4. Synchronous Parallelism

Setup of SIMD System

Control Bus (instructions & data)

simple ALU

Memory PE

connection network

Control Bus (instructions & data)

simple ALU

Memory PE

connection network

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)

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

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

---

**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), or 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), or 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] \rightarrow 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)

Data Exchange as a Vector-Permutation

Example:

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

\[
\begin{align*}
\text{PARALLEL} & \text{ ring } [0..11] \\
\text{PROPAGATE} & \text{ right(s)} \\
\text{ENDPARALLEL}
\end{align*}
\]

is mapped to:

- **Case a**: one step to right
  \[
  \begin{align*}
  \text{PARALLEL} & \text{ grid } [1..2],[1..4]; \\
  & \text{grid } [3,1] \\
  & \text{grid[i,j]} \rightarrow \text{grid[i,j+1]} \\
  \text{ENDPARALLEL} \\
  \end{align*}
  \]

- **Case b**: One line higher
  \[
  \begin{align*}
  \text{PARALLEL} & \text{ grid } [1..2],[5] \\
  & \text{grid[i,j]} \rightarrow \text{grid[i+1,1]} \\
  \text{ENDPARALLEL} \\
  \end{align*}
  \]

- **Case c**: back to the start
  \[
  \begin{align*}
  \text{PARALLEL} & \text{ grid } [3],[2] \\
  & \text{grid[3,2]} \rightarrow \text{grid[1,1]} \\
  \text{ENDPARALLEL} \\
  \end{align*}
  \]

**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)

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 \( \times \) 128 elements (PEs)
  8-way nearest neighbor, fast

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**

1.024 Cluster of 16 PEs each (total 16,384 PEs)

**Demultiplexer**

---

**Multiplexer**

---

3-stage cross bar switch

---

**SIMD Vector Reduction**

Reduction of a vector into a scalar

**Implementation:** explicit data exchange + arithmetic operation or dedicated hardware

![Vector Reduction Example](image)

**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:

1. Virtual number of PEs: 1 2 ... 1000
   Physical number of PEs: 1 2 ... 1000

2. Virtual number of PEs: 1001 1002 ... 2000
   Physical number of PEs: 1 2 ... 1000

3. Virtual number of PEs: 2001 2002 ... 2500
   Physical number of PEs: 1 2 ... 1000

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 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
- Creation of explicit loops for vector-instructions
- Translation of virtual PE-Addresses to physical PE-Addresses for data exchange operations

Example:

```plaintext
VAR i: list INTEGER; 
⇒ VAR i: ARRAY[1..R] OF small_list OF INTEGER;
```
4.2 Problems with Synchronous Parallelism

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

Gather:

\[
\text{for } i := 1 \text{ to } n \text{ do } \\
\quad a[i] := b[index[i]] \\
\text{end;}
\]

Scatter:

\[
\text{for } i := 1 \text{ to } n \text{ do } \\
\quad a[index[i]] := b[i] \\
\text{end;}
\]

• 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’

**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

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)
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];
```

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;
```

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;
```

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;`

Parallaxis Data Structures

Primary differentiation of

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

**Example:**

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

Data exchanges between processors is a **collective data exchange.**
- **No** individual data exchange between two PEs
- **All** active PEs participate in the data exchange with a relative neighbor-PE

**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];`

**MOVE.<direction>** (variable) is related to the connection specification
**Parallaxis Data Exchange**

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

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

**Parallel Execution**

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

**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;
```

**Parallel Iteration**

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

<table>
<thead>
<tr>
<th>PE-ID:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial values of x:</td>
<td>10</td>
<td>4</td>
<td>17</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>starting when-branch:</td>
<td>10</td>
<td>–</td>
<td>17</td>
<td>–</td>
<td>20</td>
</tr>
<tr>
<td>after when-branch:</td>
<td>7</td>
<td>–</td>
<td>14</td>
<td>–</td>
<td>17</td>
</tr>
<tr>
<td>starting else-branch:</td>
<td>–</td>
<td>4</td>
<td>–</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>after else-branch:</td>
<td>–</td>
<td>8</td>
<td>–</td>
<td>2</td>
<td>–</td>
</tr>
<tr>
<td>after if-selection:</td>
<td>7</td>
<td>8</td>
<td>14</td>
<td>2</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PE-ID:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial values of x:</td>
<td>10</td>
<td>4</td>
<td>17</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>starting 1st iteration:</td>
<td>10</td>
<td>–</td>
<td>17</td>
<td>–</td>
<td>20</td>
</tr>
<tr>
<td>after 1st iteration:</td>
<td>5</td>
<td>–</td>
<td>8</td>
<td>–</td>
<td>10</td>
</tr>
<tr>
<td>starting 2nd iteration:</td>
<td>–</td>
<td>–</td>
<td>8</td>
<td>–</td>
<td>10</td>
</tr>
<tr>
<td>after 2nd iteration:</td>
<td>–</td>
<td>–</td>
<td>4</td>
<td>–</td>
<td>5</td>
</tr>
<tr>
<td>starting 3rd iteration:</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>loop terminates</td>
<td>after loop:</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Bräunl 2004
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).

```
vector-components
```

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

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

Data Reduction

REDUCE: Vector $\Rightarrow$ 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);
```

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 = Same program, multiple data
• 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

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

```fortran
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)**

```c
#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)**

```fortran
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)**

```c
#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");
}
```
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 (Parallaxis)

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;  

Cellular Automata

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

Random init

After 10 steps

After 100 steps

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;

Cellular Automata

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

Program Output

Random init

After 10 steps

After 100 steps
Prime Number Generation

Parallel Sieve of Eratosthenes

2 3 4 5 6 7 8 9 10 11 12

Initialization
PE2 to PEn active

Prime number 2
removal of all multiples

Prime number 3
removal of all multiples

Prime number 5
removal of all multiples

Prime number 7
removal of all multiples

Prime number 11
Done

Legend:
○ PE "removed"
◇ active PE
● prime

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) 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)

numbers allocated to PEs

1. odd
2. odd
3. odd
4. odd
5. odd
**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;
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);

  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 */
        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 */
        if (cmp > val) val = cmp; /* swap (part 2) */
        } 
    }
    parwrite("Step", val);
  }
  /* gather result back from each processor */
  parwrite("Result", val);
  MPI_Finalize();
  return 0;
}

**Odd-Even Transposition Sorting**

Analysis: Sort of 1000 numbers

 active PEs

Time in program steps
### Systolic Matrix-Multiplication

**Input-Matrix A**

**Input-Matrix B**

\[ C := A \times B \]

**Processor-field and solution matrix C**

### Systolic Matrix-Multiplication (Parallaxis)

**MODULE matrix;**

**CONFIGURATION** grid \([0..\text{max}-1],[0..\text{max}-1]\);

**CONNECTION**

- \( \text{left: } \text{grid}[i,j] \rightarrow \text{grid}[i,(j-1) \mod \text{max}]; \)
- \( \text{up: } \text{grid}[i,j] \rightarrow \text{grid}[(i-1) \mod \text{max},j]; \)
- \( \text{shiftA: } \text{grid}[i,j] \rightarrow \text{grid}[i,(i-j) \mod \text{max}]; \)
- \( \text{shiftB: } \text{grid}[i,j] \rightarrow \text{grid}[(i-j) \mod \text{max},j]; \)

**VAR i,j : INTEGER;**

\( a, b, c : \text{grid OF REAL}; \)

**PROCEDURE matrix_mult(VAR a,b,c : grid OF REAL);**

**VAR k: INTEGER;**

\( a := \text{MOVE.shiftA}(a); b := \text{MOVE.shiftB}(b); c := a \times b; \)

FOR \( k := 2 \) TO \( \text{max} \) DO

\( a := \text{MOVE.left}(a); b := \text{MOVE.up}(b); c := c + a \times b; \)

END matrix_mult;

**BEGIN**

\( a := \text{FLOAT(DIM(grid,2)) \times 10 + DIM(grid,1)}; \)

\( b := \text{FLOAT(DIM(grid,2)) + DIM(grid,1)}; \)

\( \text{matrix_mult}(a,b,c); \)

**END.**

### Fractal Curve Generation

**Divide-and-Conquer Algorithm**

**CONFIGURATION** tree \([1..\text{maxnode}]\);

**CONNECTION**

- \( \text{child}_l : \text{tree}[i] \leftrightarrow \text{tree}[2*i] : \text{parent}; \)
- \( \text{child}_r : \text{tree}[i] \leftrightarrow \text{tree}[2*i+1] : \text{parent}; \)

**VAR i,j : INTEGER;**

\( \text{delta} : \text{REAL}; \)

\( \text{field} : \text{ARRAY}[1..\text{maxnode}] \text{OF REAL}; \)

\( x, \text{low}, \text{high} : \text{tree OF REAL}; \)

**PROCEDURE Gauss(): tree OF REAL;**

**PROCEDURE inorder(node: INTEGER);**

\( \text{(* Output of tree elements in linear sequence *)} \)

**END.**
**Fractal Curve Generation (Parallaxis)**

PROCEDURE MidPoint(delta: REAL; level: INTEGER);
BEGIN
  IF $2^{*(level-1)} \leq \text{ID(tree)} \leq 2^{*level} - 1$ THEN (* select tree level *)
    \begin{align*}
      x &:= 0.5 * (\text{low} + \text{high}) + \text{delta*Gauss()}; \\
      \text{IF level < maxlevel THEN} &
        \text{SEND.child_l(low,low); (* values for children *)} \\
        \text{SEND.child_l(x,high);} \\
        \text{SEND.child_r(x,low);} \\
        \text{SEND.child_r(high,high);}
    \end{align*}
  END;
END MidPoint;
BEGIN (* set starting values *)
  \text{low} := \text{low_val}; \text{high} := \text{high_val}; \text{x} := 0.0;
  \text{FOR } i:=1 \text{ TO maxlevel DO}
    \text{delta := 0.5}^{*} (\text{FLOAT}(i)/2.0);
    \text{MidPoint(delta, i);}
  \text{END;}
  \text{STORE(x, field);} \\
  \text{WriteFixPt(low_val, 10,3); WriteLn;}
  \text{inorder(1); (* print displaced values in linear order *)}
  \text{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)
- Immiscible Fluids (Lattice Gas)
- Rigid Body Simulation
- Traffic Simulation

**Traffic Simulation**

- General Scheme
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;
    speed := min(speed + accel, max_speed);
    (* do not back up on autobahn ! *)
    IF dist < min_dist THEN pos := pos + speed ELSE speed := 0.0 END;
    (* update position *)
    IF pos >= 1.0 THEN pos := pos - 1.0 END;
END;

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

Green: “my” car
Red: spontaneous traffic jams
Parallel Image Processing

<table>
<thead>
<tr>
<th>Original</th>
<th>Edges</th>
<th>Skeleton</th>
<th>Edge-direction</th>
<th>Corners</th>
<th>Regions</th>
<th>Fourier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bräunl 2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Edge Detection

- Many different algorithms:
  - Robert’s Cross
  - Laplace
  - Sobel
  - ...

- Sobel
  Two operators for each pixel:

\[
\begin{array}{ccc}
-1 & 1 & -1 \\
-2 & 2 & -2 \\
-1 & 1 & -1 \\
\end{array}
\]

\[
edges = \sqrt{\text{horizontal}^2 + \text{vertical}^2}
\]

Sobel Edge Detection (Parallax)

```
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;
```

```
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 Edge Detection (Parallax)
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

- Overlapping of left and right image & search for matches

Random Dot Stereograms: Analysis

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

Match
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.

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 (Parallax)

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 (Parallax)

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;

Bräunl 2004
Random Dot: Analysis (Parallax)

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.

Performance Data for Stereo Image Analysis
Performance Data for Stereo Image Analysis

![Graph showing computation time in s against image size in pixels for different machines and number of steps.](image)

Bräunl 2004