UNIVERSITY OF JOENSUU

DEP ARTMENT OF COMPUTER SCIENCE

Report Series A

An Implementation of the Programming Language **pm2** for PRAM

Simo Juvaste

Report A-1992-1

ACM D.1.3; D.3.2

UDK 681.3.06

ISSN 0789-7316

ISBN 951-708-1 12-X

An implementation of 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

1.12.1992

ABSTRACT

pm2 is a programming language for PRAM. It is based on Modula-2 with additional elements for the control of parallelism. This report describes the main implementation concepts of pm2, mainly the usage of the PRAM. The compiler from pm2 to PRAM-assembler has been implemented. Processor and memory management of the implementation are simple but adequate and effective enough for the experimental compiler . Moreover the execution times of the primitives match the costs of the primitives used in PRAM-algorithm theory .

Keywords: parallel, language, compiler , PRAM, Modula-2

1. INTRODUCTION

The programming language pm2 is a superset of a subset of W irth' s (1982) Modula-2. It is intended for writing programs and algorithms for PRAM and for proving that such programs can directly be compiled to PRAM machine language. The compiled code can be tested on a PRAM-emulator that is also implemented in this project (Hämäläinen 1992).

The biggest difference from some existing languages for PRAMs is that pm2 is compiled linearly to PRAM-assembler . No explicit processor , process or memory handling is needed during the execution of the PRAM. This means that there is no other run-time process than those executing the main program.

One of the main principles of the implementation has been minimizing interprocessor communication. This is achieved by performing most of the processor and memory allocation during compile and initialization time. Processors do run-time allocation tasks independently , so these tasks are efficiently executed in parallel. The only routine requiring interprocessor communication is the explicit processor synchronization.

A drawback of the minimized interprocessor communication is a slight loss of dynamism. A PRAM could be used to run quite sophisticated run-time system that could be able to handle very versatile dynamic processor allocation, but consequently some allocation tasks could not be done in constant time.

Similar project on implementing a programming language FORK for PRAM is presented by Hagerup & al (1991), they also refer to a joint project on constructing a PRAM prototype. Similar project for various real parallel computers is presented by T ichy & Herter (1992). The difference between pm2 and those projects, as some other for -pardo based languages, is that in pm2 the whole main program, not only the pardo-statements, is executed by all of the processors.

2. THE PRAM

This project uses a PRAM model that has a shared memory and a number of processors that do not have private memory . The model is similar to the model presented by V ishkin (1983). He proposes "a small local memory". In this project the local memory is called local registers. The PRAM-emulator of this project is documented in another report (Hämäläinen 1992). The processors of the PRAM have a common program and they execute it stepwise synchronously . Note, however , that the processors can branch differently during the execution.

The shared memory is unrestricted and word-wise readable and writable. It is assumed that the data exists in the memory before, during and after the execution. Thus the input and the output of the computation are delivered to/from the PRAM through the shared memory .

The number of processors is determined before running the PRAM starts and remains constant throughout the computation. Each processor has its own copy of the single program. Alternatively the program can be considered as shared, being concurrently accessible to all processors (which is possible in the CREW PRAM model).

Each processor has an accumulator and a constant number of other general purpose registers. There are also two special read-only registers: program counter and processor -id. Processor - id is a distinctive integer (1..N) for each processor . Instructions include arithmetic with the accumulator and a register , logical branches, reading/writing from/to the shared memory to/- from the accumulator and instructions to load processor -id and program counter to accumulator . All instructions are expected to take a uniform constant time.

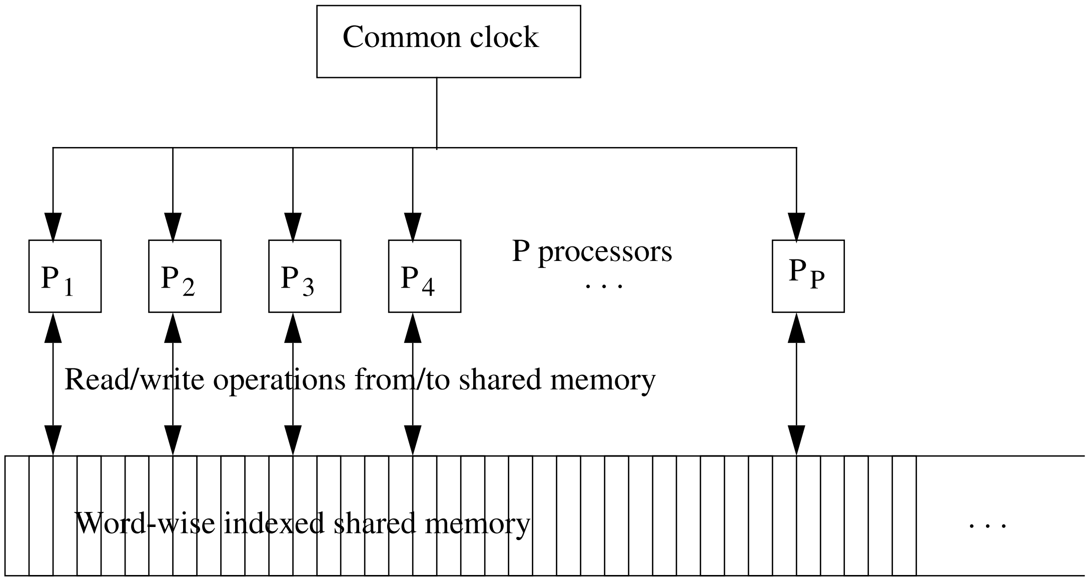

Figure 2.1 The structure of the PRAM

Memory access models of the PRAM emulator include all ER/CR and EW/CW combinations. In concurrent write (CR) model the following variants on write conflict handling are allowed: WEAK, COMMON, TOLERANT, COLLISION, COLLISION+, ARBITRARY and PRIORITY.

The PRAM used supposes that the input is in the global memory before the execution and the output will be left in the global memory after the execution (no e.g. run-time input and output tape). Because in practice input and output have to be handled somehow, the emulator provides special initialization and termination control codes that are used for input and output. The control codes are executed sequentially and they have access to the shared memory and additional instructions for input and output. Initialization control code also initializes the actual PRAM by setting the number of processors and memory access model. Each PRAM program consists of initialization control code, the actual PRAM-code and possibly termination control code.

3. AN OVERVIEW OF LANGUAGE pm2

This chapter describes the most distinctive features of the language pm2. A more careful representation of the language is given in a separate language report (Juvaste 1992). The syntax of the language is presented in Appendix 1.

The syntax is adapted from Modula-2, thus it should be at least understandable for anybody, and also writing programs in should be quite easy. Since the module structure does not match well with the memory management method used in the implementation, a complete program consists of one Pascal-style main program and some procedures within it.

To express initialization and termination control codes of the PRAM emulator , 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. In addition to these other can procedures be declared to be included in initialization or termination control code.

The types of pm2 are related with the word of the PRAM. There are only the word (~integer) and the boolean types. New types can be created using arrays, records and pointers.

Variables have nearly normal visibility scopes. Global variables can be declared as either private or shared. All local variables are private. Privateness means that there is an own copy of the variable for each processor , on the contrary there is only one copy of a shared variable that is common to all processors. The initialization and termination procedures can read and write shared variables and thus do the input and the output of the PRAM. All processors may simultaneously allocate new dynamic variables and share them by delivering pointers to them.

All processors of the PRAM start simultaneously to execute the same main pm2 program. Similar scheme was proposed by Schwartz (1980), he also proposed similar synchronization methods as used in pm2. The programmer can differentiate and use processors by the par -do statement that is a parallel version of the sequential for -do. The difference is that in a for -statement all iterations are executed by all processors, in par -statement each iteration is executed by only one processor .

4. PROCESSOR MANAGEMENT

In the PRAM model used in this project the number of processors is constant throughout the execution of a single program. All processors execute the same program with common clock, but with separate program counters. There is no other processor controller than the processors themselves and all interprocessor communication must be done through writing and reading the shared memory . So processors should make the management as individually as possible. This has been achieved quite well since the only situation when processors have to communicate is the synchronization of all processors. In all other cases each processor computes its task and memory allocation itself.

4.1 Capabilities of a processor

Besides standard RAM-operations, each processor can use its own virtual unrestricted stack and heap (though these are implemented by standard operations in the shared memory). These are managed by three registers: SP (Stack Pointer), FP (Frame Pointer) and HP (Heap Pointer). Each processor has its own copy (in its registers PROS and IL) of the number of processors and the memory interleave factor (memory management will be described later on). Additionally there are two registers for temporary storage TMP and MTMP which are used by the compiler and by macros, respectively . All operations are executed through accumulator (ACC). The mnemonic names given here (ACC, SP , etc.) are just macro-defined names for numbered (1 to 8) registers of the PRAM.

4.2 Starting a PRAM

A special initialization control code is run before the main PRAM-code execution. This control code is expressed in pm2 by a procedure named init. Within the control code the input data must be read to the memory and the number of processors, the memory access model and the number of registers must be evaluated. The initialization procedure has an access to the shared variables and to its own local variables, but not to the private variables of any PRAM processor .

When the number of processors has been set, the input of the program has been read to the shared variables, and interleave factor has been computed, the control code calls the initialization routine Init PRAM of the PRAM to create the PRAM. The memory access model (e.g. CREW) is given as an option of the command line calling the compiler . When the initial control code is completed, the execution of the PRAM starts.

At the beginning of the execution of a PRAM, all processors have the program, their index (1 to P) and empty registers and nothing else. So the first task for each processor is to get some data from the shared memory and to initialize its registers. The processors read from the shared memory the number of processors (P) and the interleave factor (IL). After reading the IL, the processors can begin to use their own stacks and heaps (which are interleaved to the shared memory by IL). Stacks and heaps are initialized by computing the proper values for SP, FP and HP . After assigning the initial pointers (different for each processor) elements of stacks and heaps can always referenced relatively to SP and HP .

Reading P and IL in the CR model is trivial, but in the ER model they must be spread by a binary tree like algorithm using O($\log_2 P$) time.

4.3 par -statement

The par -statement indexes processors for the programmer . It also maps the processes automatically to the processors.

An unsynchronized par -statement (without a sync-option before do) is implemented by a for -statement: So

```
par p := exp1 to exp2 do
       StmtSeq;
end;
```

is implemented as

```
for p := exp1 + # - 1 to exp2 by P do
       StmtSeq;
end;
```

in all processors, "#" stands for processor -id. In other words, at the first stage, the first P executions of the statement sequence StmtSeq are executed in parallel. If the range is greater than the number of processors, the next P executions of the statement sequence are executed, and so on, until the whole range has been covered. The number of executions executed by one processor differs at most by one compared to the number of iterations executed by any other processor .

In a synchronized par -statement also synchronization is needed for those processors that will not execute the statement sequence at the last phase. Thus the par -statement

```
par p := exp1 to exp2 sync do
       StmtSeq;
end;
```

is implemented as

```
for p := exp1 + # - 1 to ⌈exp2 / P⌉ * P by P do
       if p <= exp2 then
              StmtSeq;
       else
              do no-operations
       end;
end;
```

Now all processors execute exactly the same number of statement sequences or equally time-consuming no-operations, but some processors may skip the inner statements at the last iteration. Synchronization option causes the if-statement to be added an implicit else-part that is equally time-consuming than the statements, but that does nothing.

4.4 Implicit process synchronization

The sync-option in a par -statement asks the compiler to try to make all branches of the par -statement and including statements equally time-consuming. Thus if all processors enter it simultaneously , they will also exit it simultaneously . On the other hand, if some processors enter the statement unsynchronously they also exit it unsynchronously . The advantage of this approach is that a program or a section of a program can be kept synchronized without any additional cost. Only the fast branches of the program will slow down to the level of the slowest, therefore the synchronized program terminates as fast as the unsynchronized one.

The slowdown is implemented by calculating the number of clock cycles needed by each branch (e.g. a if-statement) and inserting a proper number of empty commands (NOPs) to the end of the less time-consuming branches. The clock cycle calculation is done during parsing and code-generation and each parsing step returns an estimation of the number of clock cycles needed. Iterative for -statement could be synchronized with a constant range and adding an if-then-else structure similar as in par -sync-do statement.

If the compiler detects that it is unable to balance all branches or it is not sure if the balancing was correct, it gives a warning and optionally adds an explicit synchronization after the par -statement.

Currently the balancing handles only conditional statements. Balancing the number of iterations in iteratitive statements is left programmer 's responsibility .

4.5 Explicit processor synchronization

Implicit compile-time synchronization is not possible if the parallel code includes complex structures in which branches depend on the processor -id and on the input. Run-time synchronization requires processors to communicate to recover synchronized execution (i.e. situation where each processor is executing the same instruction at the same clock cycle). This includes two stages. First, one must guarantee that all processors have finished the previous statements and are ready to be synchronized. After that the knowledge about this situation, and the accurate synchronization information, must be spread to all processors. The first stage implies the semantics of the synchronization: all processors must enter it some time. If one or more processors skip the synchronization, the others will wait those for ever , since they cannot know whether the missing ones are just late or not.

The synchronization is implemented as an assembler subroutine that has no parameters, return value or side-effects, and from which all processors exit simultaneously . There exist different versions of the subroutine for ER and CR PRAMs.

At the first stage of the subroutine, the processors form a treelike structure similar to those that are used e.g. for calculating maximum of a vector in parallel. A processor indicates its readiness and the readiness of its predecessors by writing a value to a certain memory location. The processors waiting for another processor to be ready read the memory location of the waited processor until the waited processor chances the value of its memory location. After $\lceil \log P \rceil$ steps all processors are ready. Processor #1 (leader) knows it and all others are waiting for leader's signal about that. The other processors read a memory location in a two-instruction loop until the leader writes to it. Since the loop is two cycles long, all processors will be within two clock cycles after exiting it. A couple of cycles after the first write the leader will write to the memory location a new value, and the other read it immediately after exiting their loop. Those processors that got the new value are one clock-cycle late and will jump over one instruction to reach the others. Finally the leader will jump to join the others and so all processors execute the same instruction simultaneously. Before returning from the procedure, the memory locations used for synchronization are prepared for the next synchronization. This initialization is also done initially at the starting of the PRAM. Figure 4.1 represents a skeleton of the synchronization algorithm in an algorithm notation. The algorithm is described as a single sequential procedure that is to be executed by all processors. "#" stands for processor-id. The actual assembler code is much more complicated, since it has to take into account such things as accurate clock cycle consumptions, odd number of processors and the actual addresses of the array elements.

```
proc synchronize
       A[1..P] : boolean              // readily all false
       // phase 1 : ensure that all processors have arrived
       i := P/2
       while # ≤ i do
              repeat
                     b := A[#+i]
              until b = true
              i := i/2
       endwhile
       // phase 2 : synchronization
       if # = 1 then
              A[1] := true            // all pros have arrived
              NOP
              A[1] := false           // the second synch pulse
              NOP
       else
              A[#] := true
              repeat                  // wait until all pros
                     b := A[1]        // have arrived
              until b = true
              b := A[1]
              if b = true then        // if did not got the 2nd
                     NOP              // pulse, wait
              endif
       endif
       A[#] := false                  // clear the table for the
endproc                               // next synchronization
```

Figure 4.1 The synchronization algorithm of a CREW PRAM

The synchronization of an EREW PRAM requires the second step to be done a bit differently. Since all other processors cannot simultaneously read the leader's signal, also this must be done in tree-like structure like spreading a value for all processors. At the first synchronization iteration, only one processor reads leader's signal. At the next iteration, those two processors are both leaders, and two other processors read the two signals. After $\lceil \log P \rceil$ steps all processors are synchronized. If P is not an exponent of 2, at the last iteration part of the processors execute NOPs equally long time than it takes to synchronize the last processors.

5. MEMOR Y MANAGEMENT

The PRAM has only one shared memory, but the memory has to be used as a storage both for shared variables and "pseudo private" variables of all processors. The same memory is also used by the initialization and the termination control codes and passing data (at least the input) between control code execution and the main program execution, and between processors. All processors need an unrestricted stack and an unrestricted heap. One possibility to implement these would have been dynamic memory allocation for different processors, but having many processors simultaneously trying to allocate more memory from the same shared memory would probably cause problems and extra time usage. Moreover, for each memory access there would be a calculation from "virtual" address to "physical" address.

Because variable and stack accesses are the hardest operations to implement with the accumulator based instruction set of the PRAM, they should be as simple as possible. So the decision has been to allocate memory statically at the starting the PRAM.

As our goal is to implement a simple working complier, not an optimal one, we choose a naive memory allocation strategy. The use of memory is inefficient, but it is conceptually clear and sufficient for computing relatively small programs. The advantage of it is that it allows us to do even dynamic memory allocations in parallel without interprocessor communication. The use of memory would be more efficient if each processor had an own local memory.

5.1 Memory interleaving

To achieve unboundedness of each memory space, they are interleaved in the shared memory. To achieve symmetry for all memory accesses all structures are interleaved by the same interleave factor. To interleave P stacks, P heaps, global variables and a global heap, the interleave factor is chosen to be 3P. Of any three memory locations one belongs to global heap, the second to stack of some processor and the third to heap of the processor. Thus there are exactly P global variables or dynamic allocations, P local stacks and P local heaps, and all of them have their own unbounded memory space. Successive elements (words) of a structured variable, a stack or a heap are located by steps of 3P. The physical address of a word `n` of e.g. an array is `n * IL + C`, where `C` is the starting address of the array. Since the interleave factor is located in a register of each processor, the calculation can usually be made with two or three instructions. A reference to the next word of the a structure can be done simply by adding the IL to the address of the previous word.

G0,0	S1,0	H1,0	G1,0	S2,0	H2,0	G2,0	S3,0	. . .	HP,0	G0,1	S1,1	H1,1	G1,1	. . .
0	1	2	3	4	5	6	7	. . .	I-1	I	I+1	I+2	I+3	. . .

Figure 5.1. Memory usage

Figure 5.1 shows the structure of the memory. "`Gn,x`" stands for word `x` of global variable or structure `n`, "`Sn,x`" stands for word `x` of stack of processor `n`, "`Hn,x`" stands for word `x` of heap of processor `n`. Numbers at bottom line stand for memory addresses. `I` stands for the interleave factor.

As stated earlier, interleave factor usually is 3P. This, however, restricts the sum of the number of shared variables plus the number of shared dynamic memory allocations to P. This can be released by assigning the heap size variable `H` at the initialization procedure. Then the interleave factor will be 3 * max(`P`, `H` + `G`), where `G` is the number of global variables.

The interleave factor and predeclared shared variables P and H are provided by the control code to main PRAM-code.

The interleave approach is very likely to be inefficient if the PRAM is executed by some physical distributed block-structured-memory computer. But taking the memory structure of the underlying physical machine into account would restrict portability of the compiler between different implementations of PRAM. Therefore the only assumption has been the homogeneous shared memory. If the virtual shared memory of the machine would be mapped to the physical computing elements in an interleaving way the inefficiency problem should be released.

6. COMPILER

Modula-2 syntax was simplified a bit to make possible to parse and compile pm2 with an LL(1) -parser in one pass. Separate modules were left out because PRAM accepts only a single PRAM program and single initialization and termination control codes. Compiling a fragment (e.g. procedure) of code without corresponding control code and the beginning of PRAM-code would have caused some restrictions. The preprocessor inclusion (#include) should be sufficient for procedure and type (i.e. data structure) libraries. Separate interfaces can be supported by including files with just headers (with forwards) of procedures and separately including files with implementations (i.e. proper bodies) of those procedures.

The parser is a recursive LL(1) parser whose structure quite exactly matches the syntax. The parser also controls the code generation, although there is another module to create the instructions.

The compiler is implemented in ANSI-C (Kernighan & Ritchie 1988), thus it is portable to all platforms having a proper C compiler .

7. THE CODE PRODUCED BY THE COMPILER

The compiler produces an assembler file. The binary code used by the PRAM is produced by a separate PRAM-assembler (Hämäläinen 1992). The assembler code contains PRAM-instructions, labels and PRAM-directives. In addition, the code contains cpp (C Pre-Processor) directives which produce pure PRAM-instructions etc. as described above.

Since processors do not have local memory for stacks, all stacks must be implemented in the shared memory . Using the stack in the shared memory requires both reading the memory and updating a register , which is used as stack pointer (SP). Since both of these operations must be done through accumulator (ACC), stack operations are quite expensive (5 or so clock cycles). Since each RAM has only a few registers, the stack is used quite extensively .

The parser produces code quite locally , and the compiler does not do very much optimization. Therefore most expressions are currently evaluated by using stack. The code could be optimized by adding some additional registers and using those instead of stack.

The produced code is quite long because it must serve all processors, and because of the extensive use of the stack.

7.1 Macros

Macros make the produced code more readable and provide an additional abstraction level between the compiler and the PRAM-assembler . These macros would also help porting this compiler to another flavor of PRAM-model (e.g. one with each processor having some private memory). Rewriting the macros (stack, local variable access etc.) to use private memory instead of the shared one should be nearly sufficient. In the compiler , the only thing that should be changed would be the routine that initializes the stacks (i.e. stack pointers). The drawback of using macros is that they hide some optimizing possibilities.

For each register there is an alias macro (e.g. `#define ACC 1`) to improve readability of the code. These aliases are the only names that are used for registers, so the numbering can be changed by changing just the aliases. Register #7 has two aliases: at the parallel code it is used for processors id (PID), at the control code it is used to store the final interleave factor (INITIL).

7.1.1 Stack operations

Stack operations include standard operations between accumulator and stack (`POP` etc.) and between other registers and stack (`POPR(REG)` etc.), and PO P_AND_XXX operations, where XXX stands for an arithmetic or comparison operation. For POP-arithmetic combinations the arithmetic operation is given as a parameter to a single macro (`POPAND(OPER)`) that is used for all those operations. For POP-comparison combinations there is a proprietary macro for each operation (`POPANDEQ` etc.).

7.1.2 Others

Local private and shared variable references are handled by macros `LOCVAR` and `SHARVAR`. To access non-local private variables there is no macro, but a code-generator 's procedure that produces the instructions.

Empty instructions are implemented by macros `NOP`, `NOP3` and `NOP(N)`, which take 1, 3 or N (N $\geq$ 8) clock cycles, but do not have any side effects.

The macro `WRI(REG)` writes the value of register `REG` to memory location pointed by the accumulator . The normal PRAM instruction `WRITE` does just opposite way .

Macros `SCALL(LABEL)` and `SRET` call and terminate parameterless and returnvalueless procedures.

8. Experiments

The compiler has been implemented and tested with several parallel algorithms. In spite of its limitations the language is expressive enough to implement most of the algorithms. The biggest problem is the current inadequacy of nested par -statements. This prevents programmer to write recursively parallel algorithms in pm2. Statically nested par -statements, however , can be simulated by evaluating several indexes explicitly from a larger par -range. An example of this approach can be seen in figure 8.2.

8.1 Example

As an example of programming in pm2 we take matrix multiplication. Figure 8.1 represents a straightforward parallel algorithm derived directly from a sequential one. The algorithm is written for CREW memory model. The algorithm is described as a procedure, thus it needs also some supporting procedures to do the input and the output to be an executable program.

```
proc  matmul(a, b : matrix) : matrix
        for  i ∈ [1..N]  pardo
                for  j ∈ [1..N]  pardo
                        for  k ∈ [1..N]  do
                                c[i,j] := c[i,j] + a[k,j] * b[i,k]
        return   c
```

Figure 8.1 : Matrix multiplication in algorithm notation

Figure 8.2 represents the algorithm written as a pm2 program. The algorithm itself is expressed in the main program. The input matrices are created at initialization procedure (`init`). The values are not read from the user, but some "arbitrary" values are assigned to the matrices. The result matrix is printed by termination procedure (`finish`).

In the main program there are two major differences to the original algorithm: Two nested par-statements have been replaced by a larger one and the code has been optimized by adding an additional variable f to reduce calculating indexes on each iteration. The mapping from one-dimensional par-statement to two-dimensional indexes has been done by div and mod.

```
program matmul;                                    // matrix multiplication for CREW PRAM

const n = 10;                                      // the size of the matrix
sharedvar a, b, c : array[1..n, 1..n] of word;     // input and result matrices

procedure init;                                    // procedure that is run before pram
     var i, j : word;
begin
     P := n * n;                                   // any positive expression would also do
     for i := 1 to n do
          for j := 1 to n do
               a[i, j] := i + j;                   // "arbitrary" values for
               b[i, j] := i * j;                   // input matrices
          end;
     end;
end init;

procedure finish;                                  // procedure that is run after pram
     var i, j : word;
begin
     for i := 1 to n do
          for j := 1 to n do
               writeln(c[i, j]);                   // print the matrix
          end;
          writeln;
     end;
end finish;

var i, j, d, e, f : word;                          // private variables of the prosessors

begin // main
     par d := 0 to n * n - 1 do
          i := d mod n + 1;                        // simulates two nested par-statements
          j := d div n + 1;
          f := 0;
          for e := 1 to n do
               f := f + a[e, j] * b[i, e];
          end;
          c[i, j] := f;
     end;
end matmul.
```

Figure 8.2 Matrix multiplication in pm2

Figure 8.3 represents the main program of the matrix multiplication compiled to PRAM assembler . The assembler codes for initialization and termination procedures has been removed. The code works on a PRAM consisting of any number of processors. One must notice that some of the instructions used in the code are macros, e.g. `PUSH`, and before PRAM execution the macros are substituted by a sequence of pure PRAM instructions. Similarly the register names, e.g. ACC, are replaced by corresponding numbers.

Table 8.1 shows the execution times of the program on different number of processors and different sizes of the matrices. The times are the number of clock cycles used by the execution. The total amount of work is hence product the-number -of-processors * the-number -of-clock-cycles-used. One can notice that the total amount of work is (within a very small constant factor) independent of the number of processors.

```
/* registers's, stacks, etc. pr

matmul0 :READ   0
         STORE  PROS
         READ   6
         STORE  IL
         LOADINDEX
         MUL    =3
         SUB    =2
         STORE  SP
         STORE  FP
         ADD    =1
         STORE  HP
         LOAD   =0
         WRITE  *SP
         PUSH
         PUSH

/* main compiled program */

/* a = SHARVAR(4) */
/* b = SHARVAR(5) */
/* c = SHARVAR(6) */
/* i = LOCVAR(3) */
/* j = LOCVAR(4) */
/* d = LOCVAR(5) */
/* e = LOCVAR(6) */
/* f = LOCVAR(7) */
         LOAD   =5
         MUL    IL
         ADD    SP
         STORE  SP

/* par-statement */
         LOCVAR (5)
         PUSH
         LOAD   =0
         STORE  TMP
         LOADINDEX
         SUB    =1
         ADD    TMP
         STORE  TMP
         POP
         WRI    (TMP)
         LOAD   =10
         PUSH
         LOAD   =10
         POPAND (MUL)
         PUSH
         LOAD   =1
         POPAND (SUB)
         PUSH
         JUMP   compare4
newstep4 : LOCVAR (5)
         STORE  TMP
         READ   *ACC
         PUSH
         LOAD   PROS
         POPAND (ADD)
         WRITE  *TMP
compare4 : TOPR   (TMP)
         LOCVAR (5)
         READ   *ACC
         SUB    TMP
         JPOS   endpar0

/* assigment */
         LOCVAR (3)
         PUSH
         LOCVAR (5)
         READ   *ACC
         PUSH
         LOAD   =10
         POPAND (MOD)
         PUSH
         LOAD   =1
         POPAND (ADD)
         STORE  TMP
         POP
         WRI    (TMP)

/* assigment */
         LOCVAR (4)
         PUSH
         LOCVAR (5)
         READ   *ACC
         PUSH
         LOAD   =10
         POPAND (DIV)
         PUSH
         LOAD   =1
         POPAND (ADD)
         STORE  TMP
         POP
         WRI    (TMP)

/* assigment */
         LOCVAR (7)
         PUSH
         LOAD   =0
         STORE  TMP
         POP
         WRI    (TMP)

/* for-statement */
         LOCVAR (6)
         PUSH
         LOAD   =1
         STORE  TMP
         POP
         WRI    (TMP)
         LOAD   =10
         PUSH
         JUMP   compare5
newstep5 : LOCVAR (6)
         STORE  TMP
         READ   *ACC
         ADD    =1
         WRITE  *TMP
compare5 : TOPR   (TMP)
         LOCVAR (6)
         READ   *ACC
         SUB    TMP
         JPOS   endfor4

/* assigment */
         LOCVAR (7)
         PUSH
         LOCVAR (7)
         READ   *ACC
         PUSH
         SHARVAR(4)
         PUSH
         LOCVAR (6)
         READ   *ACC
         SUB    =1
         MUL    =10
         PUSH
         LOCVAR (4)
         READ   *ACC
         POPAND (ADD)
         SUB    =1
         MUL    IL
         POPAND (ADD)
         READ   *ACC
         PUSH
         SHARVAR(5)
         PUSH
         LOCVAR (3)
         READ   *ACC
         SUB    =1
         MUL    =10
         PUSH
         LOCVAR (6)
         READ   *ACC
         POPAND (ADD)
         SUB    =1
         MUL    IL
         POPAND (ADD)
         READ   *ACC
         POPAND (MUL)
         POPAND (ADD)
         STORE  TMP
         POP
         WRI    (TMP)

         JUMP   newstep5
endfor4 :  DEL
/* end of for-statement */

/* assigment */
         SHARVAR(6)
         PUSH
         LOCVAR (3)
         READ   *ACC
         SUB    =1
         MUL    =10
         PUSH
         LOCVAR (4)
         READ   *ACC
         POPAND (ADD)
         SUB    =1
         MUL    IL
         POPAND (ADD)
         PUSH
         LOCVAR (7)
         READ   *ACC
         STORE  TMP
         POP
         WRI    (TMP)

         JUMP   newstep4
endpar0 :  DEL
/* end of par-statement */

/* end of main program */
         HALT
```

Figure 8.3 The main program of the matrix multiplication in PRAM assembler

Table 8.1 PRAM clock-cycles used by the matrix multiplication

	P=1	P=3	P=N	P=N^2
N = 2	2,247	1,173	1,173	636
N = 5	24,474	8,874	4,974	1,074
N = 10	170,599	58,069	17,149	1,804
N = 50	18,862,599	6,292,629	377,349	7,644

8.2 Future improvements

The memory allocation scheme of shared variables will probably be rewritten. If the global arrays would not be interleaved, the use of memory would be decreased in some cases by factor P. This may, however, restrict the possibility of mixing shared and distributed dynamic data-structures.

Nested par -statements should be possible to implement. A improved processor allocation algorithm would also make the handling of “useless” processors more automatic.

Some features of Modula-2 that were left out will probably be added.

References

Hagerup T., Schmitt A., Seidl H. 1991 : FORK: A High-Level Language for PRAMs. *PARLE 1991: Parallel Architectures and Languages Europe.* Springer-Verlag.

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 : *The programming language pm2*. University of Joensuu, Department of Computer Science, Report B-1992-1.

Kernighan B., Ritchie D. 1988 : *The C Programming Language, 2nd ed*. Prentice-Hall. ISBN 0-13-110362-8.

Schwartz J. T. 1980 : Ultracomputers. *ACM Transactions on Programming Languages and Systems* **2**,4 pp. 484 - 521. ISSN 0164-0925.

Tichy W., Herter G. 1992 : *Modula-2*: An Extension of Modula-2 for Highly Parallel, Portable Programs*. University of Karlsruhe, Technical Report.

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 }