the communication mechanisms used by PCN and those used by the called data-parallel programs do not interfere with each other. Since the communication mechanisms used by PCN are different for different architectures, whether additional modifications are required for either PCN or the data-parallel programs depends on both the architecture being used and the data-parallel programs to be called. This section does not attempt to address all combinations of architecture and communication mechanisms. Instead, it discusses an example of such modifications based on our experience in testing the prototype implementation.

As discussed in §D, our prototype implementation was tested with a library of data-parallel programs whose communication mechanism is based on the untyped message-passing routines of the Cosmic Environment [20]. The PCN implementation for the Symult s2010, on which testing was performed, uses the same message-passing routines as its communication mechanism. As noted in §3.4.1, when communication is based on point-to-point message-passing, the messages must be typed. Thus, for this combination of architecture and communication mechanisms, the required modification to the PCN implementation is to replace untyped messages with typed messages of a "PCN" type and to replace non-selective receives with receives that select messages of the "PCN" type. The required modification to the library of data-parallel programs is to similarly replace the use of untyped messages with messages of a "data-parallel-program" type.

Chapter 6

Examples

This chapter presents two examples of using the prototype implementation to integrate data-parallel programs into a task-parallel program. For each example we present the following:

- A brief description of the computational task and an overview of the program.
- Complete code for the task-parallel program. The code is written in PCN and uses the library procedures described in §4 and §C. An overview of PCN syntax appears in §A.
- Specifications for the data-parallel programs, using the specification format described in §4.1.1. The data-parallel programs are based on the example library described in §D.

6.1 Inner product

6.1.1 Description

This somewhat contrived example briefly illustrates the use of distributed arrays and a distributed call. The program does the following:

1. Creates two distributed vectors (1-dimensional arrays).
2. Passes them to a data-parallel program that does the following:
   (a) Initializes them, setting the i-th element of each vector to $i + 1$.
   (b) Computes their inner product.
3. Prints the resulting inner product.

6.1.2 PCN program

The following PCN program performs the described computation:

```
/*=========================================================================
===========================================================================

simple example of interface to message-passing C:
        call a routine to initialize two distributed vectors
        and compute their inner product

===========================================================================
=========================================================================*/

#include "am.h"

-foreign(
        /* object code for called program */
        "C_simple/test_iprdv.o",
        /* support libraries for test_iprdv.o */
        "../Interface/General/cfull/cmatvecnd.$ARCH$.a",
        "../Interface/General/full/matvecnd.$ARCH$.a",
        "../Interface/General/att/attnd.$ARCH$.a"
        )

go(Done)
{;
        stdio:printf("starting test\n", {}, _) ,

/* start array manager */
        am_util:load_all("am", _) ,

/* define constants */
        sys:num_nodes(/*out*/ P) ,
        am_util:node_array(/*in*/ 0,1,P, /*out*/ Processors) ,
        Local_m = 4 ,
        M = P * Local_m ,
        am_util:tuple_to_int_array(/*in*/ {M}, /*out*/ Dims) ,
        Distrib = {"block"} ,
        Borders = [] ,

/* create distributed vectors */
        am_user:create_array(/*out*/ Vector1_ID,
                /*in*/ "double", Dims, Processors, Distrib, Borders, "row",
                /*out*/ _) ,
        am_user:create_array(/*out*/ Vector2_ID,
                /*in*/ "double", Dims, Processors, Distrib, Borders, "row",
                /*out*/ _) ,

/* call data-parallel program test_iprdv once per processor */
        am_user:distributed_call(/*in*/ Processors, [], "test_iprdv",
                {Processors, P, "index", M, Local_m,
                        local(Vector1_ID), local(Vector2_ID),
                        reduce("double", 1, "am_util", "max", InProd)},
                [], [], /*out*/ _) ,

/* print result */
        stdio:printf("inner product:  %g\n", {InProd}, _) ,

/* free vectors */
        am_user:free_array(Vector1_ID, _) ,
        am_user:free_array(Vector2_ID, _) ,

        stdio:printf("ending test\n", {}, _) ,
        Done = []
}
```

6.1.3 Specification for data-parallel program

The specification for data-parallel program `test_iprdv` is as follows:

```
/*======================================================================

test inner product

======================================================================*/

void
test_iprdv(Processors, P, Index, M, m, local_V1, local_V2, ipr)
int     Processors[] ;                  /* in */
int     *P ;                            /* in */
int     *Index ;                        /* in */
int     *M, *m ;                        /* in */
double  local_V1[] ;                    /* out */
double  local_V2[] ;                    /* out */
double  *ipr ;                          /* out */
/*
Precondition:
        Processors are the processors on which the program is being
                executed.
        PP = (*P) == length(Processors).
        (*Index) indicates which processor this is (Processors[*Index]).
        MM = (*M) == (global) length of distributed vector V1.
        mm = (*m) == length of local section of V1.
        local_V1 is the local section of V1.
        local_V2 is the local section of V2.
Postcondition:
        V1[i] == V2[i] == i+1 for all i (0 .. MM-1).
        (*ipr) == inner product of V1 and V2.
*/
```

6.2 Polynomial multiplication using a pipeline and FFT

6.2.1 Description

This example is a pipelined computation that performs a sequence of polynomial multiplications using discrete Fourier transforms, where the transforms are performed using the FFT (fast Fourier transform) algorithm. This computation is representative of a class of computations involving Fourier transforms, as discussed in §2.3.2.

The computational task is to multiply pairs of polynomials of degree $n-1$, where n is a power of 2. Input is a sequence of pairs of polynomials $(F_0, G_0; F_1, G_1; \ldots; F_j, G_j; \ldots)$, each of degree $n-1$. Each input polynomial is represented by its n coefficients. Output is a sequence of polynomials $(H_0, H_1, \ldots, H_j, \ldots)$, each of degree $2n-2$, with $H_j = F_j * G_j$ for all j. Each output polynomial is represented by $2n$ coefficients.

The product of two polynomials can be computed as follows:

1. Extend each input polynomial to a polynomial of degree $2n-1$ by padding with zeros, and evaluate each extended polynomial at the $2n$ $2n$-th complex roots of unity. I.e., for input polynomial with coefficients $(f_0, f_1, \ldots, f_{2n-1})$, compute the complex values $(\hat{f}_0, \hat{f}_1, \ldots, \hat{f}_{2n-1})$, where

$$\hat{f}_j = \sum_{k=0}^{2n-1} f_k e^{\frac{2\pi ijk}{2n}}, \qquad j = 0, \ldots, 2n-1$$

   Each $e^{\frac{2\pi ij}{2n}}$, where i denotes $\sqrt{-1}$, is a distinct $2n$-th complex root of unity. Since these $\hat{f}_j$'s are simply the inverse discrete Fourier transform of the f_j's, they can be computed using an inverse FFT.

2. Multiply the two resulting $2n$-tuples of complex numbers elementwise; the resulting $2n$-tuple represents the values of the desired output polynomial at the $2n$ roots of unity.

3. Fit a polynomial of degree $2n-1$ to the resulting points. I.e., if $(\hat{f}_0, \hat{f}_1, \ldots, \hat{f}_{2n-1})$ represent the complex values of a polynomial of degree $2n-1$, determine its coefficients $(f_0, f_1, \ldots, f_{2n-1})$; this is equivalent to solving the system of equations

$$\sum_{k=0}^{2n-1} f_k e^{\frac{2\pi ijk}{2n}} = \hat{f}_j, \qquad j = 0, \ldots, 2n-1$$

   It can be shown that this system of equations is satisfied by $(f_0, f_1, \ldots, f_{2n-1})$, where

$$f_j = \frac{1}{2n} \sum_{k=0}^{2n-1} \hat{f}_k e^{-\frac{2\pi ijk}{2n}}, \qquad j = 0, \ldots, 2n-1$$

   These f_j's, however, are simply the discrete Fourier transform of the $\hat{f}_j$'s and can thus be computed using a forward FFT.

Figure 6.1 illustrates the overall structure of a program based on this algorithm. Both forward and inverse FFTs can be performed by data-parallel programs, as can the elementwise multiplication step. For each input pair of polynomials, the two inverse FFTs (one for each input polynomial) can be done concurrently; the three steps in the computation (inverse FFT, combine, and forward FFT) can execute concurrently as the stages of a pipeline, as described in §2.3.2.

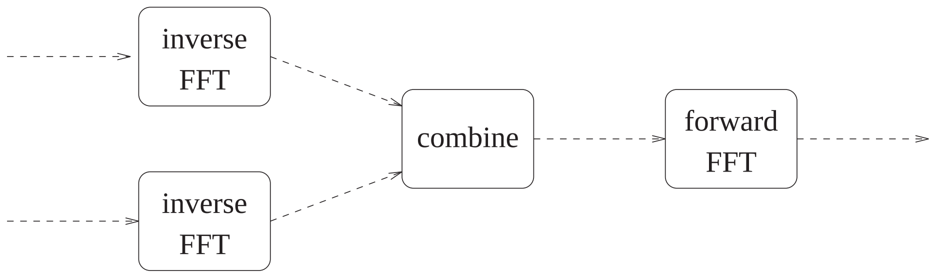

Figure 6.1: Polynomial multiplication using a pipeline and FFT

The data-parallel programs used for the FFTs are based on the discussion of the FFT in [23]. The programs perform the transform in place, i.e., replacing the input data with the output data; such an in-place transform is possible if either the input data or the output data is in permuted order, where the index permutation function for a problem of size 2^m is the bit-reversal ρ_m that maps an m-bit integer i to the integer formed by reversing the m bits of i. The inverse transform in the first stage of the pipeline accepts input in bit-reversed order and produces output in natural order; the forward transform in the third stage of the pipeline accepts input in natural order and produces output in bit-reversed order.

6.2.2 PCN program

The following PCN program performs the described computation:

```
/*======================================================================
=======================================================================

FFT example program

=======================================================================
======================================================================*/

#include "am.h"
#include "stdio.h"
#include "C_fft/fftdef.h"

#define THISMOD          "fft_ex"
```

```
-foreign(
        /* object code for fft_* programs */
        "C_fft/fft.o",
        /* object code for compute_roots, rho_proc; support for fft_* */
        "C_fft/fftlib.$ARCH$.a",
        /* support libraries for fft_*, etc. */
        "../Interface/General/cfull/cmatvecnd.$ARCH$.a",
        "../Interface/General/full/matvecnd.$ARCH$.a",
        "../Interface/General/att/attnd.$ARCH$.a",
        "../Interface/General/cfull/cmatvecnd.$ARCH$.a"
        )

/*======================================================================
========================================================================

main program

========================================================================
======================================================================*/

go(
        N,                      /*in int*/
        Infile,                 /*in string*/
        Outfile,                /*in string*/
        Done                    /*out*/
        )
/*
Precondition:
        N is the size of the input polynomials; N is a power of 2.
        Infile is the input file name; it contains pairs of
                polynomials of degree N-1, each pair represented by
                its N (real) coefficients.
        Outfile is the output file name; it contains the output
                polynomials (each the product of a pair of input
                polynomials); each is of degree 2*N - 1 and is
                represented by its coefficients (printed as complex
                numbers, though the imaginary part will be 0 if
                round-off error is ignored).
        The number of available processors, P, is an even power of 2,
                with P >= 4 and (P/4) <= N.
Postcondition:
        Output as described has been written to Outfile.
*/
FILE    infile_FD ;
FILE    outfile_FD ;
{;
        /* start array manager */
        am_util:load_all("am", /*out*/ _) ,
        /* set things up */
        {||
        /* "real" problem size is 2*N; = 2^LL */
        NN = 2*N ,
        find_log2(/*in*/ NN, /*out*/ LL) ,
        /* groups of processors */
        sys:num_nodes(/*out*/ P4) ,
        P = P4 / 4 ,
        Procs1a_0 = 0 ,
        Procs1b_0 = P ,
        ProcsC_0 = 2*P ,
        Procs2_0 = 3*P ,
        am_util:node_array(/*in*/ Procs1a_0, 1, P,
                /*out*/ Procs1a) ,
        am_util:node_array(/*in*/ Procs1b_0, 1, P,
                /*out*/ Procs1b) ,
        am_util:node_array(/*in*/ ProcsC_0, 1, P,
                /*out*/ ProcsC) ,
        am_util:node_array(/*in*/ Procs2_0, 1, P,
                        /*out*/ Procs2) ,
        /* array dimensions, etc. */
        am_util:tuple_to_int_array(/*in*/ {2*NN}, /*out*/ Dims) ,
        Distrib = {"block"} ,
        am_util:tuple_to_int_array(/*in*/ {2*NN, P},
                /*out*/ Eps_dims) ,
        Eps_distrib = {"*", "block"} ,
        No_borders = [] ,
        C_indexing = "C" ,
        /* open files */
        stdio:fopen(/*in*/ Infile, "r", /*out*/ infile_FD, _) ,
        stdio:fopen(/*in*/ Outfile, "w", /*out*/ outfile_FD, _)
} ,
/* create distributed arrays, etc. */
{||
        /* arrays for input/output data */
        am_user:create_array(/*out*/ A1a,
                /*in*/ "double", Dims, Procs1a, Distrib,
                No_borders, C_indexing, /*out*/ S1a) ,
        am_user:create_array(/*out*/ A1b,
                /*in*/ "double", Dims, Procs1b, Distrib,
                No_borders, C_indexing, /*out*/ S1b) ,
        am_user:create_array(/*out*/ A2,
                /*in*/ "double", Dims, Procs2, Distrib,
                No_borders, C_indexing, /*out*/ S2) ,
        /* arrays for roots of unity (for FFT) */
        am_user:create_array(/*out*/ Eps1a,
                /*in*/ "double", Eps_dims, Procs1a, Eps_distrib,
                No_borders, C_indexing, /*out*/ ST1a) ,
        am_user:create_array(/*out*/ Eps1b,
                /*in*/ "double", Eps_dims, Procs1b, Eps_distrib,
                No_borders, C_indexing, /*out*/ ST1b) ,
        am_user:create_array(/*out*/ Eps2,
                /*in*/ "double", Eps_dims, Procs2, Eps_distrib,
                No_borders, C_indexing, /*out*/ ST2) ,
        /* tuples of streams for communication */
        make_tuple(P, Streams1a) ,
        make_tuple(P, Streams1b) ,
        make_tuple(P, Streams2)
} ,
/* initialize roots of unity (for FFT) */
{||
        ST1a == STATUS_OK ->
        am_user:distributed_call(/*processors*/ Procs1a,
                /*program*/ [], "compute_roots",
                /*parms*/ {/*in*/ NN, /*out*/ local(Eps1a)},
                /*combine program*/ [], [],
                /*status*/ SE1a) ,
        ST1b == STATUS_OK ->
        am_user:distributed_call(/*processors*/ Procs1b,
                /*program*/ [], "compute_roots",
                /*parms*/ {/*in*/ NN, /*out*/ local(Eps1b)},
                /*combine program*/ [], [],
                /*status*/ SE1b) ,
        ST2 == STATUS_OK ->
        am_user:distributed_call(/*processors*/ Procs2,
                /*program*/ [], "compute_roots",
                /*parms*/ {/*in*/ NN, /*out*/ local(Eps2)},
                /*combine program*/ [], [],
                /*status*/ SE2)
} ,
/* main pipeline loop */
S1a == STATUS_OK, S1b == STATUS_OK, S2 == STATUS_OK,
SE1a == STATUS_OK, SE1b == STATUS_OK, SE2 == STATUS_OK ->
{||
        read_infile(/*in*/ N, infile_FD,
                /*out*/ Instream_a, Instream_b, /*in*/ [], []) ,
        phase1(/*in*/ Procs1a, NN, LL, /*inout*/ A1a,
                /*in*/ Eps1a, Instream_a,
                /*out*/ Streams1a)@‘Procs1a_0‘ ,
        phase1(/*in*/ Procs1b, NN, LL, /*inout*/ A1b,
                /*in*/ Eps1b, Instream_b,
                /*out*/ Streams1b)@‘Procs1b_0‘ ,
        combine(/*in*/ ProcsC, Streams1a, Streams1b,
                /*out*/ Streams2)@‘ProcsC_0‘ ,
        phase2(/*in*/ Procs2, NN, LL, /*inout*/ A2,
                /*in*/ Eps2, Streams2,
                /*out*/ Outstream, /*in*/ [])@‘Procs2_0‘ ,
        write_outfile(/*in*/ NN, outfile_FD, Outstream)
} ,
/* finish up */
{||
        /* free arrays */
        am_user:free_array(/*in*/ A1a, /*out*/ _) ,
        am_user:free_array(/*in*/ A1b, /*out*/ _) ,
                am_user:free_array(/*in*/ A2, /*out*/ _) ,
                am_user:free_array(/*in*/ Eps1a, /*out*/ _) ,
                am_user:free_array(/*in*/ Eps1b, /*out*/ _) ,
                am_user:free_array(/*in*/ Eps2, /*out*/ _) ,
                /* close files */
                stdio:fclose(/*in*/ infile_FD, /*out*/ _) ,
                stdio:fclose(/*in*/ outfile_FD, /*out*/ _)
        } ,
        Done = []
}

/*=======================================================================
=========================================================================

phase1:  inverse FFT

=========================================================================
=======================================================================*/

phase1(
        Procs,                  /*in array of int*/
        NN,                     /*in int*/
        LL,                     /*in int*/
        Array_ID,               /*inout distributed array*/
        Eps_array_ID,           /*in distributed array*/
        Instream,               /*in stream of double*/
        Outstreams              /*out tuple of streams*/
        )
/*
Precondition:
        NN = problem size (twice the size of the input polynomials).
        LL = log2(NN).
        Eps_array_ID contains the NN NN-th complex roots of unity.
        Instream contains NN/2 real numbers.
Postcondition:
        Array_ID contains the (complex) inverse FFT of NN numbers, of
                which the first NN/2 are the numbers from Instream and
                the last NN/2 are 0, in natural (not bit-reversed) order.
        Each element of Outstreams contains the complex numbers (each
                represented by two doubles) corresponding to one local
                section of Array_ID.
*/
{?
        Instream != [] ->
        {;
                /* get input (NN/2) and pad */
                get_input(/*in*/ NN/2, LL, Array_ID, Instream,
                        /*out*/ Instream_tail) ,
                pad_input(/*in*/ NN/2, NN, LL, Array_ID) ,
                /* perform inverse FFT on distributed array */
                am_user:distributed_call(/*processors*/ Procs,
                        /*program*/ [], "fft_reverse",
                        /*parms*/ {/*in*/ Procs, length(Procs), "index",
                                        NN, INVERSE, local(Eps_array_ID),
                                /*inout*/ local(Array_ID)},
                        /*combine program*/ [], [],
                        /*status*/ _) ,
                /* write results into output streams, saving
                        stream tails */
                make_tuple(length(Outstreams), Outstreams_tail) ,
                am_user:distributed_call(/*processors*/ Procs,
                        /*program*/ THISMOD, "dbl_array_to_stream",
                        /*parms*/ {/*in*/ "index", local(Array_ID),
                                /*out*/ Outstreams, /*in*/ Outstreams_tail} ,
                        /*combine program*/ [], [],
                        /*status*/ _) ,
                /* recurse */
                phase1(/*in*/ Procs, NN, LL, Array_ID, Eps_array_ID,
                                Instream_tail,
                        /*out*/ Outstreams_tail)
        } ,
        default ->
                /* set all elements of Outstreams to [] */
                tuple_fill(/*in*/ Outstreams, [])
}

/*======================================================================
========================================================================

combine:  combine streams of complex numbers via pairwise multiplication

========================================================================
======================================================================*/


combine(
        Procs,                  /*in array of int*/
        Instreams1,             /*in tuple of streams*/
        Instreams2,             /*in tuple of streams*/
        Outstreams              /*out tuple of streams*/
        )
/*
Precondition:
        Each element of Instreams1 and Instreams2 is a stream of
                doubles; each successive pair of doubles represents
                a complex number.
Postcondition:
        Each element of Outstreams is a stream of doubles; each
                successive pair of doubles represents a complex
                number that is the product of the corresponding
                complex numbers from Instreams1 and Instreams2.
*/
```

```
{||
        am_user:distributed_call(/*processors*/ Procs,
                /*program*/ THISMOD, "combine_sub1",
                /*parms*/ {/*in*/ "index", Instreams1, Instreams2,
                        /*out*/ Outstreams} ,
                /*combine program*/ [], [],
                /*status*/ _)
}

/*--------------------------------------------------------------------*/

combine_sub1(
        Index,                  /*in int*/
        Instreams1,             /*in tuple of streams*/
        Instreams2,             /*in tuple of streams*/
        Outstreams              /*out tuple of streams*/
        )
/*
Precondition, postcondition:
        As for combine() above, but restricted to the Index-th
                element of each tuple.
*/
{||
        combine_sub2(/*in*/ Instreams1[Index], Instreams2[Index],
                /*out*/ Outstreams[Index])
}

/*--------------------------------------------------------------------*/

combine_sub2(
        Instream1,              /*in stream of double*/
        Instream2,              /*in stream of double*/
        Outstream               /*out stream of double*/
        )
/*
Precondition, postcondition:
        As for combine() above, but restricted to a single stream.
*/
{?
        Instream1 ?= [Re1, Im1 | Instream1_tail] ,
                Instream2 ?= [Re2, Im2 | Instream2_tail] ->
        {||
                Re = Re1*Re2 - Im1*Im2 ,
                Im = Re2*Im1 + Re1*Im2 ,
                Outstream = [Re, Im | Outstream_tail] ,
                combine_sub2(/*in*/ Instream1_tail, Instream2_tail,
                        /*out*/ Outstream_tail)
        } ,
        default ->
                Outstream = []
}
```

```
/*=====================================================================
=======================================================================

phase2:  forward FFT

=======================================================================
=====================================================================*/

phase2(
        Procs,                  /*in array of int*/
        NN,                     /*in int*/
        LL,                     /*in int*/
        Array_ID,               /*inout distributed array*/
        Eps_array_ID,           /*in distributed array*/
        Instreams,              /*in tuple of streams*/
        Outstream,              /*out stream*/
        Outstream_tail          /*in stream*/
        )
/*
Precondition:
        NN = problem size (twice the size of the input polynomials,
                same size as the output polynomial).
        LL = log2(NN).
        Eps_array_ID contains the NN NN-th complex roots of unity.
        Each element of Instreams contains the complex numbers (each
                represented by two doubles) corresponding to one local
                section of Array_ID; if these numbers are inserted into
                Array_ID, the result is NN complex numbers representing
                the input to the forward FFT, in natural (not
                bit-reversed) order.
Postcondition:
        Array_ID contains the (complex) forward FFT of the NN complex
                numbers from Instreams (as described above), in
                bit-reversed order.
        Outstream contains the elements of Array_ID (each represented by
                two doubles), in natural order, followed by
                Outstream_tail.
*/
{?
        tuple(Instreams), Instreams[0] != [] ->
        {;
                /* read data from input streams, saving stream tails */
                make_tuple(length(Instreams), Instreams_tail) ,
                am_user:distributed_call(/*processors*/ Procs,
                        /*program*/ THISMOD, "stream_to_dbl_array",
                        /*parms*/ {/*in*/ "index", /*out*/ local(Array_ID),
                                /*in*/ Instreams, /*out*/ Instreams_tail} ,
                        /*combine program*/ [], [],
                        /*status*/ _) ,
                /* perform forward FFT on distributed array */
                am_user:distributed_call(/*processors*/ Procs,
                        /*program*/ [], "fft_natural",
                        /*parms*/ {/*in*/ Procs, length(Procs), "index",
                                        NN, FORWARD, local(Eps_array_ID),
                                /*inout*/ local(Array_ID)},
                        /*combine program*/ [], [],
                        /*status*/ _) ,
                /* write results from distributed array to output stream */
                put_output(/*in*/ NN, LL, Array_ID, /*out*/ Outstream,
                        /*in*/ Outstream_mid) ,
                /* recurse */
                phase2(/*in*/ Procs, NN, LL, Array_ID, Eps_array_ID,
                                Instreams_tail,
                        /*out*/ Outstream_mid, /*in*/ Outstream_tail)
        } ,
        default ->
                Outstream = Outstream_tail
}

/*======================================================================
======================================================================

get_input:  get input from stream into distributed array

======================================================================
======================================================================*/

get_input(
        N,                      /*in int*/
        LL,                     /*in int*/
        Array_ID,               /*inout distributed array*/
        Instream,               /*in stream*/
        Instream_tail           /*out stream*/
        )
/*
Precondition:
        Instream contains N real numbers.
        LL = log2(2*N).
Postcondition:
        Array_ID (size 2*N) contains the complex equivalents of the
                N real numbers from Instream, in bit-reversed order
                (i.e., for jj = 0 .. N-1, the element of Array_ID
                with index bit-reverse(LL, jj) is the jj-th number
                from Instream).
*/
{;
        get_input_sub1(/*in*/ LL, 0, N, Array_ID, Instream,
                /*out*/ Instream_tail)
}

/*--------------------------------------------------------------------*/
```

```
get_input_sub1(
        LL,                     /*in int*/
        Index,                  /*in int*/
        Limit,                  /*in int*/
        Array_ID,               /*inout distributed array*/
        Instream,               /*in stream*/
        Instream_tail           /*out stream*/
        )
{?
        Index < Limit, Instream ?= [Element | Instream_mid] ->
        {;
                bit_reverse(/*in*/ LL, Index, /*out*/ P_Index) ,
                am_user:write_element(/*in*/ Array_ID, {2*P_Index},
                        Element, /*out*/ _) ,
                am_user:write_element(/*in*/ Array_ID, {2*P_Index+1},
                        0.0, /*out*/ _) ,
                get_input_sub1(/*in*/ LL, Index+1, Limit, Array_ID,
                        Instream_mid, /*out*/ Instream_tail)
        } ,
        default ->
                Instream_tail = Instream

}

/*=====================================================================
=======================================================================

put_output:  put output from distributed array into stream

=======================================================================
=====================================================================*/

put_output(
        NN,                     /*in int*/
        LL,                     /*in int*/
        Array_ID,               /*in distributed array*/
        Outstream,              /*out stream*/
        Outstream_tail          /*in stream*/
        )
/*
Precondition:
        NN = problem size.
        LL = log2(NN).
        Array_ID contains NN complex numbers, in bit-reversed order.
Postcondition:
        Outstream contains the NN numbers from Array_ID (with each
                complex number represented by two doubles), in
                natural (not bit-reversed) order, followed by
                Outstream_tail.
*/
```

```
{||
        put_output_sub1(/*in*/ LL, 0, NN, Array_ID, /*out*/ Outstream,
                /*in*/ Outstream_tail)
}

/*-------------------------------------------------------------------*/

put_output_sub1(
        LL,                     /*in int:  log2(problem size)*/
        Index,                  /*in int*/
        Limit,                  /*in int*/
        Array_ID,               /*in distributed array*/
        Outstream,              /*out stream*/
        Outstream_tail          /*in stream*/
        )
{?
        Index < Limit ->
        {;
                bit_reverse(/*in*/ LL, Index, /*out*/ P_Index) ,
                am_user:read_element(/*in*/ Array_ID, {2*P_Index},
                        /*out*/ Re, _) ,
                am_user:read_element(/*in*/ Array_ID, {2*P_Index+1},
                        /*out*/ Im, _) ,
                Outstream = [Re, Im | Outstream_mid] ,
                put_output_sub1(/*in*/ LL, Index+1, Limit, Array_ID,
                        /*out*/ Outstream_mid, /*in*/ Outstream_tail)
        } ,
        default ->
                Outstream = Outstream_tail
}

/*===================================================================
=====================================================================

read_infile:  read input from file into stream

=====================================================================
===================================================================*/

read_infile(
        N,                              /*in int*/
        infile_FD,                      /*in file descriptor*/
        Stream_a, Stream_b,             /*out streams*/
        Stream_a_tail, Stream_b_tail    /*in streams*/
        )
/*
Precondition:
        N is the size of the input polynomials.
        infile_FD is an (open) input file containing an even number of
                sets of N real numbers (each set representing a
                polynomial).
```

```
Postcondition:
        Stream_a contains the first N numbers from the input file,
                followed by Stream_a_tail.
        Stream_b contains the next N numbers from the input file,
                followed by Stream_b_tail.
*/
FILE    infile_FD ;
{;
        read_infile_sub1(/*in*/ N, infile_FD, /*out*/ Stream_a,
                /*in*/ Stream_a_mid, 1, /*out*/ Status_a) ,
        read_infile_sub1(/*in*/ N, infile_FD, /*out*/ Stream_b,
                /*in*/ Stream_b_mid, Status_a, /*out*/ Status_b) ,
        {?
                Status_b > 0 ->
                        read_infile(/*in*/ N, infile_FD,
                                /*out*/ Stream_a_mid, Stream_b_mid,
                                /*in*/ Stream_a_tail, Stream_b_tail) ,
                default ->
                {||
                        Stream_a_mid = Stream_a_tail ,
                        Stream_b_mid = Stream_b_tail
                }
        }
}

/*-------------------------------------------------------------------*/

read_infile_sub1(
        Count,                          /*in int*/
        infile_FD,                      /*in file descriptor*/
        Stream,                         /*out stream*/
        Stream_tail,                    /*in stream*/
        Status_in,                      /*in int*/
        Status_out                      /*out int*/
        )
FILE    infile_FD ;
{?
        Count > 0, Status_in > 0 ->
        {;
                stdio:fscanf(/*in*/ infile_FD, "%f", {Num},
                        /*out*/ Status_mid) ,
                {?
                        Status_mid > 0 ->
                        {;
                                Stream = [Num | Stream_mid] ,
                                read_infile_sub1(/*in*/ Count-1,
                                        infile_FD, /*out*/ Stream_mid,
                                        /*in*/ Stream_tail, Status_mid,
                                        /*out*/ Status_out)
                        } ,
                        default ->
                        {||
                                Stream = Stream_tail ,
                                Status_out = Status_mid
                        }
                }
        } ,
        default ->
        {||
                Stream = Stream_tail ,
                Status_out = Status_in
        }
}

/*======================================================================
======================================================================

write_outfile:  write output from stream into file

======================================================================
======================================================================*/

write_outfile(
        NN,                             /*in int*/
        outfile_FD,                     /*in file descriptor*/
        Stream                          /*in stream*/
        )
/*
Precondition:
        NN is the size of the output polynomials.
        outfile_FD is an (open) output file.
Postcondition:
        Stream contains NN pairs of doubles, each representing a
                complex number.
        The NN pairs of doubles have been written to the output file.
*/
FILE    outfile_FD ;
{?
        Stream != [] ->
        {;
                write_outfile_sub1(/*in*/ NN, outfile_FD, Stream,
                        /*out*/ Stream_tail) ,
                write_outfile(/*in*/ NN, outfile_FD, Stream_tail)
        }
}

/*--------------------------------------------------------------------*/

write_outfile_sub1(
        Count,                          /*in int*/
        outfile_FD,                     /*in file descriptor*/
        Stream,                         /*in stream*/
        Stream_tail                     /*out stream*/
        )
FILE    outfile_FD ;
{?
        Count > 0, Stream ?= [Re, Im | Stream_mid] ->
        {;
                stdio:fprintf(/*in*/ outfile_FD, "(%g, %g)\n", {Re, Im},
                        /*out*/ _) ,
                write_outfile_sub1(/*in*/ Count-1, outfile_FD, Stream_mid,
                        /*out*/ Stream_tail)
        } ,
        default ->
        {;
                stdio:fprintf(/*in*/ outfile_FD, "\n", {}, /*out*/ _) ,
                Stream_tail = Stream
        }
}

/*======================================================================
======================================================================

dbl_array_to_stream:  copy array of doubles to stream

======================================================================
=====================================================================*/

dbl_array_to_stream(
        Index,                  /*in int*/
        array_section,          /*in array of double*/
        Outstreams,             /*out tuple of streams*/
        Outstreams_tail         /*in tuple of streams*/
        )
/*
Precondition:
        TRUE.
Postcondition:
        Outstreams[Index] consists of the elements of array_section,
                followed by Outstreams_tail[Index].
*/
double  array_section[] ;
{||
        dbl_array_to_stream_sub1(/*in*/ 0, length(array_section),
                array_section, /*out*/ Outstreams[Index],
                /*in*/ Outstreams_tail[Index])
}

/*--------------------------------------------------------------------*/

dbl_array_to_stream_sub1(
        I,                      /*in int*/
        Size,                   /*in int*/
        array,                  /*in array of double*/
        Outstream,              /*out stream*/
        Outstream_tail          /*in stream*/
        )
double  array[] ;
{?
        I < Size ->
        {||
                Outstream = [array[I] | Outstream_mid] ,
                dbl_array_to_stream_sub1(/*in*/ I+1, Size, array,
                        /*out*/ Outstream_mid, /*in*/ Outstream_tail)
        } ,
        default ->
                Outstream = Outstream_tail
}

/*=======================================================================
=========================================================================

stream_to_dbl_array:  copy stream of doubles to array

=========================================================================
=======================================================================*/

stream_to_dbl_array(
        Index,                  /*in int*/
        array_section,          /*out array of double*/
        Instreams,              /*in tuple of streams*/
        Instreams_tail          /*out tuple of streams*/
        )
/*
Precondition:
        Instreams[Index] consists of N doubles followed by a tail T,
                where N is the size of array_section.
Postcondition:
        Each element of array_section has been assigned a value from
                Instreams[Index].
        Instreams_tail[Index] is the tail T.
*/
double  array_section[] ;
{||
        stream_to_dbl_array_sub1(/*in*/ 0, length(array_section),
                /*out*/ array_section, /*in*/ Instreams[Index],
                /*out*/ Instreams_tail[Index])
}

/*---------------------------------------------------------------------*/

stream_to_dbl_array_sub1(
        I,                      /*in int*/
        Size,                   /*in int*/
        array,                  /*out array of double*/
        Instream,               /*in stream*/
        Instream_tail           /*out stream*/
        )
double  array[] ;
{?
        I < Size, Instream ?= [Elem | Instream_mid] ->
        {||
                array[I] := Elem ,
                stream_to_dbl_array_sub1(/*in*/ I+1, Size, /*out*/ array,
                        /*in*/ Instream_mid, /*out*/ Instream_tail)
        } ,
        default ->
                Instream_tail = Instream
}

/*=================================================================
=================================================================

tuple_fill:  set all elements of tuple to given value

=================================================================
================================================================*/

tuple_fill(
        T,                      /*in tuple*/
        Item                    /*in*/
        )
/*
Precondition:
        TRUE.
Postcondition:
        Each element of T is Item.
*/
{||
        tuple_fill_sub1(/*in*/ length(T)-1, T, Item)
}

/*---------------------------------------------------------------*/

tuple_fill_sub1(
        Index,                  /*in int*/
        T,                      /*in tuple*/
        Item                    /*in*/
        )
{?
        Index >= 0 ->
        {||
                T[Index] = Item ,
                tuple_fill_sub1(/*in*/ Index-1, T, Item)
        }
```

```
}

/*=================================================================
==================================================================

find_log2:  find (integer) base-2 log of an integer

==================================================================
=================================================================*/

find_log2(
        In,                     /*in int*/
        Out                     /*out int*/
        )
/*
Precondition:
        TRUE.
Postcondition:
        Out = floor(log2(In)).
*/
{?
        In >= 2 ->
        {||
                Out = 1 + Mid ,
                find_log2(In/2, Mid)
        } ,
        default ->
                Out = 0
}

/*=================================================================
==================================================================

pad_input:  set elements of "complex" array to 0, with indexing
        in bit-reversed order

==================================================================
=================================================================*/

pad_input(
        Index,                  /*in int*/
        Limit,                  /*in int*/
        LL,                     /*in int*/
        Array_ID                /*inout distributed array*/
        )
/*
Precondition:
        LL = log2(NN), where NN is the size of Array_ID.
Postcondition:
        For jj = Index .. Limit-1, the element of Array_ID with
                index bit-reverse(LL, jj) is 0.
```

```
*/
{?
        Index < Limit ->
        {;
                bit_reverse(/*in*/ LL, Index, /*out*/ P_Index) ,
                am_user:write_element(/*in*/ Array_ID, {2*P_Index},
                        0.0, /*out*/ _) ,
                am_user:write_element(/*in*/ Array_ID, {2*P_Index+1},
                        0.0, /*out*/ _) ,
                pad_input(/*in*/ Index+1, Limit, LL, Array_ID)
        }
}

/*======================================================================
=======================================================================

bit_reverse:  reverse bits

=======================================================================
======================================================================*/

bit_reverse(
        Bits,                   /*in int*/
        Index,                  /*in int*/
        P_Index                 /*out int*/
        )
/*
Precondition:
        TRUE.
Postcondition:
        P_Index is the bitwise reversal of Index, using Bits bits.
*/
int     temp ;
{;
        rho_proc(/*in*/ Bits, Index, /*out*/ temp) ,
        P_Index = temp
}
```

6.2.3 Specifications for data-parallel programs

The data-parallel program for the combine step is procedure `combine` in the PCN program; its specification appears as part of the program in §6.2.2.

The specifications for data-parallel programs `fft_reverse` and `fft_natural` are as follows:

```
/*======================================================================

FFT:  input in bit-reversed order, output in natural order
```

```
=====================================================================*/

void
fft_reverse(Processors, P, Index, N, Flag, epsilon, bb)
int     Processors[] ;          /* (in) */
int     *P ;                    /* (in) length(Processors) */
int     *Index ;                /* (in) */
int     *N ;                    /* (in) problem size */
int     *Flag ;                 /* (in) inverse/forward */
complex epsilon[] ;             /* (in) N-th roots of unity */
complex bb[] ;                  /* (in/out) input/output of transform */
/*
Precondition:
        Processors are the processors on which the program is being
                executed.
        PP = (*P) == length(Processors); (*P) is a power of 2.
        (*Index) indicates which processor this is (Processors[*Index])
        NN = (*N) is the size of the global array BB, the array to be
                transformed; NN is a power of 2  and NN >= PP.
        (*Flag) == INVERSE or FORWARD.
        epsilon is the NN NN-th roots of unity.
        bb is the local section of BB(in), the array to be transformed;
                global indexing is in bit-reversed order.
Postcondition:
        bb is the local section of BB(out), the transform of BB(in);
                global indexing is in natural order.
        if (*Flag) == INVERSE, BB(out) is the inverse FFT of BB(in);
                else BB(out) is the forward FFT of BB(in) (including
                division by (*N)).
*/

/*=====================================================================

FFT:  input in natural order, output in bit-reversed order

=====================================================================*/

void
fft_natural(Processors, P, Index, N, Flag, epsilon, bb)
int     Processors[] ;          /* (in) */
int     *P ;                    /* (in) length(Processors) */
int     *Index ;                /* (in) */
int     *N ;                    /* (in) problem size */
int     *Flag ;                 /* (in) inverse/forward */
complex epsilon[] ;             /* (in) N-th roots of unity */
complex bb[] ;                  /* (in/out) input/output of transform */
/*
Precondition:
        Processors are the processors on which the program is being
                executed.
        PP = (*P) == length(Processors); (*P) is a power of 2.
        (*Index) indicates which processor this is (Processors[*Index]).
        NN = (*N) is the size of the global array BB, the array to be
                transformed; NN is a power of 2  and NN >= PP.
        (*Flag) == INVERSE or FORWARD.
        epsilon is the NN NN-th roots of unity.
        bb is the local section of BB(in), the array to be transformed;
                global indexing is in natural order.
Postcondition:
        bb is the local section of BB(out), the transform of BB(in);
                global indexing is in bit-reversed order.
        if (*Flag) == INVERSE, BB(out) is the inverse FFT of BB(in);
                else BB(out) is the forward FFT of BB(in) (including
                division by (*N)).
*/
```

These programs in turn require sequential programs `compute_roots` and `rho_proc` with the following specifications:

```
/*=======================================================================

compute roots of unity

=======================================================================*/

void
compute_roots(N, epsilon)
int     *N ;                    /* (in) problem size nn */
complex epsilon[] ;             /* (out) nn-th roots of unity */
/*
Precondition:
        nn = *N is a power of 2.
        length(epsilon) = nn.
Postcondition:
        If omega denotes the primitive nn-th root of unity
                        (e^(2*PI*i/nn)),
                epsilon[j] = j-th power of omega.
*/

/*=======================================================================

do bit-reverse map

=======================================================================*/

void
rho_proc(np, tp, returnp)
int     *np ;                   /* (in) number of bits */
int     *tp ;                   /* (in) */
int     *returnp ;              /* (out) */
/*
```

```
Precondition:
        TRUE.
Postcondition:
        (*returnp) contains the rightmost (*np) bits of (*tp) in
                reverse order and right-justified.
*/
```

Chapter 7

Conclusions

7.1 Summary of work

In this report, we present a programming model for one approach to integrating task parallelism and data parallelism. The model is based on allowing task-parallel programs to call data-parallel programs with distributed data structures as parameters. Although it is a simple and restricted model, it is nonetheless applicable to a number of interesting problem classes (§2.3). The prototype implementation (§3, §4, §5) demonstrates that it is possible, in the context of SPMD implementations of data parallelism, to develop a consistent detailed description of our model and to implement it readily using an existing task-parallel notation, PCN [5, 9]. Further, the adapting of an example library of existing data-parallel programs (§D) demonstrates that the model allows reuse of existing data-parallel code.

7.2 Proposals for additional work

7.2.1 Extending the model: direct communication between data-parallel programs

The model as proposed allows direct communication within a single data-parallel program (as, for instance, between the concurrently-executing copies of an SPMD implementation of the program). However, it requires that all communication between different data-parallel programs go through the common task-parallel calling program; two concurrently-executing data-parallel programs cannot interact. This makes the model simpler and easier to reason about, but it creates a bottleneck for problems in which there is a significant amount of data to be exchanged among different data-parallel programs.

To avoid such bottlenecks, our programming model should be extended to allow concurrently-executing data-parallel programs called from a task-parallel program to communicate directly. One promising approach is to allow the data-parallel programs to communicate using channels defined by the task-parallel calling program and passed to the data-parallel programs as parameters. This approach could be readily implemented using Fortran M [8] (in which processes communicate using channels) as the task-parallel notation. Such a Fortran M implementation could be based on the same design used for the prototype implementation. Since Fortran M is an extension of Fortran 77, a Fortran M implementation would also allow straightforward integration of Fortran-based data-parallel notations such as High Performance Fortran [15] and Fortran D [12].

7.2.2 Reasoning about the model

In this report, we take an informal approach to specifying the programming model (§3) and to discussing the correctness of programs based on the model (§3.4). This informal approach is appropriate for an initial description and implementation, but it is not sufficiently rigorous to provide a foundation either for confident reasoning about large and complex programs or for the extension described above (§7.2.1).

A more formal approach to the existing model has two parts: First, the model should be specified more precisely, and not just in the context of SPMD implementations of data parallelism. Second, the underlying assumptions should be investigated and validated. One such assumption (mentioned in §1.2.3) is that it is valid to reason about data-parallel programs as if they were sequential programs; this assumption should be justified. Another more important assumption is that a data-parallel program that is correct in isolation remains correct when incorporated into a task-parallel program; this assumption should also be justified.

7.2.3 Reasoning about the extended model

The value of a more formal approach is even more evident when considering the proposed extension in §7.2.1 (direct communication between data-parallel programs). Since this extension increases the complexity of the model, its definition should be as precise as possible to ensure that the model remains consistent and reasonable. More importantly, allowing interaction between component data-parallel programs complicates the task of reasoning about correctness; a more formal approach will allow us to reason more confidently about programs containing such interactions. We hope that if we can justify our assumptions about the existing model (that correct data-parallel programs remain correct when called in isolation from a task-parallel program), as proposed in §7.2.2, this justification can be extended and applied to the more general situation, i.e., that we can determine what restrictions on the interaction of the data-parallel programs are required to preserve their correctness.

Appendix A

Overview of PCN syntax and terminology

This appendix gives a very brief description of PCN syntax as an aid for those readers completely unfamiliar with PCN. Complete information about PCN may be found in [5] and [9].

A.1 Program construction

A PCN program is a composition of smaller programs and/or primitive operations. Three types of composition are allowed:

- *Sequential composition*, in which the elements being composed are executed in sequence. For example,

  ```
  {;
          pgmA() ,
          pgmB()
  }
  ```

  means "execute `pgmA()`, and then execute `pgmB()`".

- *Parallel composition*, in which the elements being composed are executed concurrently. For example,

  ```
  {||
          pgmA() ,
          pgmB()
  }
  ```

  means "execute `pgmA()` and `pgmB()` concurrently".

- *Choice composition*, in which at most one of the guarded elements being composed is executed, based on the values of their guards. For example,

  ```
  {?
          X > 0 ->
                  pgmA() ,
          X < 0 ->
                  pgmB()
  }
  ```

  means "execute `pgmA()` if `X` is greater than zero, execute `pgmB()` if `X` is less than zero, and do nothing otherwise".

Composition can be nested; for example,

```
{||
        {; pgmA(), pgmB() } ,
        {; pgmC(), pgmD() }
}
```

indicates that two blocks are to be executed concurrently; in one block, `pgmA()` and `pgmB()` are to be executed in sequence, while in the other block, `pgmC()` and `pgmD()` are to be executed in sequence.

Also, elements in a sequential or parallel composition can be guarded; for example,

```
{;
        X => 0 ->       pgmA() ,
        X == 0 ->       pgmB()
}
```

is equivalent to

```
{;
        {?
                X > 0 ->        pgmA()
        } ,
        {?
                X == 0 ->       pgmB()
        }
}
```

i.e., it indicates a sequence of two operations: First, if `X` is greater than zero, execute `pgmA ()`; otherwise, do nothing. Second, if `X` is equal to zero, execute `pgmB()`; otherwise, do nothing.

A.2 Variables

PCN supports two distinct categories of variables: single-assignment variables, referred to as *definition variables* or *definitional variables*, and multiple-assignment variables, referred to as *mutable variables* or *mutables*.

Definition variables can be assigned a value ("defined") at most once; initially they have a special "undefined" value, and a statement that requires the value of an undefined variable suspends until a value has been assigned. Mutable variables are like the variables of traditional imperative programming languages; they can be assigned a value any number of times, and their initial value is arbitrary.

Definition variables and mutable variables can be syntactically distinguished by the fact that mutables are declared with types and lengths (for arrays) as in a C program, while definition variables are not declared. A definition variable is assigned a type and a length at the same time it is assigned a value.

Definition variables can be scalars, arrays, or tuples; syntactic support is provided for lists as a special case of 2-tuples. The empty list, denoted `[]`, is often used as a null value for definition variables, used for example when the variable must be assigned a value for synchronization purposes but the particular value is not of interest.

A.3 Communication and synchronization

Communication and synchronization among concurrently-executing processes are accomplished via definition variables. For example, a single stream of messages between two processes can be represented as a shared definitional list whose elements correspond to messages.

A.4 Interface to other languages

PCN programs can call programs written in imperative languages such as C and Fortran; such programs are referred to as "foreign code". A definition variable may be passed to a foreign-code program for use as input; the foreign-code program does not begin execution until the definition variable has been assigned a value. Mutable variables may be passed to a foreign-code program for use as either input or output.

Appendix B

Implementation details: compiling, linking, and executing programs

This appendix presents additional PCN-specific information about compiling, linking, and executing programs using the prototype implementation. The reader is assumed to be familiar with PCN syntax and with how to compile, link, and execute PCN programs. An overview of PCN syntax appears in §A; complete information about PCN may be found in [5] and [9]. It is also assumed that the reader is familiar with how to compile programs in the data-parallel notation being used. (As described in §3.1 and §3.5, the prototype implementation may be used with a variety of data-parallel notations.)

Note that the information in this section is based on PCN Version 1.2 and may not be consistent with later versions.

B.1 Transforming and compiling programs

Support for distributed calls and for some parameter options for array creation and verification requires that the user's source program be preprocessed by a source-to-source transformation that constitutes part of the prototype implementation. The transformation is written in PTN (Program Transformation Notation) [7], which is part of the PCN environment. To preprocess and compile a program, use the command:

```
form(module_name, am_user_ptn:go())
```

where *module_name* is the module to be preprocessed and compiled. This command applies the source-to-source transformation `am_user_ptn:go()` to the programs in the module and compiles the result, producing the usual PCN compiler output files *module_name*`.pam` and *module_name*`.mod`. Additional forms of the `form` command are described in [9].

Because of the way local sections are handled in the implementation (described in detail in §5.1.5), warning messages of the following form may occur during compilation:

```
Error:  Definition variable Definition_variable is passed to
a mutable in module:calling_procedure calling module:called_procedure
```

where *calling_procedure* has the form `wrapper_`*number* or `wrapper2_`*number*. These messages can be ignored; the program still compiles and executes properly. The user may also ignore warning messages of the following form:

```
Warning:  Index is a singleton variable in module:procedure
```

where *procedure* has the form `wrapper_`*number* or `wrapper2_`*number*.

B.2 Linking in data-parallel code

Like other foreign-code programs, data-parallel programs must be linked into a custom PCN executable (runtime system) before they can be called by a PCN program. The procedure for linking in foreign code is straightforward and is described fully in [9] and briefly here. It requires object code for the foreign program and any associated libraries (e.g., of message-passing routines). Observe that since the requirement is for object code and not for source code, a data-parallel source program may be transformed into object code with a standard C or Fortran compiler, with an extended C or Fortran compiler, or with a combination of preprocessor and compiler.

For example, suppose module `simple_ex.pcn` contains the following directive:

```
-foreign(
        "C_simple/test_iprdv.o",
        "../Interface/General/cfull/cmatvecnd.$ARCH$.a",
        "../Interface/General/full/matvecnd.$ARCH$.a",
        "../Interface/General/att/attnd.$ARCH$.a"
        )
```

This directive indicates that references to foreign code in module `simple_ex.pcn` are to be satisfied using the object (`*.o`) and library (`*.a`) files listed. The symbol `$ARCH$` is replaced at link time with the appropriate architecture—e.g., `sun4`—thus allowing the creation of different executables for different architectures. (This would be appropriate, for example, for programs that execute on a combination of host computer and multicomputer nodes.) The PCN linker automatically replaces the suffix `.o` with the appropriate suffix for the target architecture—for example, `.o` for Sun 4 and `.s2010_o` for the Symult s2010 multicomputer.

After `simple_ex.pcn` has been transformed and compiled into `simple_ex.pam`, as described in §B.1, the following command creates a PCN executable that includes the foreign code needed by `simple_ex`:

```
pcncc -o simple_ex.pcnexe simple_ex.pam
```

The new executable (runtime system) is file `simple_ex.pcnexe`, and it is used instead of the standard runtime system (`pcn`) to execute programs in `simple_ex.pcn`.

B.3 Executing programs with the array manager

Before executing a program that uses distributed arrays, the array manager must be started on all processors. This can be done in one of two ways:

- From within the PCN runtime environment, issue the following command:

  ```
  load("am")@all
  ```

- In the program that will make use of the array manager, make the following call:

  `am_util:load_all("am",` *Done_variable*`)`

  where *Done_variable* is a definition variable that will be set when the array manager has been loaded on all processors.

The `load` command is described in more detail in [9]; the `load_all()` procedure is described in more detail in §C.3.

The prototype implementation also includes a version of the array manager that produces a trace message for each operation it performs. To load this version, replace `"am"` in the above command or procedure call with `"am_debug"`.

Appendix C

Implementation details: additional library procedures

In addition to the library procedures described in §4, the prototype implementation includes additional procedures that are used in the sample programs in §6 and that users may find helpful in writing their own programs. This appendix documents those procedures. Documentation conventions are as described in §4.1.1.

C.1 Creating a definitional array from a tuple

The following procedure creates a definitional array from a tuple of integers.

```
am_util:tuple_to_int_array(
        /*in tuple of int*/             Tuple,
        /*out array of int*/            Array
        )
```

Precondition:

- `TRUE`.

Postcondition:

- Length(`Array`) = length(`Tuple`).
- For $0 \leq i <$ length(`Tuple`),
  `Array[i]` = `Tuple[i]`.

C.2 Creating a patterned definitional array

The following procedure creates a definitional array of the following form:

$$(\mathit{first}, \mathit{first} + \mathit{stride}, \mathit{first} + 2 * \mathit{stride}, \ldots)$$

It is intended to be useful in creating arrays of processor (node) numbers.

```
am_util:node_array(
        /*in int*/                      First,
        /*in int*/                      Stride,
        /*in int*/                      Count,
        /*out array of int*/            Processors
        )
```

Precondition:

- `Count` ≥ 0.

Postcondition:

- Length(`Processors`) = `Count`.
- `Processors[0]` =`First`.
- For $0 \leq i <$ `Count`,
  `Processors[i+1]` =`Processors[i]` + `Stride`.

C.3 Loading a module on all processors

The following procedure loads a module on all processors; that is, it is equivalent to executing the PCN runtime command `load` (as described in [9]) on all processors. It can be used, for example, to start the array manager on all processors, as described in §B.3.

```
am_util:load_all(
        /*in string*/                   Server_name,
        /*out*/                         Done
        )
```

Precondition:

- There is a module `Server_name`.

Postcondition:

- `Server_name` is loaded on all processors, and `Done` = `[]`.

C.4 Printing debugging and trace information

Concurrently-executing uses of the usual PCN mechanisms for writing to standard output (`print` and `stdio:printf`) may produce interleaved output. The following procedure writes to standard output "atomically"—i.e., output produced by a single call to this procedure is not interleaved with other output.

```
am_util:atomic_print(
        /*in list*/                     To_print
        )
```

Precondition:

- `TRUE`. Elements in the input list `To_print` can be constants (e.g., character strings) or variables of any type, including tuples and arrays.

Postcondition:

- A line containing the elements of the list `To_print`, plus a trailing carriage return, has been generated and written to standard output "atomically". The line prints only after all definition variables referenced in the list `To_print` become defined.

  For example, if `X` has the value 1, the following procedure call:

  ```
  am_util:atomic_print(["The value of X is ", X, "."])
  ```

  prints the following line:

  ```
  The value of X is 1.
  ```

C.5 Reduction operator `max`

The following procedure finds the maximum of its inputs. It can be used, for example, to combine status or reduction variables as described in §4.3.1.

```
am_util:max(
        /*in*/          In1,
        /*in*/          In2,
        /*out*/         Out
        )
```

Precondition:

- `In1`, `In2` are numeric.

Postcondition:

- `Out` is the maximum of `In1` and `In2`.

Appendix D

Implementation details: adapting existing data-parallel programs

As described in §3.1, the prototype implementation was designed to allow reuse of existing data-parallel programs with at most minor modifications. This appendix presents a case study of adapting an existing library of data-parallel programs for use with the prototype implementation.

D.1 An example library of data-parallel programs

The data-parallel programs used to test the prototype implementation were obtained by adapting an existing library of data-parallel linear-algebra programs. These programs, supplied by Eric Van de Velde of the Applied Mathematics department at Caltech, were written by hand-transforming algorithms based on the data-parallel programming model into SPMD message-passing C, using the methods described in [23].

The programs comprise a library of linear-algebra operations, including the following:

- Creation of distributed vectors (1-dimensional arrays) and matrices (2-dimensional arrays).
- Basic vector and matrix operations on distributed vectors and matrices.
- More complex operations on distributed vectors and matrices, including LU decomposition, QR decomposition, and solution of an LU-decomposed system.

D.2 Adapting the library

§3.5 describes the requirements that must be met by data-parallel programs to be called from the prototype implementation. The following modifications were made to the example library to comply with those requirements:

SPMD implementation

No changes were required.

Relocatability

Explicit use of processor numbers was confined to the library's communication routines. These routines were modified, replacing references to explicit processor numbers with references to an array of processor numbers passed as a parameter to the called data-parallel program.

Compatibility of parameters

The data-parallel programs in the library accepted as parameters only data local to the processor; parameters could include references to distributed arrays. This model is compatible with the prototype implementation in most regards. However, the programs in the library represented a distributed array as a C data structure containing array dimensions and a pointer to the local section, and the local section of a multidimensional array was an array of arrays. The programs were modified to instead represent distributed arrays as "flat" local sections; this required revision of all references to multidimensional arrays and was tedious but straightforward.

Compatibility of communication mechanisms

Communication among concurrently-executing copies of the data-parallel programs in the library was based on sending and receiving untyped messages point-to-point using the Cosmic Environment (CE) [20] communication primitives. For some architectures (for example, the Symult s2010, which was used for testing the interface between the prototype implementation and the adapted data-parallel programs), PCN employs the same CE communication primitives. As described in §3.4.1, message conflicts in such an environment can be prevented by requiring PCN and the data-parallel programs to use distinct types of messages, say a "PCN" type and a "data-parallel-program" type. As described in §5.3, as part of the prototype implementation, the PCN runtime system was modified to use typed messages of a “PCN” type rather than untyped messages. The example library’s communication routines were similarly modified to use typed messages of a “data-parallel-program” type.

Language compatibility

Since the data-parallel programs were written in C, with communication performed by calls to the CE communication routines, they could be called from PCN without change. However, some C preprocessor `#define` directives were changed to avoid conflicts with similar directives in the prototype implementation.

Appendix E

Implementation details: required files and installation procedure

This appendix describes the files that comprise the prototype implementation and how to install them in conjunction with PCN Version 1.2.

E.1 Obtaining the software

PCN may be obtained by anonymous FTP from Argonne National Laboratory, as described in [9]. Note that the prototype implementation is based on PCN Version 1.2 and may not be compatible with later versions of PCN.

The additional files described in this appendix, plus a `README` file detailing installation procedures, can be obtained by sending electronic mail to the author at `berna@vlsi.caltech.edu`.

E.2 Required files

This section describes the files that comprise the implementation. Files with the suffix `.pcn` contain PCN source code; files with the suffix `.ptn` contain source code for PTN source-to-source transformations; files with the suffix `.c` contain C source code; and files with the suffix `.h` contain C preprocessor directives (to be incorporated into one or more of the other files via an `#include` directive).

The files are divided into two categories: modified versions of files that are part of the standard PCN implementation, and added files.

E.2.1 Modified files

The prototype implementation of distributed arrays requires support for the `build` and `free` primitives, as described in §5.1.6. This support is contained in the following files:

- Modified compiler files:
  `co_ass.pcn`, `co_en.pcn`, `co_instr.pcn`, `pcnmod.h`, `ptn_mass.pcn`, `ptn_pp.pcn`.
- Modified emulator (runtime system) files:
  `debug.c`, `defs.h`, `emulate.c`, `gc.c`, `macros.h`, `utils.c`.

As described in §5.3, additional minor modifications to the PCN implementation may be required, depending on the architecture and on the communication mechanisms used by the data-parallel programs to be called. The prototype implementation does not attempt to be completely general in this regard; it includes only those modifications required to support calling the example library of data-parallel programs described in §D on the Symult s2010. These modifications are contained in the following files:

- Modified emulator files:
  `sr_ce.c`, `sr_ce.h`.

E.2.2 Added files

The following added files contain the library procedures described in §4 and §C and all supporting code, including the source-to-source transformations described in §5 and §F:

- `am.pcn`—main module for the non-debug version of the array manager, containing a capabilities directive and a top-level server program.
- `am_create.pcn`—procedures to process `create_array` requests.
- `am_debug.pcn`—main module for the debug version of the array manager (described in §B.3), containing a capabilities directive and a top-level server program.
- `am_find.pcn`—procedures to process `find_local` and `find_info` requests.
- `am_free.pcn`—procedures to process `free_array` requests.
- `am_main.pcn`—main server program for the array manager, called from the top-level server programs in `am.pcn` and `am_debug.pcn`.
- `am_rw.pcn`—procedures to process `read_element` and `write_element` requests.
- `am_user.pcn`—distributed-array and distributed-call library procedures, as described in §4.
- `am_util.pcn`—utility and support procedures, including those described in §C.
- `am_vfy.pcn`—procedures to process `verify_array` requests.

- `am.h`—preprocessor macros.
- `am_border_parm_ptn.ptn`—source-to-source transformation for `foreign_borders` option of procedures `am_user:create_array` and `am_user:verify_array`.
- `am_distrib_call_ptn.ptn`—source-to-source transformation for distributed calls.
- `am_user_ptn.ptn`—top-level source-to-source transformation; it calls the transformations in `am_border_parm_ptn.ptn` and `am_distrib_call_ptn.ptn`.
- `am_util_ptn.ptn`—utility and support procedures for the source-to-source transformations.
- `am_sys.c`—additional C utility and support procedures; these procedures perform functions that are difficult or cumbersome in PCN.
- `am_sys.pcn`—PCN interfaces to the procedures in `am_sys.c`.

E.3 Installation procedure

The installation procedure for the prototype implementation is described in detail in a `README` file, which can be obtained as described in §E.1. Briefly, the steps in installing our implementation are as follows:

1. Install PCN Version 1.2 in a local PCN installation directory *install_directory*.
2. Copy the array manager files into *install_directory*.
3. Edit the array manager `Makefile` for the desired configuration(s) (`sun4` and/or `sun4.ce`) to specify *install_directory*.
4. Install the array manager in *install_directory* by performing compiles and links with the provided script and `make` files.
5. Run the provided tests to verify proper installation.

Appendix F

Implementation details: transformation for distributed calls

§5.2 presents an overview and examples of the source-to-source transformation used to support distributed calls. This appendix gives a detailed specification for the transformation. Documentation conventions are as described in §4.1.1, and values for status variables are as described in §4.1.2.

F.1 The transformed call

Every program call in the input module of the following form:

```
    am_user:distributed_call(Processors,
                Module, Program,
                Parms,
                Combine_status_module, Combine_status_program,
                Status)
```

is converted into a block of the following form:

```
    {||
                am_util:do_all(Processors,
                            Current_module, Wrapper_program,
                            Wrapper_actual_parms,
                            Current_module, Combine_program,
                            Combine_result) ,
                data(Combine_result) ->
                Status = Combine_result[0] ,
            additional_statements_for_reduction_variables
    }
```

where *Current_module* : *Wrapper_program* (the first-level wrapper program) and *Current_module* : *Combine_program* (the combine program) are inserted programs, as described in §F.3 and §F.6 respectively, and *Wrapper_actual_parms* and *additional_statements_for_reduction_variables* are as described in §F.2 and §F.5 respectively.

F.2 The parameters

Recall from §4.3.1 that a call to `am_user:distributed_call` uses parameter *Parms* to specify the parameters to be passed to the data-parallel program. Define the following notation for *Parms*:

- Write *Parms* as $\{P_1, P_2, \ldots, P_n\}$.
- Let $A_1, A_2, \ldots, A_m$ be the distributed arrays such that $P_i = \{$`"local"`$, A_j\}$ for some i and j, i.e., the distributed arrays corresponding to local-section parameters.
- Let $V_1, V_2, \ldots, V_p$ be the variables such that $P_i = \{$`"reduce"`$, \mathit{Type}_k, \mathit{Lng}_k, \mathit{Mod}_k, \mathit{Pgm}_k, V_k\}$ for some i and k, i.e., the reduction variables.

To define the actual parameters for the wrapper program (variable *Wrapper_actual_parms* in §F.1) and the parameters to pass to the data-parallel program (line (w2) in §F.4), begin by defining $\mathit{WA}_1, \ldots, \mathit{WA}_n$ and $\mathit{NP}_1, \ldots, \mathit{NP}_n$ as follows:

- If $P_i = \{$`"local"`$, A_j\}$:
  - $\mathit{WA}_i = A_j$
  - $\mathit{NP}_i = \mathit{LS}_j$, the local section of A_j
- else if $P_i =$ `"index"`:
  - $\mathit{WA}_i =$ `_`, the PCN placeholder dummy variable
  - $\mathit{NP}_i =$ `Index`, an input integer that represents an index into *Processors*, passed by `am_util:do_all` to the wrapper program
- else if $P_i =$ `"status"`:
  - $\mathit{WA}_i =$ `_`, the PCN placeholder dummy variable
  - $\mathit{NP}_i =$ `local_status`, a mutable integer that must be set by *Module* : *Program*
- else if $P_i = \{$`"reduce"`$, \mathit{Type}_k, \mathit{Lng}_k, \mathit{Mod}_k, \mathit{Pgm}_k, V_k\}$:
  - $\mathit{WA}_i =$ `_`, the PCN placeholder dummy variable
  - $\mathit{NP}_i = \mathit{local}_k$, a mutable of type Type_k and length Lng_k that must be set by *Module* : *Program*

- else:
  - $WA_i = P_i$
  - $NP_i = P_i$

Define *ProgramID* as follows:

- If $Module$ = `[]`:
  - $ProgramID = Program$
- else:
  - $ProgramID = Module : Program$

Define *Wrapper_actual_parms*, as mentioned in §F.1, as follows:

$$Wrapper\_actual\_parms = \{\{WA_1, \ldots, WA_n\}, Lng_1, \ldots, Lng_p\}$$

where the Lng_k's are the lengths specified for the reduction variables.

F.3 The first-level wrapper program

Now define *Current_module* : *Wrapper_program* as follows:

```
Wrapper_program(
        /*in int*/                      Index,
        /*in tuple*/                    Parms,
        /*out tuple*/                   Combine_result
        )
{?
        Parms ?= {WAparms, L_1, ..., L_p} ->
                Wrapper2_program (Index, WAparms, Combine_result, L_1, ..., L_p) ,
        default ->
                Combine_result = STATUS_INVALID
}
```

where *Wrapper2_program* (the second-level wrapper program) is another inserted program in module *Current_module*, as described in §F.4.

F.4 The second-level wrapper program

Define *Current_module* : *Wrapper2_program* as follows:

```
$Wrapper2\_program$(
        /*in int*/                      Index,
        /*in tuple*/                    Parms,
        /*out tuple*/                   $Combine\_result$,
        /*in int*/                      $L_1$, ..., $L_p$
        )
int local_status ;                                                  (w1)
$Type_1$ $local_1$[$L_1$] ;
...
$Type_p$ $local_p$[$L_p$] ;
{?
        Parms ?= {$M_1$, ..., $M_n$} ->
        {||
                am_user:find_local($M_{j_1}$, $LS_1$, $S_1$) ,
                ...
                am_user:find_local($M_{j_m}$, $LS_m$, $S_m$) ,
                {?
                        $S_1$ == STATUS_OK, ..., $S_m$ == STATUS_OK,
                                data($LS_1$), ..., data($LS_m$) ->
                        {;
                                $ProgramID$($NP_1$, ..., $NP_n$) ,        (w2)
                                make_tuple($p+1$, $Combine\_result$) ,
                                $Combine\_result$[0] = local_status ,
                                                                    (w3)
                                $Combine\_result$[1] =  $local_1$ ,
                                ...
                                $Combine\_result$[$p$] =  $local_p$
                        } ,
                        default ->
                                $Combine\_result$  = STATUS_INVALID
                } ,
        } ,
        default ->
                $Combine\_result$  = STATUS_INVALID
}
```

If there is no i such that NP_i = `local_status`, line (w1) is omitted, and in line (w3), *Combine_result*[0] is defined to be STATUS_OK.

F.5 The transformed call, continued

Define the *additional_statements_for_reduction_variables* mentioned in §F.1 as follows:

If $p = 0$, there are none; if $p > 0$, there is one additional statement of the following form for each k such that $1 \leq k \leq p$:

```
data(Combine_result) ->
        V_k = Combine_result[k]
```

F.6 The combine program

For each p such that $1 \leq k \leq p$, define

$PgmID_k = Mod_k : Pgm_k$

Define *Status_PgmID* as follows:

- If *Combine_status_module* $\neq$ `[]`:
  - *Status_PgmID* = *Combine_status_module* : *Combine_status_program*
- else:
  - *Status_PgmID* = `am_util:max`

Define *Current_module* : *Combine_program* as follows:

```
Combine_program(
        /*in tuple*/                    C_in1,
        /*in tuple*/                    C_in2,
        /*out tuple*/                   C_out
        )
{?
        tuple(C_in1), tuple(C_in2),
                length(C_in1) == p+1, length(C_in2) == p+1  ->
        {||
                make_tuple(p+1, C_out) ,
                data(C_out) ->
                        Status_PgmID(C_in1[0], C_in2[0], C_out[0]) ,
                data(C_out) ->
                        PgmID_1(C_in1[1], C_in2[1], C_out[1]) ,
                ...
                data(C_out) ->
                        PgmID_p(C_in1[p], C_in2[p], C_out[p])
        } ,
        default ->
                C_out = STATUS_INVALID
}
```

Bibliography

[1] G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[2] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual.* Springer Verlag, 1983. ANSI/MIL-STD-1815A-1983.

[3] W.C. Athas and C.L. Seitz. The Cantor user report, version 2.0. Technical Report 5232, Computer Science Department, California Institute of Technology, January 1987.

[4] K.M. Chandy and C. Kesselman. The CC++ language definition. Technical Report CS-TR-92-02, California Institute of Technology, 1992.

[5] K.M. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, 1991.

[6] E.J. Cramer, P.D. Frank, G.R. Shubin, J.E. Dennis, and R.M. Lewis. On alternative problem formulations for multidisciplinary design optimization. Technical Report TR92-39, Rice University, December 1992.

[7] I.T. Foster. Program Transformation Notation: A tutorial. Technical Report ANL-91/38, Argonne National Laboratory, Mathematics and Computer Science Division, 1991.

[8] I.T. Foster and K.M. Chandy. FORTRAN M: A language for modular parallel programming. Technical Report MCS-P327-0992, Mathematics and Computer Science Division, Argonne National Laboratory, October 1992.

[9] I.T. Foster and S. Tuecke. Parallel programming with PCN. Technical Report ANL-91/32, Argonne National Laboratory, 1991.

[10] I.T. Foster, S. Tuecke, and S. Taylor. A portable run-time system for PCN. Technical Report ANL/MCS-TM-137, Argonne National Laboratory, 1991.

[11] G.C. Fox. What have we learnt from using real parallel machines to solve real problems? In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 897–955. ACM, 1988.

[12] G.C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, December 1990.

[13] P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming on MIMD Computers.* The MIT Press, 1991.

[14] P.J. Hatcher, M.J. Quinn, A.J. Lapadula, B.K. Seevers, R.J. Anderson, and R.R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):377–383, 1991.

[15] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, 1992 (revised Jan. 1993).

[16] W.D. Hillis and G.L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[17] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[18] M. Metcalf and J. Reid. *Fortran 90 Explained.* Oxford Science Publications, 1990.

[19] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1988.

[20] C.L. Seitz, J. Seizovic, and W.-K. Su. The C programmer's abbreviated guide to multicomputer programming. Technical Report CS-TR-88-01, Computer Science Department, California Institute of Technology, January 1988.

[21] Thinking Machines Corporation. *CM Fortran Reference Manual*, September 1989. Version 5.2-0.6.

[22] J. Thornley. Parallel programming with Declarative Ada. Technical Report CS-TR-93-03, Computer Science Department, California Institute of Technology, 1993.

[23] E.F. Van de Velde. Concurrent scientific computing. Draft, California Institute of Technology, 1993. To be published by Springer-Verlag.