SUPERPASCAL: A PUBLICATION LANGUAGE FOR PARALLEL SCIENTIFIC COMPUTING

PER BRINCH HANSEN

(1994)

Parallel computers will not become widely used until scientists and engineers adopt a common programming language for publication of parallel scientific algorithms. This paper describes the publication language SuperPascal by examples. SuperPascal extends Pascal with deterministic statements for parallel processes and synchronous message communication. The language permits unrestricted combinations of recursive procedures and parallel statements. SuperPascal omits ambiguous and insecure features of Pascal. Restrictions on the use of variables enable a single-pass compiler to check that parallel processes are disjoint, even if the processes use procedures with global variables. A portable implementation of SuperPascal has been developed on a Sun workstation under Unix.

1 INTRODUCTION

One of the major challenges in computer science today is to develop effective programming tools for the next generation of parallel computers. It is equally important to design educational programming tools for the future users of parallel computers. Since the 1960s, computer scientists have recognized the distinction between *publication languages* that emphasize clarity of concepts, and *implementation languages* that reflect pragmatic concerns and historical traditions (Forsythe 1966; Perlis 1966). I believe that parallel computers

P. Brinch Hansen, SuperPascal—A publication language for parallel scientific computing. *Concurrency—Practice and Experience 6*, 5 (August 1994), 461–483.

will not become widely used until scientists and engineers adopt a common programming language for publication of parallel scientific algorithms.

It is instructive to consider the historical role of Pascal as a publication language for sequential computing. The first paper on Pascal appeared in 1971 (Wirth 1971). At that time, there were not very many textbooks on computer science. A few years later, universities began to use Pascal as the standard programming language for computer science courses. The spreading of Pascal motivated authors to use the language in textbooks for a wide variety of computer science courses: introductory programming (Wirth 1973), operating systems (Brinch Hansen 1973), program verification (Alagić 1978), compilers (Welsh 1980), programming languages (Tennent 1981), and algorithms (Aho 1983). In 1983, IEEE acknowledged the status of Pascal as the *lingua franca* of computer science by publishing a Pascal standard (IEEE 1983). Pascal was no longer just another programming tool for computer users. It had become a thinking tool for researchers exploring new fields in computer science.

We now face a similar need for a common programming language for students and researchers in computational science. To understand the requirements of such a language, I spent three years developing a collection of *model programs* that illustrate the use of structured programming in parallel scientific computing (Brinch Hansen 1993a). These programs solve regular problems in science and engineering: linear equations, n-body simulation, matrix multiplication, shortest paths in graphs, sorting, fast Fourier transforms, simulated annealing, primality testing, Laplace's equation, and forest fire simulation. I wrote these programs in *occam* and tested their performance on a *Computing Surface* configured as a pipeline, a tree, a cube, or a matrix of *transputers* (Inmos 1988; McDonald 1991).

This practical experience led me to the following conclusions about the future of parallel scientific computing (Forsythe 1966; Dunham 1982; May 1989; Brinch Hansen 1993a):

1. A *general-purpose parallel computer* of the near future will probably be a multicomputer with tens to thousands of processors with local memories only. The computer will support automatic routing of messages between any pair of processors. The hardware architecture will be transparent to programmers, who will be able to connect processors arbitrarily by virtual communication channels. Such a parallel computer will enable programmers to think in terms of problem-oriented process configurations. There will be no need to map these configura-

tions onto a fixed architecture, such as a hypercube.

2. The regular problems in computational science can be solved efficiently by *deterministic parallel computations*. I have not found it necessary to use a statement that enables a parallel process to poll several channels until a communication takes place on one of them. Nondeterministic communication is necessary at the hardware level in a routing network, but appears to be of minor importance in parallel programs for computational science.

3. Parallel scientific algorithms can be developed in an *elegant publication language* and tested on a sequential computer. When an algorithm works, it can easily be moved to a particular multicomputer by rewriting the algorithm in another programming language chosen for pragmatic rather than intellectual reasons. Subtle parallel algorithms should be published in their entirety as executable programs written in a publication language. Such programs may serve as models for other scientists, who wish to study them with the assurance that every detail has been considered, explained, and tested.

A publication language for computational science should, in my opinion, have the following properties:

1. The language should extend a widely used standard language with *deterministic parallelism* and *message communication*. The extensions should be defined in the spirit of the standard language.

2. The language should make it possible to program *arbitrary configurations* of parallel processes connected by communication channels. These configurations may be defined iteratively or recursively and created dynamically.

3. The language should enable a single-pass compiler to check that parallel processes do not interfere in a time-dependent manner. This check is known as *syntactic interference control*.

The following describes SuperPascal—a publication language for parallel scientific computing. SuperPascal extends Pascal with deterministic statements for parallel processes and synchronous communication. The language permits unrestricted combinations of recursive procedures and parallel statements. SuperPascal omits ambiguous and insecure features of Pascal.

Restrictions on the use of variables permit a single-pass compiler to check that parallel processes are disjoint, even if the processes use procedures with global variables.

Since the model programs cover a broad spectrum of algorithms for scientific computing, I have used them as a guideline for language design. SuperPascal is based on well-known language features (Dijkstra 1968; Hoare 1971, 1972, 1985; Ambler 1977; Lampson 1977; IEEE 1983; Brinch Hansen 1987; Inmos 1988). My only contribution has been to select the smallest number of concepts that enable me to express the model programs elegantly. This paper illustrates the parallel features of SuperPascal by examples. The SuperPascal language report defines the syntax and semantics concisely and explains the differences between SuperPascal and Pascal (Brinch Hansen 1994a). The interference control is further discussed in (Brinch Hansen 1994b).

A *portable implementation* of SuperPascal has been developed on a Sun workstation under Unix. It consists of a compiler and an interpreter written in Pascal. The SuperPascal compiler is based on the Pascal compiler described and listed in (Brinch Hansen 1985). The compiler and interpreter are in the public domain. You can obtain the SuperPascal software by using anonymous FTP from the directory *pbh* at *top.cis.syr.edu*. The software has been used to rewrite the model programs for computational science in SuperPascal.

2 A PROGRAMMING EXAMPLE

I will use pieces of a model program to illustrate the features of SuperPascal. The Miller-Rabin algorithm is used for *primality testing* of a large integer (Rabin 1980). The model program performs p probabilistic tests of the same integer simultaneously on p processors. Each test either proves that the integer is composite, or it fails to prove anything. However, if, say, 40 trials of a 160-digit decimal number all fail, the number is prime with virtual certainty (Brinch Hansen 1992a, 1992b).

The program performs multiple-length arithmetic on natural numbers represented by arrays of w digits (plus an overflow digit):

```
type number = array [0..w] of integer;
```

A single trial is defined by a procedure with the heading

```
procedure test(a: number; seed: real;
  var composite: boolean)
```

Each trial initializes a random number generator with a distinct seed.

The parallel computation is organized as a ring network consisting of a master process and a pipeline connected by two communication channels (Fig. 1).

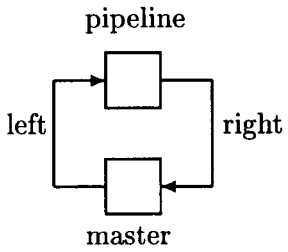

Figure 1 A ring network.

The pipeline consists of p identical, parallel nodes connected by $p + 1$ communication channels (Fig. 2).

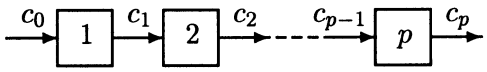

Figure 2 A pipeline.

The master sends a number through the pipeline and receives p boolean values from the pipeline. The booleans are the results of p independent trials performed in parallel by the nodes.

3 MESSAGE COMMUNICATION

3.1 Communication channels

The communication channels of SuperPascal are *deterministic synchronous channels:*

1. A channel can transmit one message at a time in either direction between two parallel processes.

2. Before a communication, a process makes a deterministic selection of a communication channel, a communication direction, and a message type.

3. A communication takes place when one process is ready to send a message of some type through a channel, and another process is ready to receive a message of the same type through the same channel.

3.2 Channel and message types

A channel is not a variable, but a communication medium shared by two parallel processes. Each channel is created dynamically and identified by a distinct value, known as a *channel reference*. A variable that holds a channel reference is called a *channel variable*. An expression that denotes a channel reference is called a *channel expression*. These concepts are borrowed from *Joyce* (Brinch Hansen 1987).

As an example, the declarations

type channel = ∗(boolean, number);
var left: channel;

define a new type, named *channel*, and a variable of this type, named *left*. The value of the variable is a reference to a channel that can transmit messages of types *boolean* and *number* only.

In general, a type definition of the form

$$\textbf{type}\ T = *(T_1, T_2, \ldots, T_n);$$

introduces a new *channel type* T. The values of type T are an unordered set of channel references created dynamically. Each channel reference of type T denotes a distinct channel that can transmit messages of types $T_1, T_2, \ldots, T_n$ only (the *message types*).

3.3 Channel creation

The effect of an *open* statement

$$\text{open}(v)$$

is to create a new channel and assign the corresponding channel reference to a channel variable v. The channel reference is of the same type as the channel variable.

The abbreviation

$$\text{open}(v_1, v_2, \ldots, v_n)$$

is equivalent to

$$\textbf{begin } \text{open}(v_1);\ \text{open}(v_2, \ldots, v_n) \textbf{ end}$$

As an example, two channels, *left* and *right*, can be opened as follows

$$\text{open(left, right)}$$

or as shown below

$$\textbf{begin } \text{open(left)};\ \text{open(right)} \textbf{ end}$$

A channel exists until the program execution ends.

3.4 Communication procedures

Consider a process that receives a number a through a channel, *left*, and sends it through another channel, *right:*

```
var left, right: channel; a: number;
receive(left, a); send(right, a)
```

The message communication is handled by two required procedures, *send* and *receive*.

In general, a *send* statement

$$\text{send}(b,\ e)$$

denotes *output* of the value of an expression e through the channel denoted by an expression b. The expression b must be of a channel type T, and the type of the expression e must be a message type of T.

A *receive* statement

$$\text{receive}(c,\ v)$$

denotes *input* of the value of a variable v through the channel denoted by an expression c. The expression c must be of a channel type T, and the type of the variable v must be a message type of T.

The send and receive operations defined by the above statements are said to *match* if they satisfy the following conditions:

1. The channel expressions b and c are of the same type T and denote the same channel.

2. The output expression e and the input variable v are of the same type, which is a message type of T.

The execution of a send operation delays a process until another process is ready to execute a matching receive operation (and vice versa). If and when this happens, a *communication* takes place as follows:

1. The sending process obtains a value by evaluating the output expression e.

2. The receiving process assigns the value to the input variable v.

After the communication, the sending and receiving processes proceed independently.

The abbrevation

$$\text{send}(b, e_1, e_2, \ldots, e_n)$$

is equivalent to

$$\textbf{begin } \text{send}(b, e_1); \text{send}(b, e_2, \ldots, e_n) \textbf{ end}$$

Similarly,

$$\text{receive}(c, v_1, v_2, \ldots, v_n)$$

is equivalent to

$$\textbf{begin } \text{receive}(c, v_1); \text{receive}(c, v_2, \ldots, v_n) \textbf{ end}$$

The following *communication errors* are detected at run-time:

1. *Undefined channel reference:* A channel expression does not denote a channel.

2. *Channel contention:* Two parallel processes both attempt to send (or receive) through the same channel at the same time.

3. *Message type error:* Two parallel processes attempt to communicate through the same channel, but the output expression and the input variable are of different message types.

Message communication is illustrated by two procedures in the primality testing program. The *master* process, shown in Fig. 1, sends a number a through its left channel, and receives p booleans through its right channel. If at least one of the booleans is true, the number is composite; otherwise, it is considered to be prime (Algorithm 1).

```
procedure master(
  a: number; var prime: boolean;
  left, right: channel);
var i: integer; composite: boolean;
begin
  send(left, a); prime := true;
  for i := 1 to p do
    begin
      receive(right, composite);
      if composite then
        prime := false
    end
end;
```

Algorithm 1 Master.

The pipeline *nodes*, shown in Fig. 2, are numbered 1 through p. Each node receives a number a through its left channel, and sends a through its right channel (unless the node is the last one in the pipeline). The node then tests the number for primality using the node index i as the seed of its random number generator. Finally, the node outputs the boolean result of its own trial, and copies the results obtained by its $i - 1$ predecessors (if any) in the pipeline (Algorithm 2).

3.5 Channel arrays

Since channel references are typed values, it is possible to define an array of channel references. A variable of such a type represents an array of channels.

The pipeline nodes in Fig. 2 are connected by a row of channels created as follows:

```
procedure node(i: integer;
  left, right: channel);
var a: number; j: integer;
  composite: boolean;
begin
  receive(left, a);
  if i < p then send(right, a);
  test(a, i, composite);
  send(right, composite);
  for j := 1 to i - 1 do
    begin
      receive(left, composite);
      send(right, composite)
    end
end;
```

Algorithm 2 Node.

```
type channel = *(boolean, number);
  row = array [0..p] of channel;
var c: row; i: integer;
for i := 0 to p do open(c[i])
```

Later, I will program a matrix of processes connected by a horizontal and a vertical matrix of channels. The channel matrices, h and v, are defined and initialized as follows:

```
type
  row = array [0..q] of channel;
  net = array [0..q] of row;
var h, v: net; i, j: integer;
for i := 0 to q do
  for j := 0 to q do
    open(h[i,j], v[i,j])
```

3.6 Channel variables

The value of a channel variable v of a type T is undefined, unless a channel reference of type T has been assigned to v by executing an open statement

$$\text{open}(v)$$

or an assignment statement

$$v := e$$

If the value of the expression e is a channel reference of type T, the effect of the assignment statement is to make the values of v and e denote the same channel.

If e and f are channel expressions of the same type, the boolean expression

$$e = f$$

is true, if e and f denote the same channel, and is false otherwise. The boolean expression

$$e <> f$$

is equivalent to

$$\textbf{not } (e = f)$$

In the following example, the references to two channels, *left* and *right*, are assigned to the first and last elements of a channel array c:

$$\text{c[0]} := \text{left; c[p]} := \text{right}$$

After the first assignment, the value of the boolean expression

$$\text{c[0]} = \text{left}$$

is *true*.

4 PARALLEL PROCESSES

4.1 Parallel statements

The effect of a *parallel statement*

$$\textbf{parallel } S_1|S_2|\ldots|S_n \textbf{ end}$$

```
procedure ring(a: number;
  var prime: boolean);
var left, right: channel;
begin
  open(left, right);
  parallel
    pipeline(left, right)|
    master(a, prime, left, right)
  end
end;
```

Algorithm 3 Ring.

is to execute the *process statements* $S_1, S_2, \ldots, S_n$ as parallel processes until all of them have terminated.

Algorithm 3 defines a *ring net* that determines if a given integer a is prime. The ring, shown in Fig. 1, consists of two parallel processes, a master and a pipeline, which share two channels. The master and the pipeline run in parallel until both of them have terminated.

A parallel statement enables you to run different kinds of algorithms in parallel. This idea is useful only for a small number of processes. It is impractical to write thousands of process statements, even if they are identical.

4.2 Forall statements

To exploit parallel computing with many processors, we need the ability to run multiple instances of the same algorithm in parallel.

As an example, consider the *pipeline* for primality testing. From the abstract point of view, shown in Fig. 1, the pipeline is a single process with two external channels. At the more detailed level, shown in Fig. 2, the pipeline consists of an array of identical, parallel nodes connected by a row of channels.

Algorithm 4 defines the pipeline.

The first and last elements of the channel array c

$$\text{c[0]} = \text{left} \qquad \text{c[p]} = \text{right}$$

refer to the external channels of the pipeline. The remaining elements

```
procedure pipeline(left, right: channel);
type row = array [0..p] of channel;
var c: row; i: integer;
begin
  c[0] := left; c[p] := right;
  for i := 1 to p − 1 do
    open(c[i]);
  forall i := 1 to p do
    node(i, c[i−1], c[i])
end;
```

Algorithm 4 Iterative pipeline.

c[1], c[2], . . . , c[p–1]

denote the internal channels.

For $p \geq 1$, the statement

```
forall i := 1 to p do
  node(i, c[i–1], c[i])
```

is equivalent to the following statement (which is too tedious to write out in full for a pipeline with more than, say, ten nodes):

```
parallel
  node(1, c[0], c[1])|
  node(2, c[1], c[2])|
    ...
  node(p, c[p–1], c[p])
end
```

The variable i used in the *forall* statement is not the same variable as the variable i declared at the beginning of the pipeline procedure.

In the *forall* statement, the clause

i := 1 **to** p

is a *declaration* of an *index variable* i that is local to the procedure statement

node(i, c[i–1], c[i])

Each node process has its own instance of this variable, which holds a distinct index in the range from 1 to p.

It is a coincidence that the control variable of the *for* statement and the index variable of the *forall* statement have the same identifier in this example. However, the scopes of these variables are different.

In general, a *forall* statement

$$\textbf{forall } i := e_1 \textbf{ to } e_2 \textbf{ do } S$$

denotes a (possibly empty) array of parallel processes, called *element processes*, and a corresponding range of values, called *process indices*. The lower and upper bounds of the index range are denoted by two expressions, e_1 and e_2, of the same simple type. Every index value corresponds to a separate element process defined by an *index variable* i and an *element statement* S.

The *index variable declaration*

$$i := e_1 \textbf{ to } e_2$$

introduces the variable i that is local to S.

A *forall* statement is executed as follows:

1. The expressions, e_1 and e_2, are evaluated. If $e_1 > e_2$, the execution of the *forall* statement terminates; otherwise, step 2 takes place.

2. $e_2 - e_1 + 1$ element processes run in parallel until all of them have terminated. Every element process creates a local instance of the index variable i, assigns the corresponding process index to the variable, and executes the element statement S. When an element process terminates, its local instance of the index variable ceases to exist.

A model program for solving *Laplace's equation* uses a *process matrix* (Brinch Hansen 1993b). Figure 3 shows a $q \times q$ matrix of parallel nodes connected by two channel matrices, h and v.

Each node process is defined by a procedure with the heading:

```
procedure node(i, j: integer;
  up, down, left, right: channel)
```

A node has a pair of indices (i, j) and is connected to its four nearest neighbors by channels, *up, down, left*, and *right*.

The process matrix is defined by nested *forall* statements:

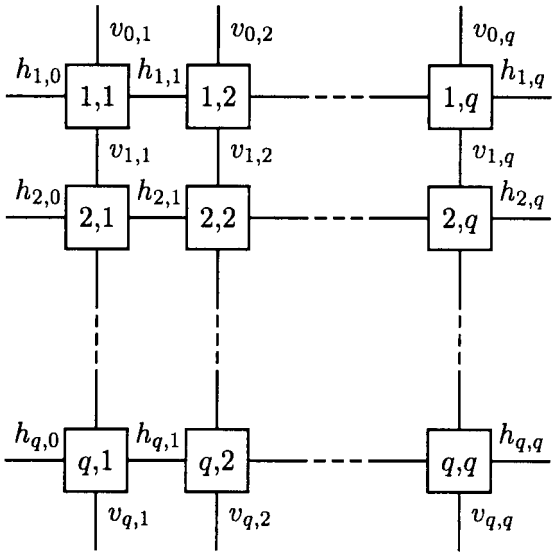

Figure 3 A process matrix.

```
forall i := 1 to q do
  forall j := 1 to q do
    node(i, j, v[i–1,j], v[i,j], h[i,j–1], h[i,j])
```

4.3 Recursive parallel processes

SuperPascal supports the beautiful concept of recursive parallel processes. Figure 4 illustrates a recursive definition of a *pipeline* with p nodes:

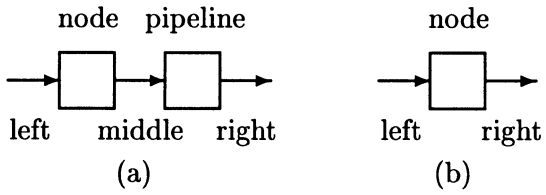

Figure 4 A recursive pipeline.

1. If $p > 1$, the pipeline consists of a single node followed by a shorter pipeline of $p - 1$ nodes (Fig. 4a).

2. If $p = 1$, the pipeline consists of a single node only (Fig. 4b).

The pipeline is defined by combining a recursive procedure with a parallel statement (Algorithm 5).

```
procedure pipeline(min, max: integer;
  left, right: channel);
var middle: channel;
begin
  if min < max then
    begin
      open(middle);
      parallel
        node(min, left, middle)|
        pipeline(min + 1, max,
          middle, right)
      end
    end
  else node(min, left, right)
end;
```

Algorithm 5 Recursive pipeline.

The pipeline consists of nodes with indices in the range from *min* to *max* (where $min \leq max$). The pipeline has a left and a right channel. If $min < max$, the pipeline opens a middle channel, and splits into a single node and a smaller pipeline running in parallel; otherwise, the pipeline behaves as a single node.

The effect of the procedure statement

```
pipeline(1, p, left, right)
```

is to activate a pipeline that is equivalent to the one shown in Fig. 2.

The recursive pipeline has a *dynamic length* defined by parameters. The nodes and channels are created by recursive parallel activations of the pipeline procedure. The iterative pipeline programmed earlier has a fixed length because it uses a channel array of fixed length (Algorithm 4).

A model program for *divide and conquer* algorithms uses a binary *process tree* (Brinch Hansen 1991a). Figure 5 shows a tree of seven parallel processes connected by seven channels.

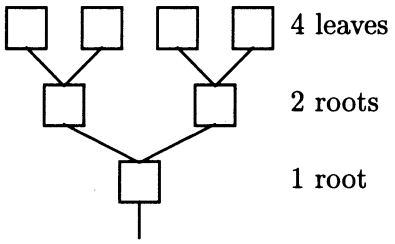

Figure 5 A specific process tree.

The bottom process of the tree inputs data from the bottom channel, and sends half of the data to its left child process, and the other half to its right child process. The splitting of data continues in parallel higher up in the tree, until the data are evenly distributed among the leaf processes at the top. Each leaf transforms its own portion of the data, and outputs the results to its parent process. Each parent combines the partial results of its children, and outputs them to its own parent. The parallel combination of results continues at lower levels in the tree, until the final results are output through the bottom channel.

A process tree can be defined recursively as illustrated by Fig. 6.

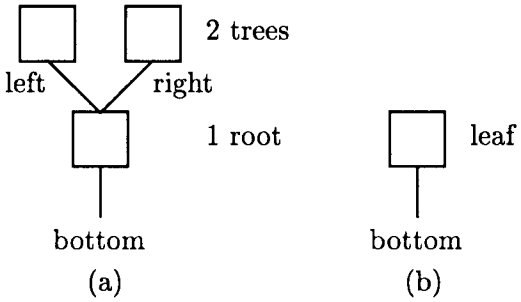

Figure 6 A recursive tree.

A binary tree is connected to its environment by a single bottom channel. A closer look reveals that the tree takes one of two forms:

1. A tree with more than one node consists of a root process and two smaller trees running in parallel (Fig. 6a).

2. A tree with one node only is a leaf process (Fig. 6b).

The process *tree* is defined by a recursive procedure (Algorithm 6). The *depth* of the tree is the number of process layers above the bottom process. Figure 5 shows a tree of depth 2.

```
procedure tree(depth: integer;
  bottom: channel);
var left, right: channel;
begin
  if depth > 0 then
    begin
      open(left, right);
      parallel
        tree(depth − 1, left)|
        tree(depth − 1, right)|
        root(bottom, left, right)
      end
    end
  else leaf(bottom)
end;
```

Algorithm 6 Recursive tree.

The behavior of *roots* and *leaves* is defined by two procedures of the form:

```
procedure root(bottom, left, right: channel)

procedure leaf(bottom: channel)
```

These procedures vary from one application of the tree to another.

The effect of the procedure statement

```
tree(2, bottom)
```

is to activate a binary tree of depth 2.

A notation for recursive processes is essential in a parallel programming language. The reason is simple. It is impractical to formulate thousands of processes with different behaviors. We must instead rely on repeated use of a small number of behaviors. The simplest problems that satisfy this requirement are those that can be reduced to smaller problems of the same kind and solved by combining the partial results. Recursion is the natural programming tool for these *divide and conquer* algorithms.

5 INTERFERENCE CONTROL

5.1 Disjoint processes

The relative speeds of asynchronous, parallel processes are generally unknown. If parallel processes update the same variables at unpredictable times, the combined effect of the processes is time-dependent. Similarly, if two parallel processes both attempt to send (or receive) messages through the same channel at unpredictable times, the net effect is time-dependent. Processes with *time-dependent errors* are said to *interfere* with one another due to *variable* or *channel conflicts*.

When a program with a time-dependent error is executed repeatedly with the same input, the output usually varies in an unpredictable manner from one run to another. The irreproducible behavior makes it difficult to locate interference by systematic program testing. The most effective remedy is to introduce additional restrictions, which make process interference impossible. These restrictions must be checked by a compiler before a parallel program is executed.

In the following, I concentrate on syntactic detection of variable conflicts. The basic requirement is simple: Parallel processes can only update disjoint sets of variables. A variable that is updated by a process may only be used by that process. Parallel processes may, however, share variables that are not updated by any of them. Parallel processes that satisfy this requirement are called *disjoint processes*.

5.2 Variable contexts

I will illustrate the issues of interference control by small examples only. The problem is discussed concisely in (Brinch Hansen 1994b).

In theory, syntactic detection of variable conflicts is a straightforward process. A single-pass compiler scans a program text once. For every state-

ment S, the compiler determines the set of variables that may be updated and the set of variables that may be used as expression operands during the execution of S. These sets are called the *target* and *expression variables* of S. Together they define the *variable context* of S. If we know the variable context of every statement, it is easy to check if parallel statements define disjoint processes.

As an example, the *open* statement

```
open(h[i,j])
```

denotes creation of a component $h_{i,j}$ of a channel array h. Since the index values i and j are known during execution only, a compiler is unable to distinguish between different elements of the same array. Consequently, the entire array h is regarded as a target variable (the only one) of the open statement. The expression variables of the statement are i and j.

An *entire variable* is a variable denoted by an identifier only, such as h, i, or j above. During compilation, any operation on a component of a *structured variable* is regarded as an operation on the entire variable. The target and expression variables of a statement are therefore sets of entire variables.

A compiler cannot predict if a component of a conditional statement will be executed or skipped. To be on the safe side, the variable context of a *structured statement* is defined as the union of the variable contexts of its components.

Consider the conditional statement

```
if i < p then send(right, a)
```

It has no target variables, but uses three expression variables, i, *right* and a (assuming that p is a constant).

5.3 Parallel statements

The choice of a notation for parallel processes is profoundly influenced by the requirement that a compiler must be able to detect process interference. The syntax of a parallel statement

```
parallel S1|S2|...|Sn end
```

clearly shows that the process statements $S_1, S_2, \ldots, S_n$ are executed in parallel.

The following restriction ensures that a parallel statement denotes disjoint processes: *A target variable of one process statement cannot be a target or an expression variable of another process statement.* This rule is enforced by a compiler.

Let me illustrate this restriction with three examples. The parallel statement

parallel open(h[i,j])|open(v[i,j]) **end**

defines two *open* statements executed simultaneously. The target variable h of the first process statement does not occur in the second process statement. Similarly, the target variable v of the second process statement is not used in the first process statement. Consequently, the parallel statement defines disjoint processes.

However, the parallel statement

```
parallel
  receive(left, a)|
  if i < p then send(right, a)
end
```

is incorrect, because the target variable a of the first process statement is also an expression variable of the second process statement.

Finally, the parallel statement

parallel c[0] := left|c[p] := right **end**

is incorrect, since the process statements use the same target variable c.

Occasionally, a programmer may wish to override the interference control of parallel statements. This is useful when it is obvious that parallel processes update distinct elements of the same array. The previous restriction does not apply to a parallel statement prefixed by the clause [*sic*]. This is called an *unrestricted statement.* The programmer must prove that such a statement denotes disjoint processes.

The following example is taken from a model program that uses the process matrix shown in Fig. 3:

```
[sic] { 1 <= k <= m }
parallel
  receive(up, u[0,k])|
  send(down, u[m,k])|
  receive(left, u[k,0])|
  send(right, u[k,m])
end
```

This statement enables a node process to simultaneously exchange four elements of a local array u with its nearest neighbors. The initial comment implies that the two input elements are distinct and are not used as output elements.

The programmer should realize that the slightest mistake in an unrestricted statement may introduce a subtle time-dependent error. The incorrect statement

```
[sic] { 1 <= k <= m }
parallel
  receive(up, u[1,k])|
  send(down, u[m,k])|
  receive(left, u[k,1])|
  send(right, u[k,m])
end
```

is time-dependent, but only if $k = 1$.

5.4 Forall statements

The following restriction ensures that the statement

forall $i := e_1$ **to** e_2 **do** S

denotes disjoint processes: *In a forall statement, the element statement S cannot use target variables.* This is checked by a compiler.

This restriction implies that a process array must output its final results to another process or a file. Otherwise, the results will be lost when the element processes terminate and their local variables disappear. For technological reasons, the same restriction is necessary if the element processes run on separate processors in a parallel computer with distributed memory.

In the primality testing program, a pipeline is defined by the statement

forall i := 1 **to** p **do** node(i, c[i–1], c[i])

Since the node procedure has value parameters only, the procedure statement

```
node(i, c[i–1], c[i])
```

uses expression variables only (i and c).

The incorrect statement

```
forall i := 1 to p – 1 do open(c[i])
```

denotes element processes that attempt to update the same variable c in parallel.

If it is desirable to use the above statement, it must be turned into an *unrestricted statement*:

```
[sic] { distinct elements c[i] }
forall i := 1 to p – 1 do open(c[i])
```

The initial comment shows that the node processes are disjoint, since they update distinct elements of the channel array c.

Again, it needs to be said that a programming error in an unrestricted statement may cause time-dependent behavior. The incorrect statement

```
[sic] forall i := 1 to p – 1 do open(c[1])
```

denotes parallel assignments of channel references to the same array element c_1.

Needless to say, syntactic interference control is of limited value if it is frequently overridden. A programmer should make a conscientious effort to limit the use of unrestricted statements as much as possible. The thirteen model programs, that I wrote, include five unrestricted statements only; all of them denote operations on distinct array elements.

5.5 Variable parameters

To enable a compiler to recognize distinct variables, a language should have the property that distinct variable identifiers occurring in the same statement denote distinct entire variables. Due to the scope rules of Pascal, this assumption is satisfied by all entire variables except variable parameters.

The following procedure denotes parallel creation of a pair of channels:

```
procedure pair(var c, d: channel);
begin
  parallel open(c)|open(d) end
end;
```

The parallel processes are disjoint only if the formal parameters, c and d, denote distinct actual parameters.

The procedure statement

```
pair(h[i,j], v[i,j])
```

is valid, since the actual parameters are elements of different arrays, h and v.

However, the procedure statement

```
pair(left, left)
```

is incorrect, because it makes the identifiers, c and d, *aliases* of the same variable, *left*.

Aliasing of variable parameters is prevented by the following restriction: *The actual variable parameters of a procedure statement must be distinct entire variables (or components of such variables).*

An *unrestricted statement* is not subject to this restriction. A model program for *n-body simulation* computes the gravitational forces between a pair of bodies, p_i and p_j, and adds each force to the total force acting on the corresponding body (Brinch Hansen 1991b). This operation is denoted by a procedure statement

```
{ i <> j } [sic] addforces(p[j], p[i])
```

with two actual variable parameters. The initial comment shows that the parameters, p_i and p_j, are distinct elements of the same array variable p.

5.6 Global variables

Global variables used in procedures are another source of aliasing. Consider a procedure that updates a global seed and returns a random number (Algorithm 7).

The procedure statement

```
random(x)
```

```
var seed: real;

procedure random(var number: real);
var temp: real;
begin
  temp := a*seed;
  seed := temp - m*trunc(temp/m);
  number := seed/m
end;
```

Algorithm 7 Random number generator.

denotes an operation that updates two distinct variables, x and *seed*.

On the other hand, the procedure statement

```
random(seed)
```

turns the identifier *number* into an alias for *seed*.

To prevent aliasing, it is necessary to regard the global variable as an *implicit parameter* of both procedure statements. Since the procedure uses the global variable as a target *and* an expression variable, it is both an *implicit variable parameter* and an *implicit value parameter* of the procedure statements.

The rule that actual variable parameters cannot be aliases applies to all variable parameters of a procedure statement, explicit as well as implicit parameters. However, since implicit value parameters can also cause trouble, we need a stronger restriction defined as follows (Brinch Hansen 1994b): The *restricted actual parameters* of a procedure statement are the explicit variable parameters that occur in the statement and the implicit parameters of the corresponding procedure block. *The restricted actual parameters of a procedure statement must be distinct entire variables (or components of such variables).*

In the primality testing program, the pipeline nodes use a random number generator. If the seed variable is global to the node procedure, then the seed is also an implicit variable parameter of the procedure statement

```
node(i, c[i–1], c[i])
```

Consequently, the statement

forall i := 1 **to** p **do** node(i, c[i–1], c[i])

denotes parallel processes that (indirectly) update the same global variable at unpredictable times. The concept of implicit parameters enables a compiler to detect this variable conflict. The problem is avoided by making the procedure, *random*, and its global variable, *seed*, local to the node procedure. The node processes will then be updating different instances of this variable.

The parallel statement

parallel write(x)|writeln **end**

is invalid because the required textfile *output* is an implicit variable parameter of both *write* statements.

Similarly, the parallel statement

```
parallel
  read(x)|
  if eof then writeln
end
```

is incorrect because the required textfile *input* is an implicit variable parameter of the *read* statement and an implicit value parameter of the *eof* function designator.

5.7 Functions

Functions may use global variables as implicit value parameters only. The following rules ensure that functions have no side-effects:

1. Functions cannot use implicit or explicit variable parameters.
2. Procedure statements cannot occur in the statement part of a function block.

The latter restriction implies that functions cannot use the required procedures for message communication and file input/output. This rule may seem startling at first. I introduced it after noticing that my model programs include over 40 functions, none of which violate this restriction.

Since functions have no side-effects, expressions cannot cause process interference.

5.8 Further restrictions

Syntactic detection of variable conflicts during single-pass compilation requires additional language restrictions:

1. *Pointer types* are omitted.
2. *Goto statements* and *labels* are omitted.
3. *Procedural* and *functional parameters* are omitted.
4. *Forward declarations* are omitted.
5. *Recursive functions* and *procedures* cannot use implicit parameters.

These design decisions are discussed in (Brinch Hansen 1994b).

5.9 Channel conflicts

Due to the use of channel references, a compiler is unable to detect process interference caused by channel conflicts. From a theoretical point of view, I have serious misgivings about this flaw. In practice, I have found it to be a minor problem only. Some channel conflicts are detected by the run-time checking of communication errors mentioned earlier. For regular process configurations, such as pipelines, trees, and matrices, the remaining channel conflicts are easy to locate by proofreading the few procedures that define how parallel processes are connected by channels.

6 SUPERPASCAL VERSUS OCCAM

occam2 is an admirable implementation language for transputer systems (Inmos 1988). It achieves high efficiency by relying on static allocation of processors and memory. The occam notation is somewhat bulky and not sufficiently general for a publication language:

1. Key words are capitalized.
2. A real constant requires eight additional characters to define the length of its binary representation.
3. Simple statements must be written on separate lines.
4. An *if* statement requires two additional lines to describe an empty *else* statement.

5. Array types cannot be named.

6. Record types cannot be used.

7. Process arrays must have constant lengths.

8. Functions and procedures cannot be recursive.

occam3 includes type definitions, but is considerably more complicated than occam2 (Kerridge 1993).

occam was an invaluable source of inspiration for SuperPascal. Years ahead of its time, occam set a standard of simplicity and security against which future parallel languages will be measured. The parallel features of SuperPascal are a subset of occam2 with the added generality of dynamic process arrays and recursive parallel processes. This generality enables you to write parallel algorithms that cannot be expressed in occam.

7 FINAL REMARKS

Present multicomputers are quite difficult to program. To achieve high performance, each program must be tailored to the configuration of a particular computer. Scientific users, who are primarily interested in getting numerical results, constantly have to reprogram new parallel architectures and are getting increasingly frustrated at having to do this (Sanz 1989).

As educators, we should ignore this short-term problem and teach our students to write programs for the next generation of parallel computers. These will probably be general-purpose multicomputers that can run portable scientific programs written in parallel programming languages.

In this paper, I have suggested that universities should adopt a common programming language for publication of papers and textbooks on parallel scientific algorithms. The language Pascal has played a major role as a publication language for sequential computing. Building on that tradition, I have developed SuperPascal as a publication language for computational science. SuperPascal extends Pascal with deterministic statements for parallel processes and message communication. The language enables you to define arbitrary configurations of parallel processes, both iteratively and recursively. The number of processes may vary dynamically.

I have used the SuperPascal notation to write portable programs for regular problems in computational science. I found it easy to express these programs in three different programming languages (SuperPascal, Joyce, and

occam2) and run them on three different architectures (a Unix workstation, an Encore Multimax, and a Meiko Computing Surface).

Acknowledgements

While writing this paper, I have benefited from the perceptive comments of James Allwright, Jonathan Greenfield and Peter O'Hearn.

References

Aho, A.V., Hopcroft, J.E., and Ullman, J.D. 1983. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA.

Alagić, S., and Arbib, M.A. 1978. *The Design of Well-Structured and Correct Programs*. Springer-Verlag, New York.

Ambler, A.L., Good, D.I., Browne, J.C., Burger, W.F., Cohen, R.M., and Wells, R.E. 1977. Gypsy: a language for specification and implementation of verifiable programs. *ACM SIGPLAN Notices 12*, 2, 1–10.

Brinch Hansen, P. 1973. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. 1985. *Brinch Hansen on Pascal Compilers*. Prentice-Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. 1987. Joyce—A programming language for distributed systems. *Software Practice and Experience 17*, 1 (January), 29–50.

Brinch Hansen, P. 1991a. Parallel divide and conquer. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1991b. The n-body pipeline. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1992a. Primality testing. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1992b. Parallel Monte Carlo trials. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1993a. Model programs for computational science: A programming methodology for multicomputers. *Concurrency—Practice and Experience 5*, 5 (August), 407–423.

Brinch Hansen, P. 1993b. Parallel cellular automata: A model program for computational science. *Concurrency—Practice and Experience 5*, 5 (August) 425–448.

Brinch Hansen, P. 1994a. The programming language SuperPascal. *Software—Practice and Experience 24*, 5 (May), 467–483.

Brinch Hansen, P. 1994b. Interference control in SuperPascal—A block-structured parallel language. *Computer Journal 37*, 5, 399–406.

Dijkstra, E.W. 1968. Cooperating sequential processes. In *Programming Languages*, F. Genuys, Ed. Academic Press, New York, 43–112.

Dunham, C.B. 1982. The necessity of publishing programs. *Computer Journal 25*, 1, 61–62.

Forsythe, G.E. 1966. Algorithms for scientific computing. *Communications of the ACM 9*, 4 (April), 255–256.

Hoare, C.A.R. 1971. Procedures and parameters: an axiomatic approach. *Lecture Notes in Mathematics 188*, 102–171.

Hoare, C.A.R. 1972. Towards a theory of parallel programming. In *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrott, Eds. Academic Press, New York, 61–71.

Hoare, C.A.R. 1985. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ.

IEEE 1983. *IEEE Standard Pascal Computer Programming Language*, Institute of Electrical and Electronics Engineers, New York.

Inmos, Ltd. 1988. *occam 2 Reference Manual*, Prentice Hall, Englewood Cliffs, NJ.

Kerridge, J. 1993. Using occam3 to build large parallel systems: Part 1, occam3 features. *Transputer Communications 1* (to appear).

Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.J. 1977. Report on the programming language Euclid. *ACM SIGPLAN Notices 12*, 2 (February).

McDonald, N. 1991. Meiko Scientific, Ltd. In *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*, A. Trew and G. Wilson, Eds. Springer-Verlag, New York, 165–175.

May, D. 1989. Discussion. In *Scientific Applications of Multiprocessors*, R. Elliott and C.A.R. Hoare, Eds. Prentice-Hall, Englewood Cliffs, NJ, 54.

Perlis, A.J. 1966. A new policy for algorithms? *Communications of the ACM 9*, 4 (April), 255.

Rabin, M.O. 1980. Probabilistic algorithms for testing primality. *Journal of Number Theory 12*, 128–138.

Sanz, J.L.C., Ed. 1989. *Opportunities and Constraints of Parallel Computing*, Springer-Verlag, New York.

Tennent, R.D. 1981. *Principles of Programming Languages*, Prentice-Hall, Englewood Cliffs, NJ.

Welsh, J., and McKeag, M. 1980. *Structured System Programming*, Prentice-Hall, Englewood Cliffs, NJ.

Wirth, N. 1971. The programming language Pascal. *Acta Informatica 1*, 35–63.

Wirth, N. 1973. *Systematic Programming: An Introduction*. Prentice-Hall, Englewood Cliffs, NJ.