1. Fundamentals

- Increase in performance: *faster processing*
- Finding solutions to bigger problems
- Physical limits for single processors have “almost” been reached, or increasing performance with a single processor is only possible at a very high cost
- It is the **natural form** of information processing

Why Parallel Processing?

Scenario 1:
- Control of autonomous vehicles via camera data
  - fast: cycle time 40ms
  - asynchronous: reaction to events
    (actually 20 ms at 50Hz, but only half-pictures are transmitted)

Scenario 2:
- Quantum chromo dynamic (QCD) [or 3D flow-simulation]
  - Calculating with conventional sequential computers architectures is impossible as it would require CPU-years (in 1994)

1.1 Introduction

How to process in parallel?
How to do Parallel Processing?

NOTE: parallelization ≠ faster processing

"Merely parallelizing a program does not necessarily improve its performance: Communication overhead can more than offset any reduction in elapsed time ..."
Scott Burleigh, JPL 1993

Which class of Systems should be used?

Determining Factors:

1. Data Parallelism vs. Function Parallelism
   Does the problem (majority of problems) consist of similar instructions that can be solved with synchronous parallelism (→ SIMD) or is the majority of instructions of different nature (→ MIMD)

There is no UNIVERSAL parallel computer!
   • Selection depends on area of application
   • "Heterogeneous Parallel Computing"

There is no IDEAL parallel algorithm!
   • Data parallelism vs. function parallelism
   • Communication costs (latency, bandwidth)
   • Only performance counts in practice (time behaviors)!
   • Algorithmically different programs with MIMD and SIMD for the same problem!!
**Where is Parallel Processing?**

Parallelism is everywhere

- Processes in nature / society / technical processes
- Even in a simple PC: E/A, DMA, Microcode, 16-bit-arithmetic
- Parallel processing is the natural form of processing information.
- Sequential processing often is an artificial restriction, brought about by historical factors (success of the sequential von-Neumann computer model).
- Problems suited to parallel processing can be better written and solved in a suitable parallel programming language than in a sequential one.
  - Efficiency
  - Readability / Clarity

**Where is Parallel Processing?**

- Disproportion:
  - 1 CPU, always active (10^6 transistors) vs. 10^10 neurons ("CPU + memory")
  - 10^10 memory elements, mostly inactive vs. always active
  - 10GHz with 10 sub-cycles → 10ps = 10^-11 s vs. 1kHz → 1ms = 10^-3 s

- Switching-processes/sec total:
  - 10^20 / s (theoretical) or 10^12 / s (practical) vs. 10^13 / s

**Where is Parallel Processing?**

At least one reason for the higher performance of the brain is its parallel processing.

**1.2 Classification**

System classification by Flynn

- Single processor computer
- Pipeline computer
- Array computer (Vector computer)
- Multicomputer

**Differentiation of MIMD:**
- Coupling via shared memory
- Coupling via network connections (message passing)
MIMD  
*everyone does something different*  
coordination

SIMD  
*everyone does the same*  
timing pulse

Conveyor Belt  (Pipeline, MISD)  
*everyone repeats individual tasks, everyone has different job*  
overlapping executions

---

SIMD (single instruction, multiple data)  

- Simpler structure  
- **Synchronous**  

Array Computer  

Vector Computer

---

Pipeline

MIMD (multiple instruction, multiple data)  

- More generic structure  
- **Asynchronous**  

i. With shared memory  

ii. Without shared memory  
*(local memory in PEs)*
### Pipeline

**Example:**
- Load data x and y
- Multiply x and y
- Add product to s

### Hybrid Parallel Systems

- Multi-Pipeline
- Multiple-SIMD
- Systolic Arrays
- Wave-Front-Arrays
- (like systolic arrays, but without common timing pulse)
- Very Long Instruction Word (VLIW)

### SPMD (same program, multiple data)

- "Cross" between SIMD and MIMD
- Closer to MIMD-Programming style in machine classification
- Only one logical flow of control, but mostly independent processors
- Concept used in Connection Machine CM-5

### 1.3 Levels of Parallelism

<table>
<thead>
<tr>
<th>Plane</th>
<th>Processing unit</th>
<th>Example system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program level</td>
<td>Job, Task</td>
<td>Multi-user operating system (e.g. time sharing)</td>
</tr>
<tr>
<td>Procedure level</td>
<td>Process</td>
<td>MIMD system</td>
</tr>
<tr>
<td>Expression level</td>
<td>Instruction</td>
<td>SIMD system</td>
</tr>
<tr>
<td>Bit level</td>
<td>Within Instruction</td>
<td>von-Neumann system (e.g. 16 Bit ALU)</td>
</tr>
</tbody>
</table>

Here are of special interest:
- Procedure level (coarse grained parallelism)
- Expression level (fine grained or massive parallelism)
**Program Level**

- Complete programs run simultaneously (or offset in time)

**Procedure Level**

Multiple sections of a program should execute in parallel.

**Areas of application:**
- Real-time programming control of time-critical processes e.g. power plant
- Process computer simultaneous access of multiple hardware components e.g. robot control
- Generic parallel processing (over)

**Expression Level**

- **Generic parallel processing**
  Splitting of a program into parallel sub-tasks, which are distributed over multiple processors to increase performance.

  - Splitting of a problem into mostly independent sub-tasks
  - Parallel program sub-tasks are called "Processes"
Bit Level

Parallel execution of bit-operations in a single word

Example: 8-Bit-Microprocessor

\[
\begin{array}{cccccc}
0 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 1 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

Bit-parallel execution

Parallel Operations

Data related mathematical notation:
Principal differentiation:
scalar data \(\rightarrow\) sequential processing
vector data \(\rightarrow\) parallel processing

Monadic Operations
a: Scalar \(\rightarrow\) Scalar
\(e.g.\) \(9 \rightarrow 3\)
"square root"
known sequential processing

b: Scalar \(\rightarrow\) Vector
\(e.g.\) \(9 \rightarrow (9,9,9)\)
"Broadcast"
multiplication of a data value

c: Vector \(\rightarrow\) Scalar
\(e.g.\) \((1,2,3,4) \rightarrow 10\)
"adding"
reduction of a vector to a scalar
(Assumption: Vector length is preserved)
"square root" component wise monad. operation
permutation of the vector
"partial sums"; complex vector operation

d: Vector \(\rightarrow\) Vector
i. \(e.g.\) \((1,4,9,16) \rightarrow (1,2,3,4)\)
ii. \(e.g.\) \((1,2,3,4) \rightarrow (2,4,3,1)\)
(iii. \(e.g.\) \((1,2,3,4) \rightarrow (1,3,6,10)\)
parallel combination of two vectors (component-wise)

Dyadic Operations (for scalar operands)
e: (Scalar, Scalar) \(\rightarrow\) Scalar
\(e.g.\) \((1,2) \rightarrow 3\)
"Sum"
known sequential processing

f: (Scalar, Vector) \(\rightarrow\) Vector
\(e.g.\) \((3, (1,2,3,4)) \rightarrow (4,5,6,7)\)
"Addition of a scalar"
component-wise application of dyadic operation

Identical to:
\(3 \rightarrow (3,3,3)\)
\(+ (1,2,3,4)\)
\(\rightarrow (4,5,6,7)\)

"vector addition"

vi. \(e.g.\) \((1,2,3,4) \rightarrow (1,3,6,10)\)
parallel combination of two vectors (component-wise)

Example: Dot Product
\[
l \cdot h
\]
\((1,2,3), (4,2,1)) \rightarrow (4,4,3) \rightarrow 11\)
component-wise vector reduction
multiplication

Example: Vector Product (Cross Product)
\((ax, ay, az), (bx,by,bz)) \rightarrow \ldots \rightarrow (ay bz - az by, az bx - ax bz, ax by - ay bx)\)
via multiple intermediate results (vector-memory is required)
1.4 Parallel Processing Concepts

Looking at programming concepts for utilizing parallel hardware
• Coroutines
• Fork & Join
• Parbegin-Parend
• Process
• Server-Client
• Implicit

Coroutines

• Not "real" parallel processing
• Single-Processor-Model
• Modula-2
• Controlled allocation / release of "Fuel" Processor
• Explicit definition of Target-Coroutine:
  \[ \text{TRANSFER(VAR Source, Destination: ADDRESS)} \]

Fork and Join

• "fork" generates a new parallel process
• "wait" waits for a process to finish
• C / UNIX
• Implementation: usually multi-tasking on 1 processor
• Mixing of declaration and synchronization
• Not a good solution in regards to software-engineering
Fork and Join

Example:

```c
int status;
if (fork() == 0) execlp ("program_B",...);
   /* Child-Process */
   ...
   wait(&status);
   /* Parent-Process */
```

Parbegin and Parend

Definition of block whose instructions are to be executed in parallel
- Similar to "par" in Algol 68
- Only primitive synchronization possible
- Parallel operations are missing

Processes

- Explicit declared and synchronized processes
- Concurrent Pascal, Modula-P
- Processes are only started at the start of the program system
- Usually the number of process remains constant during program execution

- Explicit synchronization of processes via:
  - Semaphores
  - Monitors
  - Message passing

Processes

- Frequent program errors during synchronization (protection of critical section / deadlock)
- Hardware model: SISD (sequential)
  - Semaphores / Monitors / (Message passing)
- Hardware-Model: MIMD
  - with "shared memory"
    - Semaphores / Monitors / (Message passing)
  - without "shared memory"
    - Message passing
Processes

- External **scheduler** transfers control between tasks
- Use of time slices
- No need for `OSReschedule()`

---

Server-Client

- Distribution of task blocks to other processes
- Remote-Procedural-Call
- MIMD-model without shared memory

---

Server-Client

Wait / no-wait with task delegation:

- **wait**: wait until server has finished task and may have returned result
  - little parallelism
- **no-wait**: returned parameters are only available later
  - higher degree of parallelism
- Error tolerant protocols for potential failure of server are required

---

Implicit Parallelism

Automatic management of parallel tasks by the system
- APL ("A Programming Language") is data parallel, but has totally inadequate control structures
- Automatic parallelizing of vector- and matrix operations

\[ C := A + B \]

1 processor per matrix element
Explicit Parallelism

- Programmer has total control over parallel processes
- Efficient program execution (depends on programmer, may require specialist knowledge)
- Difficult and error prone programming
- Mostly procedural programming languages (Exception: LISP)

Implicit Parallelism

- Programmer is removed from the management tasks of parallel processes
- Frequently inefficient program execution
- Easy programming, less error prone
- Mostly non-procedural programming languages or parallelizing/vectorizing compilers for procedural languages (e.g., Fortran)

1.5 Network Structures

Parallel system with n PEs (processing elements)

Costs:

- a) Number of connection per PE
  Production costs
- b) Distance between two PEs (minimal, maximal, average)
  Operating costs

Classes:

- a) Bus Systems
- b) Networks with switches
- c) Point-to-Point connection structure

Bus Systems

Distance: always 2 (PEi → bus → PEj)

- Parallel read is possible, parallel write is not.
- Not scalable
- Not usable for parallel computers with order of magnitude of more than 10 processors

Networks with Switches

Networks that contain active switching elements for reconfiguring network / re-routing

Examples:
- Crossbar Switch
- Delta Network
- Clos Network
Crossbar Switch

Cost: $n^2$ nodes (n* (n−1) if no connection along main diagonals)

Advantage: every permutation can be programmed
Disadvantage: Costs = $n^2$

Delta Network

Generic:

Switching Example:

Inputs  1  2  3  4  5  6  7  8

Outputs  1  2  3  4  5  6  7  8

Advantage: Costs $\approx n/2 \cdot \log_2{n}$
Disadvantage: Not every permutation can be programmed (Blocking is possible, Comparison: telephone net)

Clos Network

Clos network is a network made out of patches of crossbar switches
Every permutation has to be programmable $\Rightarrow$ no blocking

- Implementation: multi-stage crossbar-switch with minimal costs.
- Used in MasPar MP-1

1. Stage
$a$ crossbars with each $m : 2m-1$

2. Stage
$2m-1$ crossbars with each $a : a$

3. Stage
$a$ crossbars with each $2m-1 : m$
Clos Network

Example:

For a three-stage Clos-Net with 1,000 Inputs and Outputs we have:

\[ m = \sqrt{(1000/2)} \approx 22.4 \]

As an approximation we choose:

\[ m = 20 \quad \Rightarrow \quad a = N/m = 50 \]

The total number of nodes is:

\[ C = 39 \times (1,000,000/400 + 2,000) \]
\[ = 175,500 \]

The number of required nodes is considerably lower than the number that would be needed for a complete crossbar-switch:

\[ C_{CS} = N^2 = 1,000,000 \]

In this case the Clos-Net has a saving of 82% of nodes when compared to the crossbar.

Fat Tree

- Almost optimal structure (proof by Leiserson)
- Removal of disadvantages of binary tree structures:
  - Root (and inner nodes) are “bottlenecks” in binary tree
- Used in KSR-1 (Kendall Square) and CM-5 (Thinking Machines)
Point-to-Point Networks

i) Ring

\[ V = 2 \]  (good)
\[ A = \frac{n}{2} \]  (bad)

ii) Complete Graph

\[ V = n-1 \]  (bad)
\[ A = 1 \]  (good)

Point-to-Point Networks

iii) Grid / Torus

- Quadratic Grid (4-way nearest neighbor)
  - \[ V = 4 \]
  - \[ A = 2 \cdot \sqrt{n} - 2 \]

- Quadratic Torus (8-way nearest neighbor)
  - \[ V = 8 \]
  - \[ A = \sqrt{n} - 1 \]

Point-to-Point Networks

iv) Cubic Grid

\[ V = 6 \]
\[ A = 3 \cdot \sqrt[3]{n} \]

Point-to-Point Networks

v) Hexagonal Grid

- PEs on corners of honeycomb
  - \[ V = 3 \]
  - \[ A = 2 \cdot \sqrt{n} \]

- PEs in center of honeycomb
  - \[ V = 6 \]
  - \[ A = 2 \cdot \sqrt{n} \]

vi) Hypercube

\[ n = 2^m \]
\[ V = \log_2 n \]
\[ A = \log_2 n \]
Point-to-Point Networks

vii) Binary Tree

\[ V = 3 \]
\[ A = 2 \times \log_2 n \]

Disadvantage:
Almost all messages have to pass via the root or overloaded nodes.

Point-to-Point Networks

viii) Quadtree

\[ n = \frac{1}{3}(4^m - 1) \]
\[ V = 5 \]
\[ A = 2 \times \log_4 (3n+1) - 2 \approx 2 \times \log_4 n \]

Mostly used in image processing

Point-to-Point Networks

ix) Perfect Shuffle

\[ n = 2^m \]
\[ V = 2 \quad (1 \text{ bi-directional and } 2 \text{ uni-directional connection}) \]
\[ A = 2 \times \log n \]

Operations for mapping PE-Number into binary notation:

Shuffle \( (p_n, ..., p_1) = (p_{n-1}, ..., p_1, p_n) \)  "Rotation left"

Exchange \( (p_n, ..., p_1) = (p_n, ..., p_2, \neg p_1) \)  "Negation of lowest bits"

Examples:

a) \( 001 \)  sh. \( \rightarrow \) \( 010 \)  sh. \( \rightarrow \) \( 100 \)  sh. \( \rightarrow \) \( 001 \)

b) \( 011 \)  ex. \( \rightarrow \) \( 010 \)  ex. \( \rightarrow \) \( 011 \)
Point-to-Point Networks

x) Plus-Minus-2\(^i\)-Network (PM2I)

Let \(n=2^m\), then the PM2I-Network consists of \(2^m - 1\) structures according to the following rules:

\[
\begin{align*}
PM_{+i}(j) &= (j + 2^i) \mod n \\
PM_{-i}(j) &= (j - 2^i) \mod n
\end{align*}
\]

with: \(PM_{+(m-1)} \equiv PM_{-(m-1)}\)

Comparison of Networks

A comparison cannot just be based on \(V\) and \(A\), but it also highly depends on the particular parallel application.

Examples:

Connection Machine: Grid + Hypercube
- universally good

MasPar: Grid + "Router" (Clos-Network)
- universally good

Distributed Array Processor (DAP): Quadratic Grid
- good for image processing

The following table lists the number of requires simulation steps (transfer steps) vs the number of PEs \(n\). (according to H. J. Siegel)

<table>
<thead>
<tr>
<th></th>
<th>grid</th>
<th>PM2I</th>
<th>Perfect Shuffle</th>
<th>Hypercube</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid</td>
<td>(\approx \sqrt{n}/2)</td>
<td>(\approx \sqrt{n})</td>
<td>(\approx \sqrt{n})</td>
<td>(\sqrt{n})</td>
</tr>
<tr>
<td>PM2I</td>
<td>1</td>
<td>–</td>
<td>(\approx \log_n n)</td>
<td>2</td>
</tr>
<tr>
<td>Perfect Shuffle</td>
<td>(\approx 2\log_n n)</td>
<td>(\approx 2\log_n n)</td>
<td>–</td>
<td>(\log_n + 1)</td>
</tr>
<tr>
<td>Hypercube</td>
<td>(\log_2 n)</td>
<td>(\log_2 n)</td>
<td>(\log_2 n)</td>
<td>–</td>
</tr>
</tbody>
</table>
2. Petri Nets

A Petri-Network is an aligned bipartite graph with labels.

Nodes:  
Position (can be labeled as:  
“State”

Edges:  
Connects positions and transitions

Activated: A transition $t$ is activated, if all input-positions $p_i$ are labeled by $t$.

Switching: An activated transition $t$ can switch. Then all labels disappear from all input-positions $p_i$ of $t$ and all output-positions $p_j$ of $t$ are getting labeled.

Example:

Before switching

After switching

Bräunl 2004 65

Bräunl 2004 66

Bräunl 2004 67

Bräunl 2004 68
**Indeterminism:** If multiple transitions are active simultaneously, then it is undetermined which one of them will switch first.

Both transitions $t_1$ and $t_2$ can fire; it is undetermined which one will fire. If one of them fired, the other one can no longer fire.

---

**State:** The labeling state (or short State) of a Petri-Net at time $T$ is defined as the collective of all labels of each individual position of the net.

Label State as a bit string:

- $(111\ 00)$
- $(001\ 10)$
- $(100\ 01)$

*Computing* of all possible subsequent states of a Petri-Net

---

**Label generation:** A transition that does not have an Input-edge is always activated and can always generate labels for output-positions connected to it.

- Can always fire

**Label destruction:** A transition that does not have an output-edge and only one input-edge is only activated if the position is labeled, and it can repeatedly destroy a label.

- Can always fire if $P$ is labeled

---

**Dead:** A Petri-Net is dead (blocked), if none of its transitions is activated.

**(Blocking)**

**Alive:** A Petri-Net is alive (not blocked at any time), if at least one of its transitions is activated and this is also the case for every subsequent state.
Petri Nets

Examples:

![Diagram of alive and not alive states](image)

Not alive, blocked after a few steps (e.g. s fires first; result: blocked)

2.1 Petri Nets for Process Synchronization

Mutual exclusion of processes, e.g. for access of common data areas.

2.2 Extended Petri Nets

- Multiple labels for one position
- Negation: t can fire if P1 is labeled and P2 is not labeled
- Edge weighting: When a transition fires, number of labels is decremented/incremented

With this extension Petri-Nets are as powerful as the Turing-Machine.
Extended Petri Nets: Adder

\[ Z := Z + Y \]

Extended Petri Nets: Subtractor

\[
\begin{align*}
\text{if } Z \geq Y & \text{ then } Z := Z - Y \\
\text{else } & \text{ undefined}
\end{align*}
\]

Extended Petri Nets: Multiplier

\[ Z := Z + X \cdot Y \]

Extended Petri Nets: Copying Variables
Sequential Processing

Parallel Processing