Parallel Programming with PCN

Ian Foster Steven Tuecke

Version 2.0: January 15, 1993

PCN is a system for developing and executing parallel programs. It comprises a high-level programming language, tools for developing and debugging programs in this language, and interfaces to Fortran and C that allow the reuse of existing code in multilingual parallel programs. Programs developed using PCN are portable across many different workstations, networks, and parallel computers.

This document provides all the information required to develop parallel programs with the PCN programming system. It includes both tutorial and reference material. It also presents the basic concepts that underlie PCN, particularly where these are likely to be unfamiliar to the reader, and provides pointers to other documentation on the PCN language, programming techniques, and tools.

PCN is in the public domain. The latest version of both the software and this manual can be obtained by anonymous ftp from Argonne National Laboratory in the directory `pub/pcn` at `info.mcs.anl.gov` (cf. Appendix A).

This version of this document describes PCN version 2.0, a major revision of the PCN programming system. It supersedes earlier versions of this report.

Preface

The PCN system is the product of the efforts of many people at Argonne National Laboratory, the California Institute of Technology, and the Aerospace Corporation, including Sharon Brunett, Mani Chandy, Ian Foster, Steve Hammond, Carl Kesselman, Tal Lancaster, Dong Lin, Jan Lindhiem, Robert Olson, Steve Taylor, and Steven Tuecke. The Upshot trace analysis tool was provided by Ewing Lusk. The expanded BNF syntax for PCN was provided by John Thornley. The two-point boundary value application was provided by Steve Wright.

This work was supported in part by the National Science Foundation under Contract NSF CCR-8809615, by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the Air Force Office for Scientific Research under Contract AFOSR-91-0070, by the Office for Naval Research under Contract ONR-N00014-89-J-3201, and by the Defense Advanced Research Projects Agency under Contract DARPA-N00014-87-K-0745.

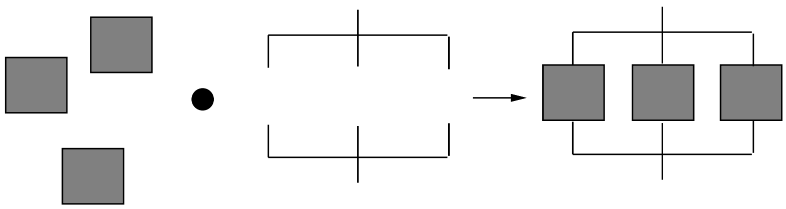
Contents

Part I
A Tutorial Introduction

1 Program Composition

Program Composition Notation (PCN) is both a programming language and a parallel programming system. As the name suggests, both the language and the programming system center on the notion of *program composition.*

Most programming languages emphasize techniques used to develop individual components (blocks, procedures, modules). In PCN, the focus of attention is the techniques used to put components together (i.e., to compose them). This is illustrated in the following figure, which shows a combining form being used to compose three programs.



This focus on combining forms is important for several reasons. First, it encourages reuse of parallel code: a single combining form can be used to develop many different parallel programs. Second, it facilitates reuse of sequential code: parallel programs can be developed by composing existing modules written in languages such as Fortran and C. Third, it simplifies development, debugging, and optimization, by exposing the basic structure of parallel programs.

It appears likely that a large proportion of all parallel programs can be developed with a relatively small number of combining forms. However, PCN does not attempt to enumerate potential combining forms. Instead, it provides a core set of three primitive composition operators — parallel, sequential, and choice composition — in a *core programming notation.* This is a simple, high-level programming language. More sophisticated combining forms (providing, for example, divide-and-conquer, self-scheduling, or domain decomposition strategies) can be implemented as user-defined extensions to this core notation. Such extensions are referred to as *templates* or *user-defined composition operators.* Program development, both with the core notation and with templates, is supported by a *portable toolkit.* These three components of the PCN system are illustrated in Figure 1.

This tutorial provides a detailed description of the core programming notation and toolkit, and an introduction to the use of templates in parallel programming.

1.1 Core Programming Notation

The core PCN programming notation is a simple, high-level language that provides three basic composition operators: *parallel*, *sequential*, and *choice*. The

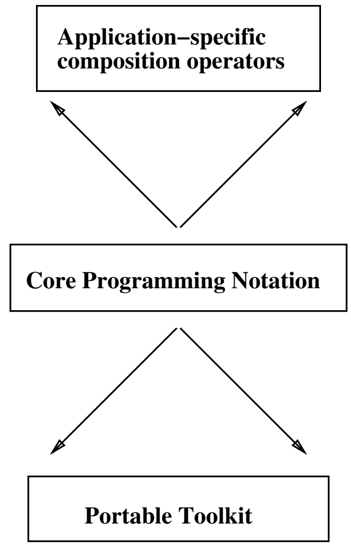


Figure 1: PCN System Structure

language provides two types of variable: conventional, or *mutable* variables, and single-assignment, or *definitional* variables. Other distinctive features of the language include extensive use of recursion, support for both numeric and symbolic computing, and an interface to sequential languages such as Fortran and C. The syntax is similar to that of C.

1.2 Toolkit

The PCN toolkit provides support for each stage of the parallel program development process. It comprises a compiler, linker, foreign language interface, standard libraries, process mapping tools, programmable transformation system, symbolic debugger, execution profiler, and trace analyzer. These facilities are all machine independent and can run on a wide variety of uniprocessors, multiprocessors, and multicomputers. They are supported by a *run-time system* that provides basic machine-dependent facilities.

Compiler The compiler translates PCN programs to a machine-independent, low-level form (PCN object code). An interface to the C preprocessor allows macros, conditional compilation constructs, and the like to be used in PCN programs.

Linker and Foreign Language Interface The PCN linker combines PCN object code (i.e., PCN compiler output), foreign object code that is called from PCN (i.e., C or Fortran compiler output), libraries, and the PCN run-time system into a single executable program. This permits C and Fortran procedures to be integrated seamlessly into PCN programs, and PCN programs to be executed similar to programs written in other languages.

Standard libraries A set of standard libraries provides access to Unix facilities (e.g., I/O) and other capabilities.

Virtual Topology tools These support process mapping on a variety of virtual machines, and templates for writing reusable parallel code.

PDB PDB is the PCN symbolic debugger. It includes specialized support for debugging of concurrent programs.

Gauge Gauge is an execution profiler for programs written in PCN and other languages. It includes run-time system support for collecting and saving profiles, and an X windows based graphical tool for interactive exploration of profile data.

Upshot Upshot is a trace analysis tool for programs written in PCN and other languages. It includes run-time system support for collecting and saving traces, and an X windows based graphical tool for interactive exploration of trace data.

1.3 Cross Reference

The basic constructs of the PCN language are described in the following sections.

- Syntax: § 4.2 and Appendix F.
- Sequential composition: § 4.3.
- Mutable Variables: § 4.3.
- Parallel composition: § 4.4.
- Definitional Variables: § 4.4.
- Choice composition: § 4.5.

The components of the PCN toolkit are described in the following sections.

- Compiler: § 3.1, § 7.
- Foreign interface and linker: § 8.
- Process mapping tools: § 10.
- Templates: § 12.
- Debugging facilities: § 14.
- PDB: § 15.

- Gauge: § 16.
- Upshot: § 17.
- Standard libraries: § 18.

Machine-specific aspects of the PCN toolkit are described in §§ 19–23. Additional documentation on the PCN language, toolkit, and applications is cited in § 24.

The `host-control` program, a utility for managing execution of PCN programs on networks, is described in a separate document. See § 24 for more information.

2 Getting Started

We assume that PCN is already installed on your computer. (If it is not, read the documentation provided with the PCN software release.) You will need to know *where* PCN is installed. Normally, this will be `/usr/local/pcn`, but some systems may place PCN in a different location.

Before you can use PCN, you must tell your Unix environment where to find the PCN software. If you are using the standard Unix C-shell (csh), you add one line to the *end* of the file `.cshrc` in your home directory. If PCN has been installed in `/usr/local/pcn`, this line is

```
set path = ($path /usr/local/pcn/bin)
```

The environment variable `path` tells the Unix shell where to find the various PCN programs (compiler, linker, etc.). This shell command adds the directory containing the various PCN executables to your shell's search path.

If you use either the Bourne shell (sh) or the Korn shell (ksh) then you will need to add the following commands to the end of the file `.profile` in your home directory.

```
PATH=$PATH:/usr/local/pcn/bin
export PATH
```

You may have to log out and log in again for these changes to take effect.

3 An Example Program

We are now ready to compile and run our first PCN program. The syntax of PCN is similar to that of the C programming language in many respects. Hence, it is appropriate that our first program print "Hello world." (the first C program in several well-known texts does just this).

Module program1.pcn

```
main(argc,argv,exit_code)
{;  stdio:printf("Hello world.\n", {}, d),
    exit_code = 0
}
```

A PCN program consists of one or more modules. Each module is contained in a separate file with a .pcn suffix. Our example program consists of a single module, program1, contained in a file program1.pcn. (We'll learn more about modules later.)

This example program has one procedure, main. Its three arguments are the number of command line arguments (argc), a list of those arguments (argv), and a variable to be used for a return code (exit_code). These are described in more detail in § 3.4.

This procedure makes what is called an *intermodule call*: it calls the printf procedure in the stdio module to print "Hello world ." The stdio module is distributed with the PCN system; it provides many of the functions of the Unix "standard I/O" library (§ 18.2).

3.1 Compiling a Program

The PCN compiler, pcncomp, is used to compile a PCN module. In general, the PCN compiler is invoked very much like most Unix based C and Fortran compilers.

Because our program is contained in a file program1.pcn, we type

```
pcncomp -c program1.pcn
```

When compiling program1.pcn, the PCN compiler will produce the file program1.pam. The .pam file contains PCN object code. These .pam files are analogous to the .o files that are produced by most C and Fortran compilers. However, unlike .o object files, the .pam files are completely machine independent: the PCN object code that is compiled for one machine will work on any other machine without recompilation.

3.2 Linking a Program

Now that we have program1.pam which contains the PCN object code for our example program, we must use the PCN linker to combine this .pam file with libraries of procedures such as the stdio module and with the PCN run-time system. The result of running the PCN linker will be an executable program.

The PCN linker will be run by pcncomp when the -c argument is not passed to pcncomp. This is the same convention as is used in most C and Fortran compilers to invoke the Unix linker (ld).

To link the example program, myprogram, we type

```
pcncomp program1.pam -o myprogram -mm program1
```

As with most C and Fortran compilers, the `-o` specifies that the name of the executable produced by the linker will be named `myprogram`. For the moment, ignore the `-mm program1` flag. This will be descussed in (§ 3.4).

The PCN linker is relatively slow. During program development you may wish to use PCN's *dynamic loading* capability. This feature allows you to avoid having to relink a program when PCN files are changed. It is described in § 15.9.

For information on linking PCN programs that call C or Fortran procedures, see § 8. For more information on using the PCN compiler and linker in general, see § 25.

3.3 Running a Program

To execute a PCN program, you can just run it like any other program. For example, to run `myprogram` you would type the following, where `%` is the Unix shell prompt:

```
% myprogram
Hello world.
%
```

In the subsequent examples, text typed by the user is written in *italic* and program output in `roman`.

Command line arguments can be passed to PCN programs as they would be to a C program. For example, the following program prints out the first few command line arguments.

Module `program2.pcn`

```
main(argc, argv, exit_code)
{?  argc == 3, argv ?= [ a1, a2, a3 ] ->
       {;  stdio:printf("%s\n%s\n%s\n", {a1, a2, a3}, _),
           exit_code = 0
       },
    default -> {; exit_code = 1 }
}
```

After this program is compiled as described above, it can be run as follows:

```
% program2 arg2 "another arg"
program2
arg2
another arg
%
```

The PCN run-time system has a number of run-time configurable parameters that can be controlled by command line arguments. In order to keep these run-time system's arguments from interfering with the program's arguments, all arguments up to but not including the first `-pcn` argument will be passed to the program. All arguments after the `-pcn` argument will be passed to the run-time system. For example, suppose you run a PCN program as follows:

```
my_program my_arg1 my_arg2 -pcn -n 2
```

This would cause `my_arg1` and `my_arg2` to be passed to the PCN program, and `-n` and `2` to the run-time system.

A complete list of these run-time system parameters, and a brief description of their meaning, can be obtained by using the `-h` argument, for example:

```
my_program -pcn -h
```

3.4 The main() Procedure

Every program must have an entry point. This is the procedure that is initially called when a program is executed. In PCN, this entry point is modeled after C.

By default, the following PCN procedure is called when a PCN program is executed:

```
main:main(argc, argv, exit_code)
```

where:

- **argc** is the argument count (an integer), as in C.
- **argv** is a list of the arguments. As in C, each argument is a string. A list is a PCN data type, which is described in § 4.8.
- **exit_code** is undefined. The program should set this to an integer exit status before terminating.

As with C programs, this exit status can be used by a Unix shell script or makefile to determine whether execution "succeeded" (exit_code = 0) or "failed" (exit_code $\neq 0$).

By default, a procedure named `main`, in a module named `main`, will be called to start execution of the program. An alternate main module and/or main procedure can be specified when linking by using the `-mm` and `-mp` flags, respectively. For example, if you wish `p:my_main(...)` to be the entry point, you might link your program by running

```
pcncomp p.pam -o my_program -mm p -mp my_main
```

The `-mm p` flag says to use `p` as the main module, and the `-mp my_main` flag says to use `my_main` as the main procedure.

When running on multiple PCN nodes, this main procedure is only called on node 0. It is the responsibility of the PCN program to run procedures on other nodes. This issue is discussed in section § 10.

Getting started with PCN:

- PCN programs are contained in files with a `.pcn` suffix; compilation produces a file with a `.pam` suffix. This `.pam` file contains the PCN object code for this program, and is machine independent.
- We compile programs by running `pcncomp -c file.pcn`.
- We link programs by running `pcncomp file.pam -mm file -o program`.
- PCN programs are executed just like other programs.
- The default entry point to a PCN program is the `main()` procedure in the `main` module (i.e., `main.pcn`). The `-mm` and `-mp` linker arguments can be used to change the main module and main procedure, respectively.
- Command line arguments that come before the `-pcn` argument are passed to the PCN program via the `argc` and `argv` arguments to the main procedure. Arguments after the `-pcn` argument are passed to the PCN run-time system.
- The last argument to the main procedure should be set to an integer exit status by the program.

4 The PCN Language

The programming language Program Composition Notation (PCN) is an integral part of the PCN programming system: it is used to express concurrent algorithms and to compose code written in sequential languages. Like any programming language, PCN has a distinct syntax that must be mastered in order to write programs. However, the key to understanding PCN is understanding the concurrent programming model that it implements. Before presenting the PCN language, we introduce this model and the fundamental concurrent programming concepts on which it is based.

4.1 Concurrent Programming Concepts

Parallel programming is often considered "hard." However, experience shows that programming models that adhere to the following principles can significantly reduce the complexity of parallel programming.

First-Class Concurrency: Concurrent execution should be a first-class citizen in a programming model, not something appended to a sequential model.

Controlled nondeterminism: The result computed by a procedure should be fully determined by the procedure's inputs, except when explicitly specified otherwise by the programmer.

Compositionality: It should be easy to understand both isolated program components and larger programs formed by the concurrent composition of these components.

Mapping independence: The way in which components of a concurrent computation are mapped to a parallel computer should not change the result computed.

PCN uses four simple ideas to realize a parallel programming model based on these principles. *Definitional variables* provide an abstract, machine-independent model of both communication and synchronization. *Concurrent composition* is the fundamental mechanism used to build up complex programs from simpler components. *Nondeterministic choice* is used to specify nondeterministic actions when required. *Encapsulation of state change* allows state change to be integrated into concurrent computations without compromising deterministic execution.

Definitional Variables. A single mechanism is provided for the exchange of information between concurrently executing program components (processes): the definitional variable. A definitional variable is initially undefined, can be assigned at most a single value, and subsequently cannot change. A process that requires the value of a definitional variable waits (suspends) until the variable is defined. If a process tries to assign a value to a definitional variable that is already defined, a run-time warning will be generated, and the assignment will fail.

Definitional variables can be used both to communicate values and to synchronize actions. If two concurrent processes, a producer and a consumer, share a definitional variable, then a value provided by the producer for this variable is automatically communicated to the consumer. Execution of the consumer is blocked until the value is provided.

The definitional variable has several benefits for concurrent programming. First, it avoids the nondeterminism that is so often associated with concurrency: choices made within program components on the basis of definitional variables cannot change. This means that components can be understood in isolation, as errors caused by time-dependent interactions cannot arise. Second, shared definitional variables provide a clearly defined and delineated interface between concurrently executing processes: interaction can occur only if processes share variables. Third, the definitional variable provides for mapping independence: processes sharing a definitional variable may interact irrespective of their location in a parallel computer.

Concurrent Composition. Complex programs are developed by the concurrent composition of simpler components. Hence, an application can be viewed as consisting of a (potentially large) number of lightweight execution threads. These execute concurrently, communicate via definitional variables, and block when required data is unavailable.

It is often desirable that the number of threads be larger than the number of processors, as this can allow the compiler and run-time system to adopt flexible scheduling strategies that overlap computation and communication, thus masking latency and improving parallel efficiency.

Nondeterministic Choice. The use of definitional variables as a communication mechanism avoids errors arising from time-dependent interactions: a choice made on the basis of a definitional variable cannot change. Hence, concurrent computations are deterministic. This is an important property that greatly simplifies parallel programming.

Nevertheless, it is sometimes useful to be able to specify nondeterministic execution, particularly in reactive applications. Nondeterminism is integrated into the programming model in a tightly controlled way. A form of guarded command is used to define the conditions under which a process may perform various actions. Only if the conditions associated with two or more actions are not mutually exclusive is execution nondeterministic.

Encapsulation of State Change. The familiar concepts of state change and sequencing that underlie sequential languages such as Fortran and C are also important in parallel programming: many algorithms are most efficiently specified in these terms. However, state change must be carefully controlled if we are to avoid introducing unwanted nondeterminism.

The approach adopted in PCN is to insist that state change be encapsulated within sequential threads. Data structures that may be subject to state change cannot be shared by concurrently executing program components. This restriction prevents concurrent updates to state, which in turn avoids the possibility of time-dependent behavior.

Programming Model Summary. Execution of a parallel program forms a set of concurrently executing lightweight processes (threads) which communicate and synchronize by reading and writing shared definitional variables. Individual threads may apply the usual sequential programming techniques of state change and sequencing. Execution is deterministic, unless specialized operators are invoked to make nondeterministic choices.

Key concurrent programming concepts:

- Definitional variables
- Concurrent composition
- Controlled nondeterministic choice
- Encapsulation of state change

4.2 PCN Syntax

The syntax of PCN is modeled on that of the C programming language. In addition, the C preprocessor is applied to programs, so macros, conditional compilation, and file inclusion constructs can be used as in C (§ 7). In the following, we make frequent reference to C when explaining features of PCN. However, these references are for illustrative purposes only, and a familiarity with C is not required to understand this material. A complete BNF grammar for the PCN syntax is provided in Appendix F.

Data Types. PCN's three simple data types — character, integer, and double-precision floating-point number (`char`, `int`, and `double`) — are as in C. One-dimensional arrays of these data types are also supported. Arrays are indexed from zero, as in C. There is also a complex data type, the tuple. This is introduced in § 4.8. A distributed variant of the tuple, the port, is described in § 11.

Constants. PCN uses the same character, integer, double precision floating point, and string constant conventions as ANSI C. Please consult your favorite ANSI C reference (e.g. *The C Programming Language*, Second Edition, Kernighan and Ritchie, 1988, pp. 193-194) for more specifics on these conventions.

Strings. Strings are represented as character arrays, as in C. A character array A representing a string S of length k contains the ASCII representation of the characters of S in $A[0]..A[k-1]$ and the null character (`\0`) in $A[k]$. A constant string is denoted by the characters of the string between quotes; for example, `"PCN"` is a string consisting of the three characters: `P`, `C`, and `N` (followed by the null character). The empty string is denoted by `""`.

Expressions. Arithmetic expressions are as in C, except that the only operators are modulus, addition, subtraction, multiplication, and division ($\%$, $+$, $-$, $*$, and $/$). The `length` function returns the number of elements in an array or 1 (one) if applied to a single number or character. User defined functions (see below) can also be called, except in guard expressions. The following are all valid expressions.

```
(1 + x)%y     i * length(g)     29 - x/g
```

Operator precedence and associativity are as in C. The following table summarizes precedence and associativity rules. Operators on the same line have the same precedence, while rows are in order of decreasing precedence. Parentheses () can be used to override these default rules.

Operators	Associativity
− *(negation of numbers)* `length`	right to left
∗ / %	left to right
+ −	left to right

Variable Names. Variable names are as in C. A variable name is a character string formed from the set {`a-z,A-Z,0-9,_`} and starting with a letter or an underscore ("_"). Case is significant and there is no maximum length. The following are all valid variable names.

`value`   `_2`   `Last_Item`   `x`

See Appendix C for a list of reserved words that cannot be used as variable names.

Comments. A comment begins with /* and ends with */, as in C.

Procedures. A procedure definition consists of a heading followed by a declaration section followed by a block. The *heading* is the procedure name and a list of arguments (i.e., formal parameters), as in C. All arguments are passed by reference, unlike in C where arguments can be passed by value. The *declaration section* is a set of declarations for arguments and local variables. The scope of a variable is the procedure in which it appears: all variables appearing in a procedure are either arguments or local variables of the procedure. In particular, there is no notion of a global variable.

The body of a procedure consists of a composition of *blocks*. The block is the basic component from which procedures are constructed. A block is either a composition, an assignment statement, a definition statement, an implication, or a procedure call. These constructs will be defined shortly.

Functions. A function consists of the keyword `function` followed by a function definition. A function definition has the same syntax as a procedure definition, except that it may include calls to the primitive `return(r)` to specify a return value, `r`. The return value of a function must be a definitional variable. Functions cannot be used within guards.

Delimiters The blocks within a composition must be separated by either a comma (,) or a semicolon (;). In addition, trailing delimiters (i.e., delimiters after the last block in a composition) are legal.

Declarations. A declaration consists of a type (`char`, `int`, or `double`) followed by one or more variable names, each with an optional suffix to denote an array. An array suffix for a local variable has the form `[`*size*`]`, where *size* is an integer, a constant integer expression, or a variable from the procedure's argument list (i.e., the array size will be determined at run-time). An array suffix for a variable that is one of the procedure's arguments has the form `[]`. The following are all valid declarations.

```
int a[size];    double b[10], c[], d;    char c;
```

We shall see that declarations are not provided for all variables: the definitional variables used in PCN for communication and synchronization are distinguished by a lack of declaration.

4.3 Sequential Composition and Mutable Variables

We now explore the PCN language proper. We shall view PCN as providing three related sets of constructs. First, there are the *composition operators* — parallel, sequential, and choice — which encode three fundamental ways of putting program components together. Second, there are two types of *variables*: conventional or mutable variables , and single-assignment or definitional variables. Third, there are specialized language features introduced to support *symbolic processing*: tuples and recursion.

We first introduce the two components that will be most familiar to many readers: sequential composition and mutable variables.

The *sequential composition* operator is used to specify that a set of statements should be executed sequentially, in the order written in the program. In languages such as Fortran and C, this is of course the normal mode of execution. However, as PCN also allows for other sorts of composition, we distinguish it by a special syntax. A sequential composition has the general form

$$\{ \ ; \ block_0, \ \ldots, \ block_k \ \}$$

where "`;`" is the sequential composition operator and $block_0$, ..., $block_k$ are other blocks.

If no composition operator is used for a block, then the PCN compiler will interpret this as a sequential block..

A *mutable variable* in PCN, like a variable in Fortran or C, is declared to have some type (`char`, `int`, or `double`), initially has some arbitrary (unknown) value, and can be modified many times during its lifetime, by means of an assignment statement. An assignment statement is represented as follows,

```
variable := expression
```

where `variable` is a mutable variable or an element of a mutable array.

Example. The procedure `swap` exchanges the values stored at the `i`th and `j`th positions of an integer `array`. Its three arguments — `array`, `i`, and `j` — are declared to be an integer array and single integers, respectively. A local variable `temp` is also declared. The three assignments are placed in a sequential composition, to ensure that they execute in the correct order.

The procedure `swaptest` can be used to execute `swap`. This procedure declares a local integer array `a[3]` and local integer variables `i` and `j`; initializes the array to contain the integers `0, 1, 2`, `i` to contain `1`, and `j` to contain `2`; calls a procedure `stdio:printf` to display the contents of `a`; calls `swap` to exchange the `i`th and `j`th components; and finally calls `stdio:printf` again to display the modified array. Note that since procedure arguments are passed by reference, the array `a` in `swaptest` is the same data structure as `array` in `swap`. Note also that in `swaptest`, the sequential composition operator ensures that both the assignments to `a` and the calls to `stdio:printf` occur in the correct order.

```
swap(array,i,j)
int array[], i, j, temp;
{;  temp     := array[j],
    array[j] := array[i],
    array[i] := temp
}

swaptest()
int a[3], i, j;
{;  a[0] := 0, a[1] := 1, a[2] := 2,
    i := 1, j := 2,
    stdio:printf("Before: %d %d %d\n",{a[0],a[1],a[2]},_),
    swap(a,i,j),
    stdio:printf("After: %d %d %d\n",{a[0],a[1],a[2]},_)
}
```

Role of Sequential Composition. The example illustrates the two primary applications of sequential composition in PCN: sequencing of updates to mutable variables and sequencing of I/O operations.

4.4 Parallel Composition and Definitional Variables

We now consider two related constructs that may be unfamiliar to some readers: parallel composition and definitional variables.

The *parallel composition* operator specifies that a set of statements are to be executed concurrently. A parallel composition has the general form

$$\{\ ||\ \texttt{block}_0,\ \ldots,\ \texttt{block}_k\}$$

where || is the parallel composition operator and $\mathtt{block}_0$, ..., $\mathtt{block}_k$ are other blocks. Execution within a parallel composition is fair: that is, it is guaranteed that execution of each block will eventually progress (unless that block has terminated). Execution of a parallel composition terminates when all of its constituent blocks have terminated.

Concurrent computations initiated within a parallel composition must be able to exchange data and synchronize their activities. It is important to understand that this *cannot* be achieved by using mutable variables (at least not without the introduction of complex locking mechanisms), as the order of read and write operations in a parallel composition, and hence the result of such operations, is not in general well defined.

Concurrent computations communicate and synchronize by means of *definitional* or single-assignment variables. We have already come across definitional variables in the introduction to this chapter. Here, we consider them in more detail.

Definitional variables are represented in the same way as mutable variables, with one exception: a solitary underscore character ("_") is used to represent an *anonymous* definitional variable. Each occurrence of "_" represents a unique variable.

Definitional variables are not declared. Any variable occurring in a procedure that is not explicitly declared in the procedure's declaration section is a definitional variable. Definitional variables initially have a special *undefined* value. They can be defined once, by means of a definition statement, and then cannot be modified. The definition statement is represented as

`variable = expression`,

where `variable` is a definitional variable. Note that a definition of the form `x = y` is allowed; this establishes `y` as an alias for `x`, so that any prior or subsequent definition for `y` also applies to `x`.

Example: Simple Divide and Conquer. The following program implements a simple divide-and-conquer strategy. As none of the variables in this procedure are declared, we see that all are definitional. Variables `prob` and `soln` are arguments; the rest are local to the procedure. When executed, procedure `div_and_conq` immediately executes a parallel composition containing four procedure calls. These execute concurrently, with execution order constrained only by availability of data. Variable `prob` is input and `soln` output. Procedure `split` consumes `prob` and hence will block until an input value is available. Likewise, the `solve` procedures block until `l_prob` and `r_prob` are defined by `split`. Once the two calls to `solve` produce values for `l_soln` and `r_soln`, the `combine` procedure can proceed to produce `soln`.

```
div_and_conq(prob,soln)
{|| split(prob,l_prob,r_prob),
    solve(l_prob,l_soln),
    solve(r_prob,r_soln),
    combine(l_soln,r_soln,soln)
}
```

Properties of Definitional Variables

- Have as initial value a special "undefined" value.
- Read operations block until the variable is given a value.
- Are defined ("written") by the definition operator ("=").
- Once defined, cannot be modified.
- Can be shared by procedures in a parallel composition.
- Are not explicitly declared.
- Can take on values of type `char`, `int`, `double`, or `tuple`.

It is instructive to compare mutable and definitional variables, as in the following table.

	Definitional	Mutable
Initial value	Special "undefined" value	Arbitrary value
Defined by	Definition operator (=)	Assignment operator (:=)
Read operation	Blocks if undefined	Always succeeds
Can be written	Once	Many times
Parallel composition	Can share	Cannot share
Explicitly declared	No	Yes
Types	tuple, int, double, char	int, double, char

Role of Parallel Composition. It is important to understand the distinct roles of the parallel and sequential composition operators. Parallel composition exposes opportunities for concurrent execution; sequential composition constrains execution order so as to sequence I/O operations or assignments to mutable variables. In general, it is a good idea to expose as much concurrency as possible in an application, as this provides the compiler and run-time system with maximum flexibility when making scheduling decisions. In particular, they can seek to reduce the cost of remote data accesses by overlapping computation and communication.

4.5 Choice Composition

The third and final composition operator that we consider is the choice composition operator, "?". A choice composition has the general form

$$\{\ ?\ \ \texttt{guard}_0\ ->\ \texttt{block}_0,\ \ldots,\ \texttt{guard}_k\ ->\ \texttt{block}_k\}$$

where each $\texttt{guard}_i$ is a sequence of one or more tests. Valid tests include

$a < b$, $a > b$, $a <= b$, $a >= b$: arithmetic comparison

$a == b$, $a\ != b$: equality and inequality tests

`int(`a`)`, `char(`a`)`, `double(`a`)`, `tuple(`a`)` : type tests

`data(`a`)` : synchronization test

? = : tuple match

`default` : default action

We refer to a single "`guard -> block`" as an *implication*.

Choosing between Alternatives. Choice composition provides a mechanism for choosing between alternatives. In this respect it may be regarded as a parallel *if-then-else* or guarded command. Each guard specifies the conditions that must be satisfied for the associated block to be executed. *At most one* of these blocks will be executed; which one depends on the result of guard evaluation.

A choice composition is executed as follows. Each guard is evaluated from left to right. A guard succeeds if all of its tests succeed. If one or more guards succeed, exactly one of the corresponding blocks is chosen to be executed.

For example, the procedure `max` executes either `z = x` or `z = y`, depending on the value of `x` and `y`, and hence defines `z` to be the larger of `x` and `y`.

Module `max.pcn`: Version 1

```
max(x,y,z)
{?  x >= y -> z = x,
    x <  y -> z = y
}
```

Synchronization. Choice composition also provides a synchronization mechanism. A test *suspends* when evaluated if it requires the value of an undefined definitional variable (e.g., `x < 3`, where `x` is undefined). Otherwise, it succeeds or fails depending on the value of its arguments.

A guard is evaluated from left to right. If any test suspends, the guard suspends. If any test fails, the guard fails. If all tests succeed, the guard succeeds.

If some guards suspend and all other guards fail, execution of the choice composition is suspended until more data is available. If all guards fail, execution of the choice composition terminates without doing anything. Hence, a call to the procedure `max` given above will suspend until both `x` and `y` have values, and then proceed as follows. If both `x` and `y` are numbers, the procedure executes either the first or second implication, depending on the values of `x` and `y`. If either `x` or `y` is not a number, the procedure terminates without doing anything.

The guard test `default` succeeds only if all other guards in a choice composition fail. For example, consider the following alternative formulation of the `max` procedure.

Module `max.pcn`: Version 2

```
max(x,y,z)
{?  x >= y  -> z = x,
    default -> z = y
}
```

The two versions of `max` give the same behavior if `x` and `y` are numbers. If either `x` or `y` is not a number, however, the first program terminates without executing either implication, while the second program selects the second implication.

Choice composition rules:

- Evaluate each guard left to right.
- If any test suspends/fails, guard suspends/fails.
- If all tests succeed, guard succeeds.
- If all guards fail, process terminates.
- If no guards succeed and some suspend, process suspends.
- If some guards succeed, execute one implication body.
- If all other tests fail, the `default` guard test succeeds.

Nondeterministic Choice. Choice composition also provides a mechanism by which nondeterminism is introduced into PCN programs. Nondeterministic choice is rarely required in parallel programming. However, it can be important in reactive applications.

We first illustrate the use of nondeterministic choice with a trivial example. We may rewrite the `max` procedure given earlier as follows. Note that the two implications are not mutually exclusive. If `x == y`, either implication may be taken.

This program is nondeterministic in the sense that the action that it performs is not determined solely by its input, although of course the answer computed is still determined precisely by the input.

```
max(x,y,z)
{?  x >= y -> z = x,
    x <= y -> z = y
}
```

We now consider a reactive programming example. A procedure `switch` has two definitional inputs corresponding to the outputs of two sensors in a mechanical device. If either sensor is activated, the corresponding input variable will be given a value. The `switch` procedure is to return a result value if either sensor is activated, with the value specifying which sensor was activated.

```
switch(sensor1,sensor2,alarm)
{?  data(sensor1) -> alarm = 1,
    data(sensor2) -> alarm = 2
}
```

The guard test `data` succeeds as soon as its argument has a value. Hence, the output variable `alarm` takes value `1` if `sensor1` is activated and `2` if `sensor2` is activated. It can take either value if both are activated.

Choice Composition is used for three purposes:

- Choosing between alternatives.
- Synchronization.
- Nondeterministic choice.

4.6 Definitional Variables as Communication Channels

Consider two procedure calls (processes), a producer and a consumer, that share a definitional variable, `x`.

`producer(x), consumer(x)`

The two processes can use the shared variable to communicate data, simply by performing read and write operations on the variable. For example, assume that the producer is defined to write the variable, as follows.

```
producer(x)
{|| x = "hello" }
```

The definition `x = "hello"` has the effect of communicating the message `"hello"` to the consumer. The consumer receives this value simply by reading (examining) the variable. For example, the following consumer procedure checks to see whether `x` has the value `hello`. Note the use of choice composition and the `default` guard.

```
consumer(x)
{? x == "hello" -> stdio:printf("Hello",{},_),
   default -> stdio:printf("Huh?",{},_)
}
```

The shared definitional variable `x` is used here to both communicate a value between the producer and consumer and to synchronize the actions of these processes. The shared definitional variable can be thought of as a *communication channel.*

The use of definitional variables to specify communication has two advantages. First, it avoids the distinction that is made in many parallel languages between interprocessor and intraprocessor communication. This means that no special "packing" or "unpacking" operations need be performed when communicating. This in turn facilitates the retargetting of programs to different parallel computers. Second, it provides great flexibility in the communication strategies that can be specified. In particular, it is possible (as we shall see below) to include variables in data structures and hence to establish dynamic communication structures.

An apparent difficulty of this formalism is that each definitional variable can be used only to communicate a single value. Fortunately, this is not the case. We show in § 4.9 below how a single shared variable can be used to communicate a *stream* of messages between processes.

4.7 Specifying Repetitive Actions

We have now encountered the constructs used in PCN to express concurrent and sequential execution, communication between concurrent computations, and state change within sequential computations. We need one more construct before we can build large programs, namely, a mechanism for specifying repeated actions.

You are probably familiar with the use of iteration to specify repetition. For example, in Fortran we may write `do i=1,10` to specify 10 repetitions of a loop, with `i` ranging from `1` to `10`. PCN provides a similar construct, called *quantification.* A quantification has the general form

```
{ op i over low .. high :: block }
```

and specifies that `block` should be executed once for each `i` in the range `low..high`, either concurrently (if `op` = ||) or sequentially (if `op` = `;`).

A quantification is useful when specifying iterative computations involving mutable variables (or ports – see § 11). However, the most commonly used iterative construct in PCN is *recursion*. You will be familiar with recursion if you have used C (or Prolog, Strand, or Lisp); it tends to be more verbose than iteration, but has the advantages of allowing richer repetition structures and of working well with definitional variables.

We introduce the use of recursion in PCN with a simple example. Consider the following procedure, which computes the `sum` of the elements with indices in the range `from..to` in `array`. This procedure is defined in terms of a choice composition with a parallel composition as the body of the first implication and a simple definition statement as the body of the second implication.

Module `sumarray.pcn`: Version 1

```
sum_array(from,to,array,sum)
{?  from <= to ->
        {|| sum_array(from+1,to,array,sumrest),
            sum = array[from] + sumrest
        },
    from > to -> sum = 0
}
```

The first implication states that if `from <= to`, then the sum of elements `from..to` is the value of element `array[from]` plus the sum of elements `from+1..to`. The second implication defines the sum to be 0 in the case when `from > to`.

This procedure uses recursion to repeat the summation over all the elements of the array. A recursive procedure normally specifies two alternative courses of action: continuation and termination. These are combined in a choice composition with guards specifying associated continuation and termination conditions.

In the example, the continuation action consists of summing `array[index]` and `sumrest`, and making a recursive call to `sum_array` to compute `sumrest`; these actions are to be performed if `from <= to`. The termination action consists of defining `sum = 0`; this is to be performed if `from > to`.

Recursive procedure specifies:

- Termination condition and actions.
- Continuation condition and actions.

Parallel algorithms based on divide-and-conquer techniques frequently make multiple recursive calls to the same procedure. For example, the following program implements a divide-and-conquer algorithm for summing the elements of an array. The task of summing an array is recursively decomposed into the tasks of summing the left and right subarrays.

Module `sumarray.pcn`: Version 2

```
sum_array(from,to,array,sum)
{?  from < to ->
        {|| sum_array(from,(from+to)/2,array,sumleft),
            sum_array((from+to)/2+1,to,array,sumright),
            sum = sumleft + sumright
        },
    from == to -> sum = array[from]
}
```

This example makes apparent the advantages of recursion as a repetition construct in a parallel language: the doubly recursive formulation of `sum_array` exposes concurrency that is not directly available in an iterative solution.

4.8 Tuples

The programs presented thus far have all dealt with simple data structures: characters, integers, double precision numbers, and arrays of the same. These data structures will be familiar to most readers from sequential languages such as Fortran and C. PCN also provides another sort of data structure called the *tuple*. Similar data structures are used in symbolic languages such as Prolog, Strand, or Lisp.

A tuple is a definitional data structure used to group together other definitional data structures. A tuple has the general form

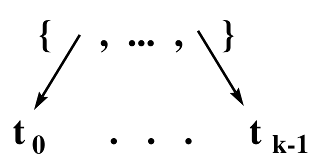
$$\{\ \texttt{term}_0,\ \ldots,\ \texttt{term}_{k-1}\ \} \qquad (k \geq 0)$$

where $\texttt{term}_0$, ..., $\texttt{term}_{k-1}$ are definitional data structures. The following are all valid tuples.

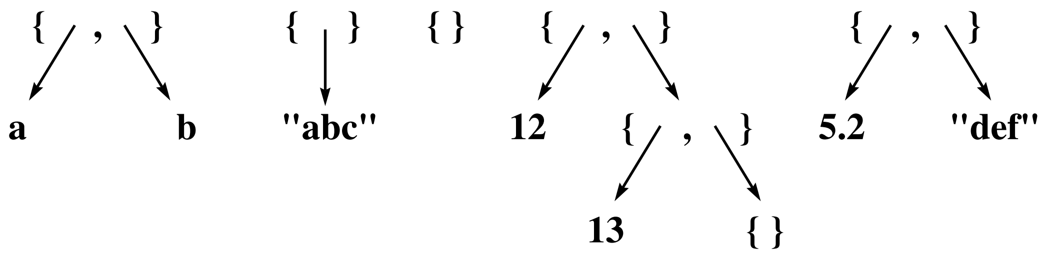
```
{a,b}    {"abc"}    {}    {12,{13,{}}}    {5.2,"def"}
```

Note that tuples can be nested: in the fourth tuple on the preceding line, the tuple `{}` is nested inside the tuple `{13,{}}`, which is in turn nested inside the tuple `{12,{13,{}}}`. Note also that tuples can contain elements of different types.

It is useful to think of tuples as representing trees. A tuple $\{\texttt{t}_0,\ \ldots,\ \texttt{t}_{k-1}\}$ represents a tree with a root and k offspring.



The tuples listed above can be drawn as follows.



Building Tuples. Tuples can be written in a program, either as an argument to a procedure call or as the right-hand side of a definition statement. For example, the block

```
{|| proc(1,{x,y,{z}}), x = "abc", y = {123} }
```

invokes a procedure `proc` with the tuple `{"abc",{123},{z}}` as its second argument.

Alternatively, the primitive operation `make_tuple` can be used to build a tuple of specified size, with each argument a definitional variable. For example, the call

```
make_tuple(3,tup)
```

defines `tup` to be the three-tuple `{_, _, _}`.

Accessing Tuples. Tuple elements can be referenced in the same way as array elements: $t[i]$ is element i of a tuple t, for $0 \leq i < \texttt{length}(t)$. Hence, the statements

```
make_tuple(3,tup), tup[0] = "abc", tup[1] = {123}, tup[2] = {z}
```

produce the tuple passed as an argument to `proc` previously.

The guard test "`?=`" (match) can be used to decompose a tuple into its constituent components. A match has the general form

$$\texttt{tup ?= \{t}_0\texttt{, ..., } t_{k-1}\texttt{\}},$$

where the $\texttt{t}_i$ are either new definitional variables or nonvariable terms. A match succeeds if `tup` has arity k and each of its arguments matches the corresponding $\texttt{t}_i$, suspends if `tup` is not defined or if one of the matches with a $\texttt{t}_i$ suspends, and fails otherwise. A new definitional variable $\texttt{t}_i$ is created with the value of the corresponding `tup` argument.

For example, the match

```
tup ?= {"abc", a, {b}}
```

succeeds if `tup = {"abc",{123},{z}}`, defining `a = {123}` and `b = z`. It suspends if `tup = {x,{123},{z}}`, as the first element of the matching typle is `"abc"`, but the first element of `tup` is the undefined variable `x`. It fails if `tup = {"def",{123},{z}}`, as the first element of the right-hand tuple (`"abc"`) does not match the first element of `tup` (`"def"`).

The match operator does *not* perform unification. That is, if the term on the left-hand side of the match contains undefined variables, those variables will *not* be defined to the values that appear in the same location on the right-hand side of the match. The only definitional variables that will be given values during a match are the new definitional variables that appear on the right-hand side of the match.

Variables that appear in right-hand side of match must be *new* definitional variables. They may not be definitional variables that already exist outside of this implication (i.e., the choice of the choice block that contains this match). The scope of these new definitional variables is the implication in which this match resides – they cannot be used outside of this implication. Therefore, to propogate a new variable that is created during the match to a definitional variable that is outside of the implication in which the match appears, you must assign the new variable to the outside definitional variable from within the body of the implication. This is illustrated in the following example which uses a match operator to extract the elements of the tuple, `t`. Those tuple elements are then used outside the scope of the implication in which the match appears.

Procedure `tuple1`

```
tuple1(t)
{||
    {?  t ?= {tmp_a, tmp_b} ->
           {|| a = tmp_a, b = tmp_b },
        default ->
           {|| a = 0, b = 0 }
    },
    r(a,b)
}
```

Comparing Tuples. The guard tests `==` and `!=` can be used to compare tuples as well as strings, numbers, and arrays. An equality test `x == y` succeeds if `x` and `y` are tuples with the same arity and corresponding subterms are also equal. The equality test is applied to subterms left to right, depth first; if any subterm test fails or suspends, the overall test also fails or suspends, respectively. The test also fails if `x` and `y` have different arities. An inequality test `x != y` succeeds if `x == y` would fail, fails if `x == y` would succeed, and suspends otherwise.

List Notation. A *list* is a two-tuple in which the first element represents the head of the list and the second element the tail. By convention, the zero-tuple ({}) represents the empty list. For example, the structure `{1,{2,{3,{}}}}` is the list containing the numbers `1`, `2`, and `3`.

This notation is clumsy, so PCN provides an alternative syntax: a list `{h,t}` may be written as `[h|t]`, the empty list as `[ ]`, a list such as `{1,{2,{3,{}}}}` as `[1, 2, 3]`, and a list such as `{1,{2,{3,tail}}}` as `[1, 2, 3|tail]`.

Example: List Length. The procedure `listlen` computes the length `len` of a list `l`. For example, a call `listlen([1,2,3,4],len)` gives the result `len = 4`. Note the use of an auxiliary procedure `listlen1`, which accumulates the length so far in `acc` and then returns the final result as `len`.

```
listlen(l,len)
{|| listlen1(l,0,len)}

listlen1(l,acc,len)
{?  l ?= [_|l1] -> listlen1(l1,acc+1,len),
    default -> len = acc
}
```

Example: Building a List. The procedure `buildlist` builds a list `l` of length `len`. For example, a call `buildlist(4,l)` gives the result `l = [4,3,2,1]`.

```
buildlist(len,l)
{?  len > 0 ->
    {|| l = [len|l1],
        buildlist(len-1,l1)
    },
    default -> l = []
}
```

Example: List Transducer. The procedure `listadd` is an example of what is called a *list transducer*. It traverses one list and constructs another containing the result of applying a simple operation to each element in the first list: in this case, the operation is simply to add one to each element. For example, a call `listadd([1,2,3,4],nl)` gives the result `nl = [2,3,4,5]`.

```
listadd(l,nl)
{?  l ?= [e|l1] ->
    {|| nl = [e+1|nl1],
        listadd(l1,nl1)
    },
    default -> nl = []
}
```

4.9 Stream Communication

We have seen how two or more concurrent computations that share a definitional variable can use that variable to exchange data. The *producer* of the data simply defines the shared variable to be the data to be communicated (e.g., `x = "hello"`). The *consumer*(s) of the data can then use the data in computation.

A shared definitional variable would not be very useful if it could be used only to exchange a single value. Fortunately, there are simple techniques that allow a single definitional variable to be used to communicate many values. The most important of these is the *stream*. A stream is a data structure that permits communication of a sequence of messages from a producer to one or more consumers. A stream acts like a queue: the producer places elements on one end, and the consumer(s) take them off the other.

By convention, stream communication is implemented in PCN in terms of list structures. Imagine a producer and a consumer sharing a variable `x`. The producer defines `x = [msg|xt]` and the consumer matches `x ?= [msg|xt]`. The effect of these operations is to both communicate `msg` to the consumer and create a new shared variable `xt` that can be used for further communication. This process can be repeated arbitrarily often to communicate a stream of messages from the producer to the consumer. Hence, a stream is a list structure, incrementally constructed by a producer and deconstructed by a consumer. The empty list (`[]`) is used to represent the end of a stream.

Example: Summing Squares. We illustrate the stream communication protocol in a program that computes the sum of the squares of the integers from `1` to `N`. We decompose this problem into two subproblems: constructing a stream of squares and summing a stream of numbers. The first subproblem is solved by the procedure `squares`, which recursively produces a stream (i.e., list) of messages $\texttt{N}^2$, $\texttt{(N-1)}^2$, ..., `1`. The second subproblem is solved by the procedures `sum` and `sum1`, which recursively consume this stream (list). The auxiliary procedure `sum1` accumulates the sum so far in `sofar` and returns the final result as `sum`.

Note the structure of the producer (`squares`) and consumer (`sum1`) procedures in the following program. Both are recursively defined. In the producer, the recursive case incrementally constructs a list `sqs` of squares by defining `sqs = [n*n|sqs1]` and calling `squares` to compute `sqs1`; the termination case defines `sqs = []`. In the consumer, the recursive case deconstructs a list `ints` of integers by matching `ints ?= [i|ints1]` and calling `sum1` to consume the rest of the messages; the termination case returns a result.

Module sumsquares.pcn

```
sum_squares(N,sum)
{|| squares(N,sqs), sum(sqs,sum) }

squares(n,sqs)                         /* Producer:          */
{?  n > 0 -> {|| sqs = [n*n|sqs1], /*   Produce element, */
                 squares(n-1,sqs1) /*    & recurse       */
            },
    n == 0 -> sqs = []             /*   Close list.      */
}

sum(ints,sum)
{|| sum1(ints,0,sum)}

sum1(ints,sofar,sum)               /* Consumer:          */
{?  ints ?= [i|ints1] ->           /*   Consume element, */
        sum1(ints1,sofar+i,sum),   /*    & recurse       */
    ints ?= [] -> sum = sofar      /*   End of list: stop*/
}
```

Send/Receive Operations. Some readers may find it useful to think of a stream as an abstract data type on which four operations are defined: `send`, `close`, `recv`, and `closed`. The first two are procedure calls used by a stream producer, and the latter two are guard tests used by a stream consumer. All take a definitional variable (`s`) as an argument; `send` and `recv` also return a new definitional variable (`s1`) representing a new stream to be used for the next communication.

`send(s,msg,s1)`: Send `msg` on stream `s`, returning as `s1` a new stream for subsequent communication.

`close(s)`: Close stream `s`.

`recv(s,msg,s1)`: Succeed if a message is pending on stream `s`, defining `msg` to be the message and `s1` the new stream.

`closed(s)`: Succeed if stream `s` has been closed.

These operations can be defined by the following macros.

File sendrecv.h

```
#define send(s,msg,s1) s = [msg|s1]
#define close(s)       s = [ ]
#define recv(s,msg,s1) s ?= [msg|s1]  /* Guard test */
#define closed(s)      s == [ ]       /* Guard test */
```

These definitions can be placed in a file (e.g., `sendrecv.h`) and included in your programs, if you prefer to think in terms of `send` and `recv` operations instead of definition and match operations on streams. For example, the `squares` and `sum1` procedures presented previously (module `sumsquares.pcn`) can be rewritten as follows.

```
#include "sendrecv.h"            /* Include macros */

squares(n,sqs)
{?  n > 0 -> {|| send(sqs,n*n,sqs1),
                 squares(n-1,sqs1)
             },
    n == 0 -> close(sqs)
}

sum1(ints,sofar,sum)
{?  recv(ints,i,ints1) -> sum1(ints1,sofar+i,sum),
    closed(ints)       -> sum = sofar
}
```

However, it would be a mistake to think of lists as simply a clumsy notation for streams, and to restrict your use of streams to the four basic operations provided in `sendrecv.h`. The fact that streams are data structures that can be manipulated in the same way as any other data structure provides enormous flexibility.

Example: Stream Filter. We illustrate this flexibility with a list transducer that filters a stream `x`, generating a stream `y` identical to `x` but with no consecutive duplicates. (For example, a call `filter([1,1,4,3,5,5,2],y)` defines `y = [1,4,3,5,2]`.)

This is not a complex example. However, it illustrates several stream-processing strategies. Note in particular the use of the match operator to check for two pending messages (as follows: `x?=[msg1,msg2|x1]`), the pushing of unused elements back onto the stream in the recursive calls (e.g., `filter([msg2|x1],y)`), and the definition of `y` to be all remaining elements of `x` in the termination case (`y = x`).

```
filter(x,y)
{?  x ?= [msg1,msg2|x1] ->
    {?  msg1 == msg2 -> filter([msg2|x1],y),
        default -> {|| y = [msg1|y1],
                       filter([msg2|x1],y1)
                   }
    },
    default -> y = x     /* x is [msg] or [] */
}
```

4.10 Advanced Stream Handling

The stream construct provides direct support for one-to-one communication, that is, communication between a single producer and a single consumer. It also supports broadcast communication, that is, generation of a single stream to be received by several consumers. For example, in the composition

{|| `producer(s), consumer(s), consumer(s)` },

both consumers receive any values generated by the producer.

Three other communication patterns are also important in practical applications: many-to-one, one-to-many, and bidirectional. The first and second are supported in PCN by specialized primitives. The third is achieved by means of a specialized programming technique.

Mergers: Many-to-One Communication. A *merger* is a PCN system program that allows the construction of an output stream that is the nondeterministic interleaving of a dynamically varying number of input streams. (The merger is hence the second source of nondeterminism in PCN, with choice composition being the first.) The only constraint on message order in the output stream is that the order of messages from individual input streams be preserved. A merger is created with a procedure call of the form

`sys:merger(in,out)`,

where `in` is an initial input stream and `out` is the output stream. An additional input stream `newin` is registered with the merger by appending a message of the form {`"merge",newin`} to any open input stream. An input stream is closed in the usual way (`s = []`); the output stream is closed automatically when all input streams are closed.

The following code fragment illustrates the use of the merger. This organizes communication between two producer processes and a single consumer, so that the consumer receives on `instream` an intermingling of the streams generated by the two producers.

```
{|| producer(s1), producer(s2)
    instream = [{"merge",s1},{"merge",s2}],
    sys:merger(instream,outstream),
    consumer(outstream)
}
```

Note that the merger must be able to determine whether each input message is a {`"merge",_`} term. Hence, messages of the form `var` or {`var,term`} (where `var` is an undefined variable) should not be sent to a merger: these will cause the merger to delay until `var` is given a value.

Distributors: One-to-Many Communication. A *distributor* is a PCN system program that routes each message received on its input stream to one of several output streams. A message of the form {`N,Msg`} causes the distributor to route `Msg` to the `N`th output stream. A distributor is created with a call of the form

`sys:distribute(N,In)`,

where `N` is the number of output streams needed and `In` is the input stream. Messages can then be sent to the distributor to register output streams. We register a stream `S` as the `N`th output stream by sending a message with the form

{`"attach",N,S,Done`},

where `Done` is a definitional variable that is defined by the distributor to signal that the stream `S` has been registered.

We request the distributor to route a message `Msg` to the `N`th output stream by sending the following message:

{`N, Msg`}

We request the distributor to broadcast a message `Msg` to all output streams by sending the following message:

{`"all", Msg`}

It is important to ensure that a stream has been registered before requesting that a message be routed to that stream. One way of doing this is to register all streams with the distributor before sending any messages. The following program achieves this. A call `make_distributor(in,ss)` creates a distributor with `ss` as its output streams. (The number of streams in `ss` is computed by the procedure `sys:list_length` defined in § 18.1.) The input stream `in` is passed to this distributor only after all output streams have been registered.

```
make_distributor(in,ss)
{|| sys:list_length(ss,len),
    sys:distribute(len,tod),
    register(0,ss,tod,in)
}

register(i,ss,tod,in)
{?  ss ?= [s|ss1] ->
        {|| tod = [{"attach",i,s,done}|tod1],
            data(done) -> register(i+1,ss1,tod1,in)
        },
    ss ?= [] -> tod = in
}
```

If the input stream to the distributor is closed (`In = []`), then the distributor closes all registered output streams and shuts down.

Two-Way Communication. Many parallel algorithms require two-way communication between concurrently executing processes. In some cases, this can be achieved by defining two communication streams, one for use in each direction. However, it is also possible to achieve two-way communication with a single definitional variable, by using a technique called an *incomplete message.*

We introduce the incomplete message technique with a simple example. Consider a program `input` capable of providing boundary conditions for two different numerical models (e.g., spectral and finite difference). This can be composed with a procedure implementing a particular numerical model, as follows.

```
input(xs), model(xs)
```

The definitional variable `xs` will be used to implement a stream.

The first thing that `input` does is to query the program it is composed with, to determine that program's input requirements. It does this by sending a message of the form

```
{"what_input",response},
```

where `response` is an undefined definitional variable. The other program (which of course must be ready to accept such a message) defines `response` to specify the required input type, allowing the first program to read `response` and generate the appropriate input data.

Possible definitions for `input` and `model` are as follows. In this example, the `model` procedure specifies that it expects input in terms of spectral coefficients by defining `response = "spectral"`. This communication causes the `input` procedure to execute `spectral_input`.

```
input(x)
{|| x = [{"what_input",response}|xs],
    {?  response == "spectral" -> spectral_input(xs),
        response == "finite_diff" -> fd_input(xs)
    }
}

model(x)
{?  x ?= [{"what_input",response}|xs] ->
    {|| response = "spectral",
        process_input(xs)
    }
}
```

In this example, a single shared variable, `xs`, has been used to achieve two-way communication. This is a simple example of a very powerful programming technique that can be used to establish a wide variety of communication patterns. The key idea is for one process to define a shared variable to be a tuple containing "holes" (undefined variables). Consumer(s) of this tuple can then fill in these holes (define the variables) to communicate additional values to the original producer or even to other consumers.

We use a more complex example to strengthen understanding of the incomplete message technique. Consider the problem of exploring a large search space with a heuristic search method. Assume that it is possible to define multiple *searchers*, each capable of exploring part of the search space, and that individual searchers can improve their efficiency by exploiting global information about the best-known partial solution. We collect and disseminate global information by defining a *controller* process to which each searcher periodically sends information about its current best partial solution. The controller responds to each such message by updating its view of the best partial solution and returning the best known partial solution.

A PCN implementation of this search method provides each searcher with a stream to the controller and uses a merger to combine the multiple searcher streams into a single controller input stream. For example, the following code links two searchers and a controller.

```
{||  searcher(s1), searcher(s2),
     sys:merger([{"merge",s1},{"merge",s2}],s),
     controller(s)
}
```

The searcher is defined as follows. A call to `first_attempt` yields an initial approximate solution (`value`), which is passed to the recursively defined procedure `search`. The `search` procedure sends the approximate local solution to the controller in a {`value,response`} tuple, where `response` is an undefined definitional variable used to communicate information back from the controller to the searcher. Depending on the `response` received from the controller, the searcher either terminates or calls `next_attempt` and repeats the process.

The controller receives a stream of approximate solutions from the workers. It processes each message by calling `improve_estimate` to improve its own estimate of the global best solution, and returning either this estimate or the signal `"stop"` (indicating that a solution has been found) to the searcher.

```
searcher(trials)
{|| first_attempt(value),
    search(trials,value)
}

search(trials,value)
{|| trials = [{value,response}|trials1],
    {?  response == "stop" -> trials1 = [],
        default ->
        {|| next_attempt(value,response,next_value),
            search(trials1,next_value)
        }
    }
}

controller(trials,bound)
trials ?= [{value,response}|trials1] ->
    {|| improve_estimate(bound,value,newbound,result),
        {?  result == "solution" -> response = "stop",
            default -> response = newbound
        },
        controller(trials1,newbound)
    }
```

Specialized Communication Structures:

- Many-to-one: `merger`.
- One-to-many: `distributor`.
- Bidirectional: incomplete message.

4.11 Interfacing Parallel and Sequential Code

The two worlds of parallel and sequential, definitional and mutable, have so far been regarded as distinct. In practice, the two worlds must interact whenever a sequential program component is integrated into a concurrent program. Such interactions are governed by three simple rules. The first restricts the way in which mutable variables can be used within parallel blocks, while the second and third specify copying operations performed by the PCN compiler when data is transferred between the definitional and mutable worlds by defining a definition in terms of a mutable, or vice versa. This copying avoids aliasing between state maintained in different sequential threads, and hence ensures that state change within individual threads does not lead to time-dependent interactions with concurrently executing processes.

Mutable Variables and Parallel Composition. Mutable variables may occur in parallel compositions, but only if their usage obeys the following rule.

> **Rule 1:** A mutable variable can be shared by blocks in a parallel composition only if no block modifies the variable.

This restriction prevents errors resulting from time-dependent, nondeterministic updates to a mutable variable (i.e., race conditions). The restriction is not currently enforced by the compiler, and so the programmer must be careful to ensure that all programs are valid.

Note that there is no similar restriction on the use of definitional variables within sequential blocks.

Mutable → Definition. The following rule states what happens when a definitional variable is defined in terms of a mutable variable.

> **Rule 2:** When a mutable occurs on the right-hand side of a definition statement, the current value of the mutable is snapshotted (copied), and the definition then proceeds as if a definitional value were involved.

For example, in the following code, `c = 5` and `d = 4` when computation is complete.

```
proc1(c,d)
int a;
{;  a := 3,
    c = 2 + a,
    a := 4
    d = a
}
```

Snapshotting a mutable array creates a definitional copy of the array that can be read but not modified. For example, in the following, `c` is defined to be a copy of the mutable array a. Subsequent changes to a do not affect the value of the definitional array c.

```
proc2(c,d)
int a[5];
{;  initialize(a),
    c = a,
    ...
}
```

Definition → Mutable. The following rule states what happens when a mutable variable is assigned an expression involving a definitional variable.

> **Rule 3:** When a definitional variable occurs on the right-hand side of an assignment, the assignment suspends until the variable has a value and then proceeds.

For example, if c is a definition with value 3 in the following program, then a has value 5 after the assignment.

```
proc3(a,c)
int a, b;
{;  b := 2
    a := b + c
}
```

Note that if the right-hand side of the assignment is not an expression, then the assignment will copy the definitional value into the mutable variable. For example, in the following code fragment, the definitional value c is copied into the mutable array a. The array a can be modified subsequently without affecting c.

```
int a[5];
a := c
```

Example. The following example illustrates the use of copying to avoid aliasing. The procedure proc has two definitional arguments: it produces as output the result of applying a transformation solve to input. It calls the procedure solve to effect the transformation; this is defined to operate on mutable data structures. Hence, proc declares a local mutable array temp, assigns temp the value input, applies solve to temp, and then defines output to be the updated value of temp. Two copying operations take place, from input to temp and from temp to output.

```
proc(input,output)
double temp[SIZE];
{;  temp := input,
    solve(temp),
    output = temp
}
```

4.12 Review

PCN encourages a compositional approach to parallel programming, in which complex programs are built up by the parallel composition of simpler components. Program components composed in parallel execute concurrently. They communicate by reading and writing definitional (single-assignment) variables. The use of definitional variables avoids time-dependent interactions, allowing individual components to be understood in isolation. In addition, read and write operations on definitional variables can be implemented efficiently on both shared-memory and distributed-memory parallel computers. Hence, parallel composition and definitional variables address three of the concerns listed at the beginning of this chapter: *concurrency*, *compositionality*, and *mapping independence*.

The choice operator is used to encode conditional execution and synchronization. It also provides a means of introducing *controlled nondeterminism* into programs. (The merger is the other mechanism used to specify nondeterministic actions in PCN programs.)

The sequential composition operator and mutable variables together provide a mechanism for integrating state change into definitional programs. This state change may be performed in PCN or in lower-level sequential languages.

A final aspect of PCN which may be unfamiliar to some readers is its use of tuples and recursion. These constructs provide support for symbolic processing. They augment arrays, iteration, and other language constructs provided by languages such as Fortran and C for numeric processing. An increasing number of applications have both numeric (regular, floating-point) and symbolic (irregular, rule-based) components. PCN's symbolic processing capabilities are intended to support such mixed-mode applications.

5 Programming Examples

We present PCN programs that solve programming problems concerned with list and tree manipulation, sorting, and a two-point boundary value problem.

5.1 List and Tree Manipulation

Membership in a List. Develop a program `member` with arguments `e`, `l`, and `r`, where `l` is a list, and at termination of execution of the program, `r = TRUE` if and only if `e` appears in list `l`. Assume that `FALSE = 0` and `TRUE = 1`, to be consistent with C.

```
#define TRUE  1
#define FALSE 0

member(e,l,r)
{?  l ?= [v|l1], v == e -> r = TRUE,
    l ?= [v|l1], v != e -> member(e,l1,r),
    l ?= [] -> r = FALSE
}
```

Membership in a List (Mutables). Now consider a program with the same specification, except that `e` and `r` are now mutables. The mutable `r` is to be set to `TRUE` or `FALSE`; `e` (and of course `l`) should not be changed.

```
#define TRUE  1
#define FALSE 0

member(e,l,r)
int e, r;
{?  l ?= [v|l1], v == e -> r := TRUE,
    l ?= [v|l1], v != e -> member(e,l1,r),
    l ?= [] -> r := FALSE
}
```

The only difference between the two programs is the addition of the type declarations and the substitution of the `:=` operator.

Reversal of a List. Develop a program `reverse` with arguments `x`, `b`, and `e`, which defines `b` to be the list of elements in `x`, in reverse order, concatenated with `e`. For example, if `x = ["A","B"]` and `e = ["C","D"]`, then `b` is to be defined as `["B","A","C","D"]`. (The name `b` stands for the beginning of the reversed list, and `e` stands for the end of the reversed list.)

```
reverse(x,b,e)
{?  x ?= [v|xs] ->  reverse(xs,b,[v|e]),
    x ?= [] -> b = e
}
```

This program can be used to simply reverse a list by calling it with `e = []`. For example, a call `reverse([1,2,3],b,[])` yields `b = [3,2,1]`.

The `reverse` procedure illustrates an important programming technique called the *difference list*. A call to `reverse` constructs a list `b` consisting of the values computed by `reverse` *followed by* the values provided as `e`. This allows lists constructed in several computations to be concatenated without further computation. For example, the calls

```
reverse([1,2,3],b,e), reverse([4,5,6],e,[])
```

construct the list `[3,2,1,6,5,4]`.

Height of a Binary Tree. Develop a program `height` with arguments `t` and `z`, where `t` is a binary tree, and `z` is to be defined to be the height of the tree. A tree `t` is either the empty tuple, {}, or a 3-tuple {`left, val, right` }, where `left` and `right` are the left and right subtrees of `t`.

```
height(t,z)
{?  t ?= {left, _, right} ->
    {|| height(left, l), height(right, r),
        {?  l >= r -> z = l+1,
            l <  r -> z = r+1
        }
    },
    t ?= {} -> z = 0
}
```

The program can be read as follows. The height of a nonempty tree is 1 plus the larger of the heights of the left and right subtrees. (The heights of the subtrees are determined by two recursive calls to `height`.) The height of an empty tree is 0.

Preorder Traversal of a Binary Tree. Develop a program `preorder` with arguments `t`, `b`, and `e`, where `t` is a binary tree, and `b` and `e` are lists. Binary trees are represented using tuples, as in the last example. List `b` is to be the list consisting of the `val` of all nodes of the tree in preorder, concatenated with list `e`. (A traversal of a tree in preorder visits the root, then the left subtree, and finally the right subtree.)

```
preorder(t,b,e)
{?  t ?= {left,val,right} ->
        {|| b = [val|m1],
            preorder(left,m1,m2),
            preorder(right,m2,e)
        },
    t ?= {} -> b = e
}
```

The program uses the difference list technique introduced previously in the reverse example: each call to `preorder` constructs a list `b` consisting of the elements in its subtree `t` followed by the supplied list `e`.

5.2 Quicksort

We present an implementation of the well-known quicksort algorithm, `qsortD`, which uses lists of definitional variables; later, we provide an *in-place* quicksort, `qsortM`, that uses mutable arrays. It is instructive to compare the two programs: the definitional program is significantly shorter and easier to understand than the mutable program. However, it makes less efficient use of memory.

Definitional Quicksort. Program `qsortD` has two input arguments, `x` and `e`, and one output argument, `b`: `x` and `e` are definitional variables that are not defined by the program, and `b` is a definitional variable that is defined by the program. All three are lists of numbers. The output `b` is specified to be the list `x` sorted in increasing order, concatenated with list `e`. For example if `e = [5, 4]` and `x = [2, 1]`, then `b = [1, 2, 5, 4]`. If `e` is the empty list, then `b` is `x` sorted in increasing order.

```
qsortD(x,b,e)
{?  x ?= [mid|xs] ->
        {|| part(mid,xs,left,right),
            qsortD(left,b,[mid|m]),
            qsortD(right,m,e)
        },
    x ?= [] -> b = e
}

part(mid,xs,left,right)
{?  xs ?= [hd|tl] ->
    {?  hd <= mid ->
            {|| left = [hd|ls],  part(mid,tl,ls,right) },
        hd >  mid ->
            {|| right = [hd|rs], part(mid,tl,left,rs) }
    },
    xs ?= [] -> {|| left = [], right = [] }
}
```

The `qsortD` procedure operates as follows. If `x` is nonempty, let `mid` be its first element and let `xs` be the remaining elements. The call `part(mid,xs,left,right)` defines `left` to be the list of values of `xs` that are at most `mid`, and `right` to be the list of values of `xs` that exceed `mid`. Call `qsortD(right,m,e)`, thus defining `m` to be the sorted list of `right` appended to `e`. Call `qsortD(left,b,[mid|m])`, thus defining `b` to be the sorted list of `left` followed by `mid` followed by `m`. Otherwise, if `x` is the empty list, then define `b` to be `e`.

The `part` procedure operates as follows. If `xs` is not empty, then let `hd` and `tl` be the head and tail (respectively) of `xs`. If `hd` is at most `mid`, define `ls` and `right` by `part(mid,tl,ls,right)`, and define `left` as `hd` followed by `ls`. If `hd` exceeds `mid`, define `left` and `rs` by `part(mid,tl,left,rs)`, and define `right` as `hd` followed by `rs`. If `xs` is the empty list, define `left` and `right` to be empty lists.

In-Place Quicksort. Program `qsortM` has two input parameters, `l`, and `r`, both of which are definitional variables, and one input-output parameter `C`, which is a one-dimensional mutable array of integers. Let $\mathtt{C}^{init}$ be the initial value of `C`, and let $\mathtt{C}^{final}$ be the value of `C` on termination of the program. Then $\mathtt{C}^{final}$ is to be a permutation of $\mathtt{C}^{init}$, where $\mathtt{C}^{final}$`[l, ..., r]` is $\mathtt{C}^{init}$`[l, ..., r]` in increasing order, and the other elements of `C` are to remain unchanged. (If $\mathtt{l} \geq \mathtt{r}$ then $\mathtt{C}^{final}$ is $\mathtt{C}^{init}$.)

```
qsortM(l,r,C)
int C[];
{?  l < r ->
      {;  split(l,r,C,mid),
          qsortM(l,mid-1,C),
          qsortM(mid+1,r,C)
      }
}
split(l,r,C,mid)
int C[], left, right, temp;
{?  l <= r ->
      {;  left := l+1, right := r, s = C[l],
          part1(l,r,C,s,left,right), temp := l,
          swap(temp,right,C), mid = right
      }
}
part1(l,r,C,s,left,right)
int C[], left, right;
{?  left <= right ->
      {;  left_rightwards(r,C,s,left),
          right_leftwards(l+1,C,s,right),
          {?  left <= right ->
                {;  swap(left,right,C),
                    left := left + 1,
                    right := right - 1
                }
          },
          part1(l,r,C,s,left,right)
      }
}
left_rightwards(r,C,s,left)
int C[], left;
{?  left <= r, C[left] <= s ->
      {;  left := left+1, left_rightwards(r,C,s,left) }
}
right_leftwards(l,C,s,right)
int C[], right;
{?  right >= l, C[right] > s ->
      {;  right := right-1, right_leftwards(l,C,s,right) }
}
swap(i,j,C)
int i, j, C[], temp;
{;  temp := C[i], C[i] := C[j], C[j] := temp }
```

Execution of `split(l,r,C,mid)` permutes `C` and assigns a value to `mid` such that `l` $\leq$ `mid` $\leq$ `r`, and such that all elements in `C[l, ..., mid-1]` are at most `C[mid]`, and all elements in `C[mid+1, ..., r]` exceed `C[mid]`.

The program `qsortM` operates as follows. If `l` $\geq$ `r`, then `qsortM` takes no action, leaving `C` unchanged. If `l` $<$ `r`, then `split` is called, and after `split` terminates execution, `C[l, ..., mid-1]` and `C[mid+1, ..., r]` are sorted independently.

The `split` program operators as follows. If `l` $>$ `r`, then `split` terminates execution without taking any action. If `l` $\leq$ `r`, then program `split(l,r,C,mid)` calls `part1(l,r,C,s,left,right)` after setting `left = l+1`, `right = r` and `s = C[l]`; program `part` leaves `s` unchanged, modifies `left` and `right`, and permutes elements of `C[l+1, ..., r]` so that, at termination of `part1`, `left = right +1`, and all elements in `C[l+1, ..., right]` are at most `s`, and all elements in `C[right+1, ..., r]` exceed `s`.

After termination of `part1`, program `swap` is called to exchange `C[l]` (which is `s`) with `C[right]`. After the swap, all elements in `C[l, ..., right-1]` are at most `s`, and `C[right] = s`, and all elements in `C[right+1, ..., r]` exceed `s`. The program terminates after `mid` is defined as `right`.

Program `part1` moves `left` rightwards and `right` leftwards until they cross (i.e., `left = right+1`).

5.3 Two-Point Boundary Value Problem

Our last programming example is a solution to a more substantial numerical problem. The problem that we consider arises when solving the linear boundary value problem in ordinary differential equations, namely,

$$\begin{gathered} y' = M(t)y + q(t), \qquad t \in [a,b], \qquad y \in R^n, \\ \text{such that} \qquad B_a y(a) + B_b y(b) = d. \end{gathered}$$

In most algorithms designed to solve this problem, the most computationally intensive task is the construction and solution of a linear algebraic system of equations, which typically has the form

$$\begin{bmatrix} B_a & & & & B_b \\ A_1 & C_1 & & & \\ & A_2 & C_2 & & \\ & & \ddots & \ddots & \\ & & & A_k & C_k \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} d \\ f_1 \\ f_2 \\ \vdots \\ f_k \end{bmatrix}.$$

Here each of the blocks has dimension $n \times n$, and k is often substantially larger than n. Construction of this system is trivially parallelizable. A more substantial challenge is to solve it in a parallel computing environment. It is important that the solution process be *stable* in a numerical sense; otherwise, the computed answer may be hopelessly inaccurate. Simple algorithms such as block elimination are therefore not appropriate. The algorithm described here uses a "structured orthogonal factorization" technique, in which orthogonal transformations are used to compress each two successive block rows of the linear system into a single block row. This produces a "reduced" system that has the same structure as the original system, but is half the size. The compression process can be applied recursively until a small system

$$\begin{bmatrix} B_a & B_b \\ \tilde{A}_1 & \tilde{C}_1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} d \\ \tilde{f}_1 \end{bmatrix}$$

remains.

The PCN code that implements this algorithm creates a set of k processes connected in a tree structure. A wave of computation starts at the k/2 leaves of the tree and proceeds up the tree to the root. The leaves perform the initial compression described above, while at the higher levels of the tree the compression is applied recursively, and at the root the small system above is solved. Finally, computation propagates down the tree to recover the remaining elements of the solution vector.

Input to the PCN code is provided at each leaf i $(0 \leq i < k/2)$ as two $n \times n$ blocks (A_i and C_i) and one n vector (f_i), and at the root as two $n \times n$ blocks (B_a and B_b) and one n vector (d).

The PCN code consists of two main parts. The first part is the code that creates the process tree. This creates a `root` process and calls a doubly recursive `tree` procedure to create $k/2$ `leaf` processes and $k/2-1$ `nonleaf` processes. Shared definitional variables (`strm, left, right`) establish communication channels between the nodes in the tree.

```
solve(k,t0,t1)
{|| root(strm),
    tree(strm,{t0,t1},1,k/2)
}

tree(strm,as,from,to)
{?  from == to -> leaf(from*2,strm,as),
    from < to ->
      {|| mid=from+(to-from)/2,
          nonleaf(left,right,strm),
          tree(left,as,from,mid),
          tree(right,as,mid+1,to)
      }
}
```

The second part of the program defines the actions performed by the `leaf`, `nonleaf`, and `root` processes. We consider the `leaf` process first. A single leaf process initializes two sets of blocks — `a1`, `c1`, `f1`) and (`a2`, `c2`, `f2` — and then calls `compress` to produce `a`, `c`, `f`. It sends a message to its parent containing the computed values and slots for return values (`ybot`, `ytop`) which will be computed by its parent. The `recover` procedure delays until values are received for `ybot` and `ytop`, and then computes the solution, `y`.

```
leaf(id,parent,as)
double a1[MM],c1[MM],f1[M],a2[MM],c2[MM],f2[M],
       a[MM],c[MM],f[M],y[M],r[MM];
{?  as ?= {t0,t1} ->
      {;  init_(id-1,a1,c1,f1,t0,t1),
          init_(id,a2,c2,f2,t0,t1),
          compress_(a1,c1,f1,a2,c2,f2,a,c,f,r),
          parent={a,c,f,ybot,ytop},
          recover(a1,c1,f1,r,ybot,ytop,y)
      }
}
```

The `nonleaf` procedure receives messages from left and right offspring. It calls `compress` to compress the `a1, c1, f1` and `a2, c2, f2` received from its offspring, producing `a,c,f`. These newly compressed values are communicated to the parent in the process tree. Once values for `ytop` and `ybot` are produced by the parent, the `recover` operation can proceed, producing `ymid`; values are then returned to the left and right offspring by the four definition statements.

```
nonleaf(left,right,parent)
double ymid[M],a[MM],c[MM],f[M],r[MM];
{?  left ?= {a1,c1,f1,ybot1,ytop1},
    right ?= {a2,c2,f2,ybot2,ytop2} ->
      {;  compress_(a1,c1,f1,a2,c2,f2,a,c,f,r),
          parent={a,c,f,ybot,ytop},
          recover_(a1,c1,f1,r,ybot,ytop,ymid),
          ybot1=ymid, ytop1=ytop,
          ybot2=ybot, ytop2=ymid
      }
}
```

The root process receives a single message containing the completely reduced blocks. It calls `comp_root` to perform the final computation, producing `ybot1` and `ytop1` which it returns to its offspring with two definitions.

```
root(child)
double ybot1[M],ytop1[M],ba[MM],bb[MM],brhs[M];
{?  child ?= {a,c,f,ybot,ytop} ->
      {;  init_root_(m,ba,bb,brhs),
          comp_root_(a,c,f,ba,bb,brhs,ybot1,ytop1),
          ytop=ytop1, ybot=ybot1
      }
}
```

6 Modules

Recall from § 3 that a PCN program consists of one or more modules. Each module is contained in a separate file with a `.pcn` suffix. A module contains zero or more procedures.

Procedures in one module can invoke procedures in other modules by means of *intermodule calls*. An intermodule call has the following general form.

`module:procedure_name(arg`$_0$`, ..., arg`$_n$`)`

A procedure can be invoked by an intermodule call only if it has been *exported* by the module in which it is defined. By default, all procedures in a module are exported. However, you can specify that only a subset of the procedures in a module are to be exported, by providing one or more `-exports` directives. An exports directive has the general form

`-exports(proc`$_0$`, ..., proc`$_k$`)`

and specifies that the module in which it appears exports procedures named by the strings `proc`$_0$, ..., `proc`$_k$. For example, the directive `-exports("procA","procB")` names `procA` and `procB` as exported.

In general, it is good practice to provide an `-exports` statement in each module, and to export only those procedures that are called from other modules. This allows the compiler to generate more compact object code.

7 The C Preprocessor

The PCN compiler applies the C language preprocessor (`cpp`) to each PCN module before compiling it. Hence, PCN programs can make use of `cpp`'s capabilities, such as include files, macros, and conditional compilation. All three of these capabilities are used in the following example program.

Module cpp_ex.pcn

```
-exports("go")

#include <pcn\_stdio.h>
#define ARRAY_SIZE 10

go()
double a[ARRAY_SIZE];
{;
#ifdef OLD_VERSION
    stdio:printf("Old version\n",{},_),
#else
    stdio:printf("New version\n",{},_),
#endif
    do_something_with_array(a)
}
```

When the PCN compiler applies `cpp` to a PCN program, it automatically defines the symbol "PCN" and a symbol that represents the target architecture for which you are compiling (i.e., sun4, rs6000, next040, ipsc860, etc). These symbols can be used for conditional compilation. For example, the following header file can be used in both PCN and C components of a program, hence ensuring that the symbol `ARRAY_SIZE` is defined in the same manner everywhere. The `#ifndef` means that the declaration of `my_c_procedure()` is used only in the C compilation.

File cpp_ex.h

```
#define ARRAY_SIZE 10
#ifndef PCN
#ifdef sun4
#include "sun4_only_hdr.h"
#endif /* sun4 */
extern void my_c_procedure();
#endif /* PCN */
```

In this example, `ARRAY_SIZE` will be defined to be the value 10 in both PCN and C programs that include this header file. Also, if the C program is compiled using `pcncomp` (i.e., `pcncomp -c file.c`), the procedure `my_c_procedure()` will be declared in the C program, and the header file `sun4_only_hdr.h` will be included in the C program when compiling for a sun4 architecture.

We can pass additional arguments to `cpp` when compiling PCN programs. For example, suppose we wish `OLD_VERSION` to be defined when compiling the program `cpp_ex.pcn` shown above. This can be achieved by using the `-D` flag when compiling with `pcncomp` as follows:

```
pcncomp -c cpp_ex.pcn -DOLD_VERSION
```

8 Integrating Foreign Code

Programming examples presented thus far have focused on the use of PCN to compose procedures written in PCN. Exactly the same syntax and techniques can also be used to compose procedures written in other ("foreign") languages. Fortran and C are currently supported.

We deal here with the PCN/foreign interface, the mechanism used to import foreign procedures, and the mechanism used to link foreign object code with the PCN run-time system.

8.1 PCN/Foreign Interface

The PCN/foreign interface is defined as follows:

- The actual parameters in a call to a foreign program can be mutables or definitional variables of type `char`, `int`, or `double`, or arrays of these types.
- Execution of a foreign procedure delays until all definitional arguments have values.
- All parameter passing is by reference.
- A foreign procedure cannot modify definitional arguments.

The last restriction is not currently enforced by the compiler, so the programmer must be careful to ensure that all programs satisfy this constraint.

Note that a consequence of this definition is that all output generated by a foreign procedure *must be returned in mutable arguments.* Sufficient storage must be allocated for these mutables prior to calling the foreign procedure.

Two important differences exist between the execution of PCN and foreign procedures called from PCN. First, PCN procedures can execute even if not all definitional arguments do have values. Indeed, they can compute values for definitional arguments. In contrast, foreign procedure calls delay until all definitional arguments have values, and can modify mutable arguments only. Second, PCN procedures can be passed tuples as arguments, whereas foreign procedures can be passed simple types only.

C. As parameter passing is by reference, arguments to a C procedure called from PCN must be declared as pointers. That is, the PCN types `char`, `int`, and `double` correspond to the C language types `char *`, `int *`, and `double *`.

Fortran. The PCN types `char`, `int`, and `double` correspond to the Fortran types `CHARACTER`, `INTEGER`, and `DOUBLE`. As Fortran also passes arguments by reference, no special treatment of arguments is required. It is necessary to append the suffix '_' to the name of a Fortran procedure called from PCN.

For example, the following PCN procedure calls a C procedure `natural_log(a,b)` to compute `b = ln(a)` and a Fortran procedure `power(a,b,c)` to compute `c =` a^b. Note the '_' suffix on the call to `power` and the use of a local mutable `tmp` for the result of the `natural_log` computation.

Module `foreign.pcn`

```
proc(a,b,c)
double a,b,c,tmp;
{;  natural_log(a,tmp), power_(tmp,b,c)}
```

The C and Fortran procedures invoked by this program can be written as follows.

File `cfile.c`

```
#include <math.h>

void natural_log(a,b)
double *a,*b;
{ *b = log(*a); }
```

File `ffile.f`

```
SUBROUTINE POWER(A,B,C)
DOUBLE PRECISION A,B,C
C = A**B
RETURN
END
```

8.2 Compiling with Foreign Code

When compiling PCN code that contains calls to foreign procedures, you need not do anything special to distinguish the foreign calls from normal PCN procedure calls. Instead, the PCN compiler assumes that any nonintermodule calls (i.e., calls that do not specify a module) to procedures not defined in that module are calls to foreign procedures. For example, this is what you see when you compile the `foreign.pcn` program shown above:

```
% pcncomp -c foreign.pcn
Notice:  Call to foreign procedure - natural_log
Notice:  Call to foreign procedure - power_
%
```

The C and Fortran source files can be compiled as normal to produce object files (`.o` files). Alternatively, `pcncomp` can be used to compile Fortran and C programs. For example:

```
pcncomp -c cfile.c
```

and

```
pcncomp -c ffile.f
```

The use of `pcncomp` to compile C and Fortran source files is recommended, since this compile command will work on any machine, no matter what the actual names of the C and Fortran compilers on the particular machines. In addition, `pcncomp` knows how to deal with Fortran programs that use C preprocessor directives (i.e., `#define`, `#include`, etc.). These source files should use a `.F` suffix. Some Fortran compilers know how to deal with `.F` files directly, in which case `pcncomp` just runs the Fortran compiler on the `.F` file. However, if a Fortran compiler cannot handle a `.F` file, `pcncomp` will first run the file through `cpp` before calling the Fortran compiler.

8.3 Linking with Foreign Code

Once all of your PCN source files are compiled to PCN object (`.pam`) files, and your C and Fortran source files are compiled to foreign object (`.o`) files, you must use `pcncomp` to link everything into an executable program.

To do this, simply add the `.o` files to the `pcncomp` link line, for example

```
pcncomp pcncode.pam ccode.o -o myprogram -mm pcncode
```

In addition, if you are linking Fortran object code, you must also add a `-fortran` flag to the link command. This ensures that Fortran initialization code is added to the executable program. For example, to link the example program above, you type:

```
pcncomp foreign.pam cfile.o ffile.o -o foreign -mm foreign -fortran
```

Like most compilers, `pcncomp` will also accept foreign libraries, which can be specified either by adding the appropriate `.a` file to the link line, or by using the -l and -L flags, for example:

```
pcncomp pcncode.pam ccode.o -o myprogram -mm pcncode libmine.a -lg
```

For a complete list of the arguments to `pcncomp`, type:

```
pcncomp -h
```

8.4 Multilingual Programming

This simple foreign interface allows sequential code (currently, Fortran and C are supported) to be integrated into PCN programs as procedure calls, indistinguishable for most purposes from calls to PCN procedures. Thus, we do not need to throw away the many years of investment in sequential code and compiler development when moving to parallel computers. Fortran and C are good sequential languages but are less well suited to parallel programming. Experience suggests that PCN is a good parallel language; nevertheless, it cannot compete with Fortran and C in code base and compiler technology. *Multilingual programming* permits us to take the best from each approach, using PCN for mapping, communication, and scheduling, and Fortran and C for sequential computation.

8.5 Deficiency of Foreign Interface

A deficiency of the Fortran interface is that no special allowance is made for "common" data (in Fortran programs) or "global" variables (in C programs). Each physical processor has a single copy of all common/global data declared in an application program, and every process on a processor has access to that data. Hence, while PCN data structures are encapsulated in processes to prevent concurrent access, the same protection is not provided for common/global data. It is the programmer's responsibility to avoid errors arising from concurrent access. Experience shows that programmers deal with this problem in one of two ways.

First, if an application is of moderate size, or is being developed from scratch, they often choose to eliminate common/global data altogether. This may be achieved by allocating arrays in PCN and passing them to the different foreign procedures. Although this approach requires substantial changes to the application, the bulk of the existing foreign code can be retained, and the full flexibility of PCN is available to the programmer.

Second, if substantial rewriting of an application is not possible, programmers maintain common/global data in its usual form and use PCN to organize operations on this data in a way that avoids nondeterminate interactions. Although certain operations are then more difficult (e.g., process migration is complicated, and the programmer must check for race conditions manually), other benefits of the PCN approach still apply.

9 Higher-Order Programs Using Metacalls

PCN provides simple support for higher-order programming. In particular, it allows module and procedure names in procedure calls to be substituted with variables, which can then be defined to be strings at run time. Variables are distinguished from strings in procedure calls by the use of enclosing back quotes, as follows.

```
..., `op`(...), ...            /* op is a variable */
```

```
..., m:`op`(...), ...          /* op is a variable */
..., `mod`:f(...), ...         /* mod is a variable */
..., `mod`:`op`(...), ...      /* mod & op are variables */
```

This sort of call is termed a *metacall.*

We illustrate the use of these higher-order features with a procedure `map_list` that applies a supplied operator to each element of a list, collecting the results of these computations in an output list. The supplied operator is assumed to be a procedure name (e.g., `"f"`); the `map_list` procedure invokes this procedure with two arguments (e.g., `f(e,v)`).

```
map_list(op,list,vals)
{?  list ?= [e|l1] ->
    {|| `op`(e,v),
        vals = [v|v1],
        map_list(op,l1,v1)
    },
    list ?= [] -> vals = []
}
```

For example, if the procedure `square` is defined as

```
square(e,v) {|| v = e*e }
```

then a call `map_list("square",[1,2,3],vals)` will define `vals` to be the list `[1,4,9]`.

The `map_list` procedure will work correctly only if the supplied operator (`op`) is located in the same module as `map_list`. The following program is more general: it allows the supplied operator to be a `mod:proc(arg)` term. Note the use of quoting in the match operation.

```
map_list2(op,list,vals)
{?  list ?= [e|l1], op ?= `mod`:`proc`(arg) ->
    {|| `mod`:`proc`(arg,e,v),
        vals = [v|v1],
        map_list2(op,l1,v1)
    },
    list ?= [] -> vals = []
}
```

Metacalls present a small problem to the PCN linker. The PCN linker normally includes in the executable program only those PCN procedures that it can determine will be called. However, since metacalls are procedure calls for which you do not specify the call target until run-time, the linker may not be able to determine that a metacalled procedure is called and therefore will not link in that procedure. To handle this situation, two additional PCN source directives are supported:

`-metacalls(mod1:proc1, mod1:proc2, ...)`: This tells the linker that if the module containing this directive is included in the executable, then so should `mod1:proc1()`, `mod1:proc2()`, etc.

`-proc_metacalls(source_proc, mod1:proc1, mod1:proc2, ...)`: This tells the linker that if the procedure `source_proc()` is included in the executable, then so should `mod1:proc1()`, `mod1:proc2()`, etc.

10 Process Mapping

Parallel compositions define concurrent processes; shared definitional variables define how these processes communicate and synchronize. Together with the sequential code executed by the different processes, these components define a concurrent algorithm that can be executed and debugged on a uniprocessor computer. However, we do not yet have a parallel program: we must first specify how these processes are to be mapped to the processors of a parallel computer.

Important features of PCN are that the mapping can be specified by the programmer and that the choice of mapping affects only the performance, not the correctness, of the program. In other words, the process mapping strategy applied in an application can change performance but cannot change the result computed. (The only exceptions to this rule are if foreign code uses global variables — e.g., common blocks — or if PCN code includes nondeterministic procedures.)

For this reason, it is common to develop PCN programs in two stages. First, program logic is developed and debugged on a workstation, without concern for process mapping. Second, a process mapping strategy is specified and its efficiency is evaluated on a parallel computer, typically by using the Gauge execution profiler.

The following language features are used when writing code to define process mappings.

Information Functions. When defining mappings, we sometimes require information about the computer on which a process is executing. This information is provided by the primitive functions `topology()`, `nodes()`, and `location()`.

`topology()`: Returns a tuple describing the type of the computer, for example `{"mesh",16,32}` or `{"array",512}`.

`nodes()`: Returns the number of nodes in the computer.

`location()`: Returns the location of the process on the computer.

Location Functions. Mapping is specified by annotating procedure calls with system- or user-defined *location functions*, using the infix operator "`@`". These functions are evaluated to identify the node on which an annotated call is to execute; unannotated calls execute on the same node as the procedure that called them. For example, the following two function definitions implement the location functions `node(i)` and `mesh_node(i,j)`, which compute the location of a procedure that is to be mapped to the `i`th node of an array and the (`i,j`)th node of a mesh, respectively. Note the use of a match (`?=`) to access the components of the mesh topology type. The per cent character, "`%`", is the modulus operator.

Example location functions: loc.pcn

```
function node(i)
{|| return( i%nodes() ) }

function mesh_node(i, j)
{?  topology() ?= {"mesh", rows, cols} ->
        return( (i*rows + j)%nodes() ),
    default -> error()
}
```

The following composition uses the function `node(i)` to locate the procedure calls `p(x)` and `c(x)`.

```
{|| p(x) @ node(10), c(x) @ node(20)}
```

Location functions are often used in the iterative construct called *quantification* (see § 4.7).

The following two procedures use quantifications and the location functions defined previously to execute the procedure `work` in every node of an array and mesh, respectively. For example, a call to `array` on a 1024-processor computer will create 1024 instances of `work()`, one per processor. (In practice, we may choose to use a more efficient tree-based spawning algorithm on a large machine.)

Examples of quantification

```
array()
{|| i over 0 .. nodes()-1 ::
        work() @ node(i)
}

mesh()
{?  topology() ?= {"mesh", rows, cols} ->
        {|| i over 0 .. rows-1 ::
            {|| j over 0 .. cols-1 ::
                work() @ mesh_node(i, j)
            }
        },
    default -> error()
}
```

Virtual Topologies and Map Functions. The ability to specify mapping by means of location functions would be of limited value if these mappings had to be specified with respect to a specific computer. Not only might this computer have a topology that was inconvenient for our application, but the resulting program would not be portable.

PCN overcomes this difficulty by allowing the programmer to define mappings with respect to convenient *virtual topologies* rather than a particular physical topology. A virtual topology consists of one or more virtual processors or *nodes*, plus a type indicating how these nodes are organized. For example, 512 nodes may be organized as a one-dimensional array, a 32×16 mesh, etc.

The embedding of a virtual topology in another physical or virtual topology is specified by a system- or user-defined *map function*. A map function is evaluated in the context of an existing topology; it returns a tuple containing three values: the type of the new embedded topology, the size of the new topology, and the function that is to be used to locate each new topology node in the existing topology. For example, the following function embeds a mesh of size `rows`×`cols` in an array topology; the mapping will be performed with the location function `node` provided previously in program `loc.pcn`. Note that the map function does not check whether the new topology "fits" in the old topology. It is quite feasible to create a virtual topology with more nodes than the physical topology on which it will execute.

Example Map Function

```
function mesh_in_array(rows, cols)
{?  topology ?= {"array", n} ->
    {|| type = {"mesh", rows, cols},
        size = rows*cols,
        map_fn = {"call",{":","loc","node"},[],[]},
        return( {type, size, map_fn} )
    },
    default -> error()
}
```

The assignment to the variable `map_fn` defines the function (in this case, `loc:node()`) that is to be used to compute the location of each new virtual node. The syntax for this is

```
var = {"call",{":",module,procedure},[arg0,...,argn],[]}
```

and specifies that the location function `module:procedure(`$\texttt{arg}_0$`,...,`$\texttt{arg}_n$`)` is to be used. The procedure arguments supplied in this "call" tuple will be prepended to the arguments that are supplied when the metacall using this "call" tuple is made.

This ugly syntax is due to a current limitation in the compiler that will be remedied in a future release.

We use the infix operator "`in`" to specify the map functions that will generate the virtual topologies used in different components of a program. For example, if the `mesh` procedure specified previously is executed on an array computer, we may invoke it as follows.

```
mesh() in mesh_in_array(rows,cols)
```

This map function, `mesh_in_array`, embeds a virtual mesh computer of size `rows`$\times$`cols` in the array computer.

Virtual topologies and map functions allow us to develop applications with respect to a convenient and portable virtual topology. When moving to a new machine, it is frequently possible to obtain adequate performance with just a naive embedding of this virtual topology. For example, our applications invariably treat all computers as linear arrays, regardless of their actual topology, and nevertheless achieve good performance. If communication locality were important (for example, if we moved to a machine without cut-through routing), we would probably have to develop a map function that provides a more specialized embedding. This can generally be achieved without changing the application code.

11 Port Arrays

Recall that individual processes communicate by reading and writing shared definitional variables, as in the composition {|| `producer(x), consumer(x)`}. The *port array* provides a similar mechanism for use when composing sets of processes.

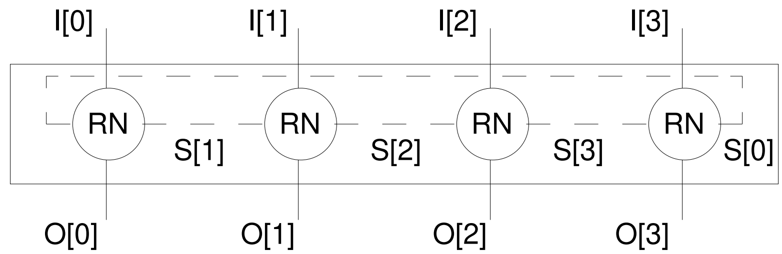
A port array is an array of definitional variables that has been distributed across the nodes of a virtual topology. A declaration "`port A[N];`" creates a port array `A` with `N` elements, distributed blockwise across the nodes of the virtual topology in which the port array is declared. (That is, elements are located on virtual topology nodes in contiguous, equal-sized blocks.) `N` must be an integer multiple of `nodes()`. Elements of a port array are accessed by indexing, in the same way as ordinary arrays; the elements can be used as ordinary definitional variables. Each element of a port array can only be accessed via indexing twice. This restriction allows memory occupied by port arrays to be reclaimed automatically.

The following procedure uses port arrays for two purposes: first, to provide each `ring_node()` process with definitional variables for use as input and output streams; and second, to establish internal communication streams between neighboring processes, so that each process has two streams, one shared with each neighbor. The `i`th node of this structure is given elements `I[i]` and `O[i]` of the two port arrays `I` and `O` passed as parameters (for communication with the outside world), and two elements of the local port array `S`. As in the C programming language, the dimension of an array passed as an argument is not specified. Notice the use of the information function, `nodes()` (§ 10), to define a port array with one element per virtual topology node.

Example of Ports: ring.pcn

```
ring(I, O)
port S[nodes()], I[], O[];
{|| i over 0 .. nodes()-1 ::
      ring_node(I[i], O[i], S[i], S[(i+1)%nodes()]) @ node(i)
}
```

The process structure created by a call to this procedure in a four-processor virtual topology can be represented as follows, with the solid lines indicating external port connections and the dotted lines internal streams. The box separates the internals of the process structure from what is visible to other processes. The `ring_node` procedure executed by each process can use the four definitional variables passed as arguments to communicate with other processes.



12 Reuse of Parallel Code

The ability to reuse existing code is vital to productive programming. The PCN system supports two forms of reuse: reuse of sequential code written in C or Fortran, and reuse of parallel code written in PCN. The former, which is discussed in § 8, is important when migrating existing sequential applications to parallel computers; the latter is becoming increasingly important as our parallel code base grows.

Cells. Our approach to the reuse of parallel code is based on what we term a *software cell*: a set of processes created within a virtual topology to perform some distinct function such as a reduction or a mesh computation, and provided with one or more port arrays for communication with other program components. We have already seen several examples of cells: for instance, the procedure `ring` in the preceding section implements a cell that performs ring pipeline computations.

The interface to a PCN cell consists simply of the port arrays and definitional variables that are its arguments. A cell definition does not name the processors on which it will execute, the processes with which it will communicate, or the time at which it expects to execute. These decisions are encapsulated in the code that composes cells to create parallel programs: a virtual topology specifies the number and identity of processors, port arrays specify communication partners, and the PCN compiler handles scheduling. As we will see in subsequent examples, the simplicity of this interface allows cells to be reused in many different contexts.

Templates. The `ring` cell would be more useful if the code to be executed at each node could be specified as a parameter. This is possible through the use of metacalls (§ 9), and in this case we refer to the cell definition as a *template*, as it encodes a whole family of similar cells. For example, the following is a template version of `ring`. The procedure to be executed is passed as the parameter `op`, which is quoted in the body to indicate that it is used as a variable.

Example Template

```
ring(op, I, O)
port S[nodes()], I[], O[];
{|| i over 0 .. nodes()-1 ::
        `op`(I[i], O[i], S[(i+1)%nodes()], S[i]) @ node(i)
}
```

This template invokes the supplied procedure with four definitional variables as additional arguments. For example, if `op` has the value `nbody(p)`, then a procedure call `nbody(p,d1,d2,d3,d4)` (`d1..d4` being the variables from the port array) is invoked on each node of the virtual topology. All parameters to `op` must be definitional variables; it is the programmer's responsibility to ensure that the number and type of these parameters match `op`'s definition.

Example. We illustrate how cells and templates are composed to construct complete applications. We make use of the ring template and also the following simple input and output cells: `load` reads values from a file and sends them to successive elements of the port array `P`; `store` writes to a file values received on successive elements of port array `Q`. Both use the sequential composition operator to sequence I/O operations.

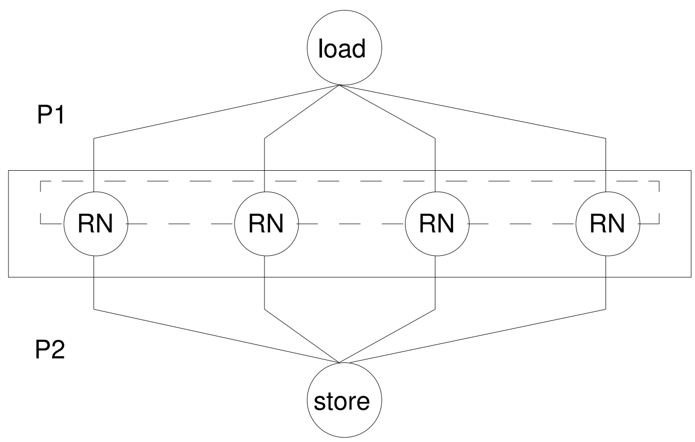
```
load(file, P)
port P[];
{;  i over 0 .. nodes()-1 ::
    {;  read(file, stuff),
        P[i] = stuff
    }
}

store(file, Q)
port Q[];
{;  i over 0 .. nodes()-1 ::
        write(file, Q[i])
}
```

We compose the three cells to obtain a procedure `compose` that reads data from `infile`, executes a user-supplied function in the ring pipeline (e.g., a naive N-body algorithm), and finally writes results to `outfile`. Note that although we use a parallel composition, data dependencies will force the three stages to execute in sequence. However, if `load` were to output a stream of values rather than a single value per node, then the three stages could execute concurrently, as a pipeline.

```
compose(param, infile, outfile)
port P1[nodes()], P2[nodes()];
{|| load(infile, P1),
    ring(nbody(param), P1, P2),
    store(outfile, P2)
}
```

Data flows from `load` to `ring` via port array `P1` and from `ring` to `store` via port array `P2`. This is illustrated in the following figure, which shows the process structure created in a four-node topology.



The complete program executes in an array topology ("`compose(param,if,of) in array()`") and will create a ring with one process per node of that topology.

13 Using Multiple Processors

A PCN program annotated with process mapping directives will execute correctly on a single processor. However, in order for the mapping directives to improve (or degrade!) performance, it is necessary to run the program on multiple processors.

The syntax used to start PCN on multiple processors varies according to the type of parallel computer. On multicomputers, we are generally required first to allocate a number of nodes and then to load the program in these nodes. For example, on the Intel iPSC/860, we must log into the host computer (System Resource Manager: SRM) and type the following commands to allocate 64 nodes, run the program, and finally free the allocated nodes.

```
% getcube -t 64
% load myprogram ; waitcube
% killcube
% relcube
```

On multiprocessors (e.g., a shared-memory Sun multiprocessor), we generally need only to add a `-n` flag to the command line when running the program. For example, to run on 4 processors, we type the following.

```
% myprogram myargs -pcn -n 4
```

The `-pcn` argument tells the run-time system that all subsequent arguments are run-time system arguments (not arguments meant for the user's program). The `-n 4` run-time system argument says to run this program on 4 processors.

The `-n` option can also be used to spawn multiple communicating PCN nodes on a uniprocessor workstation. This is not normally useful, however, as all nodes will just multitask on that workstation's simple processor. However, this option can be useful for debugging purposes under certain circumstances.

When running on a network, we generally need to either list on the command line the names of the computers on which to run nodes, or provide a configuration file indicating the names of the computers on which PCN is to run. See § 23 for more information about running PCN on networks.

For details about how to use PCN on your computer, turn to the discussion of machine dependencies in §§ 19–23.

14 Debugging PCN Programs

PCN provides a rich set of facilities for locating syntactic, logical, and performance errors in programs.

14.1 Syntax Errors

Syntax errors are detected and reported by the compiler. An error message consists of the file name, a line number, and a message indicating the type of error.

Warning messages are also generated by the compiler to indicate type mismatches between procedure definitions and calls, etc. It is good programming practice to write programs that do not generate warnings.

14.2 Logical Errors

Support for detection of *logical errors* is provided by the debugging version of the PCN run-time system. To use this version, you must add a `-pdb` argument to the `pcncomp` link line. Use of this version is recommended during program development. This debugging version provides a wide range of capabilities, including

1. Bounds checking is performed on all array and tuple accesses.

2. Checks are made for circular references, such as would be caused by {||`A=B,B=A`}.

3. The data immediately preceding and following data structures that are passed to foreign procedures are checked for validity upon return from the foreign procedure. This can help in locating array bounds violations in foreign code.

4. If you add a `-gc_after_foreign` as a run-time system command line argument (i.e., after the `-pcn` argument), a "garbage collection" (a technique for reclaiming unused storage) is invoked immediately after every foreign procedure call. This can help in tracking down foreign code that is corrupting PCN's internal data structures. A garbage collection involves a full consistency check of PCN's data structures. Since pure PCN code should never corrupt these data structures, a garbage collection failure while using this feature normally indicates that the previously called foreign procedure wrote outside its proper bounds. If the run-time system crashes with this feature turned on, the fatal error message that is printed will contain the name of the last foreign procedure that was called.

5. PCN object files (i.e., `.pam` files) can be dynamically loaded into an executable at run-time. This feature eliminates the need to relink your program each time you modify a PCN source file, and therefore can greatly speed the debug cycle of PCN programs. See § 15.9 for the details of this feature.

6. Typing Control-C (^C) during program execution provides access to the PCN symbolic debugger, PDB (§ 15).

Additional, low-level logical debugging support is provided by command line arguments that cause the PCN run-time system to print detailed information about individual procedure calls. These facilities are described in § 27; their use is not recommended in normal circumstances. This low-level debugging support can also be accessed through PDB variables.

14.3 Performance Errors

We use the term *performance error* to refer to programs that compute correct answers but for some reason do not make efficient use of available computer resources. Two tools are integrated with the PCN system to assist in the detection of performance errors: Gauge and Upshot. These are described in § 16 and § 17, respectively.

Gauge is an execution profiler: it collects information about the amount of time that each processor spends in different parts of a program. It also collects procedure call counts, message counts, and idle-time information. Two properties of Gauge make it particularly useful: profiling information is collected automatically, without any programmer intervention, and the volume of information collected does not increase with execution time. A powerful data exploration tool permits graphical exploration of profile data.

Upshot is a more low-level tool that can provide insights into the fine-grained operation of parallel programs. Upshot requires that the programmer instrument a program with calls to event logging primitives. These events are recorded and written to a file when a program runs. A graphical trace analysis tool then allows the programmer to identify temporal dependencies between events.

Part II
Reference Material

15 PDB: A Symbolic Debugger for PCN

Debuggers play an important role when programming in any language, including PCN. However, PCN is considerably different from sequential languages such as C and Fortran. For example, PCN uses both light weight processes and dataflow synchronization extensively. Therefore, a PCN debugger must have special capabilities designed to meet PCN's atypical requirements.

PDB, the PCN debugger, fits this bill. It incorporates features found in most debuggers, such as the ability to set breakpoints on procedures, interrupt execution, and examine program arguments. In addition, it incorporates capabilities that support atypical features of PCN, such as light weight processes and dataflow synchronization. In particular, PDB allows you to examine enabled and suspended processes and to control the order in which processes are scheduled for execution.

15.1 The PCN to Core PCN Transformation

The operation of PDB is complicated by the fact that the PCN run-time system does not support PCN directly, but rather a simpler language called core PCN, which lacks sequential composition and nested blocks. The PDB debugger operates on core PCN rather than PCN; hence, some understanding of the transformations used by the compiler to translate PCN to core PCN is necessary before PDB can be used effectively.

Nested Blocks. Nested blocks within PCN programs (except for sequential or parallel blocks nested in a top-level choice block) are replaced with calls to separate *auxiliary procedures* that contain these blocks. An auxiliary procedure is given the name of the procedure from which it was extracted, followed by an integer suffix. The choice of integer suffix is somewhat arbitrary; in general, however, suffixes are assigned in the order in which the corresponding auxiliary procedure calls appear in the original procedure.

Sequential Composition. Additional auxiliary procedures may be introduced as "wrappers" on operations occurring in sequential compositions. A wrapper delays execution of an operation until previous computations in the sequential composition have completed.

Wrappers are also generated to encode calls to primitive operations for which arguments may not be available at run time. Such wrappers delay computation until definitional arguments are defined. For example, a wrapper for the assignment `x:=y`, where `y` is a definition, will delay execution until `y` has a value.

Wrappers are named in the same manner as other auxiliary procedures: with a procedure name followed by a number.

Sequencing Variables. Every procedure has two additional variables added to its argument list. These variables are used for sequencing of procedure calls. They are commonly referred to as the *Left* and *Right* sequencing variables. A procedure will suspend until its *Left* variable is defined. When the procedure and its offspring have completed execution, *Right* is defined to be the same as *Left*. These variables often (but not always) occur at the end of the argument list.

Within a sequential block, the *Right* variable of one procedure call is the same as the *Left* variable of the next. This ensures that procedures execute in strict sequence. For example, the sequential block

Example of a sequential block

```
p()
{;  q(),
    r(),
    s()
}
```

is transformed to a procedure similar to the following.

Example of a transformed sequential block

```
p(L,R)
data(L) ->
    {|| q(L,M1),
        r(M1,M2),
        s(M2,R)
    }
```

Within a parallel block, all procedure calls use the parent procedure's *Left* variable as their own *Left*, and a temporary variable as their *Right*. The temporary *Right* variables are passed to a barrier procedure which defines the *Right* variable for the parallel block when all of the temporary variable have been defined. For example, the parallel block

Example of a parallel block

```
p()
{|| q(),
    r(),
    s()
}
```

is transformed to a procedure similar to the following (where `p.1` is the barrier procedure).

Example of a transformed parallel block

```
p(L,R)
data(L) ->
    {|| q(L,M1),
        r(L,M2),
        s(L,M3),
        p.1(M1,M2,M3,R)
    }

p.1(M1,M2,M3,R)
data(M1), data(M2) -> R = M3
```

Barrier Processes. As demonstrated in the preceding example, the PCN compiler sometimes generates calls to special *barrier* procedures. These are used to organize synchronization of procedures in a parallel block. These auxiliary programs are named in the same manner as other auxiliary procedures created by the compiler. However, they can usually be distinguished by the fact that all but one of their arguments are the *Right* synchronization variables of other procedures. Fortunately, these auxiliary barrier procedures can generally be ignored when debugging.

Wildcards. A procedure name is a `mod:procedure` pair. Some PDB commands that take procedure names as arguments allow the use of a limited form of a wildcard facility to specify a set of procedures. An asterisk (*) placed at the end of a procedure name designates all procedures that begin with the specified name. For example, `mod:program1*` designates all procedures in module `mod` whose names begin with `program1`. The degenerative case of a procedure wildcard is simply a `*` (e.g., `mod:*`). In this case, all procedures within the appropriate module are specified.

Module names can also be specified with this limited wildcard facility. For example, a module wildcard of `env*` designates all modules whose names start with `env`, and a simple `*` designates all loaded modules.

15.2 Obtaining Transformed Code

As described in § 15.1, a PCN program is transformed to core PCN before execution. When debugging programs with PDB it is often helpful to have this transformed version of the code available for reference, since that transformed version is really the code that is being executed.

When compiling your PCN program, you can have the compiler dump the transformed version of your program simply by adding a `-dumpafter basic` flag to the `pcncomp` compile line. Assuming the original PCN program is named `prog.pcn`, a file named `prog.basic.dump` will be created that contains a nicely formatted representation of the transformed PCN program.

15.3 Naming Processes

Execution of a PCN program can create a large number of lightweight processes. Each process executes a PCN procedure — either a procedure named in the original source, or an auxiliary procedure introduced by the transformation to core PCN.

In order to simplify the task of distinguishing between the many processes that may be created during execution of a program, PDB associates three distinct labels with each process.

1. The name of the procedure that the process is executing (nonunique).
2. An instance number (unique).
3. The process reduction in which the process was created (nonunique).

Note: A *reduction* is one completed execution of a process. The run-time system keeps a reductions counter that it increments after each reduction.

As we shall see in §15.5, PDB also provides information about the status of a process, for example, whether it is able to execute or is waiting for data.

15.4 Using the Debugger

Linking with PDB run-time system. To use PDB, you must link your program with the PDB version of the run-time system. This is accomplished by simply adding a `-pdb` argument to the `pcncomp` link command.

-pdb command line argument. When you run your program, if you add a `-pdb` run-time system command line argument (i.e., after the `-pcn` argument), your program will be interrupted and control passed to PDB before any PCN procedures are executed.

Control-C. Once your program is running, you can enter PDB by interrupting the program with an interrupt signal. This signal is typically invoked by typing Control-C (^C). If you interrupt your program while it is executing a foreign procedure, that foreign procedure will be completed before control is passed to PDB.

Once control is passed to the debugger, PDB commands can then be used to examine the state of the computation, set breakpoints on PDB procedures, enable/disable debugging on procedures, or resume execution of the PCN program. Once resumed, normal PCN execution continues until you interrupt the program execution again or until a breakpoint is encountered, causing control to revert back to the debugger. It is also possible to specify that control pass to the debugger if the active queue becomes empty. This is accomplished by setting the debugger variable `empty_queue_break` (§ 15.7).

.pdbrc When a program that is linked with the PDB version of the run-time system starts up, it searches for a `.pdbrc` file in the current directory, and then in your home directory (~). Any PDB commands found in such a file are executed. This feature allows the state of PDB to be initialized every time PDB is run.

Abbreviating PDB Commands. PDB commands can be abbreviated to the shortest string that uniquely identifies the command. (There are a few exceptions to this rule. For example, since the `show` command is typically used extensively, it can be abbreviated to `s`, even though `s` does not uniquely identify this command.)

To find out the shortest abbreviation for PDB commands, use the PDB help facility by typing `help` at the PDB command prompt.

Help. PDB has extensive online help — type `help` at the PDB command prompt for more information.

15.5 Examining the State of a Computation

We now describe the PDB commands used to examine the state of a PCN computation. For you to understand how these commands work, we need to say a little bit about how the PCN run-time system manages execution of PCN programs.

Queues. The PCN run-time system manages the execution of processes created to execute procedure calls in parallel blocks. Like a simple computer operating system, it selects processes from an *active queue* and executes them either until they block because of a read operation on an undefined definitional variable or until a timeslice is exceeded. In the former case, the process is moved to a *variable suspension queue* associated with the undefined definitional variable (unless the process requires two or more variables, in which case it is moved to a *global suspension queue*). In the latter case (a timeslice), the process is moved to the end of the active queue. PDB also maintains a fourth *pending queue*. This is used to hold processes from the active queue that the user has indicated are to be delayed (i.e., removed from consideration by the PCN scheduler).

In summary, every PCN process is to be found on one of the following four queues:

`active` The *active queue* contains processes that may be scheduled for execution.

`pending` The *pending queue* contains processes that the user has tagged to be delayed. These cannot be executed until returned to the active queue.

`globsusp` The *global suspension queue* contains processes that are suspended on more than one variable.

`varsusp` The *variable suspension queue* contains processes that are suspended on just one variable.

When describing commands, we shall use the notation <queue> to represent a queue selector— one of `active`, `pending`, `globsusp`, and `varsusp`; or `suspension` (both `globsusp` and `varsusp`) and `all` (all process queues).

We shall also use the notation <process> to represent a process specification; this is one of the following:

- n: n is an integer, representing an index into a process queue;
- $m - n$: m and n are integers, representing a range of indices into a process queue;
- #n: n is an integer, representing a process instance number;
- ^n: n is an integer, representing the reduction during which a process was created;
- _Uh: h is a hexadecimal number, representing an undefined variable that is somewhere in a process's argument list;
- *modulename*:*blockname*, representing all processes of a given name;
- `all`.

As noted in § 15.1, a limited wildcard facility allows a single <process> specifier to represent several processes.

Examining Queue Contents. The `summary`, `list`, and `show` commands allow the user to examine the four process queues at increasing levels of detail. These commands (and the queue-manipulation commands described in the next section) operate only on processes executing procedures for which debugging is enabled. The set of enabled procedures is initially all procedures; the set can be modified by using the **debug** and **nodebug** PDB commands.

In the following descriptions, all arguments that are listed within square brackets ([ ]) are optional:

`summary` [<queue>] [<process>]: prints a summary of the contents of the designated <process> on the designated <queue>. This includes module and procedure names (sorted by module and then procedure) and the number of occurrences of each procedure on each queue.

`list` [<queue>] [<process>]: prints a short listing of the processes specified by <process> on the specified <queue>.

`show` [<queue>] [<process>]: prints a detailed description of the processes specified by <process> in the specified <queue>. If the process is on the variable suspension queue, the variable that it requires in order to continue execution is also shown.

Modifying Queues. The `move` and `switch` commands are used to control how processes in the active queue are selected for execution. They can be applied only to the active and pending queues.

`move` <queue> <process> [<where>]: This moves zero or more designated processes in a designated queue (`active` or `pending`) to immediately before position *where* in the same queue. If *where* is `end`, then the designated processes are moved to the end of the queue. By default, <where> is `end`.

`switch` <queue> <process> [<where>]: This moves zero or more designated processes from a designated queue (`active` or `pending`) to the other queue (i.e., `pending` or `active`, respectively), inserting them immediately before position *where*. If *where* is `end`, the designated processes are placed at the end of the queue. By default, <where> is `end`.

15.6 Breakpoints

PDB allows breakpoints to be set on PCN procedures. When a process that is executing a procedure for which a breakpoint is set is scheduled for execution, the run-time system will interrupt the program and pass control to PDB.

Note that a process may be scheduled for execution several times before it is able to complete. For example, a process may be scheduled but will subsequently suspend because of an undefined variable. When that variable is later defined, the process will again be scheduled. A breakpoint on that process's procedure will cause a break into PDB each time the process is scheduled.

The `break`, `delete`, `enable`, and `disable` commands control breakpoints, and `status` prints information about breakpoints.

`break` [<module>:<procedure> ...]: Set a breakpoint on the specified procedure. If no procedures are given, then all current breakpoints are listed.

`delete` [<breakpoint_number> ...]: Delete the specified `breakpoint_number`. The `breakpoint_number` can be determined by running the `break` command with no arguments. If no breakpoints are given, then all breakpoints will be deleted.

`disable` <breakpoint_number> ...: Disable (but do not delete) the specified breakpoint.

`enable` <breakpoint_number> ...: Enable the specified breakpoint.

`status` [<module>:<procedure> ...]: Print breakpoint status information about the specifed procedure(s).

15.7 Debugger Variables

PDB maintains a number of internal variables that can be included in some PDB commands and, in some cases, modified by the programmer. PDB variables are distinguished in expressions by a prefix $.

Modifiable Variables. The following variables can be used to control aspects of PDB's behavior. They can be modified within PDB by using the "`=`" command.

`$print_array_size`: An integer representing the maximum size (i.e., number of elements) of an array displayed by `print`.

`$print_tuple_depth`: An integer representing the maximum depth of a tuple displayed by `print`.

`$print_tuple_width`: An integer representing the maximum width (i.e., number of elements) of a tuple displayed by `print`.

`$emulator_dl`: An integer representing the emulator debug level. This turns on the printing of debugging information in the main emulator loop. It takes an integer value between 0 and 9, where 0 is no debugging and 9 is the most debugging. See § 27 for more information on this variable.

`$gc_dl`: An integer representing the garbage collection debug level. This turns on the printing of debugging information in the garbage collector. It takes an integer value between 0 and 9, where 0 is no debugging and 9 is the most debugging. See § 27 for more information on this variable.

`$parallel_dl`: An integer representing the parallel code debug level. This turns on the printing of debugging information in the parallel emulator code. It takes an integer value between 0 and 9, where 0 is no debugging and 9 is the most debugging. See § 27 for more information on this variable.

`$global_dl`: An integer representing the global debug level. This turns on the printing of debugging information not covered by the `$emulator_dl`, `$gc_dl`, or `$parallel_dl` debug setting. It takes an integer value between 0 and 9, where 0 is no debugging and 9 is the most debugging. See § 27 for more information on this variable.

`$reduction_break`: An integer representing the next reduction at which to break into PDB.

`$empty_queue_break`: A Boolean value. When this value is set to `yes`, the system will break into PDB whenever the process queues are empty, and therefore there are no schedulable processes. When this value is set to `no`, the system will not break into PDB whenever the process queues are empty.

`$print_orphaned`: A Boolean value. When this value is set to `yes`, the system will print out a warning when it encounters an orphan process (§ 15.10) during a garbage collection.

Read-Only Variables. The following variables contain information about various aspects of the state of the computation. They can be included in expressions but cannot be modified directly.

`$module`: The name of the current module (i.e., first process on the active queue).

`$procedure`: The name of the current procedure (i.e., first process on the active queue).

`$args`: The arguments of the current process. Note that this variable is defined only at the entry to a block.

`$instance`: The instance number of the current process.

`$reduction` : The reduction during which the current process was created.

`$current_reduction`: The current reduction number.

15.8 Miscellaneous Commands

This section describes miscellaneous debugger commands that were not described in other parts of this manual.

In the following, <expr> denotes either a PCN variable name (to be interpreted in the context of the current process) or a constant.

abort: Abort execution of the PCN run-time system. See also `continue`, `next`, and `quit`.

continue: Continue with next process (head of the active queue). See also `abort`, `next` and `quit`.

debug *<module>*:*<procedure>* **...:** Enable debugging in the specified *module*:*procedure*. See also `nodebug`.

help [*<topic>* :] Give help for *topic*. If *topic* is left off, then general help will be given.

load *<filename>*: Load the `.pam` file, *filename*, into the run-time system.

modules: List the names of the modules that are currently loaded in the system, indicating for each whether it was compiled in debug mode (in the current PCN release, this column always says "n") and whether debugging is enabled.

next: Execute the next process (head of the active queue), and then break into the debugger again when it has completed. See also `abort`, `continue`, and `quit`.

nodebug *<module>*: Disable debugging in the specified *module*:*procedure*. See also `debug`.

print <expr>: Print the given expression. An expression is a variable, integer, real, or string. <expr> is either a single expression or a comma-separated list of expressions that is enclosed in parentheses.

procedures *<module>*:*<procedure>* **...:** Print various information about the specified procedure(s).

quit: Quit from the debugger; disable debugging on all modules. See also `abort`, `continue`, and `next`.

source *<filename>*: Execute the PDB commands that are in the file *filename*.

vars: List the names and values of all PDB variables.

15.9 Dynamic Loading of `.pam` Files

The PCN linker is relatively slow. In order to accelerate the modify-compile-link-test program development cycle, PDB supports dynamic loading of PCN object files (i.e., `.pam` files), which eliminates the link step from this cycle when PCN files are modified.

When a `.pam` file is dynamically loaded into a running program, the procedures in that file simply replace previously linked versions of those procedures. If there are procedures in this `.pam` file that did not previously exist in the executable, then they will be added.

The `-pdb` flag must be passed to the linker (via `pcncomp`) if dynamic loading is to be used. The `-link_all` flag is also recommended. The latter flag tells the linker to include all procedures in all modules named on the command line, rather than just those procedures reachable from the entry point. This ensures that standard libraries, such as sys and stdio, are included in their entirety. Thus, you can dynamically load code that calls library procedures that were not called in the original program.

For example, you might link your program with the command:

```
pcncomp -pdb -link_all mymod1.pam mymod2.pam -mm mymod1 -o myprogram
```

There are two ways in which `.pam` files can be dynamically loaded:

From the command line: You can use the `-load` command line argument to cause a set of `.pam` files to be dynamically loaded before any PCN procedures are executed. For example, the command

```
myprogram myargs -pcn -load mymod1.pam:mymod2.pam
```

will dynamically load the procedures in `mymod1.pam` and `mymod2.pam` into `myprogram`, overwriting those provided at link time.

From the PDB prompt.: The `load` command descibed in § 15.8 will dynamicly load `.pam` files into the executable. This, of course, means that you can dynamically load `.pam` files from a `.pdbrc` file, which is a useful feature if you are repeatedly running the program and do not wish to type the `-load` command line argument each time.

15.10 Orphan Processes

An *orphan* is a process suspended on a variable for which there are no potential producers (more precisely, a variable to which no other process possesses a reference). Such a process can never be scheduled for execution. A program that generates orphan processes is not necessarily erroneous. However, it is good programming practice to ensure that orphans are not generated (i.e., that all processes in a program terminate).

Orphan processes can be detected by the garbage collector invoked by the PCN run-time system to reclaim space occupied by inaccessible data structures. Normally, the garbage collector destroys these processes silently. However, the PDB version of the PCN run-time system prints a warning message for each orphan encountered.

The PDB variable, `$print_orphaned`, can be used to disable these orphan process messages (see § 15.7).

16 The Gauge Execution Profiler

Gauge is an execution profiler for PCN programs. It collects profile data such as the time spent in each procedure on each node, the number of times each procedure is called, idle times, and internode message counts and volumes. This profile data can subsequently be graphically displayed by using an interactive data exploration tool.

16.1 Linking a Program for Profiling

In order to collect a Gauge profile on a program, you must first link your program with a version of the run-time system that supports profiling. To do this, simply add a `-profile` to your `pcncomp` link command, for example:

```
pcncomp myprogram.pam -o myprogram -mm myprogram -profile
```

By default, only procedures from the the user's modules, the stdio library, and sys library are profiled; the system modules used to implement process mapping, etc., are ignored. However, you have full control over which modules will be profiled, through the use of linker arguments:

`-no_nmp`: Turns on full profiling, including system modules.

`-nmp` <*module*>: Turns off profiling on the *module*. (NMP stands for No Module Profile.)

16.2 Profile Data Collection

A profile is generated by executing your program with a `-gauge` and/or `-gauge_file` command line argument. The profile will be performed on all nodes, and on all modules for which profiling was enabled during linking. If the `-gauge` flag is used, the profile will be written into the file `profile.cnt`. The `-gauge_file` flag allows a different filename to used. For example, the following command runs `myprogram` and writes a profile into the file `myprof.cnt`.

```
myprogram -pcn -gauge_file myprof.cnt
```

16.3 Snapshot Profiles

By default, a profile will be taken only at the end of the run and is cumulative for the entire run. However, it is sometimes useful to examine several profiles of your program collected at various stages of execution. This can be accomplished by calling

```
profile_snapshot(snapshot_name)
```

in your program, where `snapshot_name` is a string that will be used to name the snapshot. A call to this foreign procedure on *any* node will cause a snapshot profile to be generated on *all* nodes. (Note that this means that it is a serious mistake to call `profile_snapshot()` on *every* node; this will generate P snapshots, each involving all P processors.) Each snapshot is cumulative – the profile is *not* reset after a snapshot, so procedure executions times, etc., include the times from previous snapshots.

A call to `profile_snapshot()` has no effect unless you have linked with the profiling version of the run-time system (i.e., used the `-profile` flag when linking), and you have enabled profiling (i.e., used the `-gauge` or `-gauge_file` command line flag). Hence, calls to `profile_snapshot()` can be maintained in a program and enabled when required from the command line.

16.4 Data Exploration

Gauge provides an X-windows-based graphical interactive tool for exploring profile data collected by using the methods described above. This tool combines three sorts of data to provide detailed information about execution time on a per-procedure and per-processor basis, idle time, number of messages, volume of messages and other program execution statistics. These data are

- instruction counts collected by the compiler,
- profile data collected by the run-time system when a profile is taken, and
- information about the computer on which the program was run.

The Gauge analysis tool gets the first and second of these from the `.cnt` file that is produced by the run-time system when a profile is taken. The third is taken from the host file which may need to be specified by the user (see §16.5).

The data exploration tool is invoked by typing the following Unix command:

```
gauge
```

This creates a top-level window with three parts. The top section of the window is a command window. You can click the left mouse button on one of the commands to obtain *help*, to `exit`, or to invoke the `gauge` analysis window. The middle section indicates the current directory. The bottom window gives a list of `.cnt` and `.cnt.Z` (compressed `.cnt`) files and directories in the current directory. Files are selected by pressing the left mouse button while the pointer is over the file name. If you wish to change selections, just press the left mouse button over a different file, or no file if you want to eliminate all selections.

The directory window serves two purposes. If you select a `.cnt` file in the directory window using the left mouse button and then select the *Gauge* command from the top row of buttons, Gauge is invoked on that file. Gauge can also be invoked on a `.cnt` or `.cnt.Z` file by double-clicking on its name in the directory window. Double-clicking on a directory name opens that directory, thus allowing navigation of the directory system.

Gauge has an online help facility. To use it, select the "help" button on any window. Either the scroll bar or the page-down (Control-v) and page-up (Meta-v) commands can be used to position the help text within the help window. When finished, you can dismiss the help screen using the *close* button on the bottom of the screen. If you leave the screen up, it will be reused to display the next help message.

Occasionally something might go wrong, and Gauge will generate a warning message in a popup window. Nothing else can be done until this window is dismissed by clicking the left mouse button in it.

The only command-line arguments recognized by Gauge are those recognized by the X Toolkit Intrinsics. This means that X-windows arguments such as `-icon` can be used.

16.5 The Host Database

When you invoke Gauge on a `.cnt` or `.cnt.Z` file, a warning message may be displayed indicating that your machine does not appear in the host database. (Click on the warning window to make it disappear.) This means that you must add the machine on which your application was run to the *host database* that Gauge accesses to determine various machine characteristics when displaying performance data.

The program `pcnhost` is provided to simplify the task of adding entries to the host database. A call to this program has the form

```
pcnhost machinetype
```

or

```
pcnhost -h hostname machinetype
```

The `machinetype` argument specifies an architecture type for machine computer `hostname`. If a host name is not specified, the name of the machine on which the pcnhost command is executed is added to the database. The following machine types are currently supported:

- `symmetry-b`, `symmetry`: Sequent Symmetry Rev. B
- `sparcstation-1`, `ss1`, `sun4`: A Sun SPARCstation 1
- `sun3`: A Sun 3 workstation

- `next040`: A NeXT workstation
- `iris`: An SGI Iris workstation
- `s2010`: A Symult s2010 multicomputer
- `rs6000`: An IBM RS/6000 workstation
- `ipsc860`: An Intel iPSC/860 (i860 processing nodes)
- `ipscii`: An Intel iPSC/II (386 nodes)

Note that updates to the database are not synchronized. If more than one update is being made simultaneously, information can be lost.

16.6 X Resources

Gauge requires a resource file to operate properly. This should be in

```
$(INSTALL_DIR)/lib/app-defaults/gauge
```

where `$(INSTALL_DIR)` is the directory where gauge has been installed (typically, `/usr/local/pcn`). The command

```
xrdb -merge $(INSTALL_DIR)/lib/app-defaults/gauge
```

should be added to one's `.xinitrc` or `.xsession` file. If a color workstation is being used,

```
xrdb -merge $(INSTALL_DIR)/lib/app-defaults/gauge.server
```

is also needed. Of course, any customized resource files could be used.

Alternatively, you can run `xrdb_gauge` before running `gauge`. The `xrdb_gauge` program simply executes the two `xrdb` commands shown above.

17 The Upshot Trace Analyzer

`Upshot` is a trace collection and analysis tool. There are three steps that you need to perform in order to use Upshot with PCN:

1. Instrument a program.
2. Run your instrumented program and collect a log.
3. Analyze the log.

The last step requires that you obtain and install the X windows based Upshot log event analysis tool. You can obtain it by anonymous ftp from:

```
info.mcs.anl.gov
```

in the directory

```
pub/upshot
```

17.1 Instrumenting a Program

You instrument your program by adding calls to procedures which, when executed, log a timestamped *event*. An event consists of a type and an optional task identifier and data value. You can instrument PCN, C, and Fortran code.

To instrument your program, you must first add:

```
#include "pcn_upshot.h"
```

to PCN and C source files that will contain event logging calls.

Then, the following calls can be added to your PCN, C, and/or Fortran source to log Upshot events:

- `LOG_EVENT(event_type)`
- `LOG_TASK_EVENT(task_id, event_type)`
- `LOG_TASK_EVENT_DATA(task_id, event_type, data_val)`

In these calls, `task_id` and `event_type` are positive integers, and `data_val` is an integer. None of these calls return a value.

17.2 Compiling and Linking the Instrumented Program

In PCN and C code, the above-mentioned `LOG*` calls are actually macros that call the correct procedures if the C preprocessor variable `UPSHOT` is defined, and which do nothing if it is not. Therefore, when compiling your PCN and C source with these calls, you need to add a `-DUPSHOT` argument to have them take effect:

```
pcncomp -c pcnsource.pcn -DUPSHOT
pcncomp -c csource.c -DUPSHOT
```

In Fortran, the above-mentioned `LOG*` calls are just calls to procedures that are defined in the run-time system. Therefore, you can compile your instrumented Fortran source as usual:

```
pcncomp -c fsource.f
```

When linking a program that contains event logging calls, you must add a `-profile` flag to the `pcncomp` link command:

```
pcncomp pcnsource.pam csource.o -mm pcnsource -profile -o myprogram
```

17.3 Collecting a Log

A program that contains event logging calls stores events in memory when it is executed. When it completes execution, it writes these events to files, one per processor.

A program only collects a log if the `-upshot` command line argument is specified. For example:

```
myprogram myargs -pcn -upshot
```

This causes a log file to be written for each node on which the program is running. These files are called `log.0`, `log.1`, etc. The prefix of the log filename can be changed by using the `-upshot_file` flag.

By default, a program only allocates memory for 10,000 events. An error is reported if more than this number of events are logged. The maximum number of log events can be changed by using the `-upshot_log_size` flag.

The following example collects a log during the execution of `myprogram`, puts the logs in files with a `mylog` prefix, and can record a maximum of 20,000 log events:

```
myprogram myargs -pcn -upshot_file mylog -upshot_log_size 20000
```

17.4 Analyzing a Log

As previously mentioned, execution of a program with the `-upshot` argument produces one log file for each node, for example, `log.0`, `log.1`, etc. These files must be merged by using the Unix command `mergelogs` to create a single log file, for example,

```
mergelogs log.* > log
```

We can then call the Upshot visualization program to display a set of time lines, one per processor, with the various events logged by our program displayed on the appropriate time lines:

```
upshot -l log
```

Frequently, we are not interested in the events themselves but rather in execution states defined in terms of starting events and ending events. For example, we might define a "busy" state as starting when an event is logged indicating that a message has been received on a stream, and ending when an event is logged indicating that a response has been sent. We define states in a states file, specifying each state in terms of a unique integer identifier, a starting and an ending event type, a color, and a label, for example:

File `my.sts`

```
1  10  11  blue    init_ico
2  12  13  red     init_rh
3  14  15  pink    init_geo
4  16  17  yellow  get_side
```

Upshot does not support nested states. That is, it is not meaningful for a trace to include sequences in which two start state events occur without an intervening end state event.

The name of any state file is specified to Upshot by means of the `-s` command line option, as follows.

```
upshot -l log -s my.sts
```

18 Standard Libraries

The `sys` and `stdio` modules are distributed with the PCN system and may be called from within user programs to invoke a variety of useful functions. They are invoked via intermodule calls.

In the following discussion, the notations $\downarrow$ and $\uparrow$ on program arguments denote input and output arguments, respectively.

18.1 System Utilities

The `sys` module provides the following general utility procedures.

`merger(Is`$\downarrow$`,Os`$\uparrow$`)` merges messages appearing on input stream `Is` to produce output stream `Os`. If the input stream `Is` contains a message of the form {`"merge",S`}, then the stream `S` is also merged with `Os`. The output stream `Os` is closed when all merged input streams are closed. (Cf. § 4.10 for more details.)

`distribute(N`$\downarrow$`,Is`$\downarrow$`)` distributes messages received on input stream `Is` to `N` output streams; output streams are numbered `0` to `N-1`. (Cf. § 4.10 for more details.) The distributor may receive three types of message on input stream Is:

- {`"attach",N1`$\downarrow$`,S`$\downarrow$`,D`$\uparrow$} causes stream `S` to be attached to output stream numbered `N1`; `D` is defined when the action is complete to signify that messages may subsequently be forwarded to stream `S`.
- {`N2`$\downarrow$`,M`$\downarrow$} causes the message `M` to be appended to output stream numbered `N2`.
- {`"all",M`$\downarrow$} causes the message `M` to be appended to all of the output streams. (I.e., Broadcast the message to all output streams.)

When the input stream Is is closed, all output streams are closed. (Cf. § 4.10 for more details.)

`hash(N`$\downarrow$`,Is`$\downarrow$`)` creates a hash table of size `N` and receives messages on input stream `Is`. Four messages may be sent to a hash table:

{`"add",K↓,V↓,S↑`} causes the value `V` to be added to the hash table under key `K`; if there was already an entry for key `K`, then status `S=0`; otherwise `S=1`.

{`"lookup",K↓,V↑,S↑`} causes a lookup operation on key `K`. If there is an entry for key `K`, then `V` is the associated value and status `S=1`; otherwise `S=0`.

{`"del",K↓,V↑`} deletes the entry for key `K` and returns the value `V` associated with the entry if one existed; otherwise returns `-1`.

{`"dump",L↑,D↑`} dumps the contents of the hash table into a list `L` and defines `D` when the operation is complete.

`integer_to_list(I↓,Lb↑,Le↓)` difference list `Lb-Le` is defined to be the list containing the integers of the ASCII representation of integer `I`.

`list_to_integer(L↓,I↑)` `I` is defined to be the integer that is represented by the ASCII values (integers) in the list `L`.

`integer_to_string(I↓,S↑)` `S` is defined to be the string that represents the integer `I`.

`string_to_integer(S↓,I↑)` `I` is defined to be the integer that is represented by the string `S`.

`double_to_list(D↓,Lb↑,Le↓)` difference list `Lb-Le` is defined to be the list containing the integers of the ASCII representation of the double `D`.

`list_to_double(L↓,D↑)` `D` is defined to be the double that is represented by the ASCII values (integers) in the list `L`.

`double_to_string(D↓,S↑)` `S` is defined to be the string that represents the double `D`.

`string_to_double(S↓,D↑)` `D` is defined to be the double that is represented by the string `S`.

`list_to_string(L↓,S↑)` `S` is defined to be the string that is represented by the ASCII values (integers) in the list `L`.

`string_to_list(S↓,Lb↑,Le↓)` difference list `Lb-Le` is defined to be the list containing the integers of the ASCII characters in the string `S`.

`list_to_tuple(L↓,T↑)` `T` is defined to be the tuple with elements specified by list `L`.

`tuple_to_list(T↓,Lb↑,Le↓)` difference list `Lb-Le` is defined to be the list containing the arguments of tuple `T`.

`integer_to_double(I↓,D↑)` `D` is defined to be the double cast of the integer `I`.

`double_to_integer(D↓,I↑)` `I` is defined to be the integer cast of the double `D`.

list_length(L↓,Len↑) Len is defined to be the length (integer) of the list L.

list_concat(L1↓,L2↓,Lout↑) Lout is defined to be the concatenation of list L1 followed by list L2.

list_member(element↓,list↓,status↑) status is defined to be the integer 1 if element is a member of the list list, and the integer 0 if it is not.

left_shift(Src↓,N↓,Dest↑) Dest is defined to be the integer Src left shifted N bits. (Dest = Src << N)

right_shift(Src↓,N↓,Dest↑) Dest is defined to be the integer Src right shifted N bits. (Dest = Src >> N)

ones_complement(Src↓,Dest↑) Dest is defined to be the one's complement of the integer Src. (Dest = ~Src)

bitwise_and(Src1↓,Src2↓,Dest↑) Dest is defined to be the bitwise and of integers Src1 and Src2. (Dest = Src1 & Src2)

bitwise_or(Src1↓,Src2↓,Dest↑) Dest is defined to be the bitwise inclusive or of integers Src1 and Src2. (Dest = Src1 | Src2)

bitwise_xor(Src1↓,Src2↓,Dest↑) Dest is defined to be the bitwise exclusive or of integers Src1 and Src2. (Dest = Src1 ^ Src2)

double_cast(From↓,To↑) To is defined to be the double cast of the integer or double From.

integer_cast(From↓,To↑) To is defined to be the integer cast of the integer or double From.

abs(From↓,To↑) To is defined to be the absolute value of From. If From is a double, then To will be a double. If From is an integer, then To will be an integer.

string_length(S↓,Len↑) Len is defined to be the length (integer) of the string S.

string_concat(S1↓,S2↓,Sout↑) Sout is defined to be the string that is the concatenation of string S1 followed by string S2.

string_list_concat(string_list↓,separator↓,Sout↑) string_list is a list of strings, and separator is a string. Sout is defined to be the string that is the concatenation of the strings in string_list with the separator between each.

find_substring(string↓,substring↓,index↑) index is defined to be the integer location (starting with 0) of the first occurence of the string substring in the string string, or -1 if substring is not a substring of string.

`find_substring_reverse(string↓,substring↓,index↑)` `index` is defined to be the integer location (starting with 0) of the last occurance of the string `substring` in the string `string`, or -1 if `substring` is not a substring of `string`.

`substring(string↓,start↓,len↓,substring↑)` `substring` is defined to be the substring of `string` starting at location `start` (numbering starts with 0) with the length `len`. If `len` is -1, then the substring starting at `start` through the end of `string` will be extracted.

18.2 Standard I/O

The `stdio` module provides a set of PCN procedures that are analogous to the C language standard input/output (stdio) library. It is important to realize that calls to `stdio` are sequenced only if they occur within a sequential block. Output generated by *parallel* calls to `printf` or other output procedures may be interleaved.

Most of the `stdio` procedures take an output argument, `status`. This argument should be an undefined variable when the call is made. It will be defined by the `stdio` procedure to an appropriate return code. This argument can be used both to check the status of the I/O call and to sequence subsequent execution if necessary.

The `stdio` procedures that deal with files rather than the keyboard or screen require a file pointer (`fp`) argument. This argument should be a mutable of type `FILE` (defined in the C header file `pcn_stdio.h`).

18.2.1 Reference

We now summarize the procedures provided by the `stdio` module. The arguments to all of these procedures follow as closely as possible their corresponding C procedures. Please refer to a C programming manual for more complete descriptions.

`fopen(filename↓,type↓,fp↑,status↑)` opens the file named `filename`. The file is opened for the given `type` of I/O operation, where `type` is a string containing an appropriate combination of `"r"`, `"w"`, `"a''`, and `"+"`. The mutable `fp` is assigned to be the file pointer. `status` is defined to be 0 if the open succeeds; otherwise it will be set to the error number (C `errno`).

`freopen(filename↓,type↓,fp↑↓,status↑)` like `fopen()`, except that it substitutes the named file in place of the open stream, `fp`. This is typically used to attach the preopened `stdin`, `stdout`, and `stderr` to specified files.

`fdopen(fildes↓,type↓,fp↑,status↑)` opens the file with the integer file descriptor `fildes`. The other arguments are the same as for `fopen()`.

`fclose(fp↓,status↑)` closes the file designated by `fp`. `status` is defined to be `EOF` if there is an error.

`fflush(fp↓,status↑)` flushes all buffered data for the output file designated by `fp` to be written to that file. The file remains open. `status` is defined to be `EOF` if there is an error.

`putc(c↓,fp↓,status↑)` appends the character `c` to the designated output stream `fp`. `status` is defined to be the character written, or `EOF` if there is an error.

`fputc(c↓,fp↓,status↑)` is the same as `putc()`.

`putchar(c↓,status↑)` is the same as `putc()` to standard output (the screen).

`puts(s↓,fp↓,status↑)` appends the string `s` followed by a newline to standard output. `status` is defined to be `EOF` if there is an error.

`fputs(s↓,fp↓,status↑)` appends the string `s` (*not* followed by a newline) to the designated output stream `fp`. `status` is defined to be `EOF` if there is an error.

`printf(format↓,args↓,status↑)` prints formatted output to standard output. The `format` string accepts the same format as the C language's `printf()` procedure, with two additions: it can contain a `%t`, which means to print a grounded term, and `%lt`, which means to print an ungrounded term. The `%t` and `%lt` can also take an integer immediately after the `%`, which means to print only to that depth. The `args` argument is a tuple of all the arguments to `printf`, as required by the `format`. (Since PCN procedures cannot take a variable number of arguments, as in C, all of the data arguments must be combined into a single argument using a PCN tuple.) `status` is defined to be the number of characters written, or `EOF` if there is an error.

`fprintf(fp↓,format↓,args↓,status↑)` is the same as `printf()`, except that output will go to `fp` rather than to standard output.

`sprintf(buf↑,format↓,args↓,status↑)` is the same as `printf()`, except that the output is placed into the definitional variable `buf`.

`getc(fp↓,c↑)` gets one character from the input stream `fp` and defines it to `c`. `c` is defined to be `EOF` on end of file or an error.

`fgetc(fp↓,c↑)` is the same as `getc()`.

`getchar(c↑)` is the same as `getc()` from standard input (the keyboard).

`ungetc(c↓,fp↓,status↑)` pushes the character `c` back onto the input stream `fp`. `status` is defined to be the pushed character, or `EOF` if there is an error.

`gets(s↑,status↑)` reads a string from standard input and defines it to `s`. The string is terminated by a newline character, which is replaced in `s` by a null character. `status` is defined to be the number of characters read, or EOF upon end of file.

`fgets(s↑,n↓,fp↓,status↑)` reads $n - 1$ characters, or up through a newline character, whichever comes first, from the stream `fp` and defines it to `s` as a string. The newline is *not* removed as in `gets()`. `status` is defined to be the number of characters read, or EOF upon end of file.

`scanf(format↓,args↑,status↑)` is similar to the `scanf()` procedure in C. It takes its input from standard input and places the values that it reads in the definitional variables contained in the tuple `args`. Note: This procedure does not support the %t argument for term scanning.

`fscanf(fp↓,format↓,args↑,status↑)` is the same as `scanf()`, except that the input comes from the passed stream, `fp`.

`sscanf(buf↓,format↓,args↑,status↑)` is the same as `scanf()`, except that the input comes from the passed buffer, `buf`.

`stdout(fp↑)` assigns the mutable `fp` to be the file pointer for standard output (`stdout`).

`stdin(fp↑)` assigns the mutable `fp` to be the file pointer for standard input (`stdin`).

`stderr(fp↑)` assigns the mutable `fp` to be the file pointer for standard error (`stderr`).

`fseek(fp↓,offset↓,whence↓,status↑)` calls the C `fseek` function with the `fp`, `offset`, and `whence` arguments to set the position for the next input or output operation on this file. The `status` argument is defined to be 0 if the operation completes successfully, or -1 if it fails.

`ftell(fp↓,offset↑)` calls the C `ftell` function with the `fp` argument. The `offset` argument is defined to be the offset from the beginning of the file to the current position, or -1 if there is an error.

`rewind(fp↓)` calls the C `rewind` function with the `fp` argument to set the position to the beginning for the next input or output operation on this file. This is equivalent to `fseek(fp,0,0,_)`.

`fread(buf↑,size↓,nitems↓,fp↓,status↑)` reads `nitems` of data, each of `size` bytes in length, from the stream `fp`. `buf` is defined to a character array containing this data. `status` is defined to be the number of items actually read, or 0 upon EOF or error.

`fwrite(buf↓,size↓,nitems↓,fp↓,status↑)` writes `nitems` of data, each of `size` bytes in length, to the stream `fp`. `buf` is a character array containing the data to be written. `status` is defined to be the number of items actually written, or 0 upon EOF or error.

`access(path↓,mode↓,status↑)` checks the given file, `path` for accessibility according to `mode`. `mode` is the inclusive or of the bits R_OK, W_OK, and X_OK – read, write, and execute (search) permissions, respectively. A `mode` of F_OK (i.e., 0) tests whether the directories leading to the file can be searched and the file exists. `status` is defined it 0 if the file is accessible.

`remove(filename↓,status↑)` removes the specified `filename`. `status` is defined to 0 if the operation succeeded.

`rename(oldname↓,newname↓,status↑)` rename the file `oldname` to `newname`. `status` is defined to 0 if the operation succeeded.

18.2.2 Examples

Opening and Closing Files. The following examples illustrates the use of the `fopen`, `fclose`, `stderr`, and `fprintf` procedures. Note the include statement for `pcn_stdio.h`, which includes a definition for `FILE`.

```
#include <pcn_stdio.h>

open_test(fname)
FILE fp, err;
{;  stdio:fopen(fname, "r", fp, status),
    {?  status == 0 ->
        {;  stdio:printf("File \"%s\" opened\n",{fname},_),
            /* ... */
            stdio:fclose(fp,_)
        },
        default ->
        {;  stdio:stderr(err),
            stdio:fprintf(err,
               "Error opening \"%s\" for reading\n",{fname},_)
        }
    }
}
```

Writing to a File. This example opens a file `ptest` for writing, writes the characters `ABC` to this file, and then closes the file.

```
    #include <pcn_stdio.h>

    putc_test()
    FILE fp;
    {;  stdio:fopen("ptest","w",fp,_),
        stdio:putc('A',fp,_),
        stdio:putc('B',fp,_),
        stdio:putc('C',fp,_),
        stdio:fclose(fp,_)
    }
```

Writing to the Screen. This example writes the characters `ABC` followed by a newline character to the screen (standard input).

```
#include <pcn_stdio.h>

putchar_test()
{;  stdio:putchar('A',_),
    stdio:putchar('B',_),
    stdio:putchar('C',_),
    stdio:putchar('\n',_)
}
```

Printing to the Screen. The following program uses the `printf` command to print a variety of terms of the screen. Note the use of the `%t` format command to print arbitrary terms. When executed, the program acts as follows.

```
Str:  A string
Real:  -1.230000
List:  ["A string",-1.230000,{"a",1,2,3}]
Tup:  {"a",1,2,3}
```

The program can be modified to write the same text to a file by adding an `fopen` call, substituting `fprintf` for `printf` throughout, and finally closing the file.

Module `p_test.pcn`

```
#include <pcn_stdio.h>

printf_test()
{;  str = "A string",
    r = 0 - 1.23,     /* No unary minus in PCN */
    tup = {"a",1,2,3},
    ls = [str,r,tup],
    stdio:printf("Str: %s\nReal: %f\n",{str,r},_),
    stdio:printf("List: %t\nTup: %t\n",{ls,tup},_)
}
```

Creating Strings. We illustrate the use of the `sprintf` command to create a string. When executed, the `sprintf_test` procedure prints the string `file_5`.

```
#include <pcn_stdio.h>

sprintf_test()
{;  i = 5,
    stdio:sprintf(mystring,"file_%d",{i},_),
    stdio:printf("mystring = %s\n",{mystring},_)
}
```

Reading Characters. This example shows the use of the `stdin` and `getc` procedures to read a series of characters from the keyboard (standard input). The procedure `getc_test` prints a prompt, reads characters until an end of line is reached, and then prints the result.

`Enter line:` *my line*

`Line entered: my line`

The program can also be written by using the `getchar` procedure (which reads directly from standard input), avoiding the need for the call to `stdin`.

Module `r_test.pcn`

```
#include <pcn_stdio.h>

getc_test()
FILE fp;
{;  stdio:stdin(fp),
    stdio:printf("Enter line: ",{},_),
    getc_test1(fp,ls),
    sys:list_to_string(ls,str),
    stdio:printf("\nLine entered: %s\n",{str},_)
}

getc_test1(fp,ls)
FILE fp;
{;  stdio:getc(fp,ch),
    {?  ch == '\n' -> ls = [],
        default ->
        {;  ls = [ch|ls1],
            getc_test1(fp,ls1)
        }
    }
}
```

19 Cross-Compiling

`Pcncomp` supports cross-compilation. For example, if a Sun has the necessary C and Fortran cross-compilers for the Intel iPSC/860, the Sun version of `pcncomp` can be used to compile PCN programs for the iPSC/860.

To cross-compile PCN programs for some machine, add a `-target target_name` argument to `pcncomp` compile and link commands. For example, the following commands compile and link a program containing C, Fortran, and PCN source on the Intel iPSC/860:

```
% pcncomp -c my_c.c -target ipsc860
% pcncomp -c my_f.f -target ipsc860
% pcncomp -c my_pcn.pcn -target ipsc860
% pcncomp my_pcn.pam my_c.o my_f.o -mm my_pcn
                -o myprogram -target ipsc860
```

Alternatively, the native C and Fortran cross-compiler (i.e., `icc` and `if77` on the iPSC/860) can be used directly, instead of through `pcncomp`, to compile the Fortran and C portions of the program. The advantage to using `pcncomp` is that you need not know the cross-compiler's name, location, and special arguments. Those details are taken care of by `pcncomp`, based on the cross-compilation configuration when it is installed.

Specifying a cross-compilation target of "default" (i.e., `-target default`) is equivalent to not supplying a `-target` argument at all. This can be useful in writing portable Makefiles, as described in § 26.

20 Intel iPSC/860 Specifics

To compile a PCN program for the Intel iPSC/860, follow the cross-compilation instructions in § 19, using a target of "ipsc860".

The resulting iPSC/860 executable program can be run by logging into the iPSC/860 host (SRM), allocating an appropriately sized cube, and loading the program. Once PCN terminates, we free the cube. In the following example, we assume that the host is called `gamma`:

```
% rlogin gamma
% getcube -t 4
% load myprogram; waitcube
% killcube
% relcube
```

If you wish to supply arguments to your program, those arguments must be part of the `load` command:

```
% load myprogram myargs -pcn -gauge; waitcube
```

21 Intel Touchstone DELTA Specifics

To compile a PCN program for the Intel Touchstone DELTA, follow the cross-compilation instructions in § 19, using a target of "delta".

Before you can run the resulting DELTA executable program, you must copy it onto the DELTA's CFS filesystem using either `ftp` or `rcp`.

Then you can log into the DELTA and run the program via the `mexec` command. This command specifies the height and width of the submesh to allocate, and the executable to load on the nodes in the submesh. For example, the following command would load `myprogram` onto a 4 by 8 node mesh:

```
% mexec "-t(4,8)" -f myprogram
```

If you wish to supply arguments to your program, those arguments must be part of the `-f` flag:

```
% mexec "-t(4,8)" -f "myprogram myargs -pcn -gauge"
```

22 Sequent Symmetry Specifics

Running PCN on the Sequent Symmetry is similar to running PCN on a workstation. The `pcncomp` command, used for compiling and linking, is identical to that described throughout this manual.

The `-n` run-time system command line argument is used to run PCN with several nodes. For example, the following command runs `myprogram` on 10 nodes:

```
% myprogram -pcn -n 10
```

The Symmetry has two different C compilers that can be used to compile C foreign code. They are `cc` and `atscc`. `atscc` should be used if it is available, as it supposedly produces better code than the standard `cc` compiler. Fortran code should be compiled by using the `fortran` compiler. However, if you use `pcncomp` to compiler your C and Fortran code, then you need not worry about these details.

23 Network Specifics

The network version of PCN (net-PCN) uses Berkeley stream interprocess communication (TCP sockets) to communicate between nodes. A node can run on any machine that supports TCP. Hence, a single PCN computation can run on several workstations of a particular type, several workstations of differing types, several processors of a multiprocessor, or a mix of workstations and multiprocessor nodes. Current restrictions are listed in § 23.6.

Net-PCN currently operates on the NeXT, Sun, DECstation, HP9000, IBM RS/6000, and SGI Iris.

Using net-PCN is the same as using PCN on other platforms except that the user must specify on which machines PCN nodes are to run and may also be required to specify where on those machines PCN is to be found and the commands necessary for running net-PCN nodes on the given machines.

There are several different ways of starting net-PCN, each appropriate for different types of network. We shall consider each of these in turn, starting with the easiest. First, we provide some background information on the Unix remote shell command `rsh`, which is used to start net-PCN nodes.

23.1 Using `rsh`

The Unix remote shell command `rsh` is a mechanism by which a process on one machine (e.g., `my-host`) can start a process on another machine (e.g., `my-node`). A remote shell command can proceed only if `my-host` has been given permission to start processes on `my-node`. There are two ways in which this permission can be granted.

- The file `/etc/hosts.equiv` exists on `my-node` and contains an entry for `my-host`. This file must be created by the system administrator.

- The file `.rhosts` exists in the home directory of the user running the remote shell on `my-node` and contains a line of the form

  ```
  my-host username
  ```

  where `username` is the name of the user login on `my-host`. This file is created by the user.

Some sites disallow the use of `.rhosts` files. If `.rhosts` usage is disallowed and the host machine is not in `/etc/hosts.equiv`, remote shells cannot be used to create remote processes. Alternative mechanisms must be used, as described below.

The full syntax of the `rsh` command is as follows:

```
rsh hostname -l username command arguments
```

The `username` here is the login to be used on the remote machine. If `username` is not specified, it defaults to the login name of the user on the local machine. Furthermore, if the login name used on the local machine is different from the login name on the remote machine, the `.rhosts` file for the account on the remote machine must have an entry allowing access for that account on the host machine.

23.2 Specifying Nodes on the Command Line

The simplest way to start PCN on a network of machines is to use the -nodes <*nodelist*> command line argument, where *nodelist* is a colon-separated list of machine names on which PCN nodes are to run. For example,

```
myprogram -pcn -nodes pelican:raven:plover
```

will run `myprogram` on four nodes, with one node on the machine from which this command is run (the *host*) and one node on each of the machines named in the *nodelist*: `pelican`, `raven`, and `plover`.

This startup method works only if

1. `rsh` (§ 23.1) works from the host to each machine in *nodelist*, and

2. each of the nodes shares a common filesystem with the host. The reason for this is that the host runs each node in the directory in which `pcn` is invoked. If the host and a node have different filesystems, the `rsh` used to start up that node is likely to fail.

If any of these conditions does not hold, then net-PCN must be started by using one of the alternative methods described below.

Note that we can always create multiple nodes on a single processor by using the `-n` command line flag. The command

```
mypcn -pcn -n nnodes
```

forks `nnodes - 1` nodes on the local machine (resulting in a total of `nnodes` processes) which communicate by using sockets. This feature can be useful for debugging purposes, or on multiprocessing machines.

23.3 Using a PCN Startup File

The second net-PCN startup method that we consider can be used if nodes do not share a common file system with the host. However, it still requires that `rsh` work from the host to each node.

This method uses a startup file to define the locations of remote PCN node processes. Lines in this file identify the machines on which nodes are to be started.

Startup File Syntax. A line of the form

fork *n-nodes*

causes *n-nodes* node processes to be started on the local machine. These nodes communicate with the other nodes via sockets, even though they are on the same machine as the host.

A line of the form

exec *n-nodes*: *command -pcn $ARGS$*

causes *command* to be executed. *command* is the command that invokes PCN on the appropriate machine. The host process replaces *$ARGS$* at run time with the necessary arguments to PCN to cause it to start *n-nodes* node processes.

Blank lines in startup files and lines starting with whitespace, `%`, or `#` are ignored.

Examples of Startup Files. A startup file containing the lines

```
fork 1
exec 1: rsh fulmar myprogram -pcn $ARGS$
```

starts one node on the local machine (in addition to the host node) and one node on the host `fulmar`, using the PCN executable called `myprogram`.

A startup file containing the line

```
exec 1: rsh fulmar -l bob myprogram -pcn $ARGS$
```

starts one node using the program called `myprogram` on host `fulmar` using the PCN executable `pcn` and the account for username `bob`. If we assume the PCN host is being run by user `olson` on host `host-machine`, then the `.rhosts` file in the home directory of user `bob` on `fulmar` must contain the entry

```
host-machine olson
```

A startup file containing the line

```
exec 3: rsh fulmar "cd /home/olson/pcn; ./myprogram -pcn $ARGS$"
```

runs three nodes on fulmar of the PCN executable `myprogram` after changing to the directory `/home/olson/pcn`.

A startup file containing the line

```
exec 2: sh -c 'echo "myprogram -pcn $ARGS$ &" | rsh fulmar /bin/sh'
```

is a more complex example that starts up two nodes on fulmar. This example has the desirable side effect that the `rsh` process exits after starting the PCN node, whereas in the other examples the `rsh` will not complete until the node process completes.

Using a Startup File. We execute net-PCN with a startup file `pcn-startup` by using the `-s` run-time system command line argument:

```
myprogram -pcn -s pcn-startup
```

23.4 Starting net-PCN without `rsh`

If your computer system does not support the use of `rsh`, you will need to start remote nodes by hand or by using a utility called `host-control`. See the separate manual: R. Olson, *Using host-control*, Argonne National Laboratory Technical Memo ANL/MCS-TM-154.

23.5 Ending a Computation

Normally all nodes of a net-PCN computation will exit upon completion of the computation or upon abnormal termination of PCN. If for some reason this is not the case, you must log on to each machine that was executing a net-PCN node and manually kill the PCN process.

23.6 Limitations of net-PCN

Number of Nodes. The number of nodes available in a net-PCN computation is limited by the number of file descriptors available to a process (an operating system-imposed limit). On modern versions of Unix, there are generally more than sixty file descriptors available. Hence, in practice, the number of file descriptors is not likely to be a major problem.

Heterogeneous Networks. Currently, no support exists for executing net-PCN between machines with different byte orders and/or different floating-point representations. Net-PCN does execute correctly between different machines if they use the same byte-ordering and floating point representation (we have run net-PCN successfully between Sun 3, Sun 4, and NeXT computers). However, you must be careful when using foreign code in this case because, for example, structure packing in C may differ between different compilers.

24 Further Reading

PCN Language This text provides an introduction to the PCN language and a discussion of techniques used to reason about PCN program:

> M. Chandy and S. Taylor, *An Introduction to Parallel Programming*, Jones and Bartlett, 1991.

This paper describes the PCN language, including recent extensions for process mapping and templates, as well as surveying major applications:

> I. Foster, R. Olson, and S. Tuecke, Productive Parallel Programming: The PCN Approach, *Scientific Programming*, Vol. 1, 1992, pp. 51–66.

Programming and Proof Techniques The following book provides a readable and entertaining presentation of many of the basic parallel programming techniques used in PCN:

> I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming*, Prentice Hall, Englewood Cliffs, N.J., 1989.

The proof theory for PCN is based in part on that for Unity, which is described in detail in

> M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

Software cells, templates, and parallel software reuse are discussed in:

> I. Foster, *Information Hiding in Parallel Programs*, Preprint MCS-P290-0292, Argonne National Laboratory, 1992.

PCN Toolkit The Program Transformation Notation (PTN) tool is described in

> I. Foster, *Program Transformation Notation: A Tutorial*, Technical Report ANL-91/38, Argonne National Laboratory, 1991.

The `host-control` program used to manage network implementations of PCN is described in

> R. Olson, *Using host-control*, Technical Memo ANL/MCS-TM-154, Argonne National Laboratory, 1992.

PCN Implementation The techniques used to compile PCN for parallel computers and to implement templates are described in

> I. Foster and S. Taylor, *A Compiler Approach to Scalable Concurrent Program Design*, Preprint MCS-P306-0492, Argonne National Laboratory, 1992.

A detailed description of the PCN run-time system can be found in

> I. Foster, S. Tuecke, and S. Taylor, *A Portable Run-Time System for PCN*, Technical Memo ANL/MCS-TM-137, Argonne National Laboratory, 1991.

The design, implementation, and use of the Gauge performance analysis system are described in

> C. Kesselman, *Integrating Performance Analysis with Performance Improvement in Parallel Programs*, Ph.D. thesis, UCLA, 1991.

A description of the Upshot trace analyzer can be found in

> V. Herrarte and E. Lusk, *Studying Parallel Program Behavior with Upshot*, Technical Report ANL-91/15, Argonne National Laboratory, 1991.

Applications Papers describing PCN applications include

> I. Chern and I. Foster, Design and Parallel Implementation of Two Methods for Solving PDEs on the Sphere, *Proc. Conf. on Parallel Computational Fluid Dynamics*, Stuttgart, Germany, Elsevier Science Publishers B.V., 1992, pp. 83–96.

> D. Harrar, H. Keller, D. Lin, and S. Taylor, Parallel Computation of Taylor-Vortex Flows, *Proc. Conf. on Parallel Computational Fluid Dynamics*, Stuttgart, Germany, Elsevier Science Publishers B.V., 1991, pp. 193–206.

> I. Foster and J. Michalakes, MPMM: A Massively Parallel Mesoscale Model, *Proc. 5th ECMWF Workshop on Parallel Processing in Meteorology*, ECMWF, Reading, England, 1992.

Part III
Advanced Topics

25 pcncomp and the PCN linker

For a complete list of the arguments to `pcncomp`, run:

```
pcncomp -h
```

In general, `pcncomp` tries to follow the normal Unix conventions for C and Fortran compiler arguments.

PCN linker. The PCN linker, which is called by `pcncomp`, does not replace the standard Unix linker, `ld`. Instead, it operates at a higher level than `ld`. The PCN linker's primary function is to coalesce the PCN object code contained in the `.pam` files and turn it into machine object code that can be passed to `ld` to be linked with the run-time system, the user's foreign object code, and system libraries.

This is accomplished by creating a C source file that contains initialized C data structures with names known to the run-time system. This C file is then compiled and linked with everything else to produce an executable program. The C file is usually named with a "pcnt_" prefix, followed by the name of the executable program that we are creating.

The PCN linker is a new feature of PCN version 2.0. Its advantages compared to techniques used in earlier releases include a standalone executable, faster startup on large parallel computers, faster intermodule calls, faster compilation, and greater ease of use. A significant disadvantage is slower linking. This is a problem particularly during the debugging stage of program development. To alleviate this problem, a limited form of dynamic loading of `.pam` files is supported by PDB. This removes the time-consuming link step from the debug cycle, yet preserves all of the advantages of the PCN linker during production runs. See § 15.9 for details on dynamic loading of `.pam` files, and § 26 on how to exploit the creation of the `pcnt` file to reduce the link time when debugging foreign code.

26 Makefile

This section provides an example Makefile for use with PCN programs. We also provide some discussion of the Makefile, including some "tricks" to reduce link times.

Example Makefile

```
PAMS = pcncode.pam
OBJS = fcode.o ccode.o
PCNT = pcnt_myprogram
MAIN_MOD = pcncode
PROG_NAME = myprogram
FORTRAN = -fortran
FLAVOR =
PCN_BASE = /usr/local/pcn

TARGET = default
PCNCOMP = $(PCN_BASE)/bin/pcncomp
PCNCOMPFLAGS = $(FORTRAN) $(FLAVOR) -target $(TARGET)

all: $(PROG_NAME)

pams: $(PAMS)

objs: $(OBJS)

$(PCNT).c: $(PAMS)
	$(PCNCOMP) $(PCNCOMPFLAGS) $(PAMS) -o $(PCNT).c \
		-pcnt -mm $(MAIN_MOD)

$(PROG_NAME): $(PCNT).o $(OBJS)
	$(PCNCOMP) $(PCNCOMPFLAGS) $(PCNT).o $(OBJS) \
		-o $(PROG_NAME)

.SUFFIXES: .pcn .pam .c .o .f

.pcn.pam:
	$(PCNCOMP) $(PCNCOMPFLAGS) -c $*.pcn

.c.o:
	$(PCNCOMP) $(PCNCOMPFLAGS) -c $*.c

.f.o:
	$(PCNCOMP) $(PCNCOMPFLAGS) -c $*.f

clean:
	rm -f *.pam *.mod *~ $(PROG_NAME) *.dump pcnt* *.o
```

Portability. Since `pcncomp` is used to compile the C and Fortran source files, this Makefile is highly portable. The details of the actual C and Fortran compiler names, their locations, special arguments that they take, etc., are handled automatically by `pcncomp`.

Adaptability. The entire Makefile is parameterized by the first seven variables at the top of the Makefile. It can be quickly adapted to different programs. In addition, these variables can easily be overridden from the command line to create PDB and/or profiling versions of the program. For example, to create a version of the program that is linked with the PDB run-time system, you would run:

```
make FLAVOR=-pdb
```

Cross-compilation. Cross-compilation to different machines is simple with this Makefile. (See § 19 for more on cross-compilation.) By default, it will compile the program for whatever machine the make is running on. But by overriding the `TARGET` variable from the command line, we can easily cross-compile the program for different machines. For example, to cross-compile for the Intel iPSC/860, you would run:

```
make TARGET=ipsc860
```

Debugging flexibility. Section § 25 discusses how the PCN linker operates. It creates a `pcnt.c` file (a C source file) that contains all of the necessary information from the `.pam` files, compiles that file to a `pcnt.o` file, and then links that `pcnt.o` file with the run-time system, foreign object files, and system libraries to create an executable program.

When debugging foreign code, you can exploit the fact that the PCN linker creates this intermediate `pcnt` file to greatly reduce link times. If a foreign procedure is modified, there is no need to create a new `pcnt` file before linking. Only changes in the PCN code will affect the `pcnt` file. So, instead of creating a new `pcnt` file each time foreign code is modified, the one from the previous link will suffice.

When working on PCN code, link a version with PDB by running:

```
make "FLAVOR=-pdb -link_all"
```

This will give you a version of the program that you can use with dynamic loading to quickly debug your PCN code without having to relink after each change (see § 15.9).

27 Run-Time System Debugging Options

The PDB version of the run-time system incorporates a variety of low-level execution tracing facilities. These facilities are controlled through the following four debug-level variables. The value of each variable can range from 0 to 9, with 0 meaning no trace output and 9 maximum trace output.

Emulator Debug Level: This controls debugging information in the main process scheduling loop. For example, level 2 causes all intermodule calls to be printed, level 3 additionally prints the entry and exit of foreign procedures, and level 9 prints a complete trace of the PCN abstract machine instruction being executed.

Garbage Collector Debug Level: This controls debugging information in the garbage collector. For example, level 2 causes a short summary to be printed each time a garbage collection occurs.

Parallel Debug Level: This controls debugging information relating to the parallel aspects of the system. For example, level 5 causes debugging information about the low-level message handling between nodes to be printed.

Global Debug Level: This controls debugging information not covered by the other three variables. For example, level 1 causes startup parameters and boot arguments to be printed.

The four debug levels can be manipulated in two ways. On a single node, they can be modified through the use of the PDB variables (`$emulator_dl`, `$gc_dl`, `$parallel_dl`, and `$global_dl`) described in § 15.7.

The debug levels can also be set from the command line. The following run-time system command line arguments (i.e., they must appear after the `-pcn` argument) set the various debug levels on all nodes, including the host.

`-d` <`level`> : This sets all debug levels.

`-e` <`level`> : This sets the emulator debug level. It overrides the level set by the `-d` flag.

`-g` <`level`> : This sets the garbage collector debug level. It overrides the level set by the `-d` flag.

`-p` <`level`> : This sets the parallel debug level. It overrides the level set by the `-d` flag.

The following argument enables low-level trace information after a specified number of procedure calls.

`-r` <`reduction_number`> : Do not print any debugging output until the number of procedure calls given by `reduction_number` has been executed.

The following command line arguments can be used to set debug levels selectively in different nodes of a multiprocessor.

`-node` <`node_number`> : Apply the following node debug level flags only to a particular node, `node_number`. If this argument is not used or `node_number` is -1, then apply the following node debug level flags to all nodes.

`-nd` <`level`> : This sets all debug levels on the appropriate node(s).

`-ne` <`level`> : This sets the emulator debug level on the appropriate node(s). It overrides the level set by the `-nd` flag.

`-ng` <`level`> : This sets the garbage collector debug level on the appropriate node(s). It overrides the level set by the `-nd` flag.

`-np` <`level`> : This sets the parallel debug level on the appropriate node(s). It overrides the level set by the `-nd` flag.

`-nr` <`reduction_number`> : Do not print any debugging output on the appropriate node(s) until the `reduction_number` reduction has been reached.

For example, the following command would set the emulator debug level to 3 and the garbage collector debug level to 2 on node 5 of a 10-node run.

myprogram -pcn -n 10 -ne 3 -ng 2 -node 5

All debugging messages are preceded by the node number from which the message originated and reduction number on that node when the message was printed. When debug levels are set on multiple nodes simultaneously the debugging output from these nodes will be interleaved. The node and reduction number can help you sort out these interleaved messages.

Interleaving problems can be avoided by telling the run-time system to log all debugging messages to files, instead of to the screen, by putting a `-log` on the command line. The system will then create a `Logs` directory into which all debugging output will be printed. Further, the debugging output from each node will be put in a separate file in this `Logs` directory.

Part IV
Appendices

A Obtaining the PCN Software

The PCN software is available by anonymous FTP from Argonne National Laboratory, in the `pub/pcn` directory on `info.mcs.anl.gov`. The latest version of this document is also available at the same location. The following session illustrates how to obtain the software in this way.

```
% ftp info.mcs.anl.gov
Connected to anagram.mcs.anl.gov.
220 anagram.mcs.anl.gov FTP server (Version 5.60+UA) ready.
Name (info.mcs.anl.gov:XXX): anonymous
331 Guest login ok, send ident as password.
Password:  /* Type your user name here */
230- Guest login ok, access restrictions apply.
Argonne National Laboratory Mathematics & Computer Science Division
All transactions with this server, info.mcs.anl.gov, are logged.
230 Local time is Fri Nov 8 18:26:39 1992
ftp> cd pub/pcn
250 CWD command successful.
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for file list.
pcn_v2.0.tar.Z
README
pcn_prog.ps.Z
pcn_prog.tar.Z
226 Transfer complete.
78 bytes received in 1.3e-05 seconds (5.9e+03 Kbytes/s)
ftp> binary
200 Type set to I.
ftp> get pcn_v2.0.tar.Z
200 PORT command successful.
150 Opening BINARY mode data connection for pcn_v2.0.tar.Z (XXX bytes).
226 Transfer complete.
local:  pcn_v2.0.tar.Z remote:  pcn_v2.0.tar.Z
XXX bytes received in YY seconds (ZZ Kbytes/s)
ftp> quit
221 Goodbye.
```

B Supported Machines

The following table lists the machines on which PCN is currently supported, along with the architecture name.

Architecture	Machine name
delta	Intel Touchstone Delta
ipsc860	Intel iPSC/860
iris	Silicon Graphics Iris
next040	NeXT
rs6000	IBM RS/6000
sun4	Sun 4 (SPARC based)

C Reserved Words

The following words may not be used as variable names or procedure names in PCN programs.

append_stream
char
close_stream
data
decrement_stream
default
directive
double
exports
foreign
init_recv
init_send
int
length
location
nodes
over
PCN
stream
stream_send
stream_recv
topology
tuple
_p_*
_pdb_*
PCN_*

D Deprecated and Incompatible Features

1. Under v1.2.2, a common way to write a tuple that has a string in the first element of the tuple as a label was to write it in prefix notation, such as `label(a,b)`. This is equivalent to writing the tuple in the infix notation, `{‘‘label’’,a,b}`. However, in v2.0, writing tuples in this prefix form is discouraged because its syntax is identical to that of functions, which are now supported in v2.0. Instead, it is recommended that you always write tuples using infix notation.

2. The `sys:list_length()` procedure (which was undocumented under v1.2.2 but sometimes used) has had its argument order changed between v1.2.2 and v2.0, in order to make it follow the convention used in all other libraries that the return argument is always the last argument.

3. The `stdio:scanf()` procedure no longer supports `%t` for term scanning.

4. The `-foreign()` directive is ignored under v2.0. All information about which foreign object files and libraries to link must be specified on the command line when linking with `pcncomp`.

5. The `PCN_PATH` environment variable is ignored under v2.0. Since `.pam` files are no longer loaded dynamically when they are first referenced, this is no longer needed.

6. The `sizeof()` command has been changed to `length()`. It returns the number of elements in an array, or the arity of a tuple. (This change occurred in v1.2.2.)

7. Meta operations now use ‘*var*‘ (matching back quotations) instead of ’*var* (unmatched single quote) to denote a string that is to be interpreted as a variable name. (This change occurred in v1.2.2.)

E Common Questions

What does it mean when PCN prints an `Illegal tag` message? This usually means that PCN internal data structure has been corrupted somehow. The usual way in which this happens is that user code writes past the beginning or end of an array (either in PCN or foreign code).

To help detect this situation: If you use arrays in your PCN code then you can do bounds checking by running the program under `pcn.pdb`. If you use arrays in Fortran code, many Fortran compilers have a flag to turn on bounds checking (also known as range checking).

See § 14 for information on debugging PCN programs.

Why is the PCN linker so slow? See § 25 for information on how the linker works. Also, see § 15.9 and § 26 for tips on reducing link times when debugging.

F PCN Syntax

The following syntactic conventions are employed in this expanded BNF:

nonterminal ::= production

[ ]	Surround an optional element.
{ }	Surround an element that may occur zero or more times.
|	Separates alternatives.
boldface	Indicates reserved words.
"quotes"	Indicate characters that appear literally.

The symbols *unsigned-integer*, *unsigned-real*, *character-string*, and *identifier* denote terminal symbols and are not defined further here.

Comments are delineated by the start-comment symbol `/*` and the end-comment symbol `*/`.

Compilation Module

compilation-module	::=	program-or-directive { program-or-directive }
program-or-directive	::=	program-declaration | directive

Directive

directive	::=	"-" directive-name "(" directive-arguments ")"
directive-name	::=	identifier
directive-arguments	::=	[directive-argument { "," directive-argument }]
directive-argument	::=	term

Program Declaration

program-declaration	::=	program-heading mutable-declarations program-body
program-heading	::=	program-heading-modifiers identifier "(" formal-parameters ")"
program-heading-modifiers	::=	[identifier { "," identifier }]
formal-parameters	::=	[formal-parameter { "," formal-parameter }]
formal-parameter	::=	identifier
mutable-declarations	::=	{ mutable-type mutable-declaration-list ";" }
mutable-type	::=	**int** | **double** | **char** | **port**
mutable-declaration-list	::=	mutable-declaration { "," mutable-declaration }

mutable-declaration ::= identifier [“[” [expression] “]”]

program-body ::= sequential-composition |
parallel-composition |
choice-composition

Block

block ::= assignment-statement |
definition-statement |
program-call |
sequential-composition |
parallel-composition |
choice-composition |
quantification

assignment-statement ::= variable “:=” expression

definition-statement ::= variable “=” term

program-call ::= local-program-call |
remote-program-call |
remap-program-call |
meta-program-call

function-call ::= local-program-call |
remote-program-call

local-program-call ::= simple-program-call

remote-program-call ::= simple-program-call ”@” simple-program-call

remap-program-call ::= simple-program-call **in** simple-program-call

meta-program-call ::= ‘ identifier ‘

simple-program-call ::= program-specifier “(” actual-parameters “)”

program-specifier	::=	[module-name ":"] program-name
module-name	::=	quoted-identifier
program-name	::=	quoted-identifier
actual-parameters	::=	[actual-parameter { "," actual-parameter }]
actual-parameter	::=	term

annotation ::= unsigned-integer | character-string | quoted-identifier

quoted-identifier ::= ‘ identifier ‘ | identifier

Note: The single backquote characters in the preceding line indicate literally that character.

Quantification

quantification::= identifier **over** expression ".." expression "::" block

Sequential Composition

sequential-composition ::= "{" ";" block { "," block } "}"

Parallel Composition

parallel-composition ::= "{" "||" block { "," block } "}"

Choice Composition

choice-composition ::= guarded-block |
"{" "**?**" guarded-block { "," guarded-block } "}"

guarded-block	::=	guards → block
guards	::=	guard-list | **default**
guard-list	::=	guard { conditional-and guard }
conditional-and	::=	","
guard	::=	pattern-match | equality-test | relational-test | data-test

pattern-match ::= identifier “?=” pattern
pattern ::= tuple-pattern | list-pattern
tuple-pattern ::= “{” pattern-elements “}” |
identifier “(” pattern-elements “)”
list-pattern ::= “[” pattern-elements “]” |
“[” pattern-element-list “|” pattern-element “]”
pattern-elements ::= [pattern-element-list]
pattern-element-list ::= pattern-element { “,” pattern-element }
pattern-element ::= signed-number | character-string | identifier | pattern

equality-test ::= equality-operand “==” equality-operand |
equality-operand “!=” equality-operand
equality-operand ::= expression | character-string | empty-tuple | empty-list
empty-tuple ::= “{” “}”
empty-list ::= “[” “]”

relational-test ::= relational-operand “<” relational-operand |
relational-operand “>” relational-operand |
relational-operand “<=” relational-operand |
relational-operand “>=” relational-operand
relational-operand ::= expression

data-test ::= **int** “(” term “)” |
double “(” term “)” |
char “(” term “)” |
tuple “(” term “)” |
data “(” term “)”

Variable

variable ::= identifier [“[” index “]”]
index ::= unsigned-integer | identifier

Expression

expression	::=	adding-expression
adding-expression	::=	multiplying-expression |
		adding-expression “+” multiplying-expression |
		adding-expression “−” multiplying-expression
multiplying-expression	::=	primary-expression |
		multiplying-expression “*” primary-expression |
		multiplying-expression “/” primary-expression |
		multiplying-expression “%” primary-expression
primary-expression	::=	signed-number |
		variable |
		length “(” identifier “)” |
		function-call |
		“(” expression “)”
signed-number	::=	[”−”] unsigned-integer |
		[”−”] unsigned-real

Term

term	::=	expression |
		character-string |
		tuple-constructor |
		list-constructor
tuple-constructor	::=	“{” elements “}” |
		identifier “(” elements “)”
list-constructor	::=	“[” elements “]” |
		“[” element-list “|” element “]”
elements	::=	[element-list]
element-list	::=	element { “,” element }
element	::=	signed-number | character-string | variable |
		tuple-constructor | list-constructor

Index