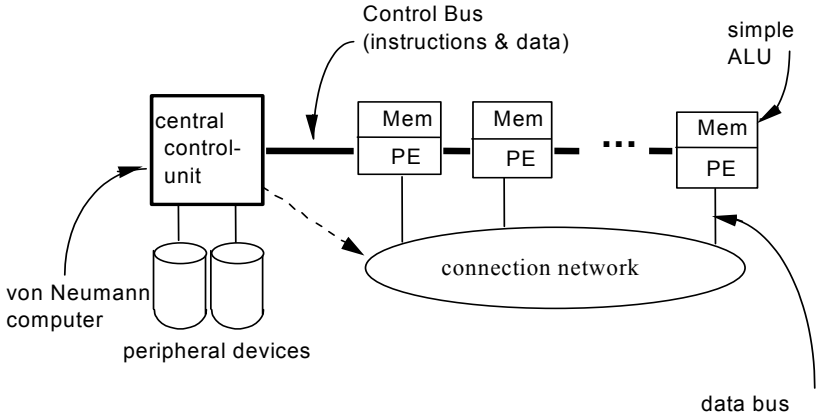
4. Synchronous Parallelism

Setup of SIMD System



Sample SIMD Systems

Historical Systems (1990)

(Note: Peak-Performance values are of limited value)

- Connection Machine — Thinking Machines
- MasPar — MasPar Co.
- DAP — Cambridge Parallel Processing

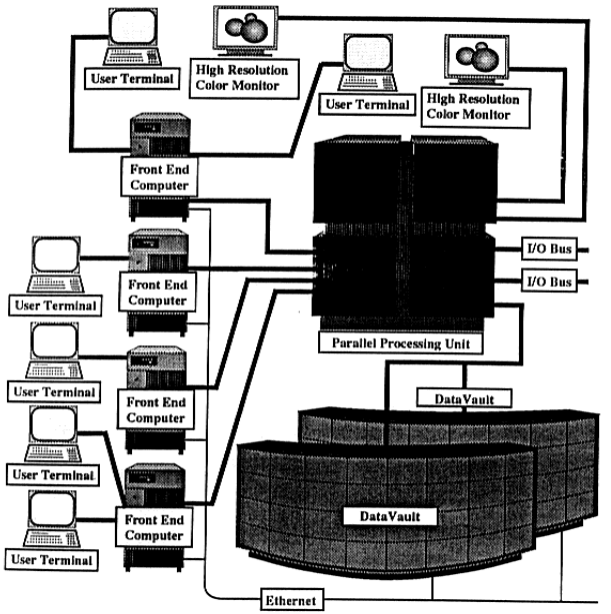
Current Systems (2005)

- SIMD extensions on standard processors
- Still to come: SIMD on a chip (e.g. for image processing)

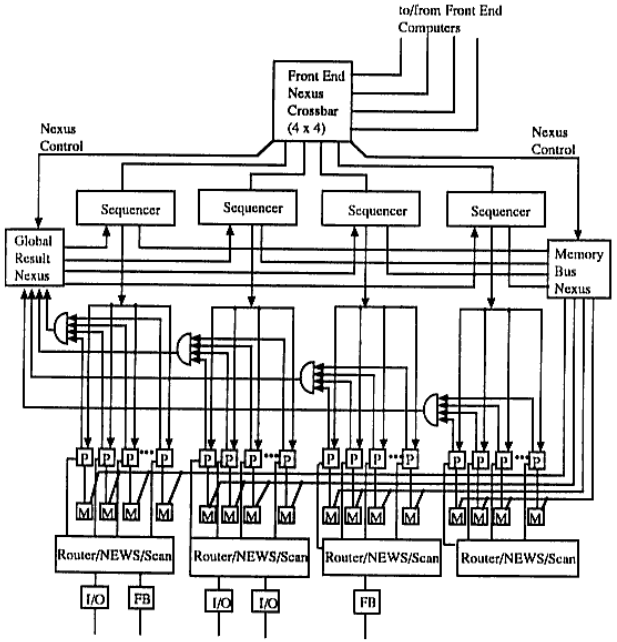
Connection Machine

Manufacturer:	Thinking Machines Corporation	
	Cambridge, Massachusetts	
Model:	CM-2	
Processors:	65.536	PEs (1-bit processors)
	2,048	floating-point co-processors (64-bit, optional)
Memory per PE:	128 KB (maximum)	
Peak-Performance:	2,500	MIPS (32-Bit operation)
	10,000	MFLOPS (scalar product, 32-bit) or
	5,000	MFLOPS (scalar product, 64-bit)
Connection network:	global Hypercube	
	0,32	GB / s
	4-fold reconfigurable neighbourhood grid (via Hypercube)	
	1,0	GB / s
Programming Languages:	CMLisp	(original Lisp-variant)
	*Lisp	(extension of Common Lisp)
	C*	(extension of C)
	CMFortran	(similar to Fortran 90)
	C/Paris	(C with calls to assembler-library routines)

Connection Machine CM-2
© Thinking Machines Co.



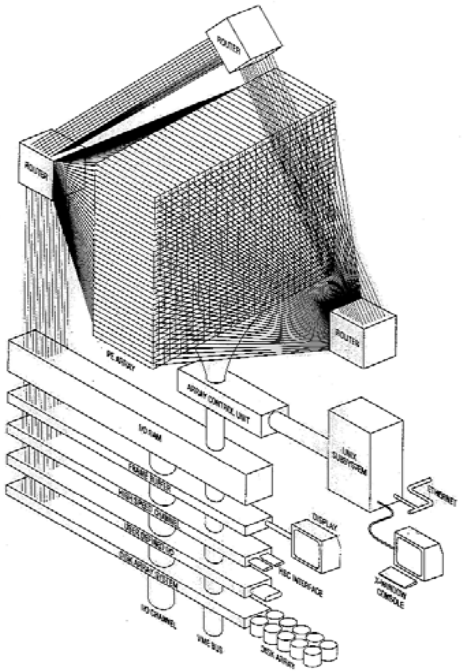
Connection Machine CM-2
© Thinking Machines Co.



MasPar

Manufacturer:	MasPar Computer Corporation Sunnyvale, California	
Model:	MP-2216	
Processors:	16,384 PEs (32-bit processors)	
Memory per PE:	64 KB (maximum)	
Peak-Performance:	68,000	MIPS (32-Bit Operation)
	6,300	MFLOPS (Add./Mult., 32-Bit), or
	2,400	MFLOPS (Add./Mult., 64-Bit)
Connection network:	3-stage global cross-bar-switch (router)	
	1.3	GB / s (without accounting for 3 µs setup)
	8-fold neighbourhood grid torus (independent of router)	
	20 GB / s	
Programming languages:	MPL	(extension of C)
	MPFortran	(similar to Fortran 90)

MasPar MP-2
© MasPar Co.



DAP (Distributed Array Processor)

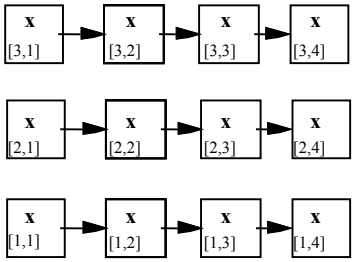
Manufacturer:	Cambridge Parallel Processing Limited Reading, England A Division of Cambridge Management Co., Irvine CA (previously: Active Memory Technology AMT)	
	Model:	DAP 610C
Processors:	4,096 PEs (1-bit processors) 4,096 Co-processors (8-bit)	
Memory per PE:	128 KB	
Peak-Performance:	40,000	MIPS (1-bit operation), or
	20,000	MIPS (8-bit)
	1,200	MFLOPS
Connection network:	4-fold neighborhood grid (**no** global network)	
	5.2	GB / s
Programming languages:	Fortran-Plus	(similar to Fortran 90)

4.1 Communication in SIMD Systems

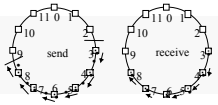
Data exchange in SIMD-Systems is a "collective process".

- No individual data exchange between two PEs
- All active PEs participate in the data exchange (with a relative neighbor-PE)
- Generally *regular structures* are supported, any aberration causes loss of parallelism

Example:



SIMD Data Exchange



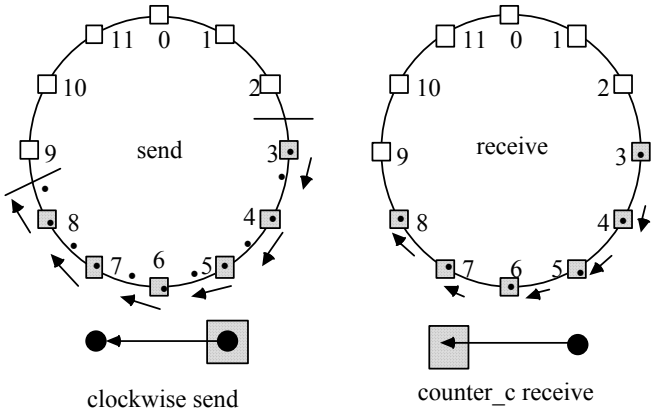
SIMD-Data Exchange:

1. Selection of a group of PEs (= Activation)
2. Selection of a connection direction (static or dynamic)
3. Execution of data exchange between pairs of (active) PEs

Example:

0. Init: Definition of connection structure
   CONFIGURATION ring [0..11];
   CONNECTION right: ring[i] ´ ring[(i+1) mod 12].left;
1. Selection of a group of PEs
   PARALLEL ring[3..8]
   ...
   ENDPARALLEL
2.+3. Execution of parallel data exchange (inside the Parallel-Block)
   propagate.right(x)

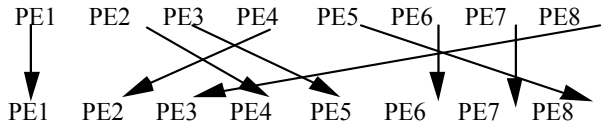
SIMD Data Exchange



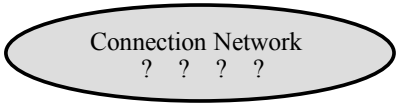
SIMD Data Exchange

Data Exchange as a Vector-Permutation

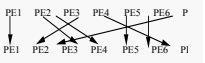
Example:



How do we achieve an efficient solution to the connection/communication problem?



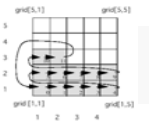
SIMD Data Exchange



How do we achieve an efficient solution to the connection/communication problem?

- The network structure setup in the hardware (fixed, non-configurable topology) can be used directly, if the number of virtual PEs $\leq$ number of physical PEs
- All other topologies have to be implemented via sequences of data exchanges (time consuming, loss of speed)

SIMD Data Exchange



Efficiency loss due to "unstructured data exchange"
e.g. mapping of ring onto a grid

```
PARALLEL ring [0..11]
   PROPAGATE.right(x)
ENDPARALLEL
```

is mapped to:

```
PARALLEL grid [1..2],[1..4];
      grid [3],   [1]
 "grid[i,j]      grid[i,j+1]"
         ENDPARALLEL;
```

Case a: one step to right

```
PARALLEL grid [1..2],[5]
 "grid[i,j]       grid[i+1,1]"
ENDPARALLEL;
```

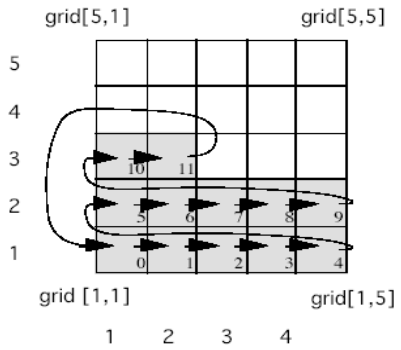
Case b: One line higher

```
PARALLEL grid [3],[2]
 "grid[3,2]       grid[1,1]"
ENDPARALLEL;
```

Case c: back to the start

SIMD Data Exchange

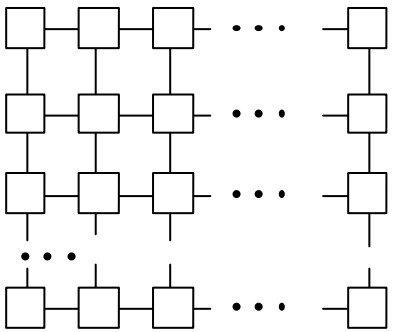
Efficiency loss due to "unstructured data exchange"



SIMD Networks: Connection Machine

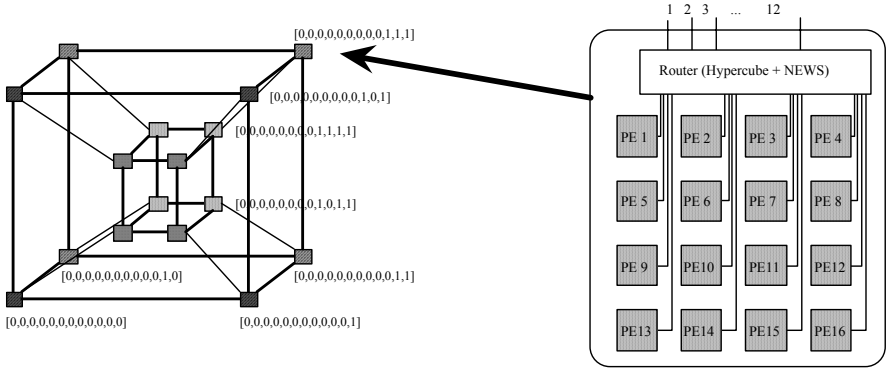
Connection Machine: **65,536 Processors**

a) Configurable multi-dimensional grid ("NEWS"), e.g. 256 × 256 elements (PEs)
4-way nearest neighbor, **fast**



SIMD Networks: Connection Machine

b) Hypercube, 12-dimensional, hence only 4096 elements (Cluster), **slow**
(diagram shows a 4-dimensional hypercube)



SIMD Networks: Connection Machine

b) Hypercube, 12-dimensional, hence only 4096 elements (Cluster), **slow**
(diagram shows a 4-dimensional hypercube)

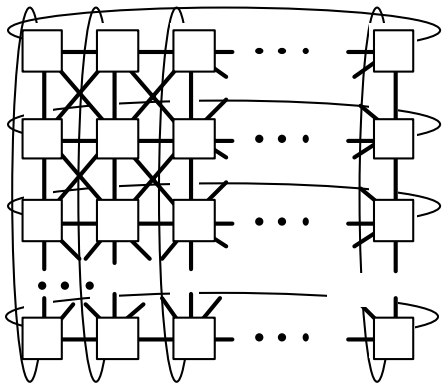
Advantages:

- Direct, faster data exchange if grid topologies are used (among others important for image processing).
- Implementation of all other topologies via the hypercube network in $\log_2 n$ steps (max. 12).

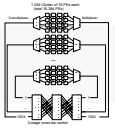
SIMD Networks: MasPar

MasPar: 16,384 Processors

a) Two-dimensional grid, 128 × 128 elements (PEs)
8-way nearest neighbor, **fast**



SIMD Networks: MasPar



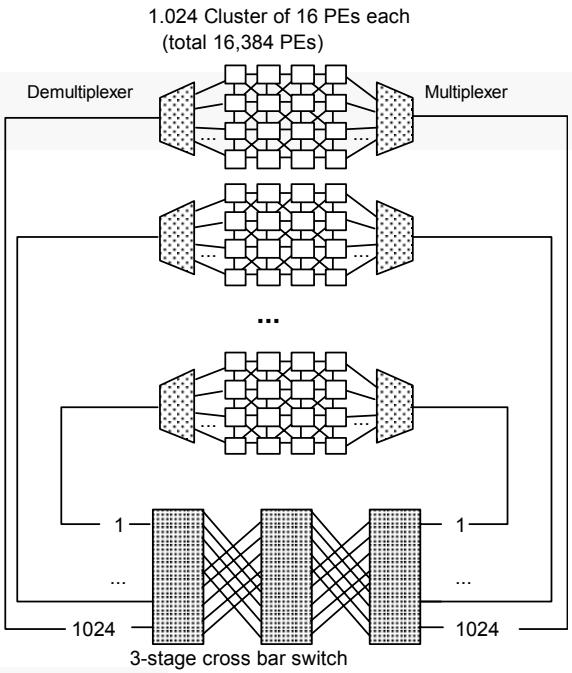
MasPar: 16,384 Processors

b) Crossbar-switch with 1024 simultaneous connections
("Global Router", 3-stage crossbar-switch), **slow**

Advantages:

- Direct, faster data exchange for grid topologies
- Even better connectivity due to 8-fold connections
- Implementation of all other connection structures via "router" with diameter 1024
- For a complete router-data-exchange, 16 steps are required

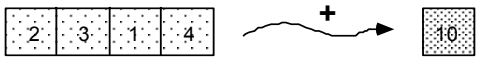
MasPar



SIMD Vector Reduction

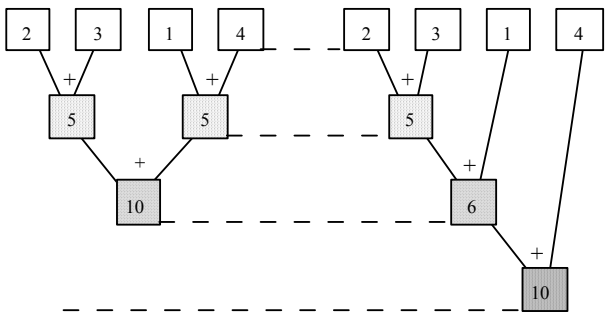
Reduction of a vector into a scalar

Implementation: explicit data exchange + arithmetic operation
or dedicated hardware



SIMD Vector Reduction

Example: s := REDUCE.SUM(v)



tree-like processing
$\log_2 n$ steps

sequential processing
(n–1) steps

Virtual Processors

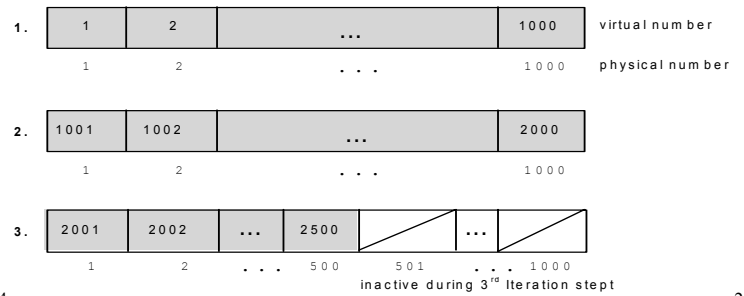
There is only 1 program running on the central control computer.
Its instructions are executed sequentially but data parallel (vectorial).

- Required *virtual* PEs (comparable to the concept of "virtual memory") are mapped to existing *physical* PEs
- PEs that are not required are inactive (switched off)
- If there are more virtual PEs than there are physical Pes the virtual PEs are mapped to the physical ones by iteration

Virtual Processors

Example:
2500 virtual PEs required by user program
1000 physical PEs exist in hardware

Solution through iteration:



Implementing Virtual Processors

This iteration is executed for each elementary operation, or (better still) for instruction sequences without data exchange.

- There is a loss of parallelism due to occasionally inactive PEs, if the number of virtual PEs is not an **exact multiple** of the number of physical PEs.

Data exchange operations are significantly difficult under this iterative mapping

- Complex mapping functions require buffer areas for all virtual PEs (Number of virtual PEs > number of physical PEs).

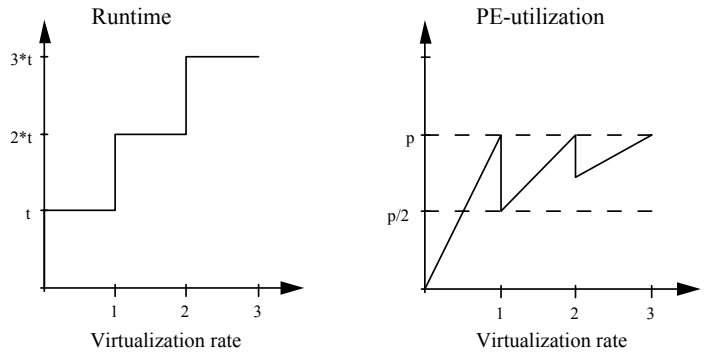
The mapping of virtual PEs to physical PEs should be *transparent* to the programmer:

- Automatically done by compiler (MasPar)
- Hardware-supported by parallel computer (Connection Machine)

Implementing Virtual Processors

For v = number of virtual PEs and p = number of physical PEs:

Virtualization rate R = v/p



Implementing Virtual Processors

Implementing virtual processors in Software

Compiler must satisfy the following tasks:

- Determine number of virtual and physical PEs and virtualization rate R
- Setup of data for all *virtual PEs*
  Example:

  VAR i: list INTEGER; ⇒ VAR i: ARRAY[1..R] OF small_list OF INTEGER;

- Creation of explicit loops for vector-instructions
  Example:

  i := 2*i; ⇒
  ```
  FOR step:=1 TO R DO
      i[step] := 2 * i[step];
  END;
  ```

- Translation of virtual PE-Addresses to physical PE-Addresses for data exchange operations

4.2 Problems with Synchronous Parallelism

Gather - Scatter
Problems with vectorization of indexed accesses (s. Quinn)

Gather:

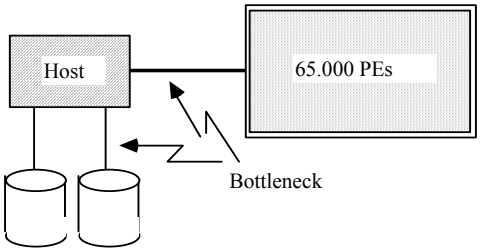
```
for i:=1 to n do
    a[i]:=b[index[i]]
end;
```

Scatter:

```
for i:=1 to n do
    a[index[i]]:=b[i]
end;
```

- Unstructured, random access (*data dependent*).
- Some vector computers solve this problem with dedicated hardware; others use time-consuming software solutions.
- In massive parallel systems: access via slower (but universal) router connection structure.

Bottleneck for Peripheral Devices



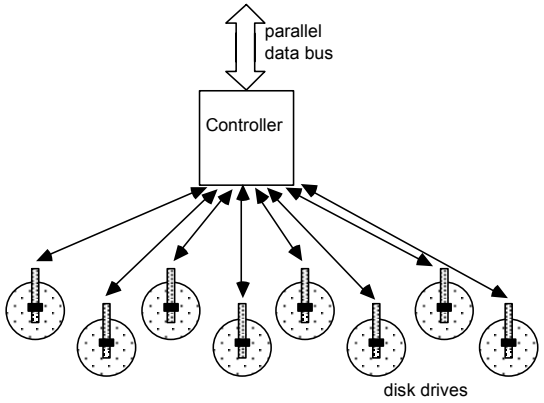
Solution: Parallel connection of peripherals.

- Connection Machine: Data Vault
- MasPar: Parallel Disk Array

Aim: fast flow of incoming and outgoing data bypassing the bottleneck 'host'

Bottleneck for Peripheral Devices

Solution: Parallel connection of peripherals.



Network Bandwidth

Throughput of the network generally determines the performance of the overall system.

- Avoid unnecessary data exchanges as those are considerably more expensive than arithmetic instructions (at least for global data exchange operations)
- Use of structured topologies reduces the communication costs (unlike in random access, fast neighbor connections can be used; possibly requires translation into fast data exchange sequence)
- For most application programs ALUs are fast enough when compared to a global network (but not necessarily for local data exchanges). Increasing performance usually requires an increase in network bandwidth.

Fault tolerance and Multi-User Operation

- SIMD-Systems can only execute one process at any one time
  ⇒ time-sharing required for multi-user operation
- Use of relatively large time slices (up to multiple seconds)
- Due to large amounts of data (up to 1GB main memory) swapping is not possible
  ⇒ distribution of each PE memory to multiple users
  ⇒ each user always has all PEs available, but possibly only a fraction of the memory
- Defect PEs can result in the loss of a whole board (e.g. 1024 PEs) or a whole computer

4.3 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

Parallaxis

Bräunl, 1989

- Machine independent
- Builds on Modula-2
- Each program contains semi-dynamic connection declarations (setup of topology) **and** algorithm description
- Can be used as a simulator on workstations and personal computers and as a true parallel system on MasPar and Connection Machine
- Simulator for single-processor-systems
- Source-Level Debugger
- Visualization tools

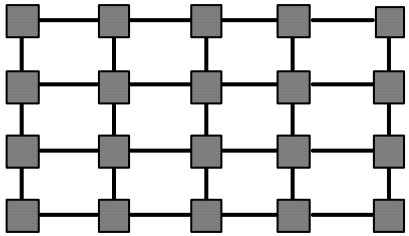
Parallaxis

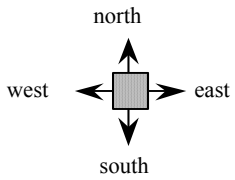
Concepts:

Configuration	configuration of processors into dimensions (analog to array declaration)
Connection	functional definition of a connection structure for all PEs via symbolic port names and direction definition as arithmetic expressions of the indices of the generic PEs, i.e.: right: list[i] → list[i+1].left;
Var (scalar)	declaration of a scalar variable (only for control computer) like seq. lang.
Var..Of (vector)	declaration of a vector variable (for all PEs) with: "<name> of type"
Parallel	parallel execution of an instruction sequence and optional selection of a group of PEs
Move	parallel data exchange within all or a group of PEs
Reduce	reduction of a vector to a scalar
Load	data exchange from Front-End (scalar array) to Back-End (vector)
Store	data exchange from parallel Back End (vector) to Front End (scalar array)

Specification of Parallel Architectures

Example: Two-Dimensional Grid

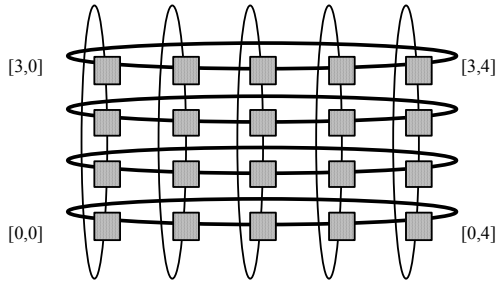





```
CONFIGURATION  grid [1..4],[1..5];
CONNECTION     north: grid[i,j] → grid[i+1, j];
               south: grid[i,j] → grid[i-1, j];
               east : grid[i,j] → grid[i, j+1];
               west : grid[i,j] → grid[i, j-1];
```


Specification of Parallel Architectures

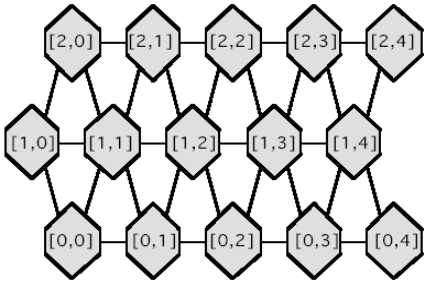
Example: Torus




```
CONFIGURATION  torus [0..h-1],[0..w-1];
CONNECTION right: torus[i,j] ↔ torus[i, (j+1) mod w]:left;
           up:    torus[i,j] ↔ torus[(i-1) mod h, j]:down;
```


Specification of Parallel Architectures

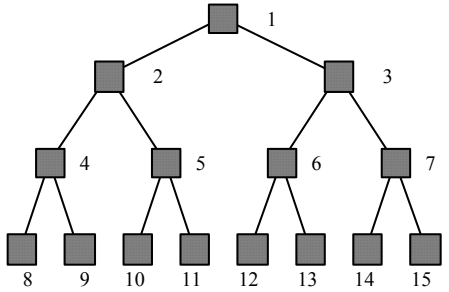
Example: Hexagonal Grid




```
CONFIGURATION hexa [3],[5];
CONNECTION  right  : hexa[i,j] ↔ hexa[i  , j+1]:left;
            up_l   : hexa[i,j] ↔ hexa[i+1, j - i mod 2]:down_r;
            up_r   : hexa[i,j] ↔ hexa[i+1, j+1 - i mod 2]:down_l;
```


Specification of Parallel Architectures

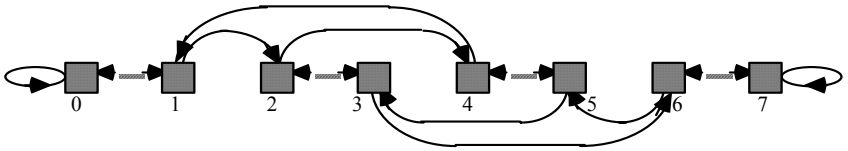
Example: Binary Tree




```
CONFIGURATION  tree [1..15];
CONNECTION   child_l: tree[i] ↔ tree[2*i]:parent;
             child_r: tree[i] ↔ tree[2*i + 1]:parent;
```

Specification of Parallel Architectures

Example: Perfect Shuffle

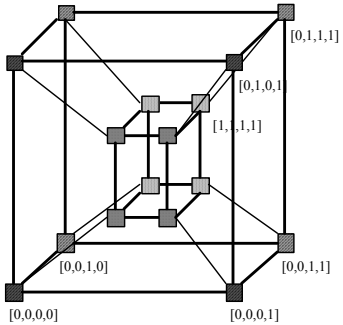



```
CONFIGURATION psn [8];
CONNECTION   exch:    psn[i] ↔ {even(i)} psn[i+1]:exch,
                               {odd(i)}  psn[i-1]:exch;
             shuffle: psn[i] ↔ {i<4}  psn[2*i]:shuffle,
                               {i>=4} psn[(2*i+1) mod 8]:shuffle;
```


Specification of Parallel Architectures

Example: Hypercube

```
CONFIGURATION  hyper [2],[2],[2],[2];
CONNECTION     go(1) :hyper[i,j,k,l] →
                      hyper[(i+1) mod 2, j, k, l);
               go(2) :hyper[i,j,k,l] →
                      hyper[i, (j+1) mod 2, k, l);
               go(3) :hyper[i,j,k,l] →
                      hyper[i, j, (k+1) mod 2, l);
               go(4) :hyper[i,j,k,l] →
                      hyper[i, j, k, (l+1) mod 2);
```

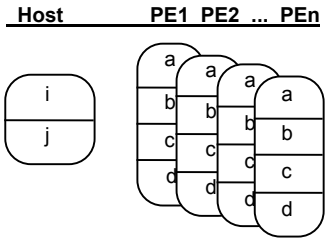


Parallaxis Data Structures

Primary differentiation of
Scalar-Variables (only on control computer) and
Vector-Variables (use configuration name,
component-wise in local memory of the parallel PEs)

Example:

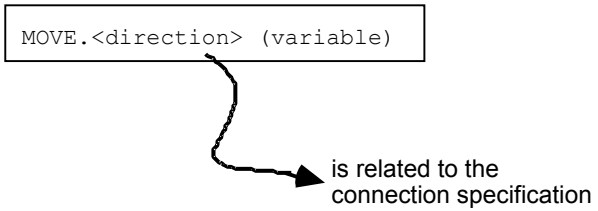
```
VAR i,j: integer;
    a,b: list of real;
    c,d: list of char;
```



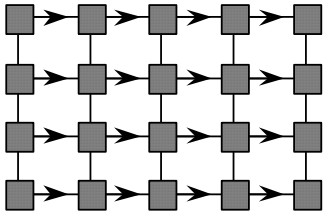
Parallaxis Data Structures

Data exchanges between processors is a *collektive data exchange*.

- **No** individual data exchange between two PEs
- All active PEs participate in the data exchange with a **relative** neighbor-PE



Parallaxis Data Exchange



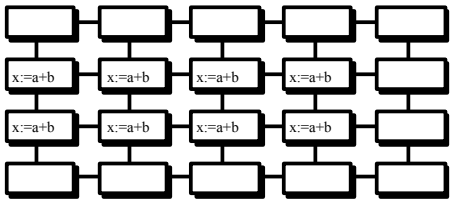
```
y := MOVE.east(x);
```

Data exchange operations:

- MOVE both sender and receiver must be active
- SEND only sender needs to be active
- RECEIVE only receiver needs to be active

Parallel Execution

```
VAR x,a,b: grid OF REAL;
...
IF DIM(grid,2) IN {2,3} THEN
  x := a+b
END;
```



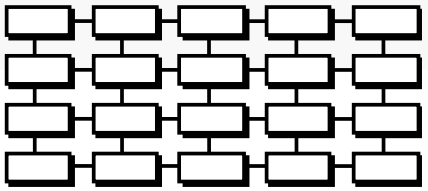
each active PE operates with its own local data

PE–Selection

a) Selections- or iteration-instruction
(IF, WHILE, REPEAT, CASE)

b) Explicit with parallel block

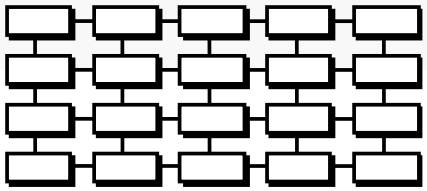
Parallel Selection



```
VAR x: grid OF INTEGER;
...
IF x>5 THEN x := x - 3
       ELSE x := 2 * x
END;
```

PE-ID:	*1*	*2*	*3*	*4*	*5*	
initial values of x:	10	4	17	1	20	
starting *then*-branch:	10	–	17	–	20	(‘–’ means inactive)
after *then*-branch:	7	–	14	–	17	
starting *else*-branch:	–	4	–	1	–	
after *else*-branch:	–	8	–	2	–	
selection done						
after *if*-selection:	7	8	14	2	17	

Parallel Iteration

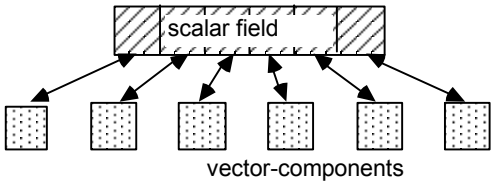


```
VAR x: grid OF INTEGER;
...
WHILE x>5 DO
  x:= x DIV 2;
END;
```

PE-ID:	*1*	*2*	*3*	*4*	*5*	
initial values of x:	10	4	17	1	20	
starting 1st iteration:	10	–	17	–	20	(‘–’ means inactive)
after 1st iteration:	5	–	8	–	10	
starting 2nd iteration:	–	–	8	–	10	
after 2nd iteration:	–	–	4	–	5	
starting 3rd iteration:	–	–	–	–	–	
loop terminates						
after loop:	5	4	4	1	5	

Host Communication

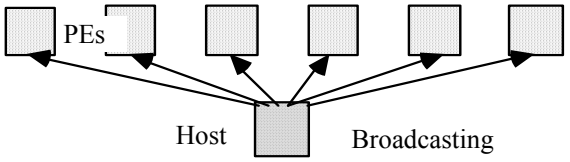
a) Component-wise distribution of an array of data from the host to the PEs (LOAD) or storing of a vector in a scalar array (STORE).



Host Communication

b) Scalar-to-vector allocation (implicit broadcast)

```
VAR s: INTEGER;
    v: list OF INTEGER;
...
v := s;
```

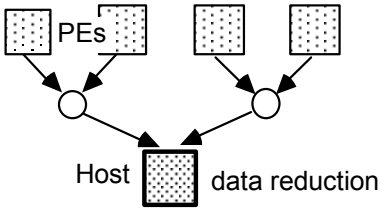


Data Reduction

REDUCE: Vector ⇒ Scalar with predefined reduction operators:

SUM, PRODUCT, MAX. MIN, AND, OR, FIRST, LAST

or arbitrary user-defined reduction operators




```
VAR s: INTEGER;
    x: grid OF INTEGER;

s := REDUCE.SUM(x);
```

Time complexity $O(\log_2 n)$ for n PEs

Dot Product in Parallaxis

```
CONFIGURATION list[max];
CONNECTION;   (* none *)

VAR s_prod:   REAL;
    x,y,prod: list OF REAL;
...
prod := x*y
s_prod := REDUCE.SUM(prod);
```

Laplace Operator in Parallaxis


```
CONFIGURATION grid [1..100],[1..100];
CONNECTION north  : grid[i,j] → grid[i+1, j];
           south  : grid[i,j] → grid[i-1, j];
           east   : grid[i,j] → grid[i, j+1];
           west   : grid[i,j] → grid[i, j-1];

VAR pixel: grid OF INTEGER;
...
pixel := 4*pixel - MOVE.north(pixel) - MOVE.south(pixel)
                 - MOVE.west(pixel)  - MOVE.east(pixel) ;
```


Using MPI in SPMD Mode

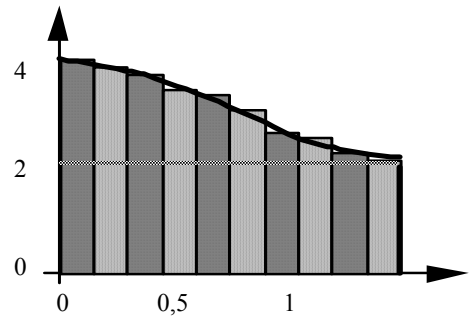
- In order to have a SIMD language based on C and not having to learn new programming concepts, we will be using the MPI library in SPMD mode
- SPMD = **S**ame **p**rogram, **m**ultiple **d**ata
- SPMD is like SIMD *without* the synchronization after each step
- We have to add some synchronization to SIMD algorithms, unless implicit through collective communication

4.4 Massively Parallel Algorithms

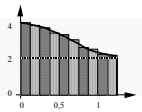
- Massively parallel means 1,000 or more PEs
- Current state: Data-parallel / SIMD
- Totally different operation as for coarse parallel algorithms.
- Processor-utilization is no longer the highest goal.
- Natural formulation of algorithms (with inherent parallelism) is possible due to the loss of artificial restrictions of sequentialization.

Pi Approximation

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \quad = \quad \sum_{i=1}^{\text{int}\,ervals} \frac{4}{1+((i-0.5)*width)^2} * width$$

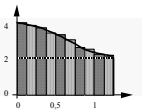


Pi Approximation (Parallaxis)



```
MODULE pi;
(* parallel reference algorithm, used by R. Babb *)
CONST intervalls = 100;
      width      = 1.0 / FLOAT(intervalls);
CONFIGURATION list [1..intervalls];
CONNECTION (* none *);

VAR val: list OF REAL;

PROCEDURE f (x: list OF REAL): list OF REAL;
(* function to be integrated *)
BEGIN
  RETURN(4.0 / (1.0 + x*x))
END f;

BEGIN (* integral approximation with rectangle-rule *)
  val := width * f( (FLOAT(ID(list))-0.5) * width );
  WriteReal(REDUCE.SUM(val), 15); WriteLn;
END pi.
```


Pi Approximation (MPI)



```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

double f(double x)
{ return (4.0 / (1.0 + x*x)); }

int main(int argc, char *argv[])
{ int myid, numprocs;
  double width, val, pi;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  width = 1.0 / (double) numprocs;  /* number of intervals */
  val = width * f( ((double) myid + 0.5) * width);
  MPI_Reduce(&val, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

  if (myid == 0) printf("Approx %.10f\n", pi);
  MPI_Finalize();
  return 0;
}
```


Cellular Automata (Parallaxis)

```
MODULE auto;
CONST n = 79;           (* number of elements *)
      m = (n+1) DIV 2;  (* number of loops *)

CONFIGURATION list [1..n];
CONNECTION left:  list[i] -> list[i-1];
           right: list[i] -> list[i+1];

VAR i  : INTEGER;
    val: list OF BOOLEAN;
    c  : list OF ARRAY BOOLEAN OF CHAR;

BEGIN
  val      :=  ID(list) = m; (* Init *)
  c[FALSE] :=  " ";
  c[TRUE]  :=  "X";
  FOR i:= 1 TO m DO
    Write(c[val]);
    val := MOVE.left(val) <> MOVE.right(val);
  END;
END auto.
```


Cellular Automata (MPI)

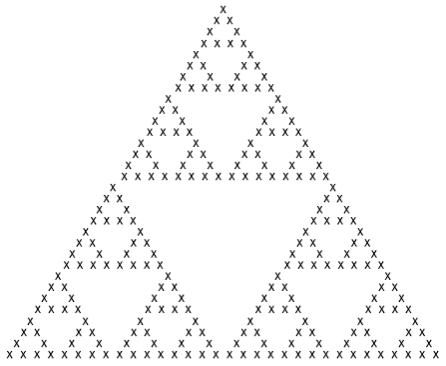
```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

int myid, numprocs;

void parwrite(char c)
/* write values from all processors */
{ char allval[100]; /* max 100 processes */
  int i;

  MPI_Gather(&c, 1, MPI_CHAR, allval, 1, MPI_CHAR, 0,
   MPI_COMM_WORLD);

  if (myid == 0)
  {  for (i=0; i<numprocs; i++) printf("%c", allval[i]);
     printf("\n");
  }
}
```

Cellular Automata (MPI)

```
int main(int argc, char *argv[])
{   int l;     char c, right, left;
    bool val;  MPI_Status mstatus;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    val = (myid == numprocs/2);
    if (val) c='X'; else c=' ';
    for (l=0; l< numprocs/2; l++)
    { parwrite(c);
      /* data exchange to the right */
      if (myid<numprocs-1) MPI_Send(&c, 1, MPI_CHAR, myid+1, 0,MPI_COMM_WORLD);
      if (myid>0)          MPI_Recv(&left, 1, MPI_CHAR,myid-1,0,MPI_COMM_WORLD,&mstatus);
        else left=' ';
      /* data exchange to the left */
      if (myid>0)          MPI_Send(&c, 1, MPI_CHAR, myid-1, 0,MPI_COMM_WORLD);
      if (myid<numprocs-1) MPI_Recv(&right, 1, MPI_CHAR,myid+1,0,MPI_COMM_WORLD,&mstatus);
        else right=' ';
      val = (left != right);
      if (val) c='X'; else c=' ';
    }

    MPI_Finalize();
    return 0;
}
```

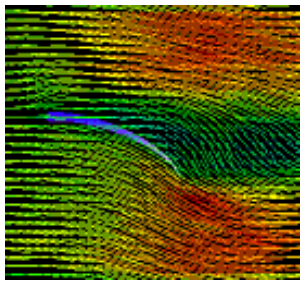

Cellular Automata

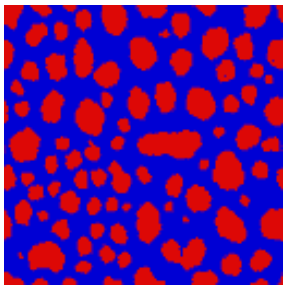
Program Output



Cellular Automata

- Cellular automata are also used for flow simulations in gases (air) and fluids: Lattice Gas Automata





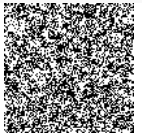
Cellular Automata (Parallaxis)

Simulating the voting behaviour of a population

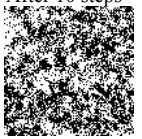
- Random initialization
- For each step: each PE takes the opinion of a randomly selected neighbor
- A cluster results

```
PROCEDURE vote;
VAR step   : INTEGER;
    opinion: grid OF BOOLEAN;
    n      : grid OF ARRAY[0..7] OF BOOLEAN;  (* neighbors *)
BEGIN
  opinion := RandomBool(grid);  (* init *)
  FOR step:=1 TO max_steps DO
    get_neighbors(opinion,n); (* returns state of all 8 neighbors *)
    opinion := n[ RandomInt(grid) MOD 8 ];
    ... (* write current state as image *);
  END;
END vote;
```

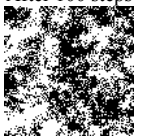
Random init



After 10 steps

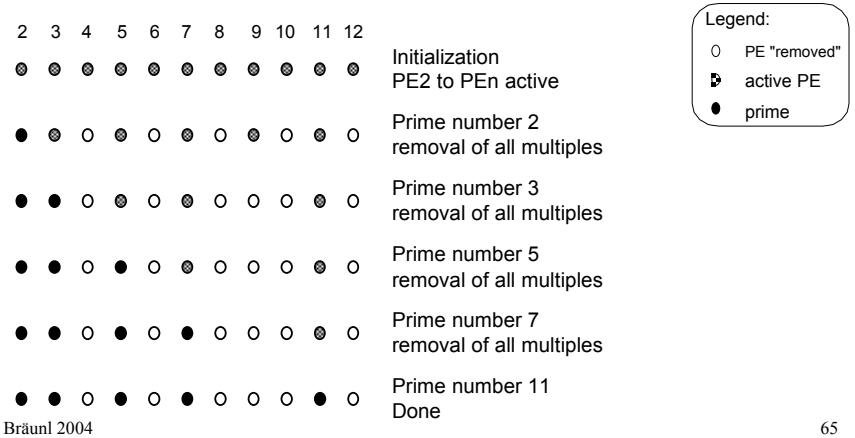


After 100 steps



Prime Number Generation

Parallel Sieve of Eratosthenes



Prime Number Generation (Parallaxis)

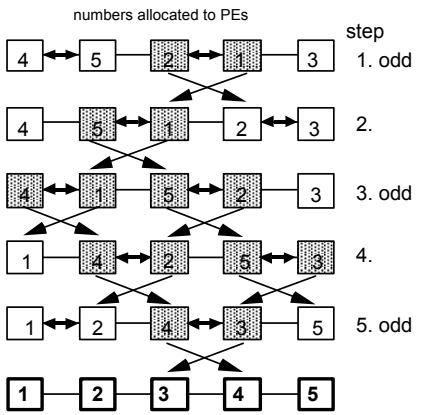
```
MODULE prime;
CONFIGURATION list [2..200];
CONNECTION  (* none *);

VAR next_prime: INTEGER;
    removed   : list OF BOOLEAN;

BEGIN
  REPEAT
    next_prime:= REDUCE.FIRST(DIM(list,1));
    WriteInt(next_prime,10); WriteLn;
    removed := DIM(list,1) MOD next_prime = 0 (* remove multip. *)
  UNTIL removed
END prime.
```

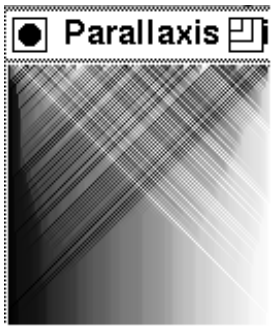

Prime Number Generation (MPI)

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[])
{ int myid, numprocs;
  int prime, candidate;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  candidate = myid + 2;  /* only numbers >=2 are potential primes */
  do {
    MPI_Allreduce(&candidate,&prime,1,MPI_INT,MPI_MIN,MPI_COMM_WORLD);
    if (myid == 0 && prime < numprocs) printf("Prime: %d\n", prime);
    if (candidate % prime == 0) candidate = numprocs; /* elim. dup. */
  } while (prime < numprocs);

  MPI_Finalize();
  return 0;
}
```


Odd-Even Transposition Sorting (OETS)

(a parallel version of Bubblesort)





OETS (Parallaxis)

```
MODULE sort;  (* Odd-Even Transposition Sorting *)
CONST n = 10;
CONFIGURATION list [1..n];
CONNECTION left: list[i] <-> list[i-1] :right;

VAR step    : INTEGER;
    a       : ARRAY[1..n] OF INTEGER;
    val,comp: list OF INTEGER;
    lhs     : list OF BOOLEAN;

BEGIN
  WriteString('Enter '); WriteInt(n,1); WriteString(' values: ');
  ReadInt(val);
  lhs := ODD(ID(list));  (* PE is left-hand-side of a comparison *)
  FOR step:=1 TO n DO
    IF lhs THEN comp := RECEIVE.left (val)
           ELSE comp := RECEIVE.right(val)
    END;
    IF lhs = (comp<val) THEN val:=comp END;       (* lhs & *)
    lhs := NOT lhs;                (*          (comp< val) *)
  END;                             (* or rhs & (comp>=val) *)
  WriteInt(val,5);
END sort.
```


OETS (MPI)

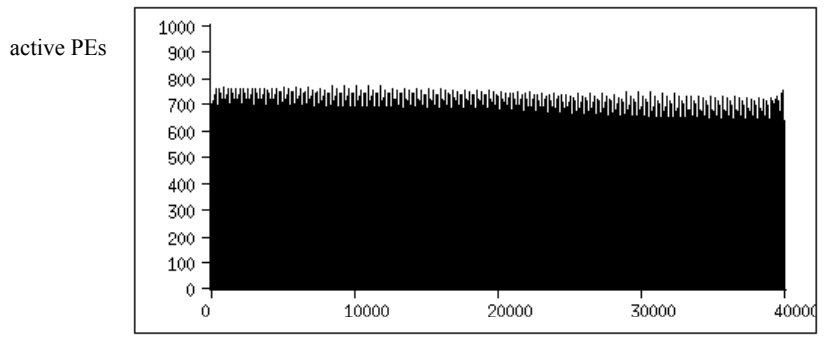
```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define SIZE 10
int myid, numprocs;

void parwrite(char* str, int v)
{ … }

int main(int argc, char *argv[])
{ int val, allval[SIZE] = { 5,2,3,4,1,8,7,9,0,6 }; /* to be sorted */
  int i, cmp;
  MPI_Status mstatus;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /* distribute values to individual processor */
  MPI_Scatter(allval, 1, MPI_INT, &val, 1, MPI_INT, 0, MPI_COMM_WORLD);

  parwrite("Initial values", val);

  /* continued on next page … */
```


OETS (MPI)

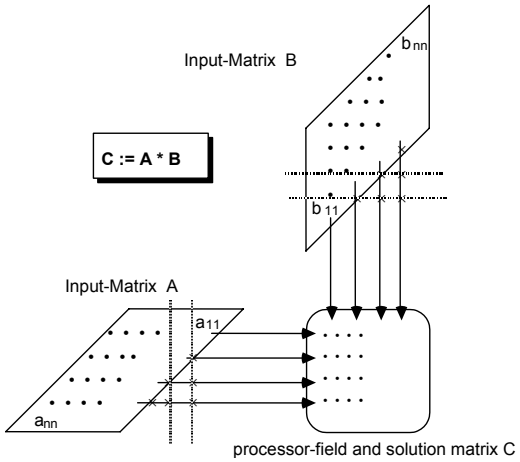
```
for (i=0; i< SIZE; i++)
{ if (myid%2 == i%2)
  { if (myid < SIZE-1) /* PE 0, 2, 4,.. in even steps, 1, 3,.. in odd */
    { MPI_Send(&val, 1, MPI_INT, myid+1, 0, MPI_COMM_WORLD); /* right */
      MPI_Recv(&cmp, 1, MPI_INT, myid+1, 0, MPI_COMM_WORLD, &mstatus);
      if (cmp <= val) val = cmp; /* swap (part 1) */
    }
  }
  else
  { if (myid > 0) /* PE 1, 3, 5,.. in even steps, 2, 4,.. in odd */
    { MPI_Send(&val, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD); /* left */
      MPI_Recv(&cmp, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &mstatus);
      if (cmp >  val) val = cmp; /* swap (part 2) */
    }
  }
  parwrite("Step", val);
}

/* gather result back from each processsor */
parwrite("Result", val);

MPI_Finalize();
return 0;
}
```


Odd-Even Transposition Sorting

Analysis: Sort of 1000 numbers



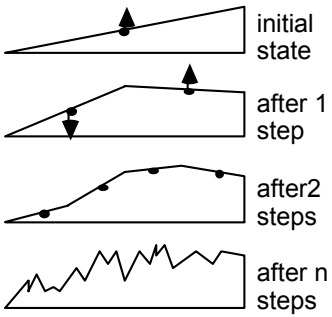
Systolic Matrix-Multiplication



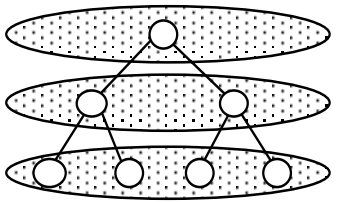
Systolic Matrix-Multiplication (Parallaxis)

```
MODULE matrix;
CONST max = 10;
CONFIGURATION  grid [0..max-1],[0..max-1];
CONNECTION     left:   grid[i,j]  ->  grid[i,(j-1) MOD max];
               up:     grid[i,j]  ->  grid[(i-1) MOD max,j];
               shiftA: grid[i,j]  ->  grid[i,(j-i) MOD max];
               shiftB: grid[i,j]  ->  grid[(i-j) MOD max,j];

VAR i,j  : INTEGER;
    a,b,c: grid OF REAL;

PROCEDURE matrix_mult(VAR a,b,c : grid OF REAL);
VAR k: INTEGER;  (* c := a * b *)
BEGIN
  a := MOVE.shiftA(a);
  b := MOVE.shiftB(b);
  c := a * b;
  FOR k := 2 TO max DO
    a := MOVE.left(a);
    b := MOVE.up  (b);
    c := c + a * b;
  END;
END matrix_mult;

BEGIN (* preset matrices as example *)
  a := FLOAT(DIM(grid,2) * 10 + DIM(grid,1));  WriteFixPt(a,7,1);
  b := FLOAT(DIM(grid,2)      + DIM(grid,1));  WriteFixPt(b,7,1);
  matrix_mult(a,b,c);  WriteFixPt(c,7,1);
END matrix.
```


Fractal Curve Generation



PEs are ordered as a binary tree



Fractal Curve Generation (Parallaxis)

Divide-and-Conquer Algorithm

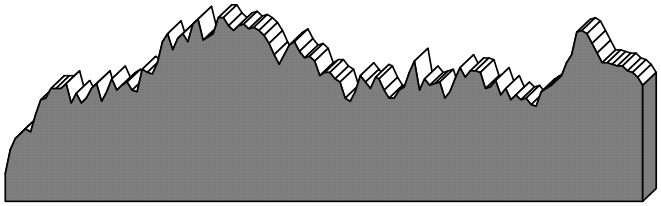
```
CONFIGURATION tree [1..maxnode];
CONNECTION    child_l : tree[i] <-> tree[2*i]   :parent;
              child_r : tree[i] <-> tree[2*i+1] :parent;

VAR i,j          : INTEGER;
    delta        : REAL;
    field        : ARRAY [1..maxnode] OF REAL;
    x, low, high: tree OF REAL;

PROCEDURE Gauss(): tree OF REAL;
…
END Gauss;

PROCEDURE inorder(node: INTEGER);
(* Output of tree elements in linear sequence *)
…
END inorder;
```

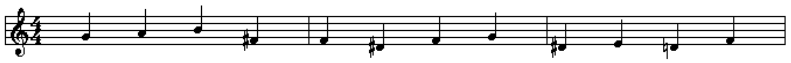
Fractal Curve Generation (Parallaxis)

```
PROCEDURE MidPoint(delta: REAL; level: INTEGER);
BEGIN
  IF 2**(level-1) <= ID(tree) <=  2**level - 1 THEN (* select tree level *)
    x := 0.5 * (low + high) + delta*Gauss();
    IF level < maxlevel THEN
      SEND.child_l(low,low); (* values for children *)
      SEND.child_l(x,high);
      SEND.child_r(x,low);
      SEND.child_r(high,high);
    END;
  END;
END MidPoint;

BEGIN (* set starting values *)
  low  := low_val;  high := high_val;  x := 0.0;
  FOR i:=1 TO maxlevel DO
    delta := 0.5 ** (FLOAT(i)/2.0);
    MidPoint(delta,i);
  END;
  STORE(x,field);
  WriteFixPt(low_val,  10,3); WriteLn;
  inorder(1); (* print displaced values in linear order *)
  WriteFixPt(high_val, 10,3); WriteLn;
END fractal.
```

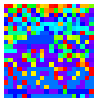

Fractal Curve Generation

Program Result for Fractal Curve
(depending on interpretation)

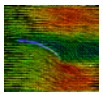




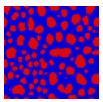
Parallel Simulation



Neuronal Nets and
Genetic Algorithms



Fluid Simulation
(Lattice-Gas)

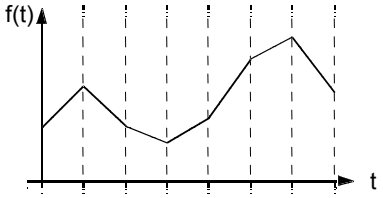


Immisicable Fluids
(Lattice Gas)

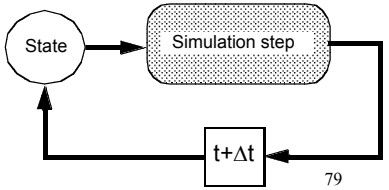
Rigid Body Simulation

Traffic Simulation

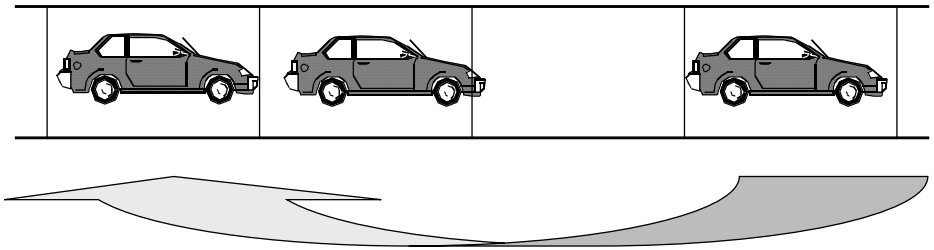
Time-discrete Simulation



General Scheme



Traffic Simulation



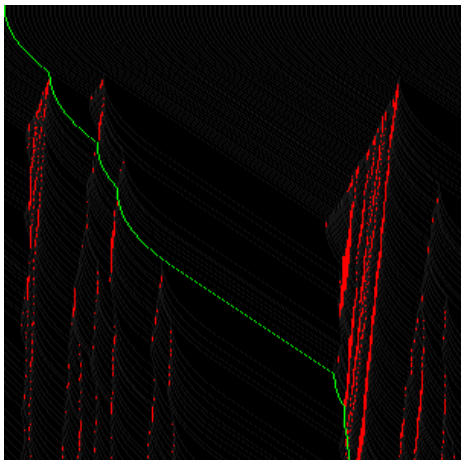
Traffic Simulation

- Model street as ring
- Use 1 PE for each car
  For graphics display in addition:
  use 1 PE for each street segment (=pixel)
- Cars start at rest in equal distance, accelerate from 0
- Cars break if distance in front too short
  (with some random factor)
- Cars accelerate if distance in front is sufficient
  (with some random factor)
- Change initial car density and
  see what happens after some iterations!

Traffic Simulation (Parallaxis)

```
FOR time := 1 TO steps DO
  my_car := DIM(street,1) = TRUNC(pos<:0:> * FLOAT(width));
  dist := ABS(MOVE.back(pos) - pos);
  collision := dist < min_dist;

  IF collision THEN speed := 0.0;
  ELSE (* no collision, accelerate *)
   accel := max_accel + rand_fac * (RandomReal(cars)-0.5);
   (* brake, if necessary *)
   IF dist < min_dist THEN accel := - max_accel END;
   (* update speed, apply speed limit *)
   speed := min(speed + accel, max_speed);
   (* do not back up on autobahn ! *)
   IF speed < 0.0 THEN speed := 0.0 END;
   (* update position *)
   IF speed < dist THEN pos := pos + speed ELSE speed := 0.0 END;
   (* leaving right, coming back in left *)
   IF pos >= 1.0 THEN pos := pos - 1.0 END;
 END;
END;
```


Traffic Simulation



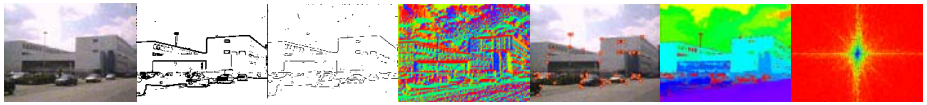
Green: “my” car

Red: spontaneous traffic jams

Parallel Image Processing

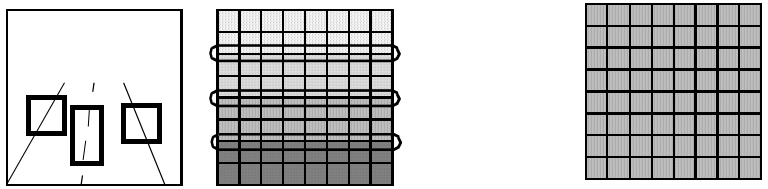
- Low level image processing is an ideal application for data parallelism, since it involves identical operations on all pixels
- Examples:
  - Edge detection
  - Corner detection
  - Stereo analysis
  - Motion analysis
  - Correlation

Parallel Image Processing



Original Edges Skeleton Edge-direction Corners Regions Fourier

MIMD (a) MIMD (b) SIMD



Edge Detection

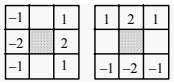
- Many different algorithms:
  - Robert's Cross
  - Laplace
  - Sobel
  - …
- Sobel
  Two operators for each pixel:

–1		1
–2		2
–1		1

1	2	1
–1	–2	–1

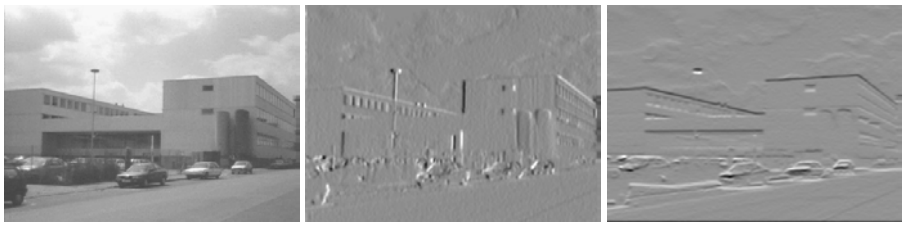
$$edges = \sqrt{horizontal^2 + vertical^2}$$

Sobel Edge Detection (Parallaxis)

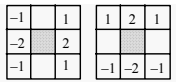


```
PROCEDURE sobel_x_3x3(img: grid OF gray): grid OF INTEGER;
VAR col: grid OF INTEGER;
BEGIN
  col := 2*img + MOVE.up(img) + MOVE.down(img);
  RETURN MOVE.left(col) - MOVE.right(col);
END sobel_x_3x3;
```

Original Sobel-x Sobel-y

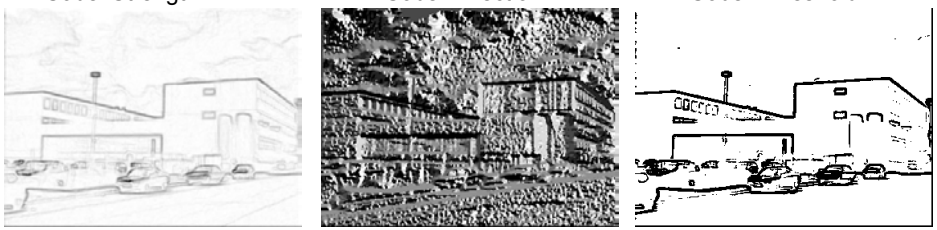


Sobel Edge Detection (Parallaxis)



```
PROCEDURE edges_sobel_3x3(img: grid OF gray;
          VAR strength,direction: grid OF gray);
VAR dx,dy: grid OF INTEGER;
BEGIN
  dx := sobel_x_3x3(img);
  dy := sobel_y_3x3(img);
  strength := limit2gray( ABS(dx) + ABS(dy) );
  direction:= round((arctan2(FLOAT(dy),FLOAT(dx))+pi) /(2.0*pi)*255.0);
END edges_sobel_3x3;
```

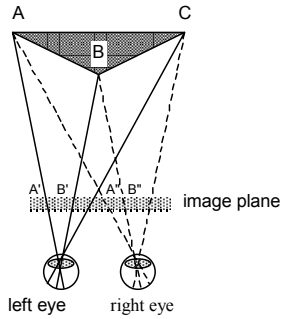
Sobel Strength Sobel Direction Sobel Threshold



Stereo Image Analysis

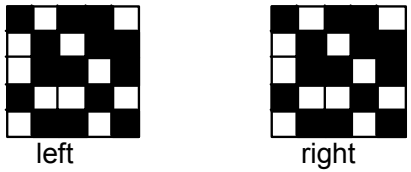
Stereo-Image-Analysis

- Obviously easy for a human
- Extremely computation intensive problem
- Can be solved via data parallel and local operators

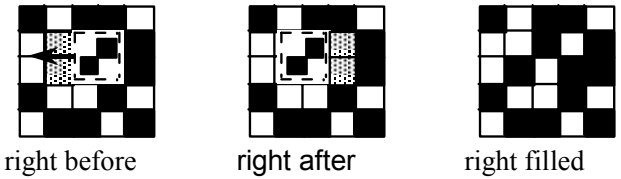


Random Dot Stereograms: Generation

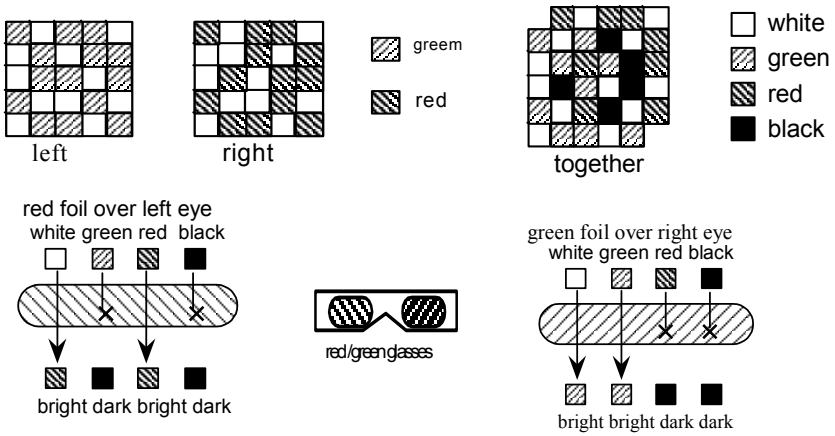
1. Filling of left and right image with the *same* random values



2. Raising or lowering of areas

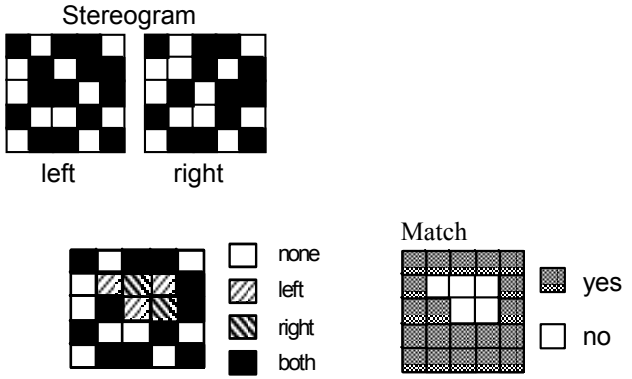


Random Dot Stereograms: Display



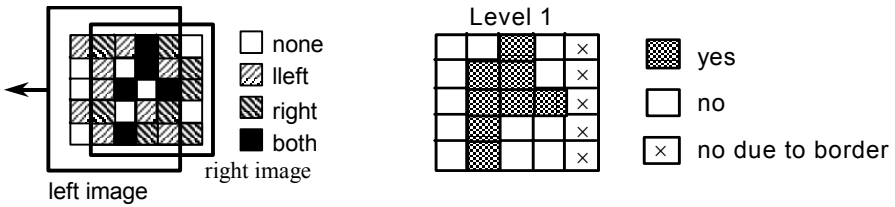
Random Dot Stereograms: Analysis

1. Overlapping of left and right image & search for matches



Random Dot Stereograms: Analysis

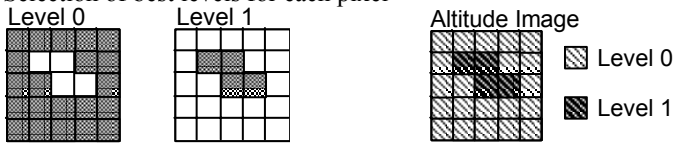
2. Shift of left image (1 pixel to left) & comparison with right image
   This shift step with subsequent comparison (see 1.) is executed iteratively for each altitude level.



Random Dot Stereograms: Analysis

3. Determination of pixel-environment for each level.
   This step is also repeated iteratively for each level and is executed in parallel for all pixels.

4. Selection of best levels for each pixel



5. Filtering (optional)
   (image processing routines)

Random Dot: Generation (Parallaxis)

```
PROCEDURE generate_random_dot(VAR l_pic,r_pic: image OF BOOLEAN);
VAR i,num, from_x, from_y, to_x, to_y, shifts: INTEGER;
BEGIN
  l_pic:= RandomBool(image);
  r_pic := l_pic;
  WriteString("Number of Areas to Elevate: "); ReadInt(num);
  FOR i := 1 TO num DO
    WriteString("Area: ");  ReadInt(from_x);   ReadInt(from_y);
    ReadInt(to_x);   ReadInt(to_y);  ReadInt(shifts);
    IF (from_y <= DIM(image,2) <= to_y) AND
       (from_x - shifts <= DIM(image,1) <= to_x) THEN
      SEND.left:shifts(r_pic,r_pic)  (* move rectangle *)
    END;
    IF (from_y <= DIM(image,2) <= to_y) AND
       (to_x - shifts <= DIM(image,1) <= to_x) THEN
      r_pic := RandomBool(image);               (* fill gap *)
    END;
  END;
END generate_random_dot;
```


Random Dot: Analysis (Parallaxis)

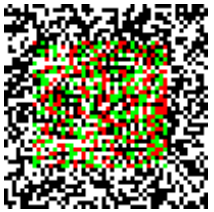
```
PROCEDURE analyze_random_dot(l_pic,r_pic: image OF BOOLEAN;
                             steps: CARDINAL; VAR elev: image OF INTEGER);
VAR equal          : image OF BOOLEAN;
    level, maxlevel: image OF INTEGER;
    i              : INTEGER;

BEGIN
  elev     := 0;
  maxlevel := 0;
  FOR i := 0 TO steps DO (* add congruencies in 3x3 neighborhood *)
    equal := l_pic = r_pic;
    level := sum_3x3( ORD(equal) );

    (* find image plane with max. value *)
    IF equal AND (level > maxlevel) THEN
      elev     := i;
      maxlevel := level;
    END;
    l_pic := MOVE.left(l_pic);  (* move image *)
  END;
END analyze_random_dot;
```

Random Dot: Analysis (Parallaxis)

```
BEGIN
  generate_random_dot(rd_left,rd_right);

  (* red := left  green := right  blue := left AND right *)
  red_green := rgb2color(binary2gray(rd_left, g_white,g_black),
                         binary2gray(rd_right, g_white,g_black),
                         binary2gray(rd_left AND rd_right, g_white, g_black));

  write_c_image(red_green, "rd.ppm",max_width,max_height);

  analyze_random_dot(rd_left,rd_right, max_steps, elevation);

  (* new color table *)
  elevation := int2gray(elevation);
  c_elev    := hsv2color(elevation, image(255),image(255));
  write_c_image(c_elev, "elev.ppm",max_width,max_height);

  (* extend to whole color range and apply median filter *)
  elevation := median_5x5(elevation);
  c_elev    := hsv2color(elevation, image(255),image(255));
  write_c_image(c_elev, "filter.ppm",max_width,max_height);
END random_dot.
```

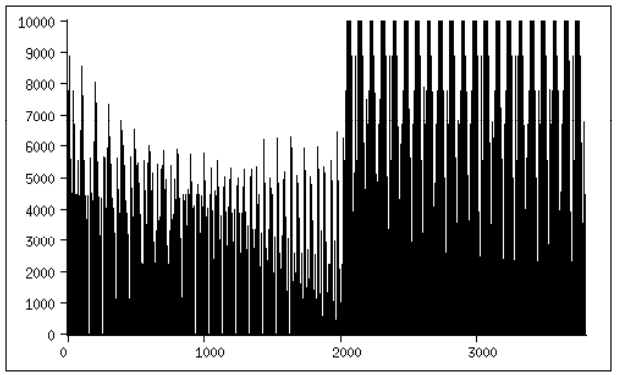

Results of Stereo Image Analysis

left   right

red/green   depth image

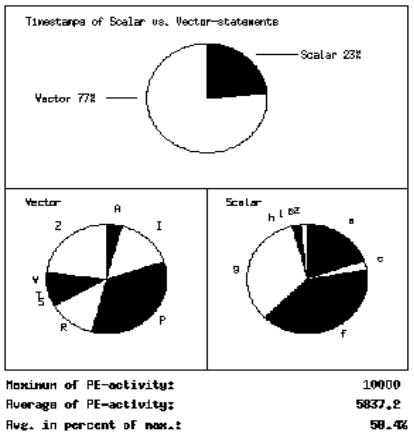
Performance Data for Stereo Image Analysis

active PEs



time in program steps

Performance Data for Stereo Image Analysis



Performance Data for Stereo Image Analysis