# Productive Parallel Programming: The PCN Approach*

Ian Foster, Robert Olson, and Steven Tuecke

Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439

**Abstract**

We describe the PCN programming system, focusing on those features designed to improve the productivity of scientists and engineers using parallel supercomputers. These features include a simple notation for the concise specification of concurrent algorithms, the ability to incorporate existing Fortran and C code into parallel applications, facilities for reusing parallel program components, a portable toolkit that allows applications to be developed on a workstation or small parallel computer and run unchanged on supercomputers, and integrated debugging and performance analysis tools. We survey representative scientific applications and identify problem classes for which PCN has proved particularly useful.

**Keywords:** PCN, program composition, parallel programming, reuse, templates.

## 1 Introduction

After many years as academic curiosities, computers combining hundreds or thousands of powerful microprocessors have overtaken vector processors and become essential tools for scientists and engineers. Unfortunately, the programming of these parallel supercomputers is still immensely time consuming. Frequently, many months of effort are required to develop, validate, and tune parallel codes; apparently minor algorithmic changes can take weeks. These factors severely limit the productivity and creativity of those using these advanced machines.

A clear need exists for tools that reduce the cost of program development to more manageable levels. Good software engineering practice tells us that these tools should possess three characteristics: (1) a *notation* that permits us to *program smarter*, by lessening the gap between our conception of a problem solution and its eventual implementation; (2) support for *code reuse* that allows us to *program less*, by reusing old code when solving new problems; and (3) a *toolkit* that permits us to *program faster*, by reducing the effort required to find errors, adapt programs to different architectures, etc.

In this article, we introduce PCN, a parallel programming system with these characteristics. PCN has been developed over the past three years at Argonne National Laboratory and the California Institute of Technology. It features a simple concurrent language (Program Composition Notation), facilities for reuse of sequential and parallel code, and a toolkit supporting compilation, debugging, and performance analysis. Important benefits

*To appear in: *Scientific Programming.*

of the approach include the ability to rapidly prototype complex concurrent algorithms, particularly those involving dynamic communication or computation structures; application portability, which permits programs developed on workstation to move to networks of workstations and to parallel supercomputers with little change; the ability to incorporate existing Fortran and C code into parallel programs; and support for the reuse of parallel program structures in different applications.

PCN is not the solution to all programming problems. A disadvantage for some programmers is the need to learn a new programming language. Others are uncomfortable with a high-level approach, preferring to program parallel computers at the lowest level possible. In addition, the PCN system is research software and, as such, not yet as sophisticated as conventional sequential programming systems. Nevertheless, it has already been used successfully to develop applications and to teach parallel programming to undergraduates. We expect it to prove useful to many users and for many purposes.

Rather than an academic exposition of PCN, this article provides an informal introduction to its capabilities and an analysis of the experiences of those using it to address substantial programming problems. By conveying the flavor of the approach and indicating the classes of problems for which it appears particularly appropriate, we hope to stimulate our readers to experiment with PCN in their own applications. The latest version of both the software and detailed documentation can be obtained by anonymous FTP from the directory `pub/pcn` at `info.mcs.anl.gov`.

The rest of this article is divided into five parts. These provide an overview of the approach, a description of the programming language, a discussion of the techniques used to reuse existing code, a description of the programming tools, and a survey of representative applications.

## 2 Approach

The focus of the PCN approach to parallel programming is the development of programs by the *parallel composition* of simpler components, in such a way that the resulting programs preserve properties of the components that they compose. In particular, deterministic compositions of deterministic components should themselves be deterministic: the result of such computations should never depend on the order in which components are scheduled for execution. Similarly, the result computed by a program should be independent of how its components are mapped to processors. This compositional property is critical to both the development of robust applications and the reuse of existing code.

The PCN language is carefully designed to realize compositionality. In particular, it requires that concurrently executing components interact by reading and writing special single-assignment or *definitional* variables. A definitional variable is initially undefined and can be assigned at most a single value. If a component attempts to read an undefined variable, execution of that component is suspended until the variable is defined. Hence, the result of a computation can never depend on the time at which read and write operations occur.

This focus on parallel composition and definitional variables leads to the following approach to parallel program design. A problem is decomposed into a large number of subproblems and a process is created for each subproblem. PCN code is written to organize the exchange of data between these processes and to coordinate their execution.

Existing *software cells* and *templates* may be integrated into the program; these define sets of processes that implement commonly-used operations such as parallel reductions or transforms. Finally, the mapping of the processes to the processors of a parallel computer is specified; this can alter performance but not the result computed.

The PCN compiler is optimized for efficient execution of programs that create many processes and that communicate and synchronize via definitional variables. It ensures that process creation, scheduling, termination, and migration are extremely inexpensive operations: typically a few tens of instructions. (Process migration incurs an additional cost proportional to the size of a process's data.) Read and write operations on definitional variables are implemented in terms of pointer operations within a single address space and message passing between address spaces. Processes are scheduled for execution so as to overlap computation and communication. Data structures are created dynamically and deallocated either when the process in which they are defined terminates (in the case of local variables) or when they are no longer accessible (in the case of definitional variables shared by several processes).

Components composed by PCN programs can be written in PCN or in sequential languages such as Fortran and C. In the latter case, existing code and compiler technology can be reused. Programs that do not use Fortran common or C global data can be composed in exactly the same way as PCN programs. If programs do use common/global data, then certain restrictions apply, as the use of common/global data violates the requirement that programs only communicate via definitional variables. This issue is discussed in Section 4.1.

# 3 Notation

Programming is rarely easy, but an appropriate notation can make it less difficult. As Whitehead observed of mathematics: "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems" [11]. In parallel programming, a good notation should express concurrency, communication, synchronization, and mapping straightforwardly and clearly. It should also discourage nondeterminism, just as a mathematical notation avoids ambiguity.

The programming notation used in the PCN system is Program Composition Notation (PCN). PCN extends sequential programming with two simple ideas — concurrent composition and single-assignment variables — and defines how these ideas interact with conventional sequential constructs [1, 6]. The PCN system also incorporates two additional constructs — virtual topologies and port arrays — that allow the definition and reuse of parallel program structures called cells and templates [4].

Our description of the PCN language is divided into five parts. These describe in turn the constructs used to specify concurrency, communication and synchronization, nondeterminism, mapping, and composition of process ensembles.

## 3.1 Concurrency

Syntax is similar to that of the C programming language. A program is a set of procedures, each with the following general form ($k, l \geq 0$).

```
name(arg₁,...,arg_k)
declaration₁; ...; declaration_l;
block
```

A `block` is a call to a PCN procedure (or to a procedure in a sequential language such as Fortran or C), a composition, or a primitive operation such as assignment. A composition is written {`op block`$_1$`, ..., block`$_m$}, $m > 0$, where `op` is one of "||" (parallel), ";" (sequential), or "?" (choice), indicating that the blocks `block`$_1$, ..., `block`$_m$ are to be executed concurrently, in sequence, or as a set of guarded commands (a sort of parallel case statement, with each block being a condition/action pair), respectively.

A parallel composition specifies opportunities for parallel execution but does not indicate how the composed blocks (which can be thought of as lightweight processes) are to be mapped to processors. The techniques used to specify mapping are described below.

### 3.2 Communication and Synchronization

Statements in a parallel composition communicate and synchronize by reading and writing special single-assignment or *definitional* variables. (Conventional, or *mutable*, variables are also supported, but can be used only within sequential blocks.) Definitional variables are distinguished by a lack of declaration, are initially undefined, can be written (defined) once using the primitive operator "`=`", and once written cannot be modified. (An attempt to overwrite a definitional variable is flagged as a runtime error.) A process that requires the value of an undefined variable suspends until the required data is available. This provides a dataflow model of computation, with execution order within parallel compositions determined by availability of data.

Processes that share a definitional variable can communicate regardless of their location in a parallel computer. For example, in the parallel composition {|| `producer(x)`, `consumer(x)`}, the two procedure calls `producer(x)` and `consumer(x)` can use `x` to communicate, whether they are executing concurrently on one processor or in parallel on two processors.

Consider the following definitions for `producer` and `consumer`. The producer defines its parameter to be the string `"hello"`, hence communicating this value to any process that shares that variable (in the composition in the previous paragraph, this is `consumer`). The consumer is defined in terms of a choice composition. The two guarded commands define tests on the parameter `v` (`v == "hello"` and `v != "hello"`) and the actions that are to be performed if these tests succeed (calls to the procedures `greet()` or `ignore(v)`, respectively). Hence, the procedure `consumer` suspends until `v` has a value and then executes one of the two procedures.

```
producer(u)              consumer(v)
{|| u = "hello"}         { ? v == "hello" -> greet(),
                             v != "hello" -> ignore(v)
                         }
```

**Stream Communication.** A shared definitional variable would not be very useful if it could only be used to exchange a single value. Fortunately, simple techniques allow a single variable to be used to communicate a *stream* of values [5]. A stream acts like a queue: the producer places elements on one end, and the consumer(s) take them off the other.

Stream communication is achieved by the incremental construction of linked list structures. The technique makes use of a data type called the *tuple*. A tuple is represented by zero or more terms enclosed in parentheses, for example {} (the empty tuple) or {`head, tail`} (a two-tuple). The *match* operator "`?=`" is used to access a tuple's components. For example, `x ?=` {`msg, xt`} checks whether `x` is a two-tuple and, if so, defines `msg` and `xt` to be references to its two components.

Imagine a producer and a consumer sharing a variable `x`. The producer defines `x` to be a two-tuple containing a message and a new definitional variable (`x =` {`msg, xt`}). The consumer matches `x ?=` {`msg, xt`} to access both the message and the new variable. These operations 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. The stream is closed by defining the shared variable to be the empty tuple.

The following program implements this protocol. The `stream_producer` generates `n` messages, calling `produce` to generate each message, and then closes the stream. The `stream_consumer` consumes messages until the stream is closed, calling `greet` or `ignore` to process each incoming message. Note that both procedures are defined recursively. For example, the producer generates one message (by defining `u` to be the tuple {`msg, u1`}) and then calls itself recursively to produce further messages. Recursion is often used in PCN because it allows the introduction of an unbounded number of new definitional variables; the PCN compiler is designed to compile such programs efficiently, and in fact translates recursive procedures into iterative code. Explicit iterative constructs are also available; these are described in a subsequent section.

```
stream_producer(n, u)
{ ? n > 0 ->
        {|| produce(n, msg),
            u = {msg, u1},
            stream_producer(n-1, u1)
        },
    n == 0 -> u = {}
}

stream_consumer(v)
{ ? v ?= {msg,v1} ->
    {|| { ? msg == "hello" -> greet(),
            msg != "hello" -> ignore(msg)
        },
        stream_consumer(v1)
    }
}
```

### 3.3 Nondeterminism

The use of definitional variables as a communication mechanism avoids errors due to time-dependent interactions. Race conditions, in which the result of a computation depends on the time at which a process reads a variable, cannot occur: a consumer of a variable always suspends until the variable has a value, and then computes with a value that cannot change.

Nevertheless, it is sometimes useful to be able to specify nondeterministic execution, particularly in reactive applications. PCN also allows the specification of nondeterministic actions, but in a tightly controlled manner. Only if the conditions associated with two or more actions in a guarded command are not mutually exclusive is execution nondeterministic. For example, the following procedure merges two input streams (`in_stream1` and `in_stream2`) into a single output stream (`out_stream`). Note that the two streams are not mutually exclusive: as guards are executed concurrently, messages can be received from either input stream, in a time-dependent manner.

```
merge(in_stream1, in_stream2, out_stream)
{ ?
    in_stream1 ?= {msg, more_in1} ->
        {||
            out_stream = {msg, more_out},
            merge(more_in1, in_stream2, more_out)
        },
    in_stream2 ?= {msg, more_in2} ->
        {||
            out_stream = {msg, more_out},
            merge(in_stream1, more_in2, more_out)
        }
}
```

PCN programs in which conditions are mutually exclusive are guaranteed to be deterministic. This is an important property that greatly simplifies parallel programming. (The reader might be concerned about the possibility of writing conditions which are mistakenly not mutually exclusive. In practice, this has not proved to be a problem.)

Two potential sources of nondeterminism which are not prevented by PCN are concurrent I/O operations and concurrent access to Fortran common or C global data by Fortran or C procedures composed by PCN. The latter issue is discussed in Section 4.1.

### 3.4 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. The following language features are used when writing code to define 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, e.g. {`"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 procedures 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.

```
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 an iterative construct called a *quantification* to create a computation that executes on many processors. 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` = `;`).

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.)

```
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 $32 \times 16$ 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. (The location function is quoted to indicate that it should not be evaluated.) 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.

```
function mesh_in_array(rows, cols)
{ ? topology ?= {"array", n} ->
    {|| type = {"mesh", rows, cols},
        size = rows*cols,
        map_fn = `node()`,
        return( {type, size, map_fn} )
    },
```

```
        default -> error()
    }
```

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 to be executed on an array computer, we may invoke it as follows.

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

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 get 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.

### 3.5 Port Arrays

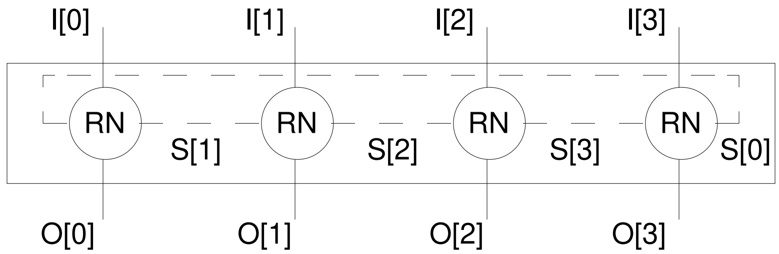
Recall that individual processes communicate by reading and writing shared definitional variables, as in the composition {|| `producer(x), consumer(x)`}. The *port array* provides a similar mechanism for use when composing sets of processes.

A port array is a array of definitional variables that has been distributed evenly across the nodes of a virtual topology. A declaration "`port P[N];`" creates a port array `P` with `N` elements, distributed blockwise across the nodes of the virtual topology in which the port array is declared. `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.

The following procedure, a variant of the `array` procedure given earlier, 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, so as to allow 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.

```
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.




# 4 Reuse

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 is important when migrating existing sequential applications to parallel computers; the latter is becoming increasingly important as our parallel code base grows.

## 4.1 Sequential Code: Multilingual Programming

A simple 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. Sequential procedures can be passed definitional and mutable data, but suspend until definitional data is available and hence never deal with incomplete information. Sequential procedures can modify only mutable variables.

A deficiency of the Fortran interface is that no special allowance is made for "common" data. Each physical processor has a single copy of all common data declared in an application's Fortran code, 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 data. It is the programmer's responsibility to avoid errors due to concurrent access. Experience shows that programmers deal with this problem in one of two ways. (1) If an application is of moderate size, or is being developed from scratch, they often choose to eliminate common data altogether. This may be achieved by allocating arrays in PCN and passing them to the different Fortran programs. Although this approach requires substantial changes to the application, the bulk of the existing Fortran can be retained, and the full flexibility of PCN is available to the programmer. (2) If substantial rewriting of an application is not possible, programmers maintain common 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.

The interface to sequential programming languages means that 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.

### 4.2 Parallel Code: Cells and Templates

**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 [4]. 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, 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.

```
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 matchs `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.

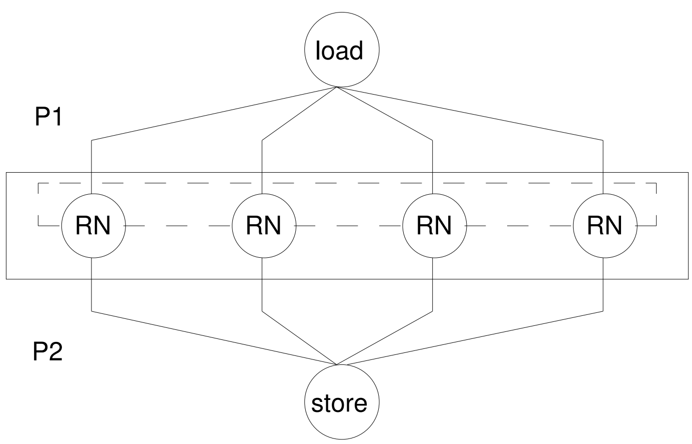
```
load(file, P)
port P[];
{ ; i over 0..nodes()-1 : read(file, stuff), P[i] = stuff }

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

We compose the three cells to obtain a program `main` 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, them the three stages could execute concurrently, as a pipeline.

```
main(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 ("`main(if,of) in array()`") and will create a ring with one process per node of that topology.

## 5 Tools

The high-level nature of the PCN language requires a sophisticated compiler (to achieve efficient execution on sequential and parallel computers) and a specialized debugger (to keep track of multiple concurrent processes). These tools are integrated with other components to form a toolkit that supports debugging, performance tuning, and integration of Fortran and C code, and that allows programs to be executed on a wide variety of parallel computers and workstation networks [6]. In this section, we describe four components of this toolkit: compiler, network implementation, parallel debugger, and performance analysis tools.

### 5.1 Portable Compiler and Runtime System

We summarize the techniques used to translate PCN programs into executable code, so as to provide some insights into the efficiency of the PCN implementation.

The PCN compiler implements both the PCN language and the constructs introduced to support reuse of parallel code. It translates PCN programs to a machine-independent, low-level form that is linked with both object code for sequential language procedures and a small runtime system, to produce an executable program. The compiler is responsible for generating code to perform specialized operations such as creating processes, suspending processes, terminating processes, and generating messages; the runtime system routes incoming messages, schedules executable processes, and manages the heap on which are allocated process records, program data, etc.

The compiler and runtime system have been carefully designed to optimize the creation, scheduling, migration, and termination of lightweight processes. A process with $n$ arguments is represented by a process record that occupies $n + 2$ words of memory, with $n$ of these words containing pointers to arguments; hence, processes can be created, scheduled, or descheduled in a few tens of instructions. A process is migrated to another processor by communicating the process record and the data structures accessible from this process record. Thus, the cost of migration is primarily the cost of transferring its data, and processes with little data can be migrated extremely cheaply. The low cost of scheduling means that the runtime system is able to schedule idle tasks when waiting for the results of remote communication operations. That is, it automatically overlaps computation and communication operations.

The compiler does not currently optimize the performance of pure PCN code, which may execute 5–10 times slower than equivalent Fortran or C code. As PCN applications typically spend much of their time executing Fortran or C, this has not been a serious difficulty. (The profiling tools described below can be used to identify bottlenecks; if necessary, PCN procedures can be rewritten in Fortran or C to improve performance.) Future compilers will improve PCN performance, allowing a larger proportion of applications to be written in PCN.

A novel aspect of the compiler is a programmable source transformation system, incorporated as an optional stage in the compiler pipeline, after the parser and before the encoder. Programmers can use this facility to implement application-specific extensions to the PCN language. For example, the transformation system has been used to implement specialized composition operators that generate self-scheduling computations [3].

### 5.2 Network Implementation

The network implementation of PCN (net-PCN) allows users to treat a set of workstations as a parallel computer. Programs developed for multiprocessors and multicomputers can be run without modification on networks, although because of higher communication costs, algorithms must normally be more coarse-grained to execute efficiently.

Net-PCN can run on any machine that supports the TCP communication protocol. Hence, a single computation can in principle 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. Currently, we require that all processors involved in a computation employ common representations for the basic PCN data types (characters, integers, and double-precision floats). In the future, type conversions will be performed automatically, allowing PCN programs to run transparently on arbitrary networks.

A useful component of Net-PCN is a utility program called `host-control`, which provides facilities for managing a network computation. This utility allows the user to inquire about the status of nodes available to Net-PCN, add and delete nodes, and execute programs [10].

### 5.3 PDB: A Parallel Debugger

Debugging tools that assist in the location of logical errors are, of course, a critical component of any programming system. PCN's unconventional language constructs, in particular its lightweight processes and dataflow synchronization, require specialized debugging support. This is provided by the PCN symbolic debugger, PDB.

The major difference between PCN and conventional sequential programming languages is that in PCN programs, many threads of control (processes) can be active at one time. Hence, PDB not only provides conventional debugger features, such as the ability to interrupt execution and examine program arguments, but also permits the user to examine enabled and suspended processes, identify definitional variables for which values have yet to be produced, and control the order in which processes are scheduled for execution.

A common error in PCN programming is for one program component not to produce a value required by another component. This results in a *deadlock* situation, in which all processes are suspended waiting for data. This situation can be detected by PDB. The programmer can examine the set of suspended processes and identify variables for which no values have been produced.

### 5.4 Understanding Performance

In parallel computing, where performance is critical and often non-intuitive, it is important to provide tools to assist in the identification of *performance errors*. Two such tools, Gauge and Upshot, have been integrated into PCN.

**Gauge.** Gauge is an execution profiler: it collects information about the amount of time that each processor spends in different parts of a program [9]. It also collects procedure call counts, message counts, and idle time information. Three properties of Gauge make it particularly useful: profiling information is collecting automatically, without any programmer intervention; the overhead incurred to collect this information is small, typically much less than 1 per cent; and the volume of data does not increase with execution time. A powerful data exploration tool permits graphical exploration of profile data. The use of Gauge is illustrated in a subsequent section.

**Upshot.** Upshot is a trace analysis tool that can provide insights into the fine-grained operation of parallel programs [8]. Upshot requires that the programmer instrument a program with calls to event logging primitives. These events are automatically recorded and written to a file when a program runs. A graphical trace analysis tool allows the programmer to examine temporal dependencies between events. Like any trace-based tool, Upshot suffers from scaling problems. However, it can be useful when used in a controlled manner, to examine local phenomena identified as problematic by Gauge.

# 6 Applications

PCN has been used in substantial programming projects that have produced programs used to further scientific research on the world's fastest computers. For example, the first two applications operational on the 528-processor, 30 Gflops Intel Touchstone Delta system — a geophysical modeling code and a fluid dynamics code — were both PCN programs [2, 7]. Here, we describe one of those programs, survey other representative applications, and identify factors that appear to favor the use of PCN for programming projects.

## 6.1 Icosahedral Climate Modeling Code

This application implements a numerical method proposed for use in climate models, a second-order, conservative control volume method on an icosahedral-hexagonal grid. The code was developed to permit detailed studies of both the method's accuracy and the long-term behavior of fundamental modes of the atmospheric circulation. The code integrates existing Fortran and C code into a parallel framework implemented in PCN [2].

An icosahedral-hexagonal grid can be structured as 10 $\mathtt{n}\times\mathtt{n}$ meshes plus two separate polar points. The parallel algorithm decomposes each mesh into $\mathtt{c}^2$ submeshes, giving 10 $\mathtt{c}^2+2$ subdomains, two with one point and the rest with $(\mathtt{n}/\mathtt{c})^2$ points. Communication must be performed to obtain values from neighboring subdomains during integration. The design of an efficient mapping is complicated by the irregular domain. On some parallel computers, it may be desirable to place two or more subdomains on the same processor.

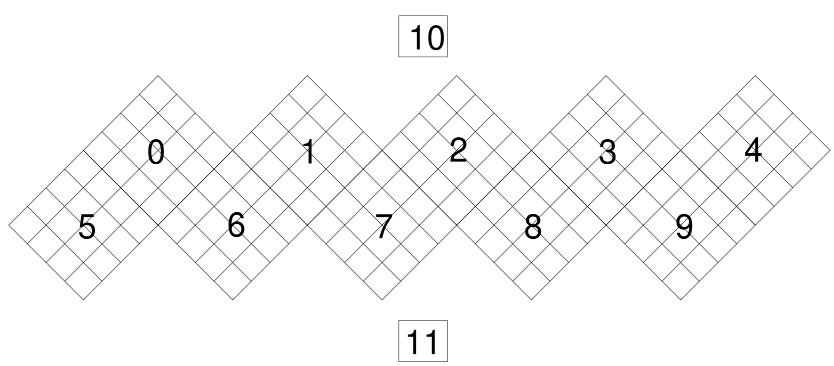



Figure 1: Icosahedral Mesh Domain Decomposition

**Implementation.** The development of the parallel code is simplified if mapping is specified with respect to a virtual topology with the same shape as the problem domain [4]. We define an `ico_mesh` topology containing ten `c`×`c` meshes and two polar processors (Figure 1) and map functions `rhombus(i)` and `pole(i)` that embed subtopologies corresponding to a single mesh or pole in an `ico_mesh`. These functions are defined as follows. They locate rhombus `i` on nodes `i`$c^2$..`(i+1)`$c^2$`-1` and pole $\hat{j}$ on node `10`$c^2$`+j` of an `ico_mesh` topology.

```
function rhombus(i)
{ ? topology() ?= {"ico_mesh", c}, i >= 0, i < 10 ->
    {|| type = {"mesh",c,c},
        size = c*c,
        map_fn = `add_offset(i*c*c)`,
        return( {type, size, map_fn} )
    },
    default -> error()
}

function pole(i)
{ ? topology() ?= {"ico_mesh", c}, i >= 0, i < 2 ->
    {|| type = {"mesh",1,1},
        size = 1,
        map_fn = `add_offset(10*c*c+i)`,
        return( {type, size, map_fn} )
    },
    default -> error()
}

function add_offset(offset,i)
{|| return( i + offset ) }
```

The following sketch of the top-level code for this application shows how mapping is expressed in terms of the icosahedral topology. Ten calls to a `mesh` template are used to set up a mesh cell inside each rhombus, two calls to `poleop` set up the polar computations, a call to a `reduce` cell establishes a global reduction structure (used for computing global minimums), and the `interconnect` procedure establishes communication streams between the various cells. For brevity, we omit the definitional variables representing communication streams.

```
sphere()
{|| {|| i over 0..9 :
            mesh(...) in rhombus(i)
    },
    poleop(...) in pole(0),
    poleop(...) in pole(1),
    reduce(...),
    interconnect(...)
```

```
        }
```

The `mesh` procedure used to create a single mesh is essentially the same as that outlined in Section 3.4. As the code executed within a subdomain is derived from the original Fortran and C, and a global reduction library is available, the only code that must be developed specifically for this application is the `interconnect` procedure and some interface code. To give an impression of what the interface code looks like, we include the main driver executed for each subrhombus. Conceptually, this alternates communication and computation. However, there are some subtleties. For example, the code communicates with a reduction cell to determine a global time step ($\Delta$`t`) consistent with the CFL condition. The use of the new $\Delta$`t` is delayed for one iteration so as to permit overlapping of the communication required for the reduction with other computation. This is achieved by using `dt` as $\Delta$`t` in the current step, and passing `new_dt` to the recursive call to `step` for use as $\Delta$`t` in the next step.

```
step(args,tau,tmax,dt,subrhombus,streams,to_r)
double subrhombus[];
{ ?
    tau < tmax ->
    { ;
        {||   /* Compute "local_dt" */
            find_local_dt(subrhombus,local_dt),
              /* Check old "dt" ok for this time step */
            { ? local_dt < dt -> error() },
              /* Initiate computation of "new_dt" */
            to_r = {{"min",local_dt,new_dt},to_r1}
              /* Exchange data with neighbors */
            communications(streams,subrhombus,streams1)
        },
        pre_filter(args,subrhombus),
          /* Compute on grid, using old "dt" */
        update_grid(args,dt,tau,subrhombus),
        post_filter(args,subrhombus),
          /* Proceed to next time step, passing "new_dt" */
        step(args,tau+dt,tmax,new_dt,subrhombus,streams1,to_r1)
    },
    default -> terminate(args,subrhombus)
}
```

**Experiences.** The parallel code was developed in collaboration with the mathematician who wrote the original sequential code. He provided advice to the undergraduate intern who wrote the parallel program, and assisted with various enhancements to the numerical method. We were fortunate in that the Fortran code used common storage only for constants; storage for program data was allocated by a C driver. This meant that we could reuse much of the Fortran without change. In addition, once we had set up the constants in the common storage on each processor, we were free to map processes to processors in any way we wanted. The complete code totals 1400 lines Fortran, 870 lines C, and 750 lines of PCN. The relatively large amount of PCN code reflects the fact that a number of enhancements to the sequential code were implemented in PCN rather than Fortran, due to the greater ease of programming in the higher-level language.

The parallel program was developed, debugged, and refined on a Sun workstation. The resulting code was moved to a 26-node Sequent Symmetry shared memory computer for performance studies and from there was ported with only minor changes to a 192-node Symult s2010 mesh, 64-node Intel iPSC/860 hypercube, and 528-node i860-based Intel Touchstone Delta mesh. The changes were due primarily to use of a different I/O structure on the Delta, and a need to work around certain deficiencies in the Delta's file system (since corrected). This portability allowed us to obtain scientific results within one week of the Delta's being installed at Caltech in May 1991; applications developed with other technologies were not operational until weeks or even months later.

Profile and trace data provided by Gauge and Upshot allowed us to identify mapping and load balancing problems in early versions of our program. One problem was that a too-coarse grained decomposition of the Fortran code gave the PCN compiler too little opportunity to overlap computation and communication. The result was much idle time. A more fine-grained implementation was easily achieved in a few hours work; this gave the good performance results reported below.

An example of a load imbalance is illustrated in Figure 2. This is a Gauge histogram display of summary data for a run on 492 Delta processors, with each pixel in the vertical dimension representing a processor and shading distinguishing time spent idle (light) and busy (dark). (About 260 processors are visible.) A slight load imbalance is evident: it appears that the processors handling location (0,0) in each rhombus are spending more time computing than other processors. Other Gauge facilities allowed us to isolate the Fortran routine in which the load imbalance occurs, at which point it was easily corrected by modifying the Fortran code. We claim that without Gauge it would have been difficult to correct this load imbalance (or even, perhaps, to suspect its existence).

Good parallel efficiencies are achieved on all four parallel computers. On the Delta, we obtain approximately 2.5 Gflops (5 Mflops per processor) and 80 per cent efficiency relative to the pure Fortran code running on a single i860 processor, for a problem size of N = 56 (approximately 150-km resolution). This compares favorably with other applications, which have typically achieved 3–6 Mflops/processor. Tuning of the sequential Fortran and improvements to the Delta compiler are expected to further improve overall performance.

The parallel code uses a simple embedding of the icosahedral mesh that is not specialized for either hypercube or mesh topologies. This mapping does not attempt to cluster neighboring icosahedral mesh nodes but simply allocates nodes in the icosahedral mesh to consecutive nodes in the underlying computer. It is specified as follows.

```
function icosahedron(c)
{|| type = {"ico_mesh", c},
    size = 10*c*c+2,
    map_fn = `node()`,
    return( {type, size, map_fn} )
}
```

Because parallel efficiency is so good, we have not been motivated to explore alternative

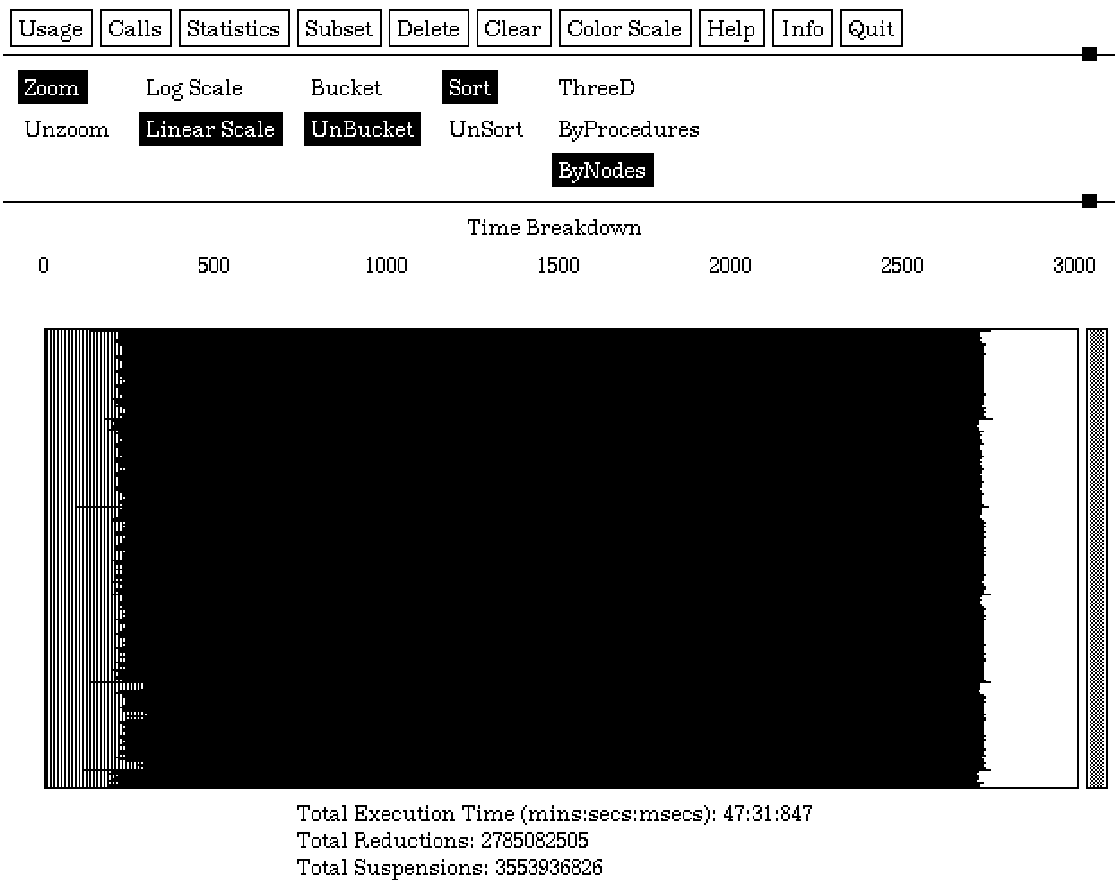



Figure 2: Gauge performance display: time breakdown

mappings of the icosahedral mesh. (Some tinkering with the mapping did not appear to generate significant improvements; this is probably to be expected, given that cut-through routing in the Symult and Delta reduces the importance of communication locality.) Nevertheless, the use of the icosahedral virtual topology leaves us with the option of exploring alternatives in the future, if either improvements in per-node performance increase relative communication costs, or the code is ported to a machine on which locality is more important. One potentially interesting mapping would fold the whole icosahedral mesh structure (locating two or more nodes per processor) so as to reduce message latency. Of course, this can be achieved without changing the application code.

## 6.2 Application Survey

Most applications developed to date are, like the icosahedral code, scientific in nature; almost all use PCN to organize the parallel execution of pre-existing Fortran or C code. Although they solve a wide variety of problems, many can be structured in terms of one or more of a small number of basic cells and templates. We describe some representative examples, indicating the structures used in the implementations. We also give code sizes when this information is available to us.

**Mesh Structures.** The structure of many different mesh-based applications can be captured in one- or two-dimensional mesh templates. A two-dimensional mesh template forms a building block for both the icosahedral code and another climate modeling code based on overlapping stereographic meshes [2] (3800 lines C, 640 lines PCN). Other mesh-based applications include a computational fluid dynamics code developed by Harrar *et al.* for computing Taylor-vortex flows, based on a torus structure [7] (5300 lines Fortran, 900 lines PCN); a finite-element code for simulating flow in Titan rocket engines (9000 lines Fortran, 180 lines PCN); and a parallel implementation of the mesoscale weather model MM4 (15000 lines Fortran, 250 lines PCN). Work is under way to build a version of MM4 in which the mesh template performs dynamic load balancing.

**Ring Structures.** Cells similar to the ring structure presented in Section 4.2 form the basis for several applications. A code for computing nonlinear dynamics properties of extended climate simulations uses an algorithm similar to that used for naive N-body simulations of molecular dynamics (250 lines Fortran, 170 lines PCN). Essentially the same algorithm and structure have also been used in programs for computing molecular interactions and covariances between bases in genetic sequences (the latter is 500 lines C, 800 lines PCN). Similar structures are used in a parallel implementation of the spectral transform method used in climate modeling (7400 lines Fortran, 370 lines PCN).

**Tree Structures.** Tree and butterfly structures are used in many codes to perform parallel reductions. A good example of a code based entirely on a tree structure is one developed by Wright to solve two-point boundary value problems [12] (700 lines Fortran, 50 lines PCN). This algorithm dynamically creates a process tree; data is produced at the leaves, flows up the tree to the root (being reduced at each node) and then back down to the leaves to yield the final solution [6]. The code is defined with respect to a tree virtual topology; the map function that defines this topology specifies how the complete structure is embedded in a parallel computer. Note that it is the low cost of process creation and migration in PCN which makes this dynamic formulation of the algorithm (which provided to be particularly convenient) feasible.

**Self-Scheduling Structures.** A self-scheduling program incorporates code to dynamically map tasks to idle processors; although this approach introduces additional overhead relative to a static schedule, it is essential for some very dynamic problems. Self-scheduling programs can be constructed easily in PCN because of the simplicity of process migration [3]. (The global address space provided by the compiler means that processes can be migrated as data structures.) Self-scheduling applications include codes for aligning genetic sequences, computing phylogenetic trees, and predicting protein structure. (Computational biology is a rich source of applications for self-scheduling techniques, because of the frequent use of heuristics.) An application under development at Argonne schedules tasks to ring structures (each involving several processors) rather than to individual processors. An interesting aspect of all these codes is that the scheduling code can be separated from the application-specific code in a distinct scheduling cell. Alternative scheduling cells can be substituted without changing the application; typically the scheduling structure is specified in 20–100 lines of code.

**Genetic Algorithms.** Genetic optimization algorithms maintain a population of candidate solution vectors and apply simulated natural selection to improve the quality of this population. One approach to parallelizing these algorithms is to maintain multiple populations, with periodic exchanges of individual vectors. Our PCN implementation of a parallel genetic algorithm is parameterized with the initialization, mutation, and mating operators that define a genetic algorithm. The PCN code handles all aspects of execution on a parallel computer, using a *router* cell for asynchronous communication of selected individuals between populations and a *reduction* cell for computing global values when checking for termination. The PCN code totals 500 lines; applications developed with this code have added anything from a few hundred lines of C to 6000 lines of Fortran.

### 6.3 Discussion

As this brief survey shows, PCN applications span a wide range, from the simple and straightforward to the sophisticated and complex. The amount of PCN code incorporated in the various programs depends both on the complexity of the parallel algorithms and the extent to which PCN was used for algorithm development in addition to porting.

It is probably too early to draw firm conclusions regarding the merits of the approach. However, we can make a few observations concerning user reactions. We find that programmers perceive a substantial benefit from the use of PCN (and frequently become ardent advocates of the technology) when their programming problem has one or more of the following characteristics.

- A complex communication structure, or a need to overlap computation and communication.

- A need for load balancing.

- Dynamic computation, communication, or mapping structures.
- A need for portability and scalability.
- Initial performance errors that are corrected by using Gauge.
- An interest in exploring algorithmic alternatives: e.g., different stencils, reduction strategies, communication algorithms, or mappings.
- An ability to reuse existing cells and templates.

In contrast, programmers working with simple, regular problems (such as one-dimensional decompositions with static mapping) find it hard to justify the inevitable learning curve associated with a new approach to programming.

# 7 Conclusions

The ability to develop parallel programs quickly and easily is becoming increasingly important to many scientists and engineers. Although we cannot expect parallel programming to become easy, we can avoid unnecessary difficulties by using appropriate tools. In this article, we have described tools that take us several steps beyond the low level facilities commonly available on parallel supercomputers. A simple concurrent programming notation allows us to express complex parallel algorithms without unnecessary contortions. Interfaces to sequential languages allow us to reuse existing Fortran and C code. Support for cells and templates allows us to define and reuse parallel program structures. Compiler, debugging, and performance analysis tools reduce the labor associated with program development and provide portability over a wide range of machines.

PCN has already been used to develop substantial applications; other application projects are under way. Optimizing compilers are being developed, with particular emphasis on the requirements of fine-grained computers. Libraries of software cells and templates are being developed to support fluid dynamics, geophysical modeling, and computational chemistry; similar libraries can and should be developed for other areas of computational science.

# Acknowledgments

This work is a collaborative effort involving research groups at Argonne and Caltech. As such, it owes a great debt to many individuals. Steve Taylor leads the research at Caltech. Mani Chandy has contributed to the language definition. Sharon Brunett and Dong Ling are responsible for compiler development. Gauge and Upshot were developed by Carl Kesselman and Ewing Lusk, respectively. I-liang Chern and Steve Hammond helped develop the icosahedral grid application.

This research was supported at Argonne by the National Science Foundation's Center for Research on Parallel Computation under Contract NSF CCR-8809615 and by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under Contract W-31-109-Eng-38.

## References


[1] Chandy, C., and Taylor, S., *An Introduction to Parallel Programming*, Jones and Bartlett, 1991.

[2] Chern, I., and Foster, I., 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., 1991.

[3] Foster, I., Automatic generation of self-scheduling programs, *IEEE Trans. Parallel and Distributed Systems*, 2(1):68–78, 1991.

[4] Foster, I., Information hiding in parallel programs, Preprint MCS-P290-0292, Argonne National Laboratory, 1992.

[5] Foster, I., and Taylor, S., *Strand: New Concepts in Parallel Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[6] Foster, I., and Tuecke, S., *Parallel Programming with PCN*, Technical Report ANL-91/32, Argonne National Laboratory, 1991.

[7] Harrar, H., Keller, H., Lin, D., and Taylor, S., Parallel computation of Taylor-vortex flows, *Proc. Conf. on Parallel Computational Fluid Dynamics*, Stuttgart, Germany, Elsevier Science Publishers B.V., 1991.

[8] Herrarte, V., and Lusk, E., Studying parallel program behavior with Upshot, Technical Report ANL-91/15, Argonne National Laboratory, 1991.

[9] Kesselman, C., *Integrating Performance Analysis with Performance Improvement in Parallel Programs*, Technical Report UCLA-CS-TR-91-03, UCLA, 1991.

[10] Olson, R., Using host-control, Technical Memo ANL/MCS-TM-154, Argonne National Laboratory, 1991.

[11] Whitehead, A., *An Introduction to Mathematics*, Oxford University Press, 1958.

[12] Wright, S., Stable parallel algorithms for two point boundary value problems, Preprint MCS–P178–0990, Argonne National Laboratory, and *SIAM J. Sci. Statist. Comput.*, 1992 (in press).