ISSN 0105-8517

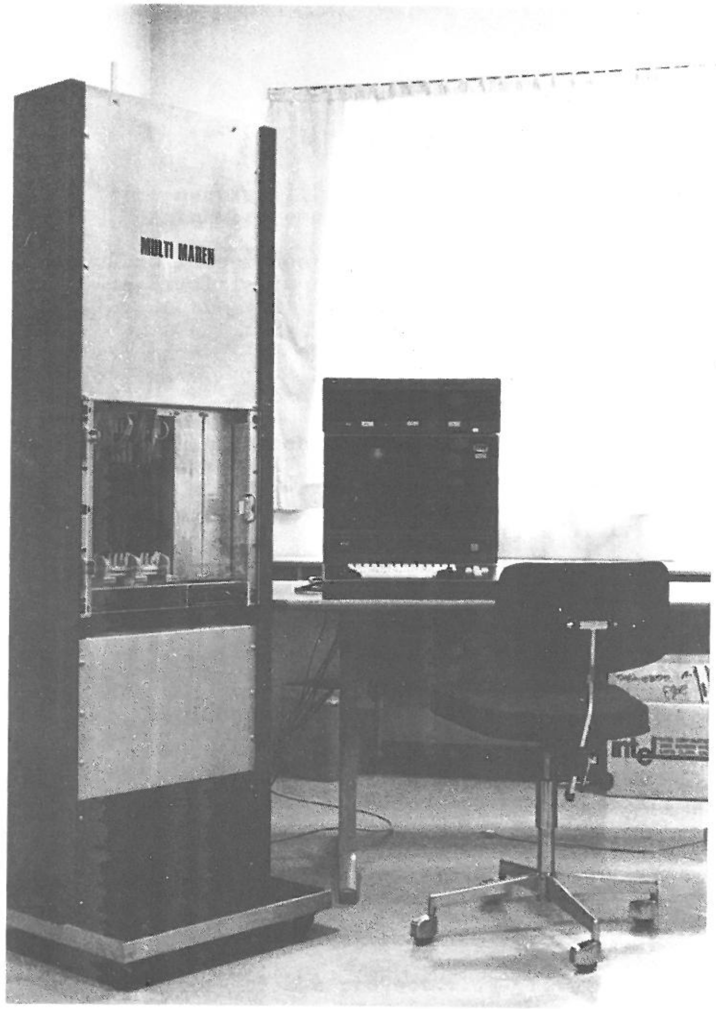
EARLY EXPERIENCE FROM A MULTI-PROCESSOR PROJECT

by

Peter Møller-Nielsen
Jørgen Staunstrup

DAIMI PB-142
January 1982

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



The multi-processor laboratory.

Table of contents:

Chapter 1:

Introduction.

This is an overview of the Multi-Maren multi-processor. Multi-Maren is built to perform experiments with programs for machines with many independent processors. Although it is possible to construct a multi-processor with many independent processors having a very large theoretical processing power, it may be a problem to construct simple and reliable programs which can utilize such a multi-processor. Multi-Maren is intended as a laboratory for experimenting with multi-programs that are simple and reliable, but execute much faster on a multi-processor than they would on a single processor. In chapter two an example of such a multi-program is presented.

Since the emphasis is on program development, as little effort as possible has gone into hardware development. Even if the available hardware is not optimal in all respects, we have avoided designing new hardware, since although specially designed hardware might fit our needs better in some respects, it would probably not be better overall. So the machine is built out of standard boards which, besides the processing unit contain RAM and EPROM stores and a number of interfaces for external communication. The architecture of the machine is described in chapter three.

We wanted to be able to program the machine in a high level language. As was the case with the hardware, existing languages were not optimal, but using an existing language seemed better than designing a new language. We found that the best compromise was Concurrent Pascal. Further justification for our choice can be found in [Søe and Søgaard 1981]. The Concurrent Pascal implementation on Multi-Maren is described in further detail in chapter four.

Chapter five contains a short history of the development of the project to illustrate when and in what context the most important decisions were made.

Acknowledgements.

The work reported below was carried out in a collaboration between the authors and

Jens Søgaard Andersen
Jan Christensen
John W. Nielsen
Kurt Søe

The project has been supported financially by The Danish Natural Sciences Research Council, The Danish Technical Sciences Research Council, The Aarhus University Research Foundation, and Aarhus University.

Some of the early inspiration and motivation for building a multi-processor stems from discussions with Per Brinch Hansen, at the University of Southern California during the period from 1976 to 1978.

Finally, the quick, competent and accurate assistance given by the two Danish Intel-dealers "Søren Lyngsø A/S" and "Multikomponent A/S" has been a great help.

Chapter 2:

Some measurements of the speedup of a multi-program.

A multi-program for adaptive quadrature, described by Rice [Rice 1976] has been implemented in Concurrent Pascal, and executed on a multi-processor machine. Execution times were measured for a varying number of active processors, and nearly linear speedup was observed.

Section 1: Background.

Multi-Maren is part of a laboratory which was established during the period from April 1980 to August 1981. A purpose of the laboratory, and Multi-Maren in particular, is to enable measurements to be taken of the performance of multi-programs executed on a machine with several independent processors. Multi-Maren consists of nine identical processors, each with local store. All of the processors have access to a common store by means of a single, shared bus. The architecture of Multi-Maren is described in detail in chapter 3.

Multi-programs to be executed on Multi-Maren are programmed in Concurrent Pascal. The implementation does not contain tools for measurements beyond the standard routine "realtime" (see [Brinch Hansen 1977], p. 261).

The experiment reported here, is one of the first extensive series of measurements using the Multi-Maren multi-processor. Therefore, the experiment has several objectives: - first, to study the behavior of an algorithm, which belongs to a very general class of algorithms; - second, to check the reliability of the measuring tool, by selecting an algorithm, which can be (and has been) modelled and investigated in detail; - third, to gain experience with the language as it is defined in [Brinch Hansen 1977] and the needs for more elaborate measuring tools to be included in the implementation.

The algorithm which was selected for our study is based on an algorithm for adaptive quadrature published by Rice [Rice 1976].

Section 2 contains a short description of the algorithm, viewed as a particular member of a very broad class of algorithms. Section 3 describes the quantities which were measured, and the structure of the actual Concurrent Pascal program. In section 4 the measured values are presented and discussed. A number of concluding remarks are collected in section 5.

Section 2: The algorithm.

A large number of algorithms are based on the principle of "Divide-and-Conquer". The essence of this principle is the following:

> A problem is solved either directly (if the problem is considered to be simple) or by dividing it into two (or more) problems of the same kind as the original one.

This principle is particularly favorable, provided:

> The cost ("overhead") of subdividing a problem is small compared to the cost of solving the subproblems.
>
> The cost of calculating the solution of the original problem from the solutions of the subproblems, is small.

The principle is particularly suitable as a basis for constructing a multi-program which can be speeded up by simply increasing the number of processors; - i.e. without any extra programming effort. It is even more advantageous if the calculation of the solution from the solutions to subproblems is merely a simple accumulation of the solutions to the subproblems, independent of the sequence in which they become available.

Multi-programs derived from the "Divide-and-Conquer" principle can be given the basic structure shown in fig. 1.
The multi-program consists of a number of identical and independent processes, called "slaves". Each slave is allocated a processor for itself. A slave executes the following sequential program:

```
cycle
   Take a problem from the "problem-heap";
   if  The problem is simple  then
   begin
      Solve the problem;
      Send the solution to the "result-accumulator"
   end
   else
   begin
      Split the problem into subproblems;
      Transfer each subproblem to the "problem-heap"
   end
end;
```

The calculations are started by inserting the initial problem into the "problem-heap". When the "problem-heap" is empty, and all slaves are idle (i.e. waiting for a problem to solve), the initial problem is solved and the solution can be obtained from the "result-accumulator".

This paper reports on the speedup observed for a particular

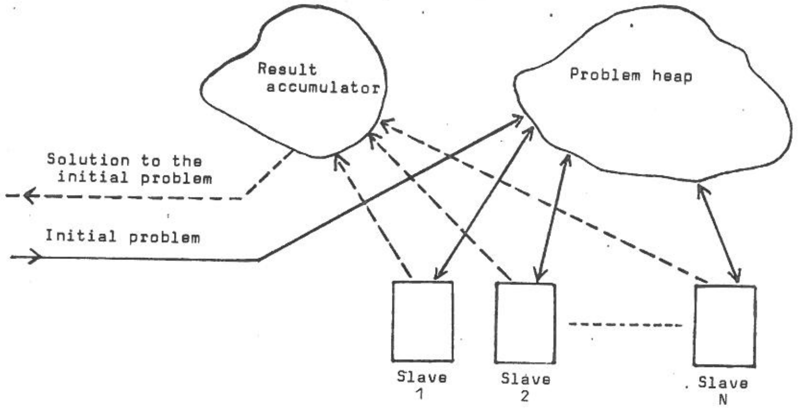


Fig. 1 Basic structure of the algorithm.

member of this class of algorithms, when the number of slaves is varied from 1 to 8.

As mentioned above, the particular algorithm selected for this study, is a version of adaptive quadrature. Quadrature algorithms are members of the class mentioned above for the following reason:

Suppose that Q(a,b) is an approximation to the integral of f on the interval from a to b, and B(a,b) is a bound on the error of this approximation, so that:

$$\left| \int_a^b f - Q(a,b) \right| \leq B(a,b)$$

Then the solution (i.e. Q(a,b) and B(a,b)) to the problem $\int_a^b f$ can be obtained from the solutions to the two subproblems $\int_a^c f$ and $\int_c^b f$ by simply adding the solutions of the sub-problems, since:

$$\left| \int_a^b f - (Q(a,c) + Q(c,b)) \right| \leq B(a,c) + B(c,b)$$

We must now decide:

What is meant by "a simple problem".

How Q and B are calculated for a problem.

How a problem is divided, if it is not simple.

Suppose that the initial problem is to calculate $\int_0^1 f$ to the accuracy Eps(0,1), i.e. to find corresponding values Q(0,1) and B(0,1) , so that B(0,1) $\leq$ Eps(0,1) . A more detailed program for a slave could then be the following:

```
cycle
   Take a problem (specified by the triple:
    [a,b,Eps(a,b)]) from the "problem-heap";
   Calculate Q(a,b) and B(a,b);
   if B(a,b) ≤ Eps(a,b) then
      The problem is classified as "simple",
       and Q(a,b) and B(a,b) are sent to the
       "result-accumulator"
   else
   begin
      Values c, Eps(a,c) and Eps(c,b) are chosen
       so that Eps(a,c) + Eps(c,b) = Eps(a,b);
      Send the problems [a,c,Eps(a,c)] and
       [c,b,Eps(c,b)] to the "problem-heap"
   end
end;
```

The initial problem is [0,1,Eps(0,1)] . When the "problem-heap" is empty and all the slaves are idle, the accumulated values of Q and B form a solution to the initial problem since:

$$\left| \int_0^1 f - \sum Q(a,b) \right| \leq \sum B(a,b) \leq \sum Eps(a,b) = Eps(0,1)$$

The program described in [Rice 1976] can be viewed as a further refinement and improvement of the skeleton program above.
In this study, the class of functions f that can be handled, is restricted to monotonic functions with monotonic first derivatives. This restriction simplifies the calculation of B considerably. The reader is refered to [Rice 1976] for more details on this and other aspects of the actual algorithm.

Section 3: The multi-program.

The complete Concurrent Pascal program, which is executed on Multi-Maren, is a merge of the following three independent and very different program modules:

1: The algorithm.
This module implements the algorithm under investigation.

2: The measurement strategy.
This module implements the decisions about what quantities to measure during a single execution of the algorithm, and when to collect them.

3: The measurement tactics.
This module makes it possible to perform several independent executions of the basic algorithm (and related data collections) in a single execution of the complete program. The values of the parameters for the individual executions of the algorithm (e.g. the accuracy required for the quadrature) are also controlled by this program module, and all input and output is done here. In short, - this module contains the overall control of the progression of the measurements during a single program execution.

The program module for the algorithm can be derived directly from fig. 1 . The "problem-heap" and the "result-accumulator" are variables of type monitor, and the operations on the heap and the accumulator are easily mapped into corresponding entry routines. The set of slaves corresponds to an array of variables of a single process type with the two monitors as formal parameters.

The second module measures the following quantities:

T :
The total execution time for a given initial problem, i.e. the elapsed time from when the initial problem is put into the "problem-heap" until the solution (to the required precision) is available in the "result-accumulator". This time period will be referred to as the "solution period".

$\#_f$:
The number of problems taken from the "problem-heap" and successfully completed during the solution period. $\#_f$ may be interpreted as the total amount of work done by the set of slaves.

$\overline{N}_A$:
The average number of active slaves. A slave is inactive, when it is in the state "delayed" in the monitor "problem-heap", because the heap is temporarily empty. The average is taken over the solution period.

The choice to measure T, $\#_f$ and $\overline{N}_A$ was based on a series of pilot-experiments not reported here.

The third module controls the execution of the complete program as follows:
The number, N, of slaves is chosen between 1 and 8. A sequence of 5 initial problems, with required accuracy Eps= 10^{-i} i=1..5 , are solved. For each of these 5 independent problems T, $\#_f$ and $\overline{N}_A$ are measured.

This module is a separate process, which is in the state "delayed" during the solution periods. The complete program has the structure shown in fig. 2.

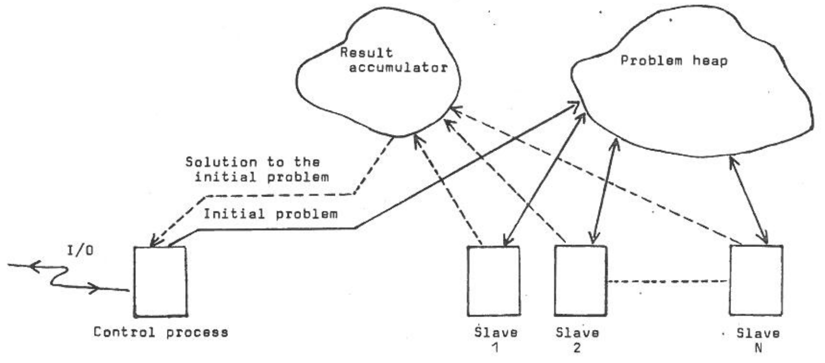


Fig. 2 Structure of the complete program.

How to construct the complete program as a merge of three independent program modules is a problem which is well-known from discrete event simulation. The modules are concerned with very different matters using different concepts. It would be advantageous if they could be written and maintained as three separate program texts, possibly written in three different languages. Unfortunately, this is not possible in the present case - it is especially difficult to confine the third module (the measurement tactics) using Concurrent Pascal language constructs.

The insertion of extra statements into the algorithm for the purpose of measuring some properties of the algorithm, influences the execution times of the algorithm in two ways:

An increase in the execution times (e.g. the execution time for one cycle in a slave).

A change in the fraction of the execution time which a slave spends in a monitor (e.g. exchanging problems with the "problem-heap") relative to the time spent outside monitors.

The first effect is of no importance for this study, since only speedup is considered.

The second effect will, in general, influence the speedup. The reasons why this influence does not appear in the present measurements, are discussed in section 4.

Note that the performance of this multi-program does not depend on the fairness of the implementation of monitors (see chapter 4), since all slaves are identical processes and the slaves never use "busy waiting" when they request a problem from the "problem-heap".

Section 4: Measured values and analysis.

This section presents the measured values for the quadrature problem:

$$\int_0^{30} 9.1282546_{10}{-3} \sqrt{x + 0.001}\ dx$$

Table 1 shows the measured values for T and $\#_f$ when N (the number of slaves) varies from 1 to 8, and Eps (the required accuracy) varies from $_{10}{-1}$ to $_{10}{-5}$. In the table, $\#_f$ shows some variations for fixed Eps and varying N. For a fixed Eps, the set of subproblems is the same for all N, and only the sequence in which the solutions arrive at the "result-accumulator" can vary, when N varies. This variation of the sequence, coupled with rounding errors in the accumulation of Q and B, influences the exact number of solutions that must be accumulated before the required accuracy can be guaranteed.

In fig. 3 the values of T from table 1 are shown in the form of the speedup defined as:

$S = T_1 / T_N$, where N is the number of slaves.

The dotted line corresponds to a speedup, which is linear in N.

The deviation from a linear speedup can be understood by observing, that not all slaves are active throughout the solution period, i.e. $N_A < N$. Therefore , a speedup which is linear in N_A rather than N should be expected. Fig. 4 shows the measured values for N_A as a function of N for Eps = $_{10}{-4}$ and $_{10}{-5}$.

	N	1	2	3	4	5	6	7	8
Eps=10^{-1}	T (sec)	4.54	3.09	2.99	2.98	2.95	2.98	2.95	2.97
	$\#_f$	3	3	3	3	3	3	3	3
Eps=10^{-2}	T (sec)	16.29	8.93	7.33	6.93	6.71	6.74	6.65	6.74
	$\#_f$	11	11	12	11	12	12	11	12
Eps=10^{-3}	T (sec)	54.04	27.78	20.00	15.61	13.39	11.74	11.13	10.31
	$\#_f$	37	37	37	37	37	37	37	39
Eps=10^{-4}	T (sec)	185.75	93.50	63.90	40.22	39.56	33.72	29.57	26.73
	$\#_f$	127	127	127	127	127	127	127	127
Eps=10^{-5}	T (sec)	654.48	327.77	221.46	166.08	133.11	111.99	97.98	87.93
	$\#_f$	452	452	456	452	452	452	458	468

Table 1 T and $\#_f$ as a function of N and Eps.

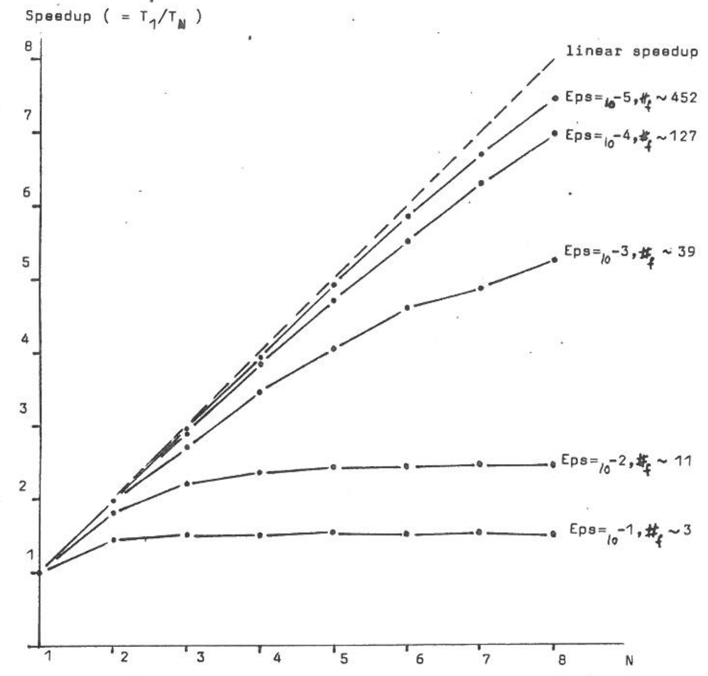


Fig. 3 Speedup as a function of N.

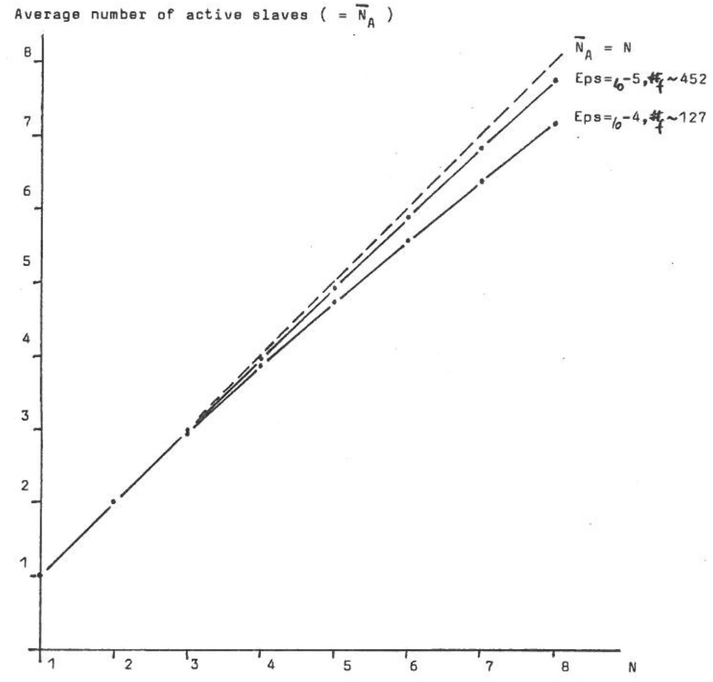


Fig. 4 Average number of active slaves (measured values) as a function of N.

The measured values of $\overline{N}_A$ can be predicted by the following model:

Assume that the only loss in activity of the slaves is due to a lack of problems during the first part of the solution period. When the initial problem is inserted into the problem heap one slave becomes active. After one execution of the cycle in the active slave, two slaves become active, then four slaves, and so on until all slaves are active. The model assumes that all slaves remain active until the end of the solution period. For N=8, $N_A(t)$ is modelled as shown in fig. 5. The unit of the abscissa is the execution time k_N for one cycle of a slave. The model assumes that the variation of k_N from problem to problem is relatively small. This assumption can be verified by inspection of the algorithm in [Rice 1976]. Note that t_1 in fig. 5 depends only on N and t_2 on N and $\#_f$. The values of $\overline{N}_A$ predicted by this model for Eps = 10^{-4} and 10^{-5} agree with the measured values on fig. 4 within 1%.

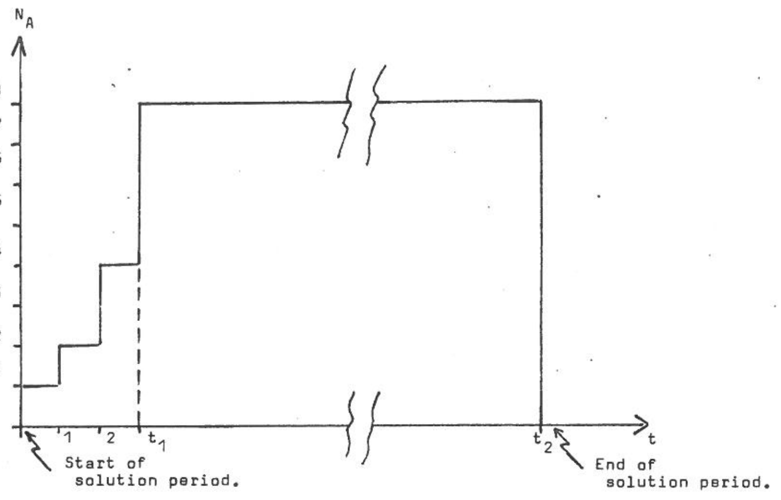


Fig. 5 The number of active slaves as a function of time.

Using the definition above for k_N, the following relation is expected to hold:

$$T_N = (k_N \cdot \#_f) / \overline{N_A}$$

where $k_N \cdot \#_f$ is the amount of work to be done and $\overline{N_A}$ is the number of slaves actually working.

Inserting the measured values (corresponding to Eps= 10^{-4} and 10^{-5}) into the right-hand-side of the expression:

$$k_N = (T_N \cdot \overline{N_A}) / \#_f$$

it can be verified that k_N is a constant (within 2%), when N varies from 1 to 8.

The execution time for one cycle in a slave (other than for queueing at a monitor gate) is almost a constant from problem to problem. Hence, k_N is expected to be independent of N until a shared resource (i.e. a monitor - either the "problem-heap" or the "result-accumulator") is saturated. Additional measurements have indicated that saturation for this complete program will happen at around N = 10.

Similar measurements have been performed with other integrands and integration intervals. They all agree with the behavior described above.

Section 5: Concluding remarks.

This section contains two separate remarks: one about the prediction of execution times, and one about the importance of global adaptivity (i.e. the use of a single, common problem heap) in the algorithm.

The relation: $T_N = (k_N \cdot \#_f) / \overline{N_A}$ can be used to predict T_N for different values of Eps and N in the following way:

Measure T_N and $\#_f$ for a suitable value of N and Eps, calculate $\overline{N_A}$ from the model (in section 4), and calculate k_N by means of the relation above. For given values of N and Eps, $\#_f$ is calculated using the relation:

$$\text{Eps} \cdot (\#_f)^2 \cong \text{constant} \qquad \text{[Rice 1976]}.$$

$\overline{N_A}$ is then calculated from the model, and finally T_N can be calculated.

In order to study the importance of global adaptivity of the algorithm, the following multi-program for adaptive quadrature was written:

Given N slaves, an initial problem and a required accuracy, Eps. Divide the interval of integration into N intervals (a_i, b_i), i = 1,...,N of equal size. Give each slave one of the problems described by an interval (a_i, b_i) and a required accuracy of Eps / N. Each slave solves its own problem completely, and they share no common problem heap. When all slaves have solved their problem, the original problem is solved by accumulating the N solutions. The execution time T for the origional problem (with interval: (a,b) and accuracy: Eps) is the maximum of the execution times $T^{(i)}$, $i = 1,\dots,N$ for the N problems with interval: (a_i, b_i) and accuracy: Eps / N.

The execution times for this program have been measured and compared with the execution times reported in section 4. Table 2 shows the comparison. The importance of the global adaptivity can be seen by comparing the execution times for the two programs. Note also the large (and increasing) value of $\max_i(T^{(i)}) / \min_i(T^{(i)})$.

N	1	2	4	8	
$\dfrac{\max_i T^{(i)}}{\min_i T^{(i)}}$	1.00	4.04	8.65	14.03	Local adaptivity.
$T = \max_i T^{(i)}$	654.49	492.99	420.35	325.88	
Speedup	1.00	1.33	1.56	2.01	
T	654.48	327.77	166.08	87.93	Global adaptivity. (see section 4)
Speedup	1.00	2.00	3.94	7.44	

Table 2 Comparisons between multi-programs using local- and global adaptivity.

Chapter 3:

The Multi-Maren machine architecture.

This chapter describes the Multi-Maren hardware. The machine is built out of standard components, but a wide range of adjustments and combinations are possible, and the machine has been shaped to our needs by selecting from the available options.

Section 1: Overview.

Multi-Maren consists of nine identical processors connected to a common store by a shared bus as shown on fig. 1.

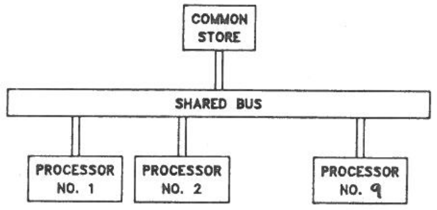


Fig. 1 The Multi-Maren architecture.

The processors are Intel iSBC 86/12A boards each of which contains a 16 bit microprocessor (Intel 8086), a local store and some I/O interfaces. The shared bus is an Intel Multibus equipped with a specially constructed arbiter (see section 3). The common store is an Intel iSBC 032 memory board. Each of these is described in further detail below.

Section 2: The processor.

The processors are Intel iSBC 86/12A boards [Intel 1978c]. These boards can be adapted to a specific application by a number of wire wraps and switches. This section describes how the boards have been adapted to our use.
The following parts of the board are currently used: The 8086

microprocessor, the RAM store, the EPROM store, the timers and the serial I/O port.
The Intel 8086 is a 16-bit microprocessor, this means that operands typically are 16 bit quantities. It is, however, possible to address 8 bit quantities (bytes). The total address space is 1 Mbyte. The Intel 8086 is described in further detail in [Intel 1979].

The RAM store.

The RAM store on a board can be accessed by the microprocessor on that board, or by any other processor in the system via the Multibus. All accesses to the RAM store are made through a so-called "dual port" which is under the control of either the microprocessor on the same board or the Multibus, see fig. 2. Since the RAM store is placed on the same board as the processor it is called the local store (Section 4 describes a store which is common to all processors).

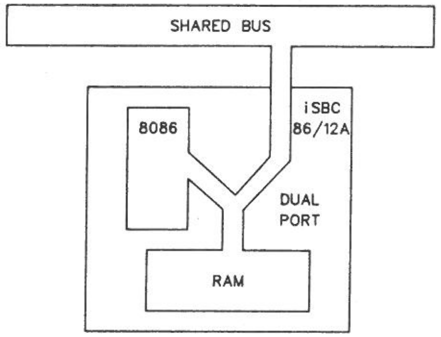


Fig. 2 The dual port RAM.

There are 32 K bytes of RAM store on each processor board. These are accessed locally by the (hexadecimal) addresses: 00000 - 07FFF. However, when accessed from the Multibus the addresses of all the RAM stores are different. This makes it possible to distinguish between the stores on all the boards. The (hexadecimal) addresses of the local stores are:

10000 - 17FFF	store on board 1
20000 - 27FFF	store on board 2
.	
.	
.	
90000 - 97FFF	store on board 9

So all RAM store is accessible by all processors. Note that the addresses of a RAM store are different when accessed on-board and off-board.

The EPROM store.

Each board has 8 K bytes of EPROM store. This store is only accessible by the on-board processor. It is accessible through the addresses: FE000 - FFFFF. When the power is turned on or when the system is reset each processor starts executing the instruction placed in address FFFF0 of the EPROM store. A program (called the "debugger") is placed at this address. This program initializes the board and makes it possible to bootstrap other programs (see chapter 4, section 5).

The timers.

Each of the boards has two 16 bit counters which can be used as timers by the microprocessor on the same board. The two counters are called "TMR0" and "TMR1". TMR0 is decremented with a frequency of 1.23 Mhz i.e. every 0.8 microsecond. TMR1 is decremented every time TMR0 gets to 0. The values of TMR0 and TMR1 can be changed and read by I/O instructions. Below is shown a PL/M-86 library package to manipulate the two timers.

```
timer:

do;
declare  tmr$control  literally    '0d6H',
         tmr0         literally    '0d0H',
         tmr1         literally    '0d2H',
         tmr0$mode    literally    '034H',
         tmr1$mode    literally    '074H',
         tmr0$count   literally   '12288',
         tmr1$count   literally   '30000',
         tmr0$latch   literally    '000H',
         tmr1$latch   literally    '040H';

declare  s    word,
         b(2) byte  at (@s );
```

```
init$tmr:

procedure public;

   output( tmr$control ) = tmr0$mode;
   output( tmr$control ) = tmr1$mode;

   output( tmr1 ) =  low ( tmr1$count );
   output( tmr1 ) = high ( tmr1$count );

   output( tmr0 ) =  low ( tmr0$count );
   output( tmr0 ) = high ( tmr0$count );

   call time( 1000 );

end init$tmr;


read$tmr0:

procedure integer public;

   output( tmr$control ) = tmr0$latch;
   b( 0 ) = input( tmr0 );
   b( 1 ) = input( tmr0 );
   return  tmr0$count - signed( s );

end read$tmr0;


read$tmr1:

procedure integer public;

   output( tmr$control ) = tmr1$latch;
   b( 0 ) = input( tmr1 );
   b( 1 ) = input( tmr1 );
   return  tmr1$count - signed( s );

end read$tmr1;

end timer;
```

Input/Output.

Each board has a serial I/O port. This port can be used by the microprocessor on the board through its I/O instructions. Below is shown a PL/M-86 library package to manipulate the I/O port.

```
io86:

do;
declare     statport    literally   '0daH',
            crtcmd      literally   '037H',
            crtmode     literally   '04eH',
            dataport    literally   '0d8H',
            pitport     literally   '0d6H',
            pitc2m3     literally   '0b6H',
            pitctr2     literally   '0d4H',
            reset       literally   '040H',
            txready     literally    '01H',
            rxready     literally    '02H',
            topch       literally   '07fH';

co:

procedure( c ) public;
   declare  c  byte;

   do while ( input( statport ) and txready ) = 0;
   end;

   output( dataport ) = c;

end co;


ci:

procedure byte public;

   do while ( input( statport ) and rxready ) = 0;
   end;

   return  input( dataport ) and topch;

end ci;
```

```
init$io:

procedure( baud ) public;
   declare baud  word;

   declare brm   word,
           delay byte;

   baud = baud / 100;
   brm  =  768 / baud;

   /* Initialization sequence; from Intel: iSBC957 */

   output( statport ) = 0;   delay = 0;
   output( statport ) = 0;   delay = 0;
   output( statport ) = 0;   delay = 0;

   output( statport ) = reset;  delay = 0;

   output( statport ) = crtmode;
   output( pitport )  = pitc2m3;

   output( statport ) = crtcmd;
   output( pitctr2 )  =  low( brm );
   output( pitctr2 )  = high( brm );

   call time( 1000 );  /* wait 100 msec. */

end init$io;

end io86;
```

Section 3: The arbiter.

The arbiter grants exclusive access to the Multibus. This provides the processors with indivisible operations for manipulating the common store.

Each processor has two control lines, BPRQ and BPRN, which are used to gain access to the Multibus. The signal BPRQ (bus request) from a processor indicates that it wants to use the bus and BPRN (bus priority) to a processor indicates that the processor may go ahead and use the bus. Hence all processors use the following algorithm when referring to the common store:

```
      .
      .
      .
   BPRQ := true;
   while not BPRN do;
    " use bus / common store "
   BPRQ := false;
      .
      .
      .
```

The arbiter is connected to processor no. i through two wires carrying the two signals BPRQ[i] and BPRN[i]. The arbiter executes the following algorithm:

```
   cycle
      repeat
         i := i mod n + 1
      until BPRQ[i];
      BPRN[i] := true;
      while BPRQ[i] do;
      BPRN[i] := false
   end
```

Each iteration of the inside repeat loop takes 100 nsec. No processor uses the bus for more than a single instruction, so we get:

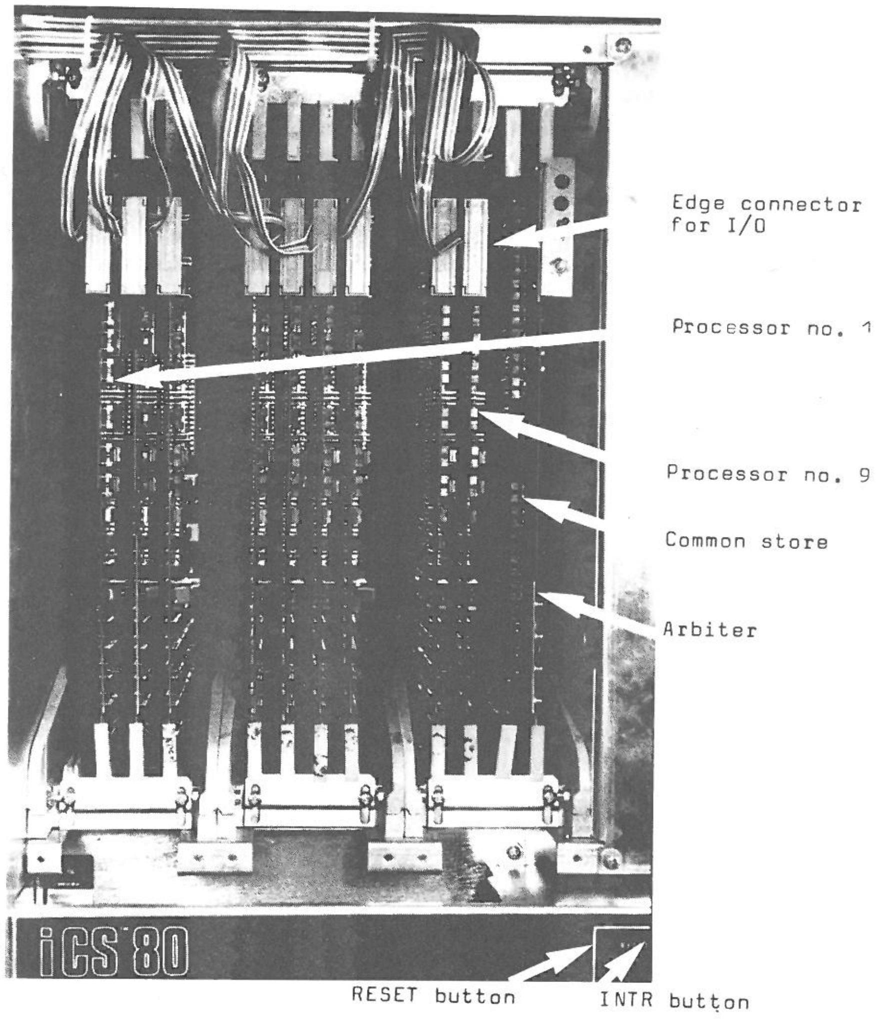
```
   minimal waiting time for bus:            100 nsec.

   maximal waiting time for bus,
      nobody else requests the bus:         900 nsec.
      all nine processors request the bus:    7 microsec.
```

Section 4: The common store.

The common store is an Intel iSBC 032 memory board. It contains 32 K bytes and it can only be accessed from the Multibus. The (hexadecimal) addresses of the common store are: B0000 - B7FFF.

Section 5: The chassis.

The processor, common store and arbiter boards are mounted in an Intel iCS 80 chassis as shown on the photograph below.
The chassis is mounted together with power supplies, connectors for I/O and cooling system, in a rack as shown on the front page photograph.
The chassis has twelve slots. Each slot can hold one board. These slots are mounted on a backplane, which contains the data


Edge connector for I/O
Processor no. 1
Processor no. 9
Common store
Arbiter
iCS 80
RESET button
INTR button

The chassis

and control lines making up the Multibus.
All twelve slots are equivalent except with regard to the arbiter and the power supplies:

There are two power supplies, one feeds slots 1 to 6 and the other feeds slots 7 to 12. The power lines of the Multibus are disconnected between slots 6 and 7. A power supply can feed at most five processor boards.

Additional wiring is mounted on the backplane in order to provide the signals BPRQ[1:9] and BPRN[1:9] (see section 3). This wiring assumes that the arbiter occupies slot 12, and that the processors occupy slots 2 to 10.

I/O lines.

The serial I/O port on each board is led via an edge connector to a standard RS232 connection on the left of the rack.

INTR and RESET buttons.

The two manually operated controls INTR and RESET work as follows:
When INTR is depressed all processors recieve a non-maskable interrupt (NMI).
When RESET is depressed all processors are reinitialized (see chapter 4, section 5).

Chapter 4:

The Concurrent Pascal implementation.

Section 1: Introduction.

The machine is used for implementation and analysis of algorithms utilizing concurrency. It is therefore necessary to have a high level language with multi-programming constructs available on the machine.

None of the existing languages were ideal for this purpose, but we found that Concurrent Pascal was the best compromise, primarily because of the high quality of its implementation. Further justification can be found in [Søe and Søgaard 1981].

It was important to finish the implementation as quickly as possible, so we could get on with the programming experiments. The absolute speed of the machine was only of secondary importance; if necessary the implementation could be optimized later.

The central decision in the present Concurrent Pascal implementation was to allocate one processor to each process. This leads to an extremely simple implementation which has several advantages: (listed in order of priority)

1. Analyzing a program running on a complex implementation can be very hard, because it is difficult to distinguish the properties of the implementation from those of the program.

2. It can be very hard to convince oneself of the correctness of a complex implementation. Testing is extremely difficult when processors run concurrently.

3. A simple implementation can be completed rather quickly.

Many of the details of the implementation reported below can be explained by the decision to allocate one processor for each process.

As described in chapter 3, all processors share a common store, which can only be accessed through the shared bus, the Multibus. This bus can easily become a bottleneck if all processors are using it frequently. An effort was therefore made to reduce the use of the Multibus as much as possible.

Section 2: Program structure.

A Concurrent Pascal program consists of a number of definitions (constants and types) and an initial process where all processes and shared data structures are declared and initialized.

The following is a typical structure of a Concurrent Pascal program:

```
const
  c = ... ;
type
  ml = monitor
         ...
       end;
  cl = class
         ...
       end;
  p2 = process
         ...
       end;
  p3 = process
         ...
       end;

  .
  .
  .

  m2 = monitor
         ...
       end;
  p9 = process
         ...
       end;

"initial process"
var
  v: ...;
begin
  init v(  );
  ...
end.
```

The store references of one processor executing a Concurrent Pascal process can be divided into the following three categories:

a) references to code,
b) references to local data,
c) references to common data.

The local data (i.e. process and routine variables) are used by one process only. The common data (i.e. global monitor variables) are used by more than one process.

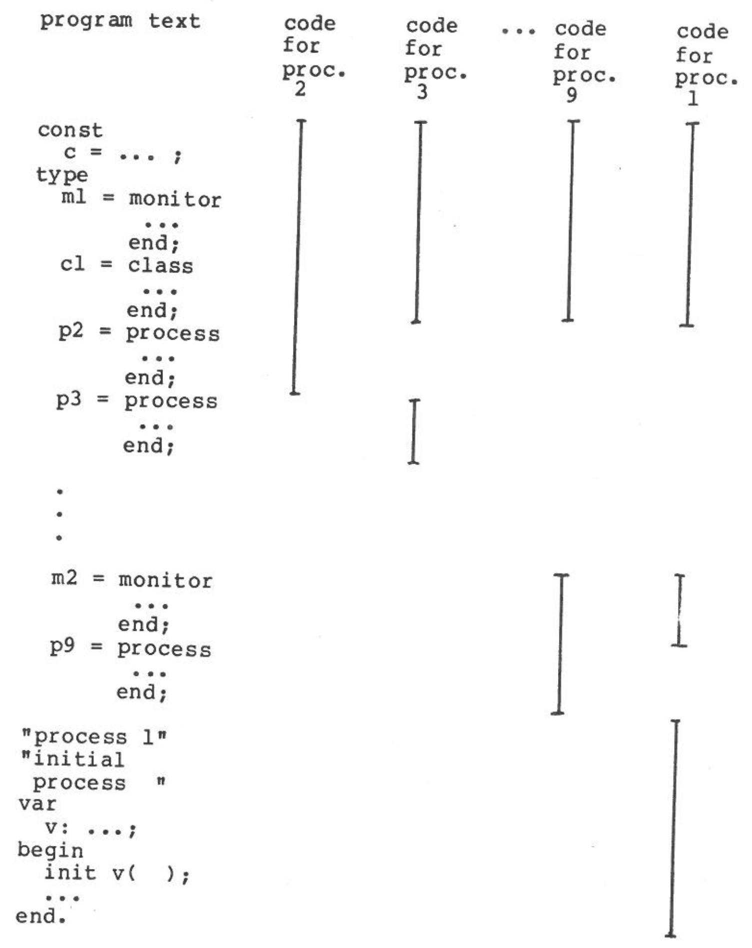
The exact distribution of store references into these three categories may vary, but [Jones 1980] and [Ougaard 1978] have reported the following distribution:

a) references to code > 50 %,
b) references to local data < 40 %,
c) references to common data < 10 %.

We have not yet made any measurements of this distribution on the Multi-Maren machine, but it is our impression that category c is significantly smaller.
Based on this observation, it was decided only to place common data (i.e. global monitor variables) in the common store. The distinction between local and global monitor variables is explained by the following skeleton of a monitor:

```
type m =
monitor(s: ...);
  var                         }
    g1: ...;                  } global monitor variables
    g2: ...;                  }
    ...                       }

  procedure entry e(p: ...);
    var                       }
      l1: ...;                } local routine variables
      l2: ...;                }
  begin                       }
    Se;                       } routine body
  end;                        }

  ...

begin                         }
  Sg;                         } monitor body
end;                          }
```

A consequence of this decision is that all code for the monitor body and routines is duplicated in each of the processors. Each has its own copy to avoid using the Multibus for fetching instructions. The following picture shows how the compiler distributes the code of a typical Concurrent Pascal program:



```
const
  c = ... ;
type
  m1 = monitor
        ...
       end;
  c1 = class
        ...
       end;
  p2 = process
        ...
       end;
  p3 = process
        ...
       end;

  .
  .
  .

  m2 = monitor
        ...
       end;
  p9 = process
        ...
       end;
"process 1"
"initial
 process  "
var
  v: ...;
begin
  init v(  );
  ...
end.
```

In addition to the code, the local store of each processor contains the global variables of the process and the parameters and local variables of the routines (including monitor routines).

Section 3: The interpreter.

Concurrent Pascal is implemented by means of an interpreter. The interpreter executes C-code instructions as generated by the compiler (see section 7). The C-code is described in more

detail by Hartmann[1977]. It has been necessary to modify a few of the C-code instructions to make it possible to utilize the full 20-bit address range of the hardware.
The interpreter has been programmed in PL/M-86, and the data type "real" is handled by the standard real emulator[Intel 1978a]. In a future version we hope to replace this emulator by the Intel 8087 numerical co-processor. Each processor has its own copy of the interpreter and the real emulator. By implementing Concurrent Pascal by an interpreter written in PL/M-86, the absolute speed of the machine has been reduced significantly (at least by an order of magnitude). The advantage on the other hand is the short time it took to do the implementation (see chapter 5).

Section 4: Synchronization.

There are two levels of synchronization in Concurrent Pascal. The lowest level, called short-term synchronization, ensures that at most one process at a time is executing one of the entry routines of a particular monitor. This is implemented by a so-called gate. The second level of synchronization, called medium-term synchronization, is used to express the logical synchronization constraints of the program, e.g. that no more elements can be put into a buffer which is full. The Concurrent Pascal construct for expressing this is called a queue.
The implementation of gates and queues is based on the assumption that the machine provides an indivisible exchange operation, :=:, which exchanges the contents of a variable located in a local store with a variable contained in the common store. An implementation of gates is described below as a Concurrent Pascal class. Entry to a monitor routine is done by calling "entermon", and exit from the monitor is done by calling "exitmon".

Short-term synchronization.

A correct solution must satisfy the following two requirements:

1) at any time at most one process may have access to a particular monitor (mutual exclusion),
2) if no process is using a monitor, a request to access the monitor must be granted (responsiveness).

Note, that requirement 2, responsiveness, is weaker than the fairness requirement made in the Concurrent Pascal report [Brinch Hansen 1977]. This is discussed below.
Consider the following implementation, which satisfies 1 and 2:

```
type
  gate-state = (open, closed);

  gate =
  class
    var
      g: gate-state;       "allocated in common store"

    procedure entry entermon;
      var
        help: gate-state; "allocated in local store"
    begin
      help:= closed;
      repeat
        g :=: help;
      until help=open;
    end;

    procedure entry exitmon;
    begin
      g:= open;
    end;

  begin
    g:= open;
  end;
```

This implementation is inefficient because the busy waiting loop in the routine entermon puts a heavy load on the shared bus. Furthermore, this load will slow down all processors currently using some monitor. Hence, the higher the demand is on a monitor, the slower it gets executed; this is clearly not acceptable.

The load on the shared bus can be decreased by prolonging the cycle time of the busy waiting loop.

```
repeat
  wait;
  g :=: help;
until help=open;
```

Ideally the wait should not burden the shared bus and it should last exactly until the gate becomes open, no longer and no shorter. It is, however, obvious that the correctness of the solution is not affected by making the wait shorter than this ideal. There are many ways of implementing the wait, here we discuss a family of implementations where the waiting ceases when a gate change from closed to open. In this family the operation "exitmon" becomes:

```
procedure entry exitmon;
begin
  g:= open;
  indicate-state-change;
end;
```

We will now analyze what requirements a realization of "wait" and indicate-state-change" must fulfil. This analysis is not as short and convincing as we would have liked it to be, but currently we cannot do better. It turns out that essential properties of the above algorithms is the order of the statements and that one statement e.g. "wait" is completely finished before the next "g :=: help" is initiated. What follows is an explanation of why these two properties are essential.

First, note that mutual exclusion is satisfied by the first algorithm of this section, the realization of "wait" and "indicate-state-change" cannot destroy this.

Secondly, the order of the statements in both "exitmon" and "entermon" is important for responsiveness. _The order of the statements may not be altered by any realization_ (restriction 1).

Thirdly, if all indications of state changes are separated enough, so that all processors can finish their wait, test the corresponding gate and return to their waiting state, before the next indication of a state change appears, there would not be any problems in realizing the algorithm. But two or more indications may appear so close together (seen from the waiting process) that they are indistinguishable. This may for example happen when the waiting process is very very slow or if several processors give indications almost simultaneously. Thus the "wait" consists of at least two steps:

```
cycle
  .
  .
  "ready to receive indication"
  await-indication;
  enable-indication;
  "ready to receive indication"
  .
  .
  .
end;
```

The second step "enable-indication" may consist of sending another signal, resetting variables or the like. _This enabling must be completely finished before the succeding statement (testing the gate) is initiated_ (restriction 2).

Currently, the "indicate-state-change" and "wait" statements are realized as follows: In the local store of each processor p is allocated a boolean variable try[p]. When a state change from closed to open occurs on any gate this boolean is set to true in all processors.

```
wait:

repeat until try[p];        "await indication"
try[p]:= false;             "enable indication"


indicate-state-change:

for i in processors do
  try[i]:= true;
```

The complete implementation is as follows:

```
gate =
class
  var
    g: gate-state;       "allocated in common store"
    try: array[processors] of boolean;
                         "allocated in local stores"
  procedure entry entermon(p: processors);
    var
      help: gate-state; "allocated in local store"
  begin
    "assume try[p] is true"
    help:= closed;
    repeat
      repeat until try[p];
      try[p]:= false;
      g :=: help;
    until help=open;
    try[p]:= true;
  end;

  procedure entry exitmon;
    var
      i: processors;
  begin
    g:= open;
    for i in processors do
      try[i]:= true;
  end;

begin
  g:= open;
  for i in processors do
    try[i]:= true;
end;
```

This is exactly the synchronization algorithm used in the

Concurrent Pascal kernel, except that the kernel is programmed in PL/M-86.

Medium-term synchronization.

A process may await a signal in a monitor by calling "delay" on a variable of type "queue". This releases the exclusive access to the monitor so that other processes can access the monitor. When another process calls "continue" on the same variable of type "queue", this process is forced to leave the monitor and the delayed process resumes its execution. The C-code instructions "delay" and "continue" are implemented by the following algorithm which will not be described in further detail here:

```
queue =
class(g: gate); "g is the gate of the monitor
                 containing the queue-variable."
  var
    q: (noone, delayed, continued);

  procedure entry delay(p: processors);
  begin
    "assume try[p] true"
    if q = noone
       then
       begin
         q:= delayed;
         g.exitmon;
         repeat
           repeat until try[p];
           try[p]:= false;
         until q=continued;
         try[p]:= true;
         q:= noone;
       end
       else exception;
  end;
```

```
          procedure entry continue;
            var
              i: processors;
          begin
            if q=delayed
               then
               begin
                 q:= continued;
                 for i in processors do
                   try[i]:= true;
               end
               else g.exitmon;
          end;

          function entry empty: boolean;
          begin
            empty:= (q=noone);
          end;

        begin
          q:= noone;
        end;
```

Fairness.

The short-term synchronization algorithm given above is not fair. We are not convinced that fairness is a reasonable requirement for a multi-processor. This is clearly a viewpoint which can be criticized, but

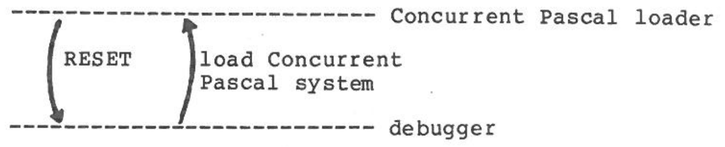
1) a fair synchronization algorithm would be more complicated;
2) fairness is not necessary for our applications (e.g. the program described in chapter 2);
3) programs where fairness is essential are probably ill suited for Multi-Maren anyway. If fairness is essential some processors are always waiting for access to a monitor. This implies a bad utilization of one or more processors and the algorithm should be revised.

We do, however, admit that our experience and intuition about this issue is very limited, so further experiments might force us to change our view on fairness.

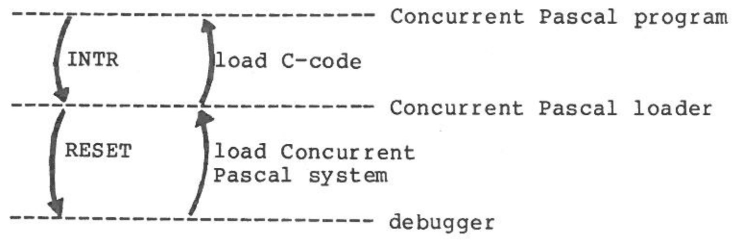
Section 5: Initialization.

This section describes how a Concurrent Pascal program is loaded and started.
When the power is turned on or when the machine is reset, a program (placed in the EPROM store) called the debugger is started. The debugger is similar to the "iSBC 957" debugger

supplied by Intel[Intel 1978b]. Among other facilities, this program can load and start the Concurrent Pascal loader. The debugger and the Concurrent Pascal loader implement the following two levels of virtual machines:



When the Concurrent Pascal loader is running, all processors have received their copy of the Concurrent Pascal interpreter and kernel, but they have not yet received the C-code (the program to be executed). This is loaded and distributed by the Concurrent Pascal loader. This adds a new virtual machine level:



The Concurrent Pascal loader is a PL/M-86 program running on processor number 1. It receives the C-code from the Intellec development system and distributes it to the relevant processors. When the processors have received their part of the C-code, they start executing it, but only until they reach the C-code instructions corresponding to the first begin in the process body (the C-code instruction "initproc"). At this instruction they are stopped until explicitly started by an "init-statement" in the initial process:

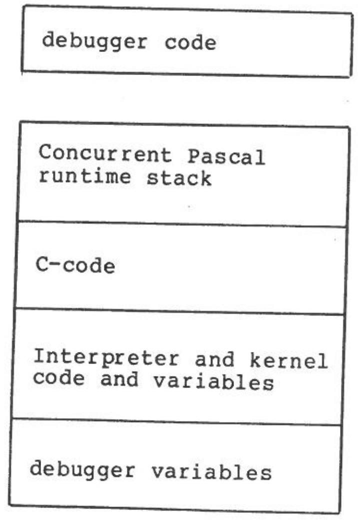
```
initial process    process 2   ...    process 9

var                    .                  .
  p2: ...              .                  .
  ...                  .                  .
  p9: ...              .                  .
                       .                  .
begin                  .                  .
  init p9; ------------------------> begin
  ...                  .
  init p2; ------> begin
  ...

end.
```

The states of all the processors are described in a table which has an entry for each process. An entry has one of the following values:

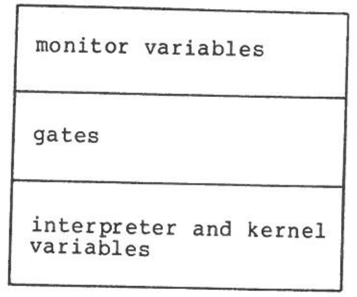
not started:	The interpreter has not yet been started.
wait initproc:	Waiting for init-statement.
running:	Executing C-code instructions.
terminated:	The processor has executed the last "end" of its body.
delayed:	Delayed in a queue.
error xxxxx:	An exception has occured, xxxxx indicates which.

The contents of this table is displayed by processor 1 either when the INTR button is depressed or when the initial process terminates. Depressing INTR brings the system back on the loader level where a new program may be loaded and started.

Immediatly after starting a Concurrent Pascal program, the contents of all the processors' local stores are as follows:

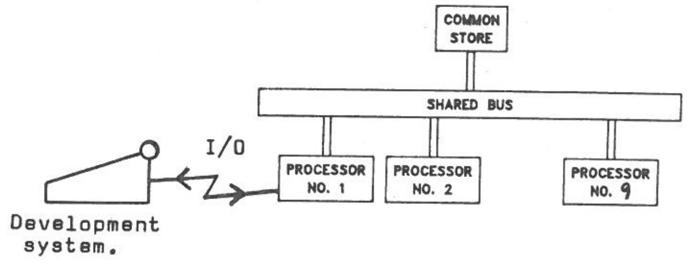

debugger code
Concurrent Pascal runtime stack
C-code
Interpreter and kernel code and variables
debugger variables

The contents of the common store is as follows:



Section 6: Input/Output.

There is only one Input/Output device available for each processor, a terminal. Each processor has its own Input/Output port which is a standard RS 232 connection. Therefore, up to nine terminals may be connected and used independently. Usually only one terminal is connected (to processor 1), this terminal is actually an Intel "Intellec" development system.



The normal I/O connection.

<u>Section</u> 7: The compiler.

The compiler runs on a PDP-10 system which is connected to the Intellec development system. After compilation the C-code must therefore be transmitted to the Intellec system.
The compiler is a modified version of the Concurrent Pascal compiler described by Hartmann[1977]. The modifications can be divided into three categories:

1) rewriting the compiler into the Pascal dialect used on the PDP-10,
2) introducing the code duplication described above,
3) correcting some errors.

<u>Section</u> 8: Additional material.

The details of how to use the various parts of the Multi-Maren system are described in two user's guides. Below is given a short summary of each.

Concurrent Pascal at DAIMI. User's Guide.

This is the user's guide to those parts of the Concurrent Pascal system which run on the PDP-10: the compiler, a PDP-10 C-code interpreter, and the disassembler. Furthermore, the restrictions in Concurrent Pascal are listed.

Running Concurrent Pascal on the Multi-Maren System.

This is a description of how a Concurrent Pascal program is transmitted to and executed on Multi-Maren.

Finally there are a number of internal reports describing various auxiliary programs which are also used for other purposes e.g. the PDP-10/Intellec transmission program and the Multi-Maren debugger mentioned in section 5.

Chapter 5:

A history of the project.

In January 1980 a pilot study was initiated. A small number of microprocessors were investigated with regard to:
- price,
- capacity (primarily cycle time and storage),
- connection possibilities (e.g. common store, bus etc.),
- connection of peripherals,
- suitability for supporting a Pascal like language (Concurrent Pascal, Modula, or Platon).

under the limitation that all hardware had to be commercially available. This pilot study and its results are described in [Søe and Søgaard 1981]. The study provided us with a knowledge of the available hardware and its capabilities.

During the spring of 1980 the purpose of the laboratory was formulated more precisely: experiments with implementation and analysis of algorithms utilizing concurrency. Having this purpose in mind the major design goals were laid down. This led us to choose a hardware system with a common bus-structure and to choose Concurrent Pascal as the programming language. These decisions where made during the summer of 1980. At that point we asked all relevant manufactures to demonstrate to us a system with:
- a 16-bit processor board connected to
- a 16-bit wide bus (which should not be affected by local computations on the processor board) and
- some kind of connection to either a main frame host or a cheap development system.

It turned out that the only manufacturer which was able to fulfil these requirements was Intel. Since there were no important deficiencies with the Intel system, we chose the Intel 8086 as the cornerstone of the multiprocessor. On September 18, 1980 the following equipment was delivered:

- two Intel iSBC 86/12A processor boards,
- an Intel iCS 80 chassis with power supply,
- an Intellec 230 development system.

During the fall, work was started on the Concurrent Pascal implementation and a communication channel between the Computer Science Department's PDP-10 and the Intellec system was established. This was done by writing a program which made the Intellec appear as a terminal to the PDP-10. Using this, files could be transmitted from the development system to the PDP-10 file system (the speed was 2400 baud).

In December 1980 the basic control structures and arithmetic operations of Concurrent Pascal were implemented, so that an initial process without classes, monitors and other processes could be executed. Two months later, all the modifications in the compiler and interpreter due to the distribution of object code were completed. Then followed the implementation of the synchronization primitives and the initialization of a program with several processes and monitors. Independently of this the type real was implemented.

The entire implementation was finished around May 1, 1981. At that point all language constructs (with the exceptions mentioned in "Concurrent Pascal at DAIMI. User's Guide.") were implemented, but still the system had only two processors.

The initial configuration of the system with only two processors was sufficient to test the Concurrent Pascal implementation, but it was clearly not sufficient for the real purpose of the project: "experiments with multiprogams". When we became confident that the Concurrent Pascal implementation would work we ordered seven more processors (iSBC Intel 86/12A boards) and a 32K byte memory board. These were delivered in May and June. This expansion required two hardware changes:

1) the bus arbitration was changed;
2) the power supply was changed. It actually became two power supplies each feeding half of the boards.

These hardware changes were completed in June 1981. During the late Spring and Summer, processor boards were gradually added to the system. This revealed a few hardware problems (malfunctioning CPU's and EPROM sockets) and some shortcomings in the user interface. After correcting these, the entire system became available to students and faculty in August 1981.

During the implementation period parts of the system were used for preliminary experiments. In the Spring of 1981 the first graduate course using the system was offered. This was an introduction to the system and to PL/M-86. Each participant wrote a multi-program in PL/M-86 and made some speed measurements on this program.

A similar course was offered in the Fall Semester of 1981, but this time the multi-programs were written in Concurrent Pascal.

References:

[Brinch Hansen 1977]: The Architecture of Concurrent Programs, Per Brinch Hansen, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.

[Hartmann 1977]: A Concurrent Pascal Compiler for Minicomputers, Alfred C Hartmann, Lecture Notes in Computer Science, Vol. 50, Springer Verlag 1977.

[Jones 1980]: The Cm* Multiprocessor Project: A Research Review, A. K. Jones and E. F. Gehringer (editors), Computer Science Department, Carnegie-Mellon University, July 1980.

[Intel 1978a]: PL/M-86 Programming Manual for the 8080/8085-based development System, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051, U.S.A.

[Intel 1978b]: iSBC 957 Intellec - iSBC 86/12 Interface and Execution Package, User's Guide, Intel Corporation.

[Intel 1978c]: iSBC 86/12 Single Board Computer Hardware Reference Manual, Intel Corporation.

[Intel 1979]: The 8086 Family User's Manual, Intel Corporation.

[Ougaard 1978]: Multiprocessorer til dedikerede systemer skrevet i Concurrent Pascal, Uffe Ougaard, Datalogisk Institut, University of Copenhagen, December 1978.

[Rice 1976]:	<u>Parallel Algorithms for Adaptive Quadrature III, Program Correctness</u>, John R. Rice, ACM Transactions on Mathmatical Software, vol. 2, no. 1, March 1976, p. 1 - 30.
[Søe and Søgaard 1981]:	<u>Undersøgelse af 16-bits mikroprocessorkort</u>, Kurt Søe and Jens Søgaard Andersen, Computer Science Department, University of Aarhus, Denmark, October. 1981.