

L.1 MIMD Programming Languages

- Pascal derivatives
 - Concurrent Pascal (Brinch-Hansen, 1977)
 - Ada (US Dept. of Defense, 1975)
 - Modula-P (Bräunl, 1986)
- C/C++ plus parallel libraries
 - Sequent C (Sequent, 1988)
 - pthreads
 - PVM “parallel virtual machine” (Sunderam et al., 1990)
 - MPI “message passing interface” (based on CVS, MPI Forum, 1995)
- Special languages
 - CSP “Communicating Sequential Processes” (Hoare, 1978)
 - Occam (based on CSP, inmos Ltd. 1984)
 - Linda (Carriero, Gelernter, 1986)

PVM (Parallel Virtual Machine)

Sunderam, Dongerra, Geist, Manchek (Knoxville Tennessee), 1990

Parallel processing library for C, C++ or FORTRAN

- Programming of algorithm in known language
- Insertion of synchronization and communication via library functions (e.g. process start, data exchange)

Implemented for:

Unix Workstations, Windows PCs, Cray YMP and C90, IBM 3090, Intel iPSC, Intel Paragon, Kendall Square KSR-1, Sequent Symmetry, Thinking Machines CM-2, CM-5

Parallel processing tool for:

- MIMD-Systems with shared memory
- MIMD-Systems without shared memory
- individual workstations (SISD)
- heterogeneous net (Cluster) of workstations

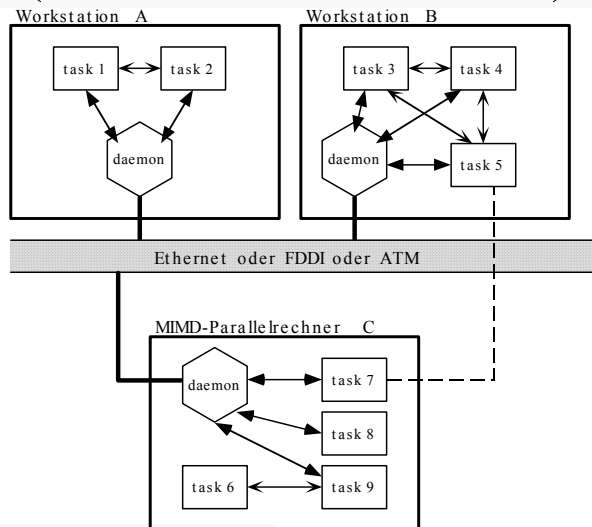
Similar systems:

p4, Express, PARMACS, SPPL

Standardization:

MPI (message passing interface)

PVM (Parallel Virtual Machine)



PVM (Parallel Virtual Machine)

Methodology:

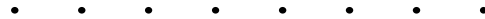
- Declaration of computers to be used
Creation of text file `pvm_hosts`
- Start of a PVM-Daemon on each computer used
Unix-command: `$ pvm pvm_hosts`
- Start of processes (**tasks**) on certain computers
C-Routine: `pvm_spawn`
- Data send (String):
`pvm_packstr`
`pvm_send`
- Data receive (String):
`pvm_recv`
`pvm_upkstr`



PVM (Parallel Virtual Machine)

Methodology:

- Pack- and Unpack-Routines for all data types
- Termination of a PVM-Task
C-Routine: `pvm_exit`
- Termination of the entire PVM-System
(termination of all PVM-Tasks and all daemons)
Unix-commando: `$ pvm`
`> halt`



PVM (Parallel Virtual Machine)

"Hello World" (according to Manchek):

Programm hello.c

```
#include <pvm3.h>
main()
{
  int tid
  char reply[30];

  printf("I am task %x\n", pvm_mytid());
  if (pvm_spawn("hello2", (char**) 0, 0, "", 1, &tid) == 1 {
    pvm_recv(-1, -1)
    pvm_buinfo(pvm_getrbuf(), (int*) 0, (int*) 0, &tid);
    pvm_upkstr(reply);
    printf("message from task %x : %s\n", tid, reply);
  } else
    printf("cannot start hello2\n");
  pvm_exit();
}
```

Programm hello2.c

```
#include <pvm3.h>
main()
{
  int ptid
  char buf[100];

  ptid = pvm_parent();
  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, 1);
  pvm_exit();
}
```



PVM (Parallel Virtual Machine)

Process control:

```
int cc = pvm_spawn(char *aout, char **argv, int flag, char *where, int cnt, int *tids)
int cc = pvm_kill(int tid)
void pvm_exit()
int tid = pvm_mytid()
int tid = pvm_parent()
int cc = pvm_config (int *host, int *arch, struct hostinfo **hostp)
int cc = pvm_addhosts (char **hosts, int cntm int *st)
int cc = pvm_delhosts (char **hosts, int cntm int *st)
```

Groups of processes (for broadcast and barrier-synchronization):

```
int inum = pvm_ingroup(char *group)
int cc = pvm_lvgroup (char *group)
int cc = pvm_bcast (char *group, int msgtag)
int cc = pvm_barrier (char *group, int cnt)
```



PVM (Parallel Virtual Machine)

Data exchange:

```
int buf = pvm_initsend(int encoding)
int cc = pvm_pkstr (char *cp)
int cc = pvm_upkstr(char *cp)
int cc = pvm_pkint (int *np, int cnt, int std)
int cc = pvm_upkint(int *np, int cnt, int std)
analogue routines for: pkbyte, pkcplx, pkdcplx, pkdouble, pkfloat, pklong, pkshort
int buf = pvm_send(int tid, int msgtag)
int buf = recv (int tid, int msgtag)
int buf = nrecv(int tid, int msgtag)
int cc = pvm_buinfo(int buf, int *len, int *msgtag, int *tid)
```





PVM (Parallel Virtual Machine)

Summary:

- Public Domain Software
anonymous ftp: netlib2.cs.utk.edu
- Available for large number of machines
- Development continues
- Implemented via Unix-Sockets
- Uses threads, if processes are on the same computer (if operating system support exists)
- xpvm
graphical user interface under X-Windows, display of messages, debugging, based on tools tcl/tk
- HeNCE (Heterogeneous Network Computing Environment)
graphical user interface with extra options: graphical editor for tasks, programming via interface



CSP

C. A. R. Hoare, 1978

Parallel language constructs:

- $X:: m:=a; \dots$ Declaration of a process (with an allocation as the first instruction)
- all instructions are separated by semicolons
 - declaration of variables are allowed to alternate with instructions
 - each instruction can success or fail

skip Empty instruction

$[P1 \parallel P2]$ Start of parallel processes

terminal ? number Data received from process terminal

printer ! line Data sent to processes printer



CSP

$x=1 \rightarrow m:=a$ "guarded command"
{IF $x=1$ THEN $m:=a$ END}

The instruction to the right of the arrow can only execute if the Boolean condition (guard) returns TRUE.

$x=1; \text{term?}a \rightarrow m:=a$ Multiple AND-linked guard-expressions can be written with separating semicolons.
The last sub-expression can be a data-receive operation.



CSP

$[x=1 \rightarrow m:=a$
 $x=2 \rightarrow m:=b]$ "alternative command"
{IF $x=1$ THEN $m:=a$ ELSEIF $x=2$ THEN $m:=b$ END}

This is a select-instruction where each case requires a preceding condition (*guard*):

- Exactly one alternative with a true guard is selected and executed
- if more than one guard is true, one is selected at random – the other true guards must not have any effect, i.e. possible data-receive operations in those guards must not be executed.
- if no guard is true the *alternative command* fails (**Error messages**).

$*[x=1 \rightarrow m:=a$
 $x=2 \rightarrow m:=b]$

"repetitive command" (denoted by $*$)
The instruction sequence is iteratively repeated until the *alternative command* fails (until no guard is true).



⋮

CSP

`[(i:1..4) x=i → m:=i]`

"value range" (generic instruction)
This example is equivalent to:

`[x=1 → m:=1 x=2 → m:=2
x=3 → m:=3 x=4 → m:=4]`

`X(i:1..3):: print!i`

"series of processes"
(multiple copies of a parametric process)

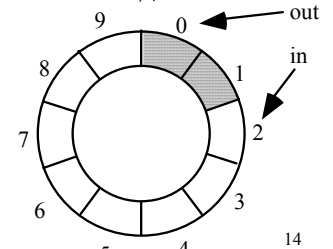
This example is equivalent to:
`X(1):: print!1 || X(2):: print!2 || X(3):: print!3`

⋮

Bounded Buffer Solution in CSP

Call by the Producer: `BB!p` Store new data
Call by the Consumer: `BB!more(); BB?p` Read new data

```
BB::
buffer : (0..9) dataset;
in, out: integer; in:=0; out:=0;
*[in < out+10; producer?buffer(in mod 10)
  → in := in+1
  consumer?more()
  → producer!buffer(out mod 10);
  out := out+1
]
```



⋮

Semaphore Implementation in CSP

- Disadvantage in CSP: Receiver-process must know name of sender-process
- Hence here the array of application-processes for semaphores: `X(i:1..100)`
- Call of semaphore-operations by an application process

`X: S!P();
S!V();`

```
S:: val: integer; val := 0;
*[(i:1..100)        X(i)?V() → val := val+1
  (i:1..100) val>0; X(i)?P() → val := val-1
]
```

⋮

Ada

US Department of Defense, 1975

Parallel language constructs:

task Process
entry process-entry, which can be called by another process
accept Wait for call of an entry by another process
select e.g. waiting for different entries
when "guarded commands"

Message passing in Ada: "Rendezvous-Concept"

- Sender and receiver are **blocked**, until data exchange is finished
- Exception: SELECT only tests if message can be received without blocking

Semaphore Solution in Ada

```
TASK BODY semaphore IS
BEGIN
  LOOP
    ACCEPT P
    ACCEPT V
  END LOOP;
END semaphore;
```

Call:

```
semaphore.P();
...
semaphore.V();
```

Restriction: Multiple V-Operations must not be sent in succession

Sequent Parallel C Library

Parallel Library functions (fork & join):

```
cpus_online()      Returns number of physical processors
m_set_procs (number) Sets number or required processors
m_fork (func, arg1,...,argn) Duplication and starting of a procedure on multiple processors (with
                             identical parameter values)
m_get_numprocs ()  Returns the total number of child (or sibling) processes that actually
                             started
m_get_myId ()      Returns the process number of a child process; or 0 for a parent process
m_kill_procs ()    Kills all child processes
                             (child processes finish in a busy-wait loop, and hence need to be
                             terminated explicitly)
```

Semaphor-Implementation:

```
s_init_lock (sema)  Initialisation of semaphors
s_lock (sema)      P-Operation
s_unlock (sema)    V-Operation
```

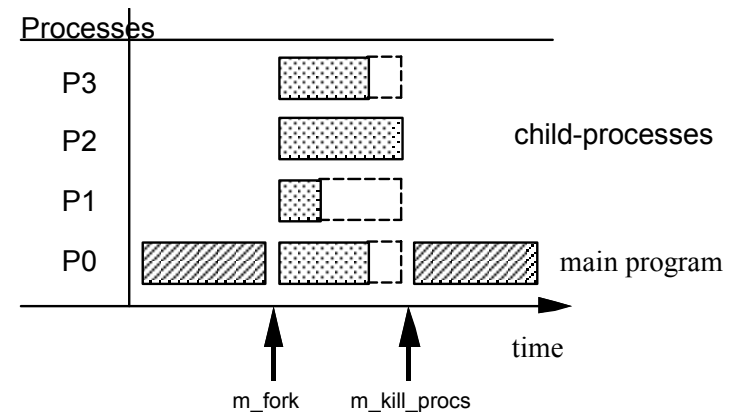
Sequent Parallel C Library

Example Program:

```
...
m_set_procs(3);      /* request of 3 more processors */
m_fork(parproc,a,b); /* start of the parallel child processes */
m_kill_procs();      /* deletion of child-processes after they terminate */
...

void parproc(a,b)    /* Parallel procedure of child processes */
float a,b;
{ ...
  n=m_get_numprocs(); /* check number of child processes */
  m=m_get_myId();     /* check own process number */
  ...
}
```

Sequent Parallel C Library



Sequent Parallel C Library

Handling of more jobs than processors available: **Iteration**

Example: Assume N iterations are necessary

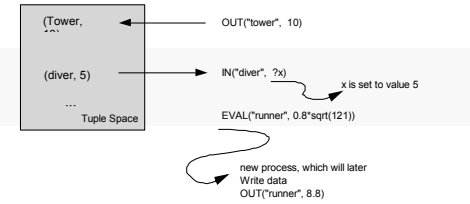
```
void parproc(a,b,c) /* parallele procedure */
{ int count,id,pe;
  pe=m_get_numprocs();
  id=m_get_myId()
  for (count=id; count<=N; count+=pe)
  { .../* actual calculation */ }
}
```

Assumption: Looping from 0 to N=20 is required, but only 6 processors are available.

- Processor 0 executes loop iterations: 0, 6, 12, 18.
- Processor 1 executes loop iterations: 1, 7, 13, 19.
- Processor 2 executes loop iterations: 2, 8, 14, 20.
- Processor 3 executes loop iterations: 3, 9, 15.
- Processor 4 executes loop iterations: 4, 10, 16.
- Processor 5 executes loop iterations: 5, 11, 17.

Linda

Carriero and Gelernter, 1986

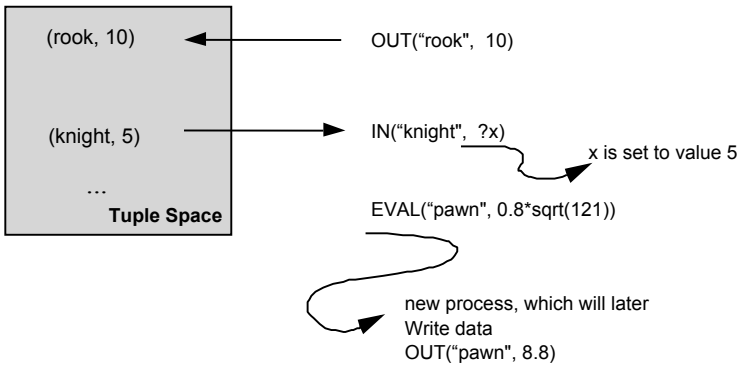


- Embedding into different programming languages possible, e.g.. C, Fortran, Modula-2, etc.
- Common data pool ("tuple space") for all processes ("active tuple")

Parallel Operations:

- OUT Creation of a data element ("passive tuple")
 - RD Read (without removal) of data element
Parts of the tuple may be pre- loaded with values; then only matching tuples are considered.
 - RDP Read-predicate (Boolean test operation on data in tuple space) test-only, no read
 - IN Read and remove of data element from tuple space (= RD+ delete)
 - INP Read-predicate (= RDP+ delete)
 - EVAL Start of a new process, which will later write its result via OUT into the tuple space.
- The program system terminates when no active tuples (processes) exist, or when all are blocked

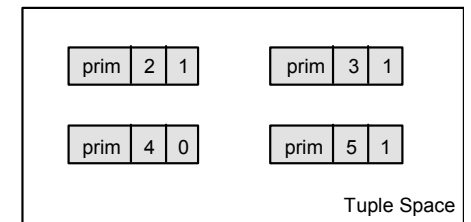
Linda



Prime Number Solution in Linda

```
lmain()
{ int i, ok;
  for (i=2; i<Limit; ++i) {
    EVAL("prim", i, is_prime(i));
  }
  for (i=2; i<=Limit; ++i) {
    RD("prim", i, ?ok);
    if (ok) printf("%d\n", i);
  }
}

is_prime(me)
int me;
{ int i, limit,ok;
  double sqrt();
  limit = sqrt((double) me);
  for (i=2; i<=limit; ++i) {
    RD("primes", i, ?ok);
    if (ok && (me%i == 0)) return 0;
  }
  return 1;
}
```



Modula-P

Bräunl, 1986

Module-Concept:

- Low level module
Interrupts and events (interrupt-service-routines), synch. via semaphores
- High level module
Explicit processes with exception management, synch. via monitors and conditions
- Processor module
For each computer node in a distributed system synch. via messages (RPC)

Example:

```
PROCESS abc(i: INTEGER);
BEGIN
  ... (* Instruction of the process *)
END PROCESS abc;

...

START(abc(1)); (* start process "abc" twice *)
START(abc(7)); (* but with different parameters *)
```

Bräunl 2004

25

Modula-P

Monitors with Conditions

```
MONITOR sync;
VAR c: CONDITION;

ENTRY read(i: INTEGER;
  VAR value: INTEGER);
BEGIN
  WHILE i<0 DO WAIT(c) END;
  ...
END read;

BEGIN (* Monitor-Initialization *)
  ...
END MONITOR sync;
```

Call: sync.read(10,w);

Semaphore

```
VAR sem: SEMAPHORE[1];
...
P(sem);
... (* critical instructions *)
V(sem);
```

Remote Procedure Call

```
COMMUNICATION pc05 (x,y: REAL;
  VAR result: REAL);
VAR r: REAL;
BEGIN
  r := x*x + y*y;
  result := SQRT(r);
END COMMUNICATION pc05;
```

Bräunl 2004

26

L.2 Coarse-Grained Parallel Algorithms

- Synchronization with Semaphores
- Synchronization with Monitor
- Pi calculation
- Distributed simulation

Bräunl 2004

27

Synchronization with Semaphores

```
1 PROCESSOR MODULE bb;
2 IMPLEMENTATION
3 IMPORT synch;
4
5 PROCESS Producer;
6 VAR i: INTEGER;
7 BEGIN
8   i:=0;
9   LOOP
10    i:=(i+1) MOD 10;
11    produce(i);
12  END
13 END PROCESS Producer;
14
15 PROCESS Consumer;
16 VAR i,quer: INTEGER;
17 BEGIN
18   quer:=0;
19   LOOP
20    consume(i);
21    quer:=(quer+i) MOD 10;
22  END
23 END PROCESS Consumer;
24
25 BEGIN
26   WriteString("Init Module");
27   WriteLn;
28   START(Producer);
29   START(Consumer);
30 END PROCESSOR MODULE bb.
```

Bräunl 2004

28

Synchronization with Semaphores

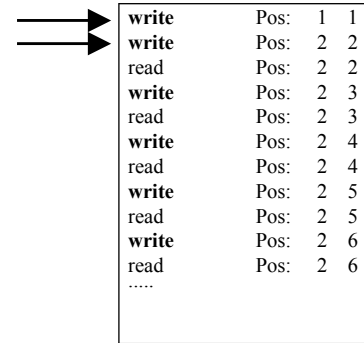
```

1  LOWLEVEL MODULE synch;
2  EXPORT
3  PROCEDURE produce (i: INTEGER);
4  PROCEDURE consume (VAR i: INTEGER);
5
6  IMPLEMENTATION
7
8  CONST n=5;
9  VAR buf: ARRAY [1..n] OF INTEGER;
10 pos,z: INTEGER;
11 Critical: SEMAPHORE[1];
12 Free: SEMAPHORE[n];
13 Used: SEMAPHORE[0];
14
15 PROCEDURE produce(i: INTEGER);
16 BEGIN
17 P(Free);
18 P(Critical);
19 IF pos>=n THEN WriteString("Err");
20 WriteLn; HALT;
21 END;
22 pos:=pos+1;
23 buf[pos]:=i;
24 (* *) WriteString("write Pos: ");
25 (* *) WriteInt(pos,5);
26 (* *) WriteInt(i,5); WriteLn;
27 V(Critical);
28 V(Used);
29 END produce;
30 PROCEDURE consume(VAR i: INTEGER);
31 BEGIN
32 P(Used);
33 P(Critical);
34 IF pos<=0 THEN WriteString("Err");
35 WriteLn; HALT;
36 END;
37 i:=buf[pos];
38 (* *) WriteString("read Pos: ");
39 (* *) WriteInt(pos,5);
40 (* *) WriteInt(i,5); WriteLn;
41 pos:=pos-1;
42 V(Critical);
43 V(Free);
44 END consume;
45
46 BEGIN
47 WriteString("Init Synch"); WriteLn;
48 pos:=0;
49 FOR z:= 1 TO n DO buf[z]:=0 END;
50 END LOWLEVEL MODULE synch.

```

Synchronization with Semaphores

Sample Run



Synchronization with Monitor

```

1  PROCESSOR MODULE hl;
2  IMPLEMENTATION
3  IMPORT msynch;
4
5  PROCESS Producer;
6  VAR i: INTEGER;
7  BEGIN
8  i:=0;
9  LOOP
10 i:=(i+1) MOD 10; (* prod *)
11 Puffer:write(i);
12 END
13 END PROCESS Producer;
14
15 PROCESS Consumer;
16 VAR i,quer: INTEGER;
17 BEGIN
18 quer:=0;
19 LOOP
20 Puffer:read(i);
21 quer:=(quer+i) MOD 10; (* cons *)
22 END
23 END PROCESS Consumer;
24
25 BEGIN
26 WriteString("Init hl"); WriteLn;
27 START(Producer);
28 START(Consumer);
29 END PROCESSOR MODULE hl.

```

Synchronization with Monitor

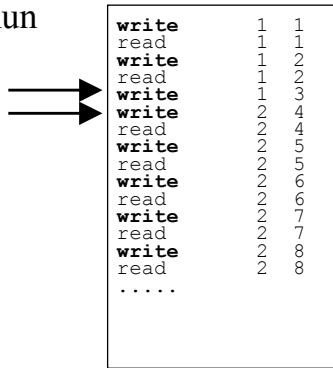
```

1  MODULE monitor_buffer;
2
3  EXPORT MONITOR Puffer;
4  ENTRY write(a: INTEGER);
5  ENTRY read (VAR a: INTEGER);
6
7  IMPLEMENTATION
8
9  MONITOR Puffer;
10 CONST max = 5;
11
12 VAR Stack: ARRAY[1..max] OF INTEGER;
13 Pointer: INTEGER;
14 Free, Used: CONDITION;
15
16 ENTRY write(a: INTEGER);
17 BEGIN
18 WHILE Pointer=max DO WAIT(Free) END;
19 inc(Pointer);
20 Stack[Pointer] := a;
21 IF Pointer=1 THEN SIGNAL(Used) END;
22 (* *) WriteString("write ");
23 (* *) WriteInt(Pointer,3);
24 (* *) WriteInt(a,3); WriteLn;
25 END write;
26 ENTRY read(VAR a: INTEGER);
27 BEGIN
28 WHILE Pointer=0 DO WAIT(Used) END;
29 a := Stack[Pointer];
30 (* *) WriteString("read ");
31 (* *) WriteInt(Pointer,3);
32 (* *) WriteInt(a,3); WriteLn;
33 dec(Pointer);
34 IF Pointer = max-1 THEN
35 SIGNAL(Free) END;
36 END read;
37
38 BEGIN (* Monitor-Initialisation *)
39 WriteString("Init M."); WriteLn;
40 Pointer:=0;
41 END MONITOR buffer;
42
43 BEGIN
44 WriteString("Init mb"); WriteLn;
45 END MODULE monitor_buffer.

```

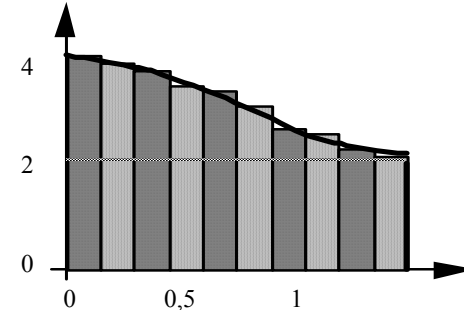
Synchronization with Monitor

Sample Run

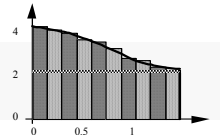


Pi Calculation

$$\pi = \int_0^1 \frac{4}{1+x^2} dx = \sum_{i=1}^{intervals} \frac{4}{1+((i-0.5)*width)^2} * width$$



Pi Calculation



```

1 PROCESSOR MODULE pi_calc;
2 IMPLEMENTATION
3 CONST intervals = 100; (* Number of sub intervals *)
4 width = 1.0 / FLOAT(intervals); (* Interval width *)
5 num_work = 5; (* Number of worker-processes *)
6
7 PROCEDURE f(x: REAL): REAL;
8 (* to integrating function *)
9 BEGIN
10 RETURN(4.0 / (1.0 + x*x));
11 END f;
12
13 MONITOR assignment;
14 VAR sum : REAL;
15 pos, answers: INTEGER;
16
17 ENTRY get_interval(VAR int: INTEGER);
18 BEGIN
19 pos := pos+1;
20 IF pos<=intervals THEN int := pos
21 ELSE int := -1 (* fertig *)
22 END;
23 END get_interval;
24
25 ENTRY put_result(res: REAL);
26 BEGIN
27 sum := sum+res;

```

```

28 answers := answers+1;
29 IF answers = intervals THEN (* show result *)
30 WriteString("Pi = "); WriteReal(sum,10); WriteLn;
31 END;
32 END put_result;
33
34 BEGIN (* monitor-init *)
35 pos := 0; answers := 0;
36 sum := 0.0;
37 END MONITOR assignment;
38
39 PROCESS worker(id: INTEGER);
40 VAR iv : INTEGER;
41 res: REAL;
42 BEGIN
43 assignment_get_interval(iv); (* read 1.task from monitor *)
44 WHILE iv > 0 DO
45 res := width * f( FLOAT(iv)-0.5 * width );
46 assignment_put_result(res); (* send result to monitor *)
47 assignment_get_interval(iv); (* read task from monitor *)
48 END
49 END PROCESS worker;
50
51 PROCEDURE start_procs;
52 VAR i: INTEGER;
53 BEGIN
54 FOR i:= 1 TO num_work DO START(worker(i)) END
55 END start_procs;
56
57 BEGIN
58 start_procs;
59 END PROCESSOR MODULE pi_calc.

```

L.3 SIMD Programming Languages

- Fortran 90 Fortran Committee 1991
- HPF “High Performance Fortran” HPF Committee 1992
- MPL “MasPar Programming Language” MasPar 1990
- C* V5 Rose, Steele 1987
- C* V6 Thinking Machines 1990
- Parallaxis Bräunl 1989

Fortran 90

Vector commands: (examples)

```

INTEGER, DIMENSION(100,50) :: A, B, C
A = B + C
...
INTEGER, DIMENSION(50) :: V
V(2:10) = 1
V(1:21:2) = 1
V = A(77,:)

WHERE (V .LT. 0) V = 7

S = SUM(V)

```

parallel 100x50 array of integer
matrix-addition

1 dim. vector
allocation of scalar to vector
allocation with "step" (here: only every 2nd element)
allocation of a matrix row

PE-selection via logical expression

data reduction, further reductios operations are:
ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT, SUM

Parallel standard functions:

```

DOTPRODUCT (Vector_A, Vector_B)
MATMUL (Matrix_A, Matrix_B)

```

Fortran 90

Computation of the dot product in Fortran 90:

```

INTEGER S_PROD
INTEGER, DIMENSION(100) :: A, B, C
...
C = A * B
S_PROD = SUM(C)

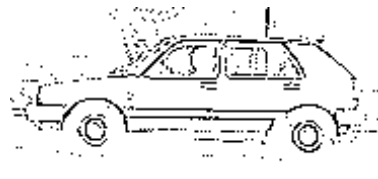
```

Fortran 90 Example: Laplace Operator

for each pixel:

	-1	
-1	4	-1
	-1	

in the overall image



Fortran 90 Example: Laplace Operator

	-1	
-1	4	-1
	-1	

```

INTEGER, DIMENSION(0:101,0:101) :: Image
...
Image(1:100,1:100) = 4*Image(1:100,1:100)
                    - Image(0: 99,1:100) -Image(2:101,1:100)
                    - Image(1:100,0: 99) -Image(1:100,2:101)

```

⋮

HPF (High Performance Fortran)

- Problem in Fortran 90: only **implicit** parallelism
- Programs cannot simply be adjusted to parallel hardware (*tuning* only via compiler directives)
- Development of a Fortran-Variants with explicit parallel constructs
 - Fortran D
 - High Performance Fortran HPF

⋮

HPF (High Performance Fortran)

Parallel Execution

```
FORALL I = 1,100
  FORALL J = 1,100
    A(I,J) = 5 * A(I,J) - 3
  ENDDO
ENDDO
```

Reduction

```
REAL X, Y(N)
FORALL I = 1,N
  REDUCE(SUM, X, Y(I))
ENDDO
```

⋮

C*

Rose, Steele, Thinking Machines 1987 (V5, based on C++)
 Thinking Machines Co. 1990 (V6, based on C with concepts similar to Parallaxis)

- Development for Connection Machine, but mostly hardware independent
- C extended with parallel constructs
- Virtual processors in C*-programs (hardware-implemented on the Connection Machine)
- Each PE can access the local memory of another PE via index expressions

Parallel Language Concepts:

- Variables are declared:
 - Only for the host (regular data declaration as in C)
 - Parallel for groups of PEs ("shape"-declaration)
- No definition of PE-connections, but automatic routing during data accesses on neighbor-PEs.

⋮

C* (V6)

Example Matrix Addition:

```
shape [100][50] two_dim;
int:two_dim A, B, C;
...
with (two_dim)
  { A = B + C; }
```

Example Selection:

```
shape [50] one_dim;
int:one_dim V;
int:one_dim Field[100];
...
with (one_dim)
  where (V < 0) { V = 7; }
```

⋮

C* (V6)

Reduction:

```
S = 0;
S += V;
```

further reduction operators:

```
+=(Sum)   *=(Product)  &=(AND)  |=(OR)   ^=(XOR)
<?=(Minimum)  >?=(Maximum)
```

⋮

C* (V6)

Problems:

```
int S;
...
S = (int) V;
```

Attention:

allocation of an undetermined component!

```
int S;
int:one_dim V, W;
...
S += V
S = S+V
```

$S := S + \sum v_i$ vector reduction

Error (attempted assignment of vector to scalar, see above "type casting")

```
V += S
```

$V := V + S$ Addition of scalar to vector

```
V += W
```

$V := V + W$ Addition vector to vector

⋮

C* (V6)

Data Exchange between PEs

via Index Expressions (router access):

```
shape [50] one_dim;
int:one_dim V, W, Index;
... /* Indices have values: 2, 3, 4, ..., 51 */
with (one_dim) {
  [Index]V = W;
}
```

⋮

C* (V6)

Data Exchange between PEs

via Grid Communication:

```
shape [100][50] two_dim;
int:two_dim A, B;
...
with(two_dim) {
  A = pcoord(0);
  B = pcoord(1);
}

shape [50] one_dim;
int:one_dim V, W, Index;
...
with (one_dim) {
  [pcoord(0) + 2]V = W;
}
```

$A = (0 \ 0 \ \dots \ 0; 1 \ 1 \ \dots \ 1; \dots; 99 \ 99 \ \dots \ 99)$

$B = (0 \ 1 \ \dots \ 49; 0 \ 1 \ \dots \ 49; \dots; 0 \ 1 \ \dots \ 49)$

•
•
•

C* (V6)

Dot Product in C*:

```

shape [max] list;
float: list x,y;
float s_prod = 0.0;
...
with (list) {
  s_prod += x*y;
}

```

• • • • • • • •

•
•
•

C* (V6)

Laplace-Operator in C*:

```

shape [100][100] grid;
int: grid pixel, dim1, dim2;
...
with (grid) {
  dim1 = pcoord(0);
  dim2 = pcoord(1);
  pixel = 4*pixel - [dim1-1][dim2] pixel - [dim1+1][dim2] pixel
           - [dim1][dim2-1] pixel - [dim1][dim2+1] pixel;
}

```

abbreviations for pcoord:

```

with (grid) {
  pixel = 4*pixel - [.-1][. ] pixel - [.+1][. ] pixel
            - [.] [-1] pixel - [.] [ +1] pixel;
}

```

• • • • • • • •

•
•
•

MPL MasPar Programming Language

- Designed for MasPar computer
- Extension of standard-C
- Machine dependent (xnet – special instruction to utilize grid structure)

Concepts:

plural	parallel variable
xnet	8-way nearest neighbor connection
router	arbitrary connection topology via global router
all	parallel execution of an instruction sequence
proc	access to individual PEs
visible	data reference Front End / parallel Back End

• • • • • • • •

•
•
•

MPL MasPar Programming Language

Data Exchange between PEs:

- Grid network (xnet): xnetN, xnetNE, xnetE, xnetSE, xnetS, xnetSW, xnetW, xnetNW

Example: j = xnetW[2].i;

- Router (generic, but slower than grid):

Example: j = router[index].i;

• • • • • • • •



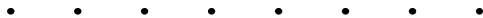
MPL MasPar Programming Language

Scalar Constants

nproc total number of PEs of a MasPar system
 nxproc number of columns of a MasPar system
 nyproc number of rows of a MasPar system

Vector Constants

iprocc PE-Identification number (0 .. nproc - 1)
 ixproc PE-Position within a row (0 .. nxproc - 1)
 iyproc PE-Position within a column (0 .. nyproc - 1)



MPL MasPar Programming Language

Access to individual PEs:

```
int s;
plural int v;
...
s = proc[1023].v; component number 1023 of vector v
s = proc[5][7].v; component of the 5th row and 7th columns of vector v
```

Data Reduction:

reduceADD, reduceMUL, reduceAND, reduceOR, reduceMax, reduceMin



MPL MasPar Programming Language

Dot Product in MPL:

```
float s_prod (a, b)
plural float a,b;
{ plural float prod;
  prod = a*b;
  return reduceAddf (prod);
}
```

Laplace-Operator in MPL:

```
plural int pixel;
...
pixel = 4 * pixel - xnetN[1].pixel - xnetS[1].pixel
          - xnetW[1].pixel - xnetE[1].pixel;
```

