UNIVERSITY OF JOENSUU
DEP ARTMENT OF COMPUTER SCIENCE
Report Series B

The Programming Language **pm2** for PRAM

Simo Juvaste

Report B-1992-1

ACM D.1.3; D.3.2
UDK 681.3.06

The programming language

pm2

for PRAM

Simo Juvaste
juvaste@cs.joensuu.fi
University of Joensuu, Department of Computer Science
PO Box 1 11, 80101 Joensuu, Finland

1.12.92

ABSTRACT

pm2 is a programming language for PRAM. It is based on Modula-2 with additional elements for the control of parallelism. To program efficiently and correctly with pm2 one should understand some principles of the PRAM model. This report describes the syntax and usage of pm2. The text also includes some comments about the implementation to explain semantics of some structures. The implementation of the language is described in another report (Juvaste 1992).

1. LANGUAGE pm2

The programming language pm2 is a superset of a subset of Wirth's (1982) Modula-2 mixed a bit with Pascal (Jensen, Wirth 1978). Its purpose is to make possible to write programs and algorithms for PRAM and to prove that those programs can be directly compiled to PRAM machine language.

Main differences to Modula-2 are parallel loops, as well as initialization and termination procedures and missing of modules.

Because pm2 is intended for PRAM programming, programmer has to think a bit in terms of PRAM when writing a pm2 program. It is left partly to programmer's responsibility to use all processors correctly. These inconveniences should be acceptable since pm2 is a PRAM programming language and it is possible to compile pm2 programs directly, and get quite efficient and readable PRAM-code.

1.1 Program structure

pm2 has not the module structure of Modula-2 because pm2 is not intended for large applications, and because linking separately created modules for a single PRAM would cause ambiguities in parallelism handling.

A PRAM program consists of two or three parts: control code for reading input to memory and for initializing PRAM, main PRAM code to be run by PRAM, and possibly control code for outputting the results of computation stored in the memory . To express control code parts, pm2 programs contain one or more special procedures whose assembler code is located at the control code before and after the main PRAM-code. The main initializing procedure must exist and must be named `init`, the main termination procedure must be named `finish`, but it may be missing. Besides these other procedures can be declared to be included in initialization or terminating control code.

Program = **program** ident ";" Block ident "."
Block = { Declaration } **begin** StatementSequence **end**

Here ident is the name of the program and must be the same both at the beginning of the program and at the end of the program. The block forming the body of a whole program must include declaration of initializing procedure.

1.2 Declarations

Constant, type, variable and procedure declarations may appear at the beginning of any block. Differences to Modula-2 are shared/private variables, different predeclared types and declaration of initializing and terminating procedures. Declarations may appear in any order , but declaration of an identifier must always textually precede the first reference.

Declaration = **const** { ConstantDeclaration ";" } | **type** { TypeDeclaration ";" } | **var** { VariableDeclaration ";" } | **sharedvar** { VariableDeclaration ";" } | ProcedureDeclaration ";"

1.2.1 Constant declarations

Only word (integer) type constants can be declared. Operators and, or , xor and not in constant expressions are bitwise operations.

ConstantDeclaration = ident "=" ConstExpression
ConstExpression = ["+" | "-"] ConstT erm { AddOperator ConstT erm }
ConstT erm = ConstFactor { MulOperator ConstFactor }
ConstFactor = ident |integer |"(" ConstExpression ")" | **not** ConstFactor

1.2.2 Type declarations

There are two predeclared types: word and boolean. Word matches with a word of the PRAM. Boolean is either `true` or `false`. There are no character or string types since control code cannot read characters. There is no bitset (SET in Modula-2) type since the word length of PRAM is unrestricted. There are no real numbers since PRAM cannot do real arithmetics, and since control code cannot print real numbers.

TypeDeclaration = ident "=" Type
Type = SimpleType | ArrayType | RecordType | PointerType
SimpleType = ident | **word** | **boolean**
PointerType = **pointer to** Type

Array types differ from Modula-2 so that the range of each dimension must be an explicit subrange of the word type. Since there are no enumerations or named subranges they cannot be used like in Modula-2.

ArrayType = **array** ["[" ConstExpression [".." ConstExpression { "," ConstExpression ".." ConstExpression }] "]"] **of** Type

Shared arrays can be open or half-open arrays. An open array (no range, or square brackets at the declaration) is an unlimited one-dimensional array that can be indexed by any integer. A half-open array (only lower bound between square brackets) can be indexed by any integer greater or equal than the lower bound. Open or half-open arrays do not consume extra memory until big indexes are referenced.

Record types have no variable parts like in Modula-2. Members of a record type cannot be open or half-open arrays.

RecordType = **record** FieldList { ";" FieldList } **end**
FieldList = IdentList ":" Type
IdentList = ident { "," ident }

Two types are the same and assignable if :

1) their types have the same name i.e. they <u>are</u> the same type, or
2) they are pointers to the same type

1.2.3 Variable declaration

There are two types of global variables (global variables are declared in the program body block) shared and private, which are declared with keywords **sharedvar** and **var**, respectively. A shared variable is common to all processors, i.e. it is presented only once in the shared memory. On the contrary, each processor has its own copy of private variables in the shared memory.

VariableDeclaration = IdentList ":" Type

Parallel referencing or assigning to shared variables may cause read or write conflicts if concurrent read or write is forbidden in the PRAM model considered. Private variables can be referenced in parallel since all processors refer to different memory locations.

All local variables (declared in procedures) are private. In initialization and termination procedures only global shared variables and local variables can be referenced, not global private variables.

Variables are not initialized to any value. Programmer should always initialize variables before usage since the uninitialized value can be 0, the value of previous instance of the same variable, or garbage.

1.2.4 Procedure declaration

In Modula-2, a procedure can be called textually before its declaration. In pm2 at least the header of a procedure must be given textually before the first call. The declaration containing only a header is adapted from Pascal. When declaring only header of a procedure, all formal parameters and the return value are given normally, but the body of the procedure is replaced by the keyword **forward**. All forwarded procedures must be declared later again without formal parameters but with full body.

ProcedureDeclaration = **procedure** [**initsub** | **finishsub**] ident [FormalParameters] ";"
ProcedureBody ";"
ProcedureBody = **forward** | Block ident
FormalParameters = "(" [FPSection { ";" FPSection }] ")" [":" Type]
FPSection = [**var**] IdentList ":" Type

Value parameters must be of simple or pointer type. Variable parameters (declared with keyword **var**) may be of complex type. Returntype of a function procedure must be of simple or pointer type.

The compiled code of the procedures named "init" and "finish" will be located in the control code of PRAM before or after the main PRAM code, respectively. These procedures cannot be called from anywhere, but they are executed once automatically. If a procedure declaration includes the option **initsub** or **finishsub**, compiled code of the procedure will also be located to the control code of PRAM. Such procedures have the same restrictions and extensions as main initializing and finishing procedures and they can be called only from same class (init, finish) procedures.

1.3 Expressions

The expressions of pm2 have to be of simple or pointer type. This restricts assignment and comparison to simple types. Thus assignments and comparisons of structured types must be done element by element.

Relational operators "#" and "<>", logical/bitwise operators "&" and **and**, and arithmetic operators "/" and **div** are synonyms. "/" and **div** truncate result to integer. Operators **or**, **xor**, **and** (= "&"), and **not** may be used to both word and boolean type expressions.

Relational operator "#" (not equal) is not allowed at the first column of a source file. If it is necessary (should never be) to begin a line with this relational operator, one must use the synonym "<>". This restriction is due to using cpp as pm2 preprocessor.

Expression = SimpleExpression [Relation SimpleExpression]
Relation = "=" | "#" | "<>" | "<" | "<=" | ">" | ">="
SimpleExpression = ["+" | "-"] Term { AddOperator Term }
AddOperator = "+" | "-" | **or** | **xor**
Term = Factor { MulOperator Factor }
MulOperator = "*" | "/" | **div** | **mod** | **and** | "&"
Factor = ConstFactor | Designator [ActualParameters] | "?" Designator | "(" Expression ")"
  | **not** Factor | **log** "(" Expression ")" | **spread** "(" Designator ")" | **true** | **false**
  | **nil**
Designator = ident { "." ident |"[" Expression { "," Expression } "]" | "^" }

In a designator `a.b` refers to field `b` in record `a` (`a` is a designator of record type which has member `b`). `a[b]` refers to element `b` of array `a`. Notice that there is no run-time range checking, so out-of-range reference may yield garbage, or, in case of range underflow, stop the PRAM. `a^` refers to the object to which the pointer a is pointing. Dereferencing an uninitialized pointer will result garbage or PRAM-error. Dereferencing a `nil`-valued pointer will result PRAM-error. Operators ".", "[ ]" and "^" in a designator are associated from left to right and have higher precedence than arithmetic operators.

As an addition to Modula-2, a "?" preceding a designator stands for *address-of* -operator. It returns a pointer to the designator following it. "?" has a lower precedence than operators ".", "[ ]" and "^". This feature was included to pm2 because having address of an array element is useful when building dynamic data structures in parallel, c.f. 1.5.3, and because expressions must be of simple or pointer type. In addition, this address-of -operator also often gives possibilities for writing simpler and faster programs.

The function **log** stands for $\log_2$. The expression **spread** (designator) is to be used to spread values of global variables to all processors in Exclusive Read model, c.f.1.5.3.

1.4 Standard Modula-2 statements

Most pm2 statements and control structures match exactly the respective Modula-2 structures.

StatementSequence = Statement { ";" Statement }
Statement = [assignment |ProcedureCall |IfStatement |ForStatement |ParStatement
|WhileStatement |RepeatStatement |NewStatement |DecrementStatement
|IncrementStatement |ReturnStatement |ReadStatement |WriteStatement
|WritelnStatement | **synchronize**]

1.4.1 Basic statements

assignment = Designator ":=" Expression

The designator and expression of an assignment must be of the same type, simple or pointer. Since there is no such thing as subtype, assignability should be quite clear.

ProcedureCall = ident [ActualParameters]
ActualParameters = "(" [Expression { "," Expression }] ")"

Actual parameters of a procedure call have to match by number and type with those of the procedure declaration. A function procedure call cannot itself form a statement.

NewStatement = **new** "(" Designator ")"

The parameter of **new** must be of pointer type. After execution of the statement the parameter points to newly allocated, but not initialized, structure of the destinationtype of the pointer.

If the initialization procedure makes numerous (#-of-news + #-of-sharedvars + 5 > number-of-processors) new-allocations, programmer must include an assignment H := <size-of-init-heap> to the initialization procedure. H is a predeclared global variable and <size-of-init-heap> is number of new-statements to be executed in the initialization procedure (not the sum of sizes of allocations in words). Within a parallel main program there are no restrictions for the number of new-statements.

DecrementStatement = **dec** "(" Designator ["," Expression] ")"
IncrementStatement = **inc** "(" Designator ["," Expression] ")"

These statements decrement/increment the value of the designator by the expression, or, if expression is missing, by 1. Note that these statements are not included in Wirth's Modula-2 report, but they belong to Modula-3.

ReturnStatement = **return** [Expression]

Return-statement terminates the execution of the current procedure, and in the case of a function procedure it sets the return value of the procedure to the expression. Expression in a return-statement must be of the same type as the returntype declared in the header of the procedure.

1.4.2 Control flow statements

IfStatement = **if** Expression **then** StatementSequence { **elsif** Expression **then** StatementSequence } [**else** StatementSequence] **end**

Expressions following **if** and **elsif** must be boolean. All expressions are evaluated one by one until an evaluation yields true, then the corresponding StatementSequence is executed, but no remaining expressions nor any other StatementSequence. If no any of expression evaluations yields true, else-part, if existing, is executed.

ForStatement = **for** ident ":=" Expression **to** Expression [**by** ConstExpression] **do** StatementSequence **end**

At the beginning the variable ident of type word is initialized to the expression preceding **to**, statementsequence is executed, and variable is incremented by the expression given after **by**, or, if by-part doesn't exist, by 1. Then the variable is compared with the expression following **to**, and if it is less than or equal to it StatementSequence is executed again. This continues until the variable becomes greater than the expression following **to**. If the expression following **by** is negative, it is tested whether the variable is less than the expression following **to**. The expressions before and after **to** are evaluated only once. The iteration variable should not be changed in the StatementSequence.

WhileStatement = **while** Expression **do** StatementSequence **end**

The boolean-type expression between **while** and **do** is evaluated, and, if evaluation yields true, the StatementSequence is executed; then again the expression is evaluated. This continues until evaluation of the expression yields false. After that the StatementSequence is not executed and control flow jumps to the next statement.

RepeatStatement = **repeat** StatementSequence **until** Expression

First the StatementSequence is executed, then expression is evaluated. If evaluation of the expression yields false, StatementSequence is executed again. This continues until the expression yields true.

1.4.3 Input/output statements

Since PRAM cannot do input or output, these statements are allowed only during the initialization of PRAM and after the execution of PRAM, i.e. in **init** or **finish** procedures or procedures tagged with **initsub** or **finishsub**. The subroutines in the control code of the PRAM can currently read only integers and print integers and characters.

The basic input and output statements affect on the streams `stdin` and `stdout` of the process executing PRAM control code.

ReadStatement = **read** "(" Designator ")"

This statement reads an integer from `stdin` into the word-type Designator enclosed in parentheses.

WriteStatement = (**write** | **chwrite**) "(" Expression ")"
WritelnStatement = (**writeln** | **chwriteln**) ["(" Expression ")"]

Write statement outputs the result of the Expression of simple or pointer type to the stream `stdout`. Boolean false appears as "F", true as "T". Pointers appear as shared memory addresses. As an addition to write, **writeln** writes a newline character after result of the expression. If "(Expression)" is missing, **writeln** only writes a newline character. **chwrite** and **chwriteln** write the corresponding ASCII character of the value of the expression.

Also more complex (for example parallel) input/output streams can be implemented by saving data to dynamic data structures, c.f. 2.5.1.

1.5 Parallelism in pm2

The main program is always executed by all processors of the PRAM. Only initializing and finishing procedures are executed sequentially. Programmer must take this into account to prevent read and write conflicts and problems caused by unsynchronized processors.

The number of processors to be used by the program is defined by a predeclared global variable P. P must be assigned to a positive value during the execution of the initialization procedure. It cannot be changed after that. Although P appears to be a shared global variable, all processors have a copy of it. Thus it can be referenced in parallel even in ER-model.

All processors start to execute the code at the beginning of the main program. They can be distinguished either by predeclared read-only variable PID, or by indexing them by a par-statement. PID stands for the processor-id of each processor, and it is an integer value from 1 to P. At initialization or termination procedures, PID cannot be referenced. The par-statement is the recommended way to index processors. PID is purposed mainly for debugging programs.

1.5.1 par-statement

ParStatement = **par** ident ":=" Expression **to** Expression [**sync**] **do** StatementSequence **end** par -statement creates a “process” for each value in the range declared by the range expressions and each of these processes will get its own value for parallel loop variable (ident following **par**). These processes will be spread evenly to all processors of the PRAM. Hence, the execution time of a par -statement is O(T*n/P), where T is the time of the StatementSequence, n is the size of the range and P is the number of processors. Each processor will execute StatementSequence once for each process it got. So StatementSequence will be executed once for each loop variable value in the range (process). Range expressions must yield the same values for all processors.

If the size of the range equals to the number of processors, all processors will execute the StatementSequence once. If the range is smaller than the number of processors some processors will skip StatementSequence. If the range is larger than the number of processors some or all processors will execute statementsequence many times.

The safest and easiest way to program is not to suppose anything about the execution order of iterations of the StatementSequence. In practice, if range (N) is smaller than or equal to the number of processors (P), then all iterations are executed simultaneously . If N > P then first P iterations are executed first, then next P , and so on until all N iterations are executed.

Even if one processor executes more than one process (StatementSequence), it still has only one set of private variables. So if a private variable is assigned at the first process of some processor , the value will be the same until to the next process execution, when it is assigned again. However this is not a big restriction because always one process is executed totally before the same processor begins the next. So always within one StatementSequence, the values of private variables keep stable. To deliver private variables between different par -statements one has to use shared arrays, or at least as many processors as there are processes in the par -statements.

When assigning values for shared variables in a par -statement (in CW model), one must take into account the possible unsynchronization of the processors. For example, if the par -range is larger than the number of processors, an incrementation of a variable can occur several times, once for each iteration, instead of one incrementation appearing in the source code.

1.5.2 Synchronization

In some programs it is vital that operations are executed in a certain order . Because processors work independently , one processor can, for example, read a memory cell before an other processor has written to it. One way to guarantee the correct execution order is to make sure that all processors execute vital statements simultaneously . In pm2 there are two ways to keep processors synchronized, implicit and explicit synchronization.

Adding **sync** option to par -statement before **do** makes all branches equally time consuming. So all processors entering a par- sync- do -statement simultaneously , also exit it simultaneously, no matter what is the relation between the number of processors and processes. Also if some processors execute StatementSequence once less then some others, they will wait equally long time before exiting the par -statement. If StatementSequence within the par -statement is too complex to be synchronized, compiler will give a warning and will add an explicit synchronization after par -statement. This automatic adding can be forbidden by compiler option -ns.

Notice that if the number of processes is larger than the number of processors, different processes in a processor will not, in spite of sync-option, be executed simultaneously , but sequentially one by one. Notice also that if processors do not enter par -sync-do -statement simultaneously , they will neither exit it simultaneously , unless explicit synchronization was added. Some processors can even exit par -statement before some others enter it. So this option only “synchronizes” processes, not processors. All this is due to implicit and quite static processor allocation.

The statement **synchronize** causes processors to wait until all processors have entered it, and after that every processor exits it simultaneously . Synchronization has to be placed in such a place where all processors surely enter . If a processor skips synchronization a ll othe rs will wait for it fo rever . One should notice that this statement only synchronizes processors, not processes (range in par -statement). If synchronization is tried within a par -statement, where some processors skip the last iteration (i.e. range is not divisible by the number of processors), the other processors will jam. So synchronize-statement must not be placed within a par -statement, unless one is sure that the range is divisible by the number of processors.

A drawback of the synchronize-statement is that it takes a time logarithmic to the number of processors. Sync-option only slows down the faster branches of the program to the level of the otherwise slowest branch.

1.5.3 Other topics and hints of parallelism

The **spread** -function helps to use shared variables in exclusive read -model. Its parameter is a variable, and return value is the value of the variable. It is meant for shared variables, but if the parameter is a private variable, the value of the variable of the processor 1 is spread to all processors. Evaluating the expression takes logarithmic time in ER-model and constant time in CR-model. Notice that all processors must call the spread function simultaneously . This is quite restrictive, but it is necessary to make spread efficient. If all processors are not executing spread synchronously it can cause read conflicts and values returned by spread may not be correct for all processors. A similar procedure is used at the beginning of the execution of the PRAM to spread the number of processors and some other values to all processors. So starting an ER PRAM takes a time logarithmic to the number of processors.

When building a big dynamic data structure (e.g. a graph) in parallel the biggest problem is usually connecting the distributed substructures. One possibility is to deliver some pointers between processors through a shared pointer array after the creation of substructures. Another possibility is to have a part of the substructures statically declared as an array (e.g. an element per processor) and mixing these elements with dynamically created elements by address-of -operator . For example vertices of a graph could be elements of an array , and edges could be pointers to these elements. This way of building a graph is especially useful when simulating some processor networks by a PRAM.

2. pm2 -COMPILER

pm2 is the compiler that compiles pm2 source code to PRAM-assembler . The output of the compiler also includes some cpp (C Pre-Processor) directives. After cpp processing, the code only includes PRAM instructions and directives.

This chapter tells how to use the compiler and explains how to read PRAM-code produced by compile, libraries made for pm2 and some of the bugs and restrictions of the pm2 compiler . To allow several source files, there is another version of pm2, pmm2, that first calls cpp to process the pm2 source code.

2.1 pm2 command line parameters

Command to compile a pm2 program to PRAM-assembler code :

pm2 sourcefile destinationfile [-c] [-nc] [-ns] [-d[a][s][t][e][p]] [-EREW | -CREW | -ERCW | -CRCW] [-WEAK | -COMM | -TOLE | -COLL | -CPLU | -ARBI | -PRIO]

where :

`sourcefile`

pm2 source code to be compiled

`destinationfile`

macro-pel code to be produced

`-c`

use all-capital keywords (Modula-2 standard way)

`-nc`

do not add comments to assembler code

`-ns`

do not add explicit synchronization when implicit failed

`-dastep`

debugging prints while compiling

`s` print tokenstream

`t` print symboltables

`e` print assembler instruction creations

`p` print parser procedure calls

`a` all of above

`-EREW -CREW -ERCW -CRCW`

memory access model of the PRAM, defaults `CREW`

-WEAK -COMM -TOLE -COLL -CPLU -ARBI -PRIO

concurrent write conflict handling model, defaults `WEAK`

2.2 PRAM-directives

Within the main PRAM-code some special directives can be placed to examine the execution. This is not allowed in the control code since the emulator does not allow it. Currently `@CLOCK` (print the number of clock cycles used), `@CORE` (print all used memory), `@LASTMEMINDEX` (print the index of the last used memory location) and `@MONITOR` (call PRAM-monitor program) are implemented for the PRAM emulator . These or any directives beginning with "@" can be freely located in the pm2 source code. All directives found in the source code are copied to the corresponding place of the PRAM-code.

2.3 pmm2 : pm2 with macro preprocessing

pmm2 is a script that first runs cpp on source code and then forwards it to pm2. The main use of cpp is to handle multiple source files, i.e. mostly to be able to use `#include` -command. For example procedure libraries can be used in this way . Separate interfaces could be supported by including files with just headers (with forwards) of procedures, and separately including files with implementations (i.e. with bodies) of those procedures.

2.4 PRAM-assembler code produced by pm2

This subsection describes mainly how to read PRAM-code produced by the compiler . The produced code extensively uses the macro facility of the PRAM-assembler . Macros are declared in file `macros.pel` that is included by #include at the beginning of the code. Most important of these macros are those which implement stack operations. Other code included in to program are the standard procedures used by compiler . These are in files `procs.?R?W.pel`, where ?R?W is the memory access model of the PRAM that is used.

Besides instructions, macro calls and directives, the produced code includes comments provided by the compiler . For readability , each source code statement is commented by the type of the statement, e.g. "/* assignment */". All variables are declared at the beginning of the block by comment “/* i = LOCV AR(3) */” or “/* c = SHAR VAR(5) */” etc. References to the variables can then be recognized in the code by this same expression.

Labels for PRAM-code are generated from source code keywords and identifiers by adding a serial number . For example the procedures are labeled by their name. Labels added by the compiler are named after their usage, e.g. “elsif7”, “else6”, “endif6”. The compiler can add several labels to the same instruction if the instruction is destination of different jumps. Some macros take as their parameters one or two labels. They appear something like “POP ANDLS(PO PANDLS6, PO PANDLS7)”.

2.5 Libraries

pmm2 compiler can use libraries by including #include preprocessor directives to the source code. In addition, one usually has to add library procedure calls to different parts of the program, for example to initialize some data structures.

2.5.1 pwrite : Parallel output streams

pwrite-library is written to facilitate output (seemingly) during parallel execution. Because output of many simultaneous writes to the same stream is hard to imagine, and because a PRAM cannot print, output is collected to many independent streams, and streams are flushed by control code one by one after the execution of PRAM.

Usage :

To activate pwrite and a set of streams, add following to the declaration part of the program (always necessary) :

```
const streams = <number-of-streams>
#include "pwrite.pm2"
```

The streams are initialized by

```
initstreams;
```

at the beginning of main program (always necessary). The statement

```
flushstreams;
```

at the termination procedure closes and flushes the streams. Statements

```
pwrite(n, a);
or
pwriteln(n, a);
```

print value `a` to stream `n` in the parallel main program. `flushstreams` prints all streams with stream number as a header, and after that, all values that have been written to the stream. Stream numbering has nothing to do with processor numbering, so different processors may print to the same stream, and each processor can print to different streams. However, simultaneous writing to a stream by many processors is forbidden.

2.6 Bugs and restrictions

Some bugs and a lot of restrictions remain in the compiler. Some of the restrictions and semantic odds, e.g. assigning the number of processors before structured references, are due the paradigm and ideology of the PRAM, the language or the compiler, so they will not be released unless the usage of the PRAM compiler is changed.

If the source code is correct, also compiled code should be correct.

2.6.1 Parser and error recovery

Error recovery is not reliable, and can cause situations where parser totally loses the structure of the program, still the first errormessage is usually correct. Declaration errors sometimes cause compiler internal error messages later on during the compilation.

Warnings of the compiler are not very reliable (the compiler does not analyze the exact control flow), and they should not always be taken seriously. Still they usually inform about possible bugs.

2.6.2 Bugs and restrictions related with parallelism

Parallelism bugs are likely to be corrected first.

sync -option does not reliably synchronize all kinds of statements, neither compiler does reliably give warning of the failure or add explicit synchronization.

Nested par -statements do not execute all iterations if outer statement does not use all processors and if the inner one has a larger range than the outer one. Currently it is recommended not to use nested par -statements, because the biggest advantage compared to many sequentially located par -statements is less writing effort.

In CRCW PRIORITY -model unsynchronized processors cause the last written value to remain. This is of course natural, but if the range in the par -statement is larger than the number of processors the value written at the last iteration by the smallest numbered processor (usually processor 1) will remain.

References

Hämäläinen P. 1992 : *A PRAM emulator*. University of Joensuu, Department of Computer Science, Report B-1992-2.

Jensen K., Wirth N. E. 1978 : *Pascal User Manual and Report*. Springer-Verlag. ISBN 3-540-90144-2.

Juvaste S. 1992 : *An implementation of the programming language pm2*. University of Joensuu, Department of Computer Science, Report A-1992-1. ISSN 0789-7316. ISBN 951-708-112-X.

Wirth N. B. 1982 : *Programming in Modula-2*. Springer-Verlag. ISBN 0-387-12206-0.

Appendix 1: The Syntax of pm2

Program = **program** ident “;” Block ident “.”

Block = { Declaration } **begin** StatementSequence **end**

Declaration = **const** { ConstantDeclaration “;” } | **type** { TypeDeclaration “;” } | **var** { VariableDeclaration “;” } | **sharedvar** { VariableDeclaration “;” } | ProcedureDeclaration “;”

ProcedureDeclaration = **procedure** [**initsub** | **finishsub**] ident [FormalParameters] “;” ProcedureBody “;”

ProcedureBody = **forward** | Block ident

FormalParameters = “(“ [FPSection { “;” FPSection }] “)” [“:” Type]

FPSection = [**var**] IdentList “:” Type

IdentList = ident { “,” ident }

ConstantDeclaration = ident “=” ConstExpression

ConstExpression = [“+” | “-”] ConstTerm { AddOperator ConstTerm }

AddOperator = “+” | “-” | **or** | **xor**

ConstTerm = ConstFactor { MulOperator ConstFactor }

MulOperator = “*” | “/” | **div** | **mod** | **and** | “&”

ConstFactor = ident |integer |”(“ ConstExpression “)” | **not** ConstFactor

TypeDeclaration = ident “=” Type

Type = SimpleType | ArrayType | RecordType | PointerType

SimpleType = ident | **word** | **boolean**

ArrayType = **array** [“[“ ConstExpression [“..” ConstExpression { “,” ConstExpression “..” ConstExpression }] “]”] **of** Type

RecordType = **record** FieldList { “;” FieldList } **end**

FieldList = IdentList “:” Type

PointerType = **pointer to** Type

VariableDeclaration = IdentList “:” Type

Expression = SimpleExpression [Relation SimpleExpression]

Relation = “=” | “#” | “<>” | “<“ | “<=” | “>” | “>=”

SimpleExpression = [“+” | “-”] Term { AddOperator Term }

Term = Factor { MulOperator Factor }

Factor = ConstFactor |Designator [ActualParameters] | “?” Designator |”(“ Expression “)” | **not** Factor | **spread** “(“ Designator “)” | **true** | **false** | **nil**

Designator = ident { “.” ident |”[“ Expression { “,” Expression } “]” | “^” }

ActualParameters = “(“ [Expression { “,” Expression }] “)”

StatementSequence = Statement { “;” Statement }

Statement = [Assigment |ProcedureCall |IfStatement |ForStatement |ParStatement |WhileStatement |RepeatStatement |NewStatement |DecrementStatement |IncrementStatement |ReturnStatement |ReadStatement |WriteStatement |WritelnStatement | **synchronize**]

Assigment = Designator “:=” Expression

ProcedureCall = ident [ActualParameters]

IfStatement = **if** Expression **then** StatementSequence { **elsif** Expression **then** StatementSequence } [**else** StatementSequence] **end**

ForStatement = **for** ident “:=” Expression **to** Expression [**by** ConstExpression] **do** StatementSequence **end**

ParStatement = **par** ident “:=” Expression **to** Expression [**sync**] **do** StatementSequence **end**

WhileStatement = **while** Expression **do** StatementSequence **end**

RepeatStatement = **repeat** StatementSequence **until** Expression

NewStatement = **new** “(“ Designator “)”

DecrementStatement = **dec** “(“ Designator [“,” Expression] “)”

IncrementStatement = **inc** “(“ Designator [“,” Expression] “)”

ReturnStatement = **return** [Expression]

ReadStatement = **read** “(“ Designator “)”

WriteStatement = (**write | chwrite**) “(“ Expression “)”

WritelnStatement = (**writeln | chwriteln**) [“(“ Expression “)”]

ident = letter { letter |digit }

integer = digit { digit }