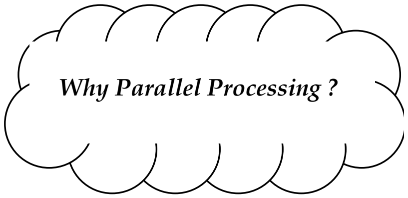
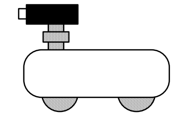
1. Fundamentals



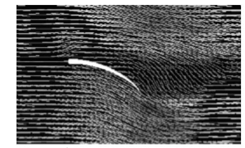
- Increase in performance: ***faster processing***
- Finding solutions to bigger problems
- Physical limits for single processors have “almost“ been reached, or increasing performance with a single processor is only possible at a very high cost
- It is the **natural form** of information processing

Why Parallel Processing?



Scenario 1:

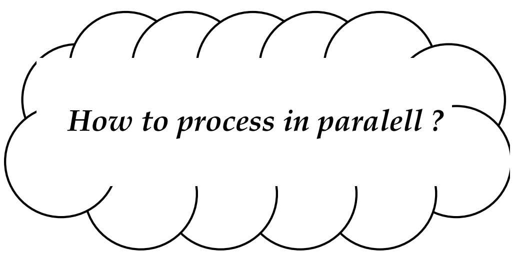
Control of autonomous vehicles via camera data
- fast: cycle time 40ms
  (actually 20 ms at 50Hz, but only half-pictures are transmitted)
- asynchronous: reaction to events



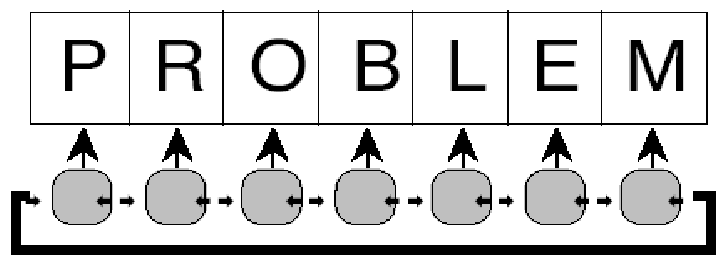
Scenario 2:

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 do Parallel Processing?

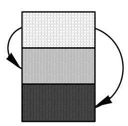


How to do Parallel Processing?

NOTE: parallelization ≠ faster processing

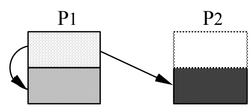
> "**Merely parallelizing a program does not necessarily improve its performance**:
> Communication overhead can more than offset any reduction in elapsed time ..."
> Scott Burleigh, JPL 1993

<u>Sequential:</u>



data exchange
here ≈ cost free

<u>Parallel:</u>



data exchange
here ≈ cost free   here **very expensive**

How to do Parallel Processing?

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)

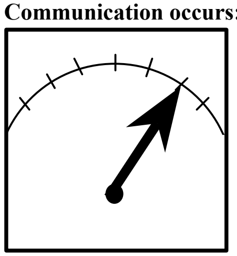
How to do Parallel Processing?

2. Relative Communication Effort

Parallel Program

Frequency Ratio:
Communication op. vs arithmetic op.

Communication occurs:

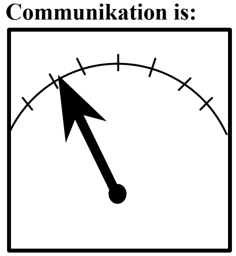


rare   frequent

Parallel Computer

Cost Ratio:
Commun. op. vs arithmetic-Op.

Communikation is:



cheap   expensive

How to do Parallel Processing?

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)
  Network speed = *f* (latency, bandwidth)
- **Only performance counts** in practice (time behaviors) !
- **Algorithmically different programs with MIMD and SIMD for the *same problem* !!**

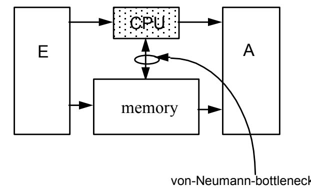
Where is Parallel Processing?

Parallelism is everywhere

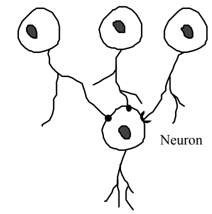
- Processes in nature / society / technical processes
- Even in a simple PC: E/A, DMA, Microcode, 16-bit-arithmetic
- Parallel processing is the natural form of processing information.
- Sequential processing often is an artificial restriction, brought about by historical factors (success of the sequential 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?

<u>von-Neumann-Computer</u>
sequential



<u>human brain</u>
parallel



Disproportion:

1 CPU, always active (10^6 transistors)	10^{10} neurons ("CPU + memory")
10^{10} memory elements, mostly inactive (1GB ≈ 10^{10} mem. Cells = 10^{10} transistors)	always active

Switching times:

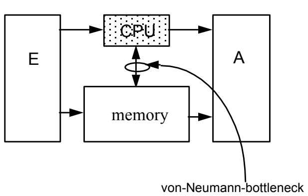
10GHz with 10 sub-cycles → 10ps = 10^{-11} s	1kHz → 1ms = 10^{-3} s

Switching-processes/sec total:

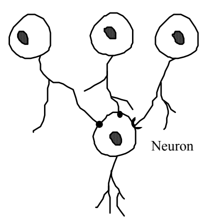
10^{20} / s (theoretical) or 10^{12} / s (practical)	10^{13} / s

Where is Parallel Processing?

<u>von-Neumann-Computer</u>
sequential



<u>human brain</u>
parallel



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

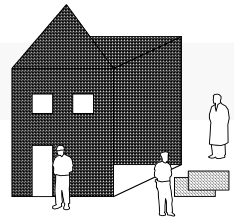
1.2 Classification

Single processor computer →	SISD Single Instruction Single Data	SIMD Single Instruction Multiple Data	← Array computer (Vector computer)
Pipeline computer →	MISD Multiple Instruction Single Data	MIMD Multiple Instruction Multiple Data	← Multicomputer

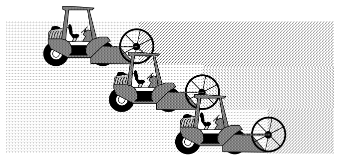
System classification by Flynn

Differentiation of MIMD:
- Coupling via shared memory
- Coupling via network connections (message passing)

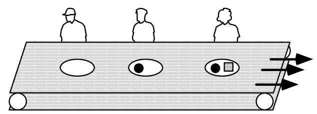
MIMD
everyone does something different
coordination



SIMD
everyone does the same
timing pulse



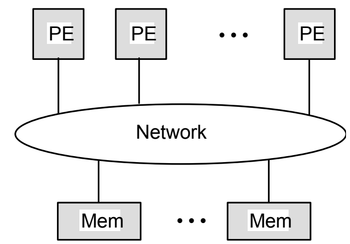
Conveyor Belt **(Pipeline, MISD)**
everyone repeats individual tasks,
everyone has different job
overlapping executions



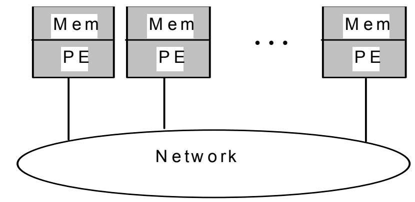
MIMD (multiple instruction, multiple data)

- More generic structure
- **Asynchronous**

i. With shared memory



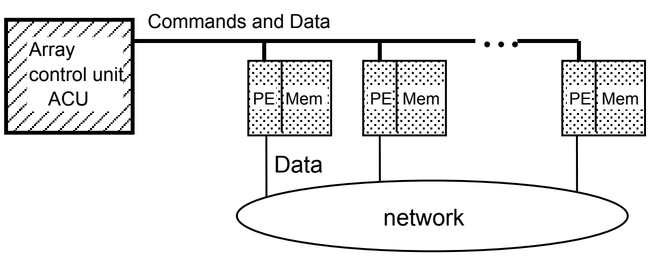
ii. Without shared memory
(local memory in PEs)



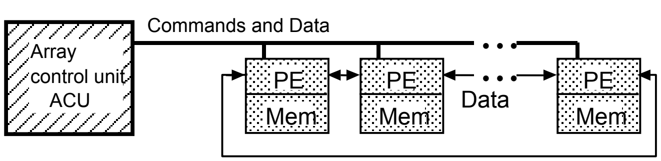
SIMD (single instruction, multiple data)

- Simpler structure
- **Synchronous**

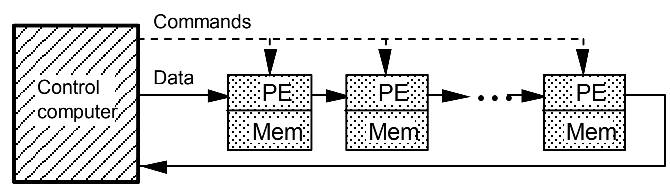
Array Computer



Vector Computer



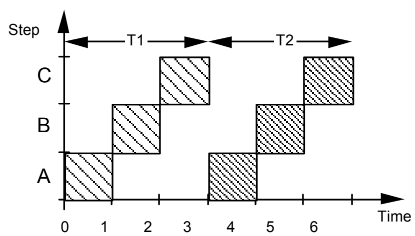
Pipeline



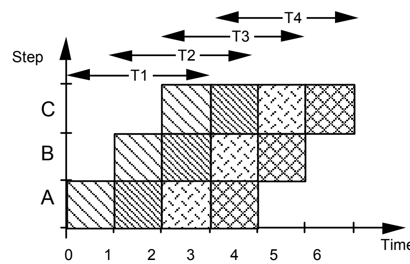
Pipeline

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

Sequential

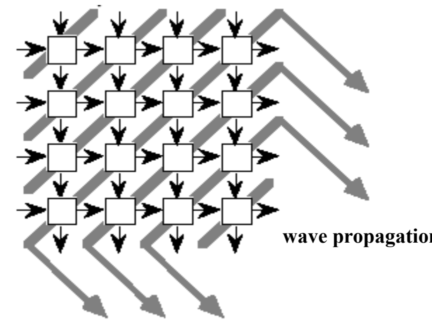


Parallel



Hybrid Parallel Systems

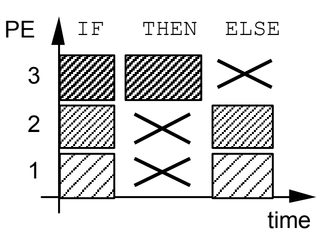
- Multi-Pipeline
- Multiple-SIMD
- Systolic Arrays
- Wave-Front-Arrays
- (like systolic arrays, but without common timing pulse)
- Very Long Instruction Word (VLIW)



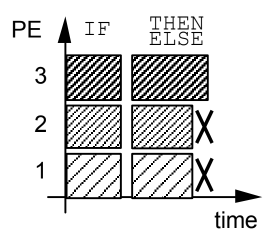
SPMD (same program, multiple data)

- "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

SIMD:



SPMD:



1.3 Levels of Parallelism

Coare grained

↕

Fine grained

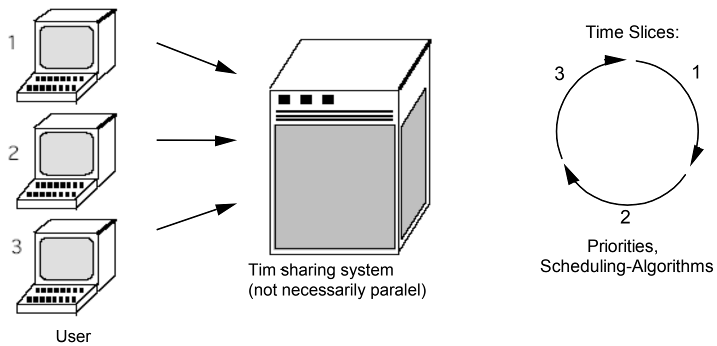
Plane	Processing unit	Example system
Program level	Job, Task	Multi-user operating system (e.g. time sharing)
Procedure level	Process	MIMD system
Expression level	Instruction	SIMD system
Bit level	Within Instruction	von-Neumann system (e.g. 16 Bit ALU)

Here are of special interest:

- Procedure level (coarse grained parallelism)
- Expression level (fine grained or massive parallelism)

Program Level

- Complete programs run simultaneous (or offset in time)

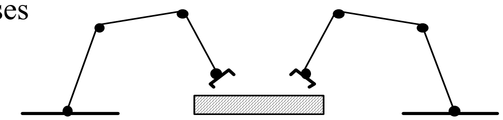


Procedure Level

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)

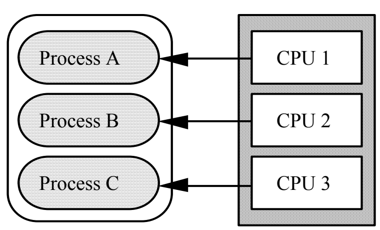


Procedure Level

- **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"



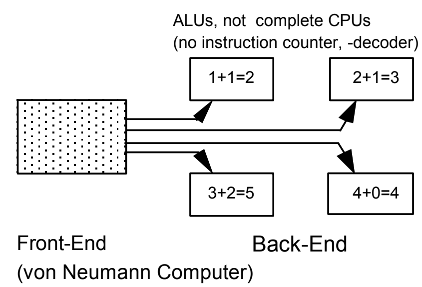
Expression Level

Arithmetic expressions are executed in parallel at component level

- Vectorization
- Data parallelism

<u>Example:</u>

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 1 & 1 \\ 2 & 0 \end{pmatrix}$$



Bit Level

Parallel execution of bit-operations in a single word

<u>Example:</u> 8-Bit-Microprocessor

	0	1	0	1	1	1	0	1
AND	1	1	0	1	1	0	0	0
	0	1	0	1	1	0	0	0

Bit-parallel execution

Parallel Operations

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

Monadic Operations

a: Scalar → Scalar
e.g. 9 → 3 — "square root"; known sequential processing

b: Scalar → Vector
e.g. 9 → (9,9,9,9) — "Broadcast"; multiplication of a data value

c: Vector → Scalar
e.g. (1,2,3,4) → 10 — "adding"; reduction of a vector to a scalar

d: Vector → Vector — (Assumption: Vector length is preserved)
i. e.g. (1,4,9,16) → (1,2,3,4) — "square root" component wise monad. operation
ii. e.g. (1,2,3,4) → (2,4,3,1) — permutation of the vector
(iii. e.g. (1,2,3,4) → (1,3,6,10) — "partial sums"; complex vector operation)

Parallel Operations

Dyadic Operations (for scalar operands)

e: (Scalar, Scalar) → Scalar
e.g. (1,2) → 3 — "Sum"; known sequential processing

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

Identical to: 3 → (3,3,3,3) multiplication *(b)*
+ <u>(1,2,3,4)</u> vector addition *(g)*
(4,5,6,7)

g: (Vector, Vector) → Vector
e.g. ((1,2,3,4), (0,1,3,2)) → (1,3,6,6) — "Addition"; parallel combination of two vectors (component-wise)

Parallel Operations

<u>Example:</u> Dot Product

$$((1,2,3), (4,2,1)) \xrightarrow{l} (4,4,3) \xrightarrow{h} 11$$

component-wise multiplication — vector reduction

<u>Example:</u> Vector Product (Cross Product)

((ax, ay, az), (bx,by,bz)) → . . . → (ay bz - az by, az bx - ax bz, ax by - ay bx)

via multiple intermediate results (vector-memory is required)

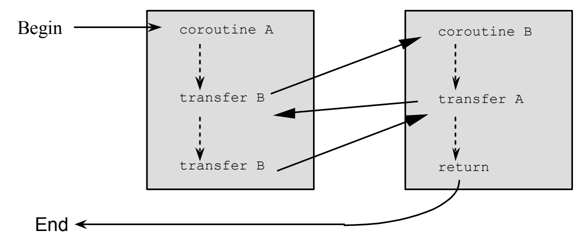
1.4 Parallel Processing Concepts

Looking at programming concepts for utilizing parallel hardware

- Coroutines
- Fork & Join
- Parbegin-Parend
- Process
- Server-Client
- Implicit

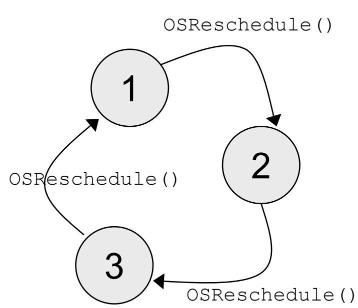
Coroutines

- Not "real" parallel processing
- Single-Processor-Model
- Modula-2
- Controlled allocation / release of "Fuel" Processor
- Explicit definition of Target-Coroutine:
  `TRANSFER(VAR Source, Destination: ADDRESS)`



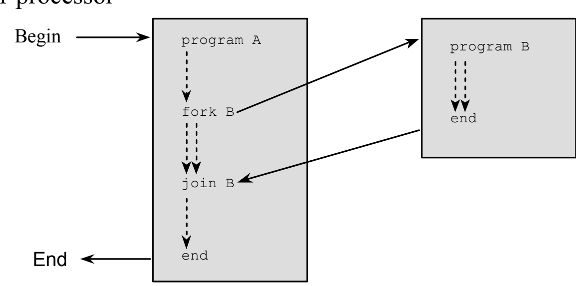
Coroutines

- Each parallel thread decides itself when to relinquish control
- All threads must be *well behaved*
- `OSReschedule()`

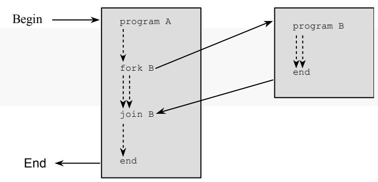


Fork and Join

- "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:

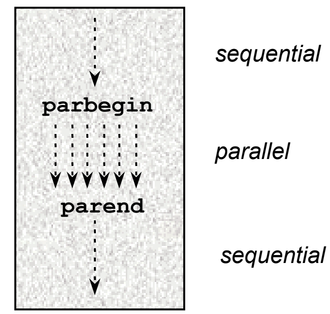
```
int status;
if (fork() == 0) execlp ("program_B",...);
                        /* Child-Process */

/* Parent-Process */
...
wait(&status);
```


Parbegin and Parend

Definition of block whose instructions are to be executed in parallel

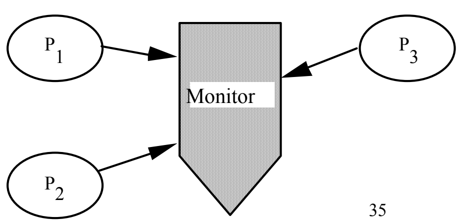
- Similar to "par" in Algol 68
- Only primitive synchronization possible
- Parallel operations are missing



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

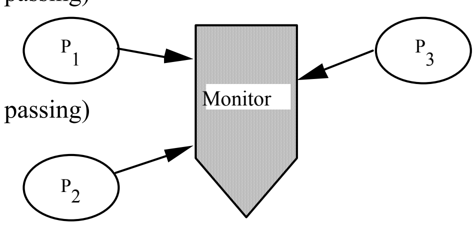


Processes

- Frequent program errors during synchronization (protection of critical section / deadlock)

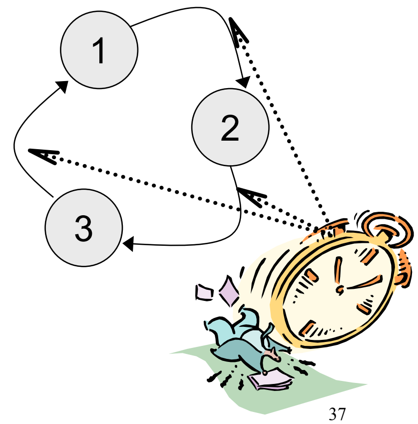
- Hardware model: SISD (sequential)
  Semaphores / Monitors / (Message passing)

- Hardware-Model: MIMD
  - with "shared memory“
    Semaphores / Monitors / (Message passing)
  - without "shared memory“
    Message passing



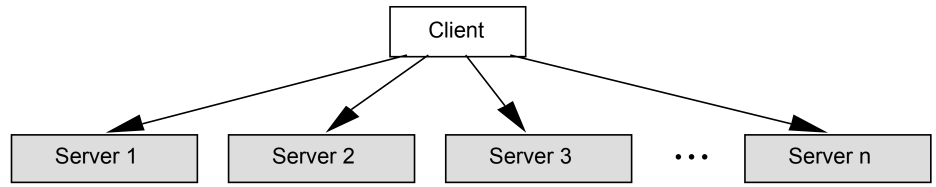
Processes

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



Server-Client

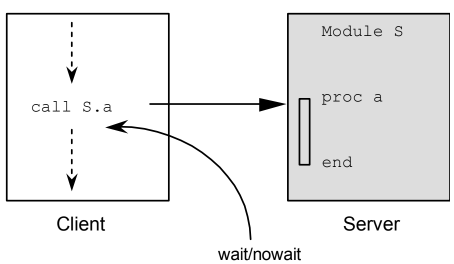
- Distribution of task blocks to other processes
- Remote-Procedure-Call
- MIMD-model without shared memory



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

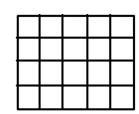


Implicit Parallelism

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

$C := A + B \Longrightarrow$



1 processor per matrix element

Matrix of PEs

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

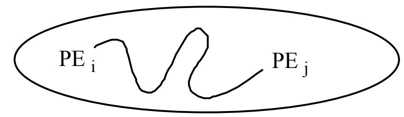
- Programmer is removed from the management tasks of parallel processes
- Frequently inefficient program execution
- Easy programming, less error prone
- Mostly non-procedural programming languages or parallelizing/vectorizing compilers for procedural languages (e.g. Fortran)

1.5 Network Structures

Parallel system with n PEs (processing elements)

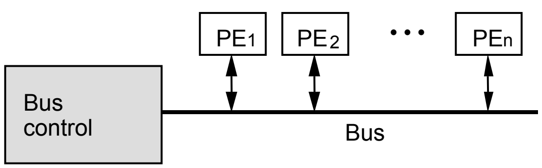
Costs: a) Number of connection per PE
Production costs

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 ($PE_i \rightarrow$ bus $\rightarrow PE_j$)

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

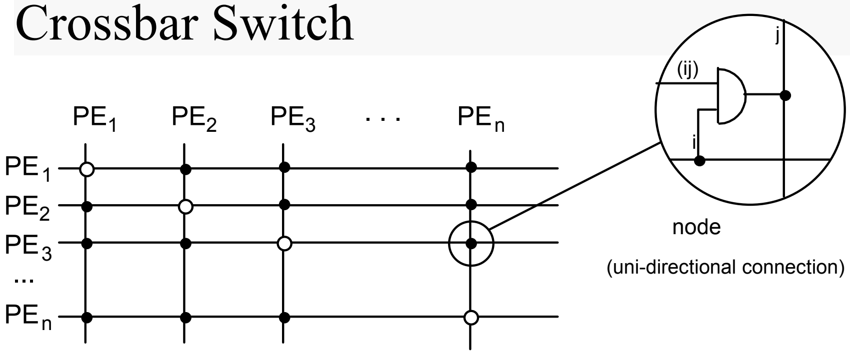
Networks with Switches

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

Examples:

- Crossbar Switch
- Delta Network
- Clos Network

Crossbar Switch

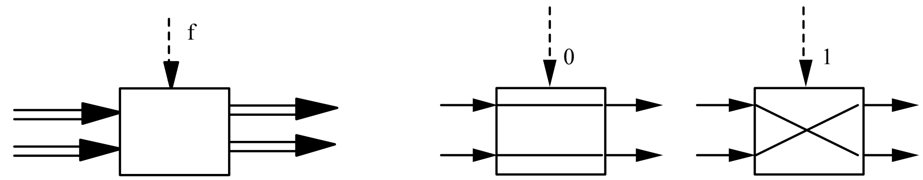


Cost: n^2 nodes (n* (n–1) if no connection along main diagonals)

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

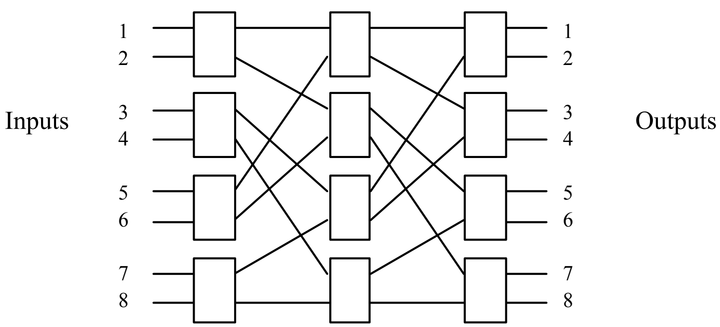
Delta Network

Generic: Switching Example:



Delta Network

Example: 8 × 8 Delta-Network



Advantage: Costs ≈ $n/2 * \log_2 n$
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 ⇒ no blocking

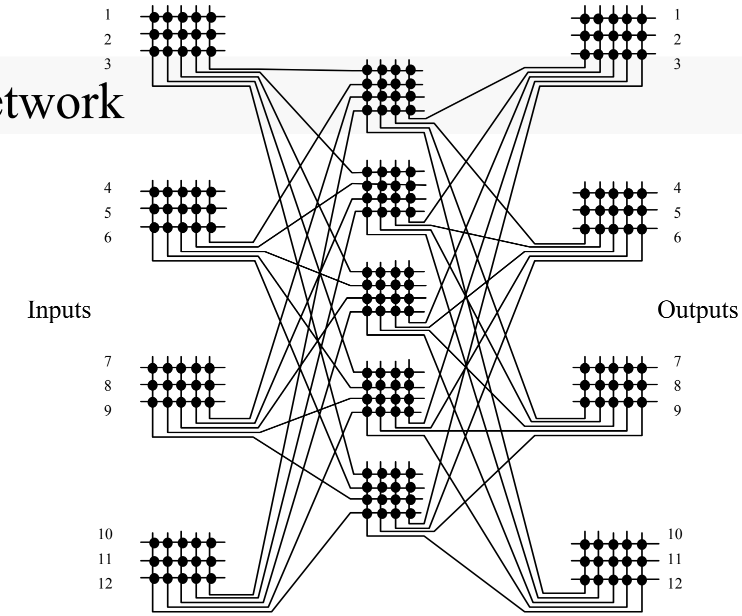
- Implememtation: multi-stage crossbar-switch with minimal costs.
- Used in MasPar MP-1

<u>1. Stage</u>
a crossbars with each
m : 2m-1

<u>2. Stage</u>
2m-1 crossbars with each
a : a

<u>3. Stage</u>
a crossbars with each
2m-1 : m

Clos Network



Clos Network

Clos-Network is approximately minimal, if m ≈ √(N/2)
(for N ≥ 24, only then then is a Clos-Network cheaper than a simple crossbar-switch)

Calculate total number of crosspoints C in Clos Network:

C = m*(2*m - 1) * **a** + a² * (**2*m - 1)** + (2*m - 1)*m * **a**
= m*(2*m - 1) * 2*a + a² * (2*m - 1)

(2*a) crossbars with "m : (2*m-1)" [Stage 1 and 3]
plus
(2*m-1) crossbars with "a : a" [Stage 2]

C = (2*m - 1) * (N²/m² + 2*N)

Clos Network

Example:

For a three-stage Clos-Net with 1.000 Inputs and Outputs we have:
m ≈ √(1000/2) ≈ 22.4

As an approximation we choose:
m = 20 ⇒ a = N/m = 50

The total number of nodes is:
C = 39 * (1,000,000/400 + 2,000)
= 175,500

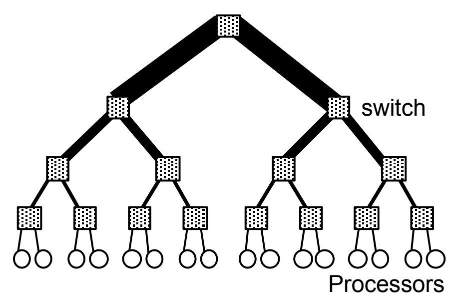
The number of required nodes is considerably lower than the number that would be needed for a complete crossbar-switch:

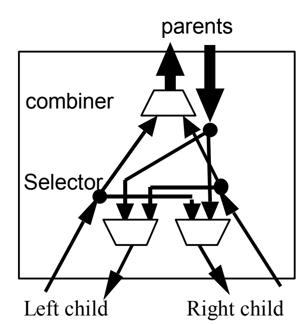
C_{CS} = N² = 1,000,000

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)

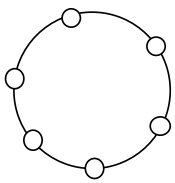




Point-to-Point Networks

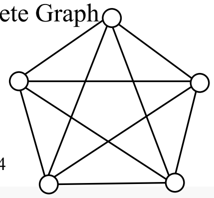
n	=	Number of PEs in Network
V	=	Connections for each PE
A	=	Maximum distance between two PEs

i) Ring



V = 2 (good)
A = n/2 (bad)

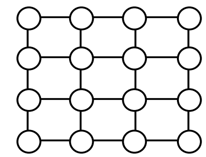
ii) Complete Graph



V = n–1 (bad)
A = 1 (good)

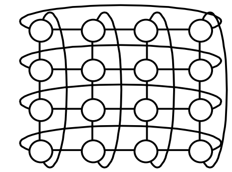
Point-to-Point Networks

iii) Grid / Torus



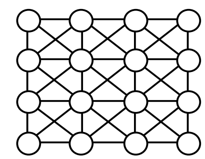
Quadratic Grid
(4-way nearest neighbor)

$V = 4$
$A = 2*\sqrt{n} - 2$
$\approx 2 * \sqrt{n}$



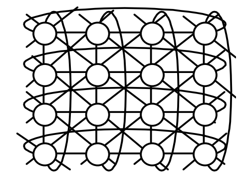
Quadratic Torus
(4-way nearest neighbor)

$V = 4$
$A = \sqrt{n}$



Quadratic Grid
(8-way nearest neighbor)

$V = 8$
$A = \sqrt{n} - 1$
$\approx \sqrt{n}$

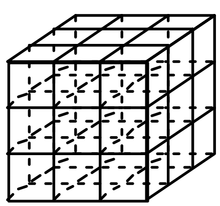


QuadraticTorus
(8-way nearest neighbor)
(diagonal edge connections not drawn)

$V = 8$
$A = \sqrt{n}/2$

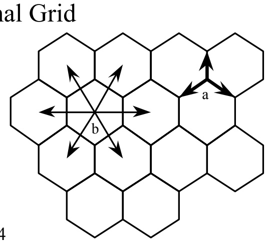
Point-to-Point Networks

iv) Cubic Grid



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

v) Hexagonal Grid



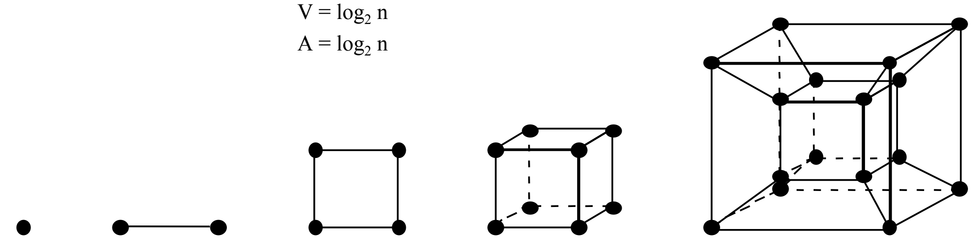
a) PEs on corners of honeycomb
$V = 3$
$A \approx 2 * \sqrt{n}$

b) PEs in center of honeycomb
$V = 6$
$A \approx 2 * \sqrt{n}$

Point-to-Point Networks

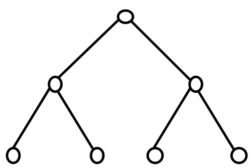
vi) Hypercube

$n = 2^m$
$V = \log_2 n$
$A = \log_2 n$



Point-to-Point Networks

vii) Binary Tree

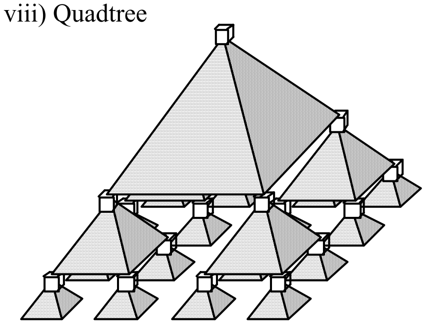


V = 3
A = 2 * $\log_2$ n

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

Point-to-Point Networks

viii) Quadtree

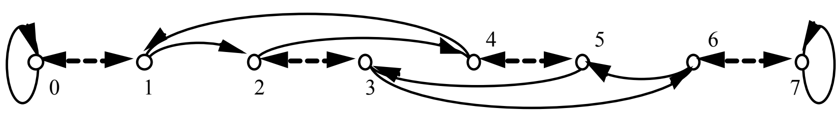


n = 1/3(4^m – 1)
V = 5
A = 2 * $\log_4$ (3*n+1) – 2
≈ 2 * $\log_4$ n

Mostly used in image processing

Point-to-Point Networks

ix) Perfect Shuffle



n = 2^m

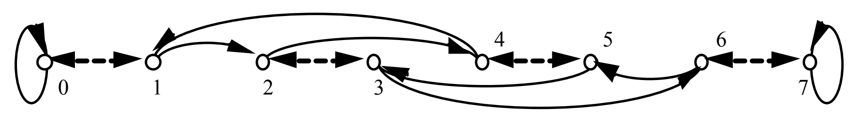
V = 2 (1 bi-directional and 2 uni-directional connection)
A = 2 * log n

Operations for mapping PE-Number into binary notation:

Shuffle	$(p_n, \ldots, p_1) = (p_{n-1}, \ldots, p_1, p_n)$	"Rotation left"
Exchange	$(p_n, \ldots, p_1) = (p_n, \ldots, p_2, \sim p_1)$	"Negation of lowest bits"

Point-to-Point Networks

ix) Perfect Shuffle



<u>Examples:</u>

a) 001 $\xrightarrow{\text{sh.}}$ 010 $\xrightarrow{\text{sh.}}$ 100 $\xrightarrow{\text{sh.}}$ 001

b) 011 $\xrightarrow{\text{ex.}}$ 010 $\xrightarrow{\text{ex.}}$ 011

Point-to-Point Networks

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

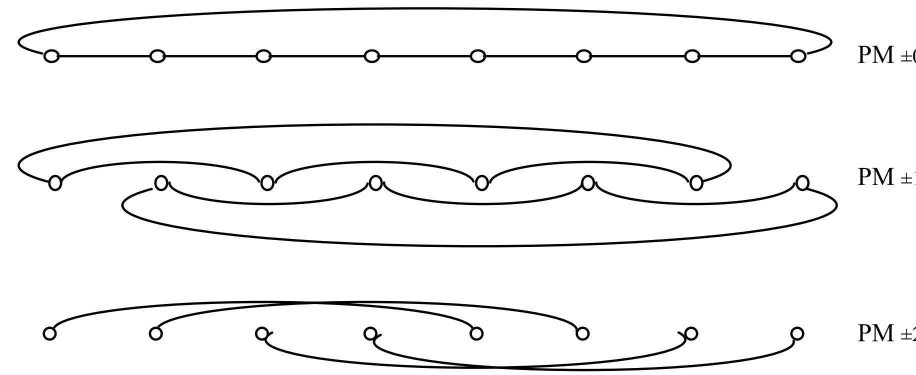
Let $n=2^m$, then the PM2I-Network consists of $2*m - 1$ structures according to the following rules:

$$PM_{+i}(j) = (j + 2^i) \bmod n$$
$$PM_{-i}(j) = (j - 2^i) \bmod n$$

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

Point-to-Point Networks

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



V = 2 * log n - 1
A = log n - 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

Comparison of Networks

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

Net 1 → simulates Net 2 ↓	grid	PM2I	Perfect Shuffle	Hypercube
grid	–	$\approx \sqrt{n}/2$	$\approx \sqrt{n}$	$\sqrt{n}$
PM2I	1	–	$\approx \log_2 n$	2
Perfect Shuffle	$\approx 2*\log_2 n$	$\approx 2*\log_2 n$	–	$\log_2 n + 1$
Hypercube	$\log_2 n$	$\log_2 n$	$\log_2 n$	–

2. Petri Nets

A Petri-Network is an aligned <u>bipartite</u> <u>graph</u> with <u>labels</u>.

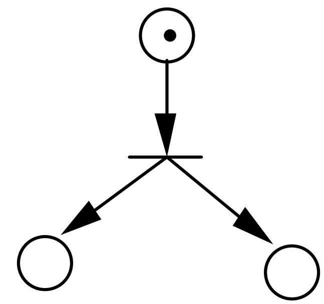
Nodes: Position (can be labeled as:) "State"

Transition "Action"

Edges: Connects positions and transitions

Petri Nets

Example:



Petri Nets

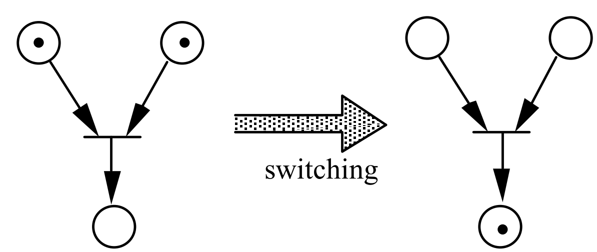
Definitions

Activated: A transition *t* is activated, if all Input-positions *pi* are labeled by *t*. *(state)*

Switching: An activated transition *t* can switch. Then all labels disappear from all input-positions *pi* of *t* and all output-positions *pj* of *t* are getting labeled. *(process)*

Petri Nets

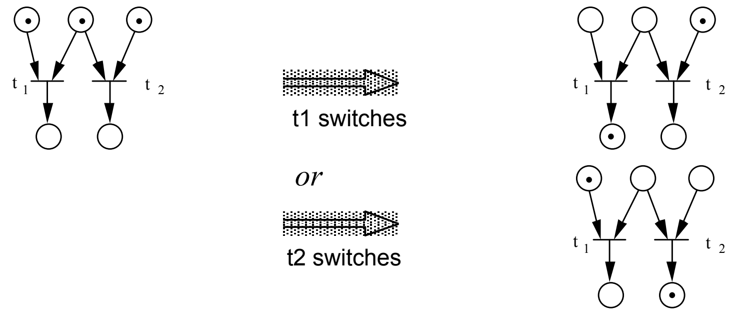
Example:



before switching after switching

Petri Nets

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



Both transitions t1 and t2 can fire; it is undetermined which one will fire.
If one of them fired, the other one can no longer fire.

Petri Nets

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.

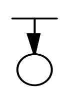
Label State as a bit string:

(111 00) → (001 10)
(111 00) → (100 01)

➨ „Computing“ of all possible subsequent states of a Petri-Net

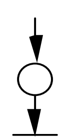
Petri Nets

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

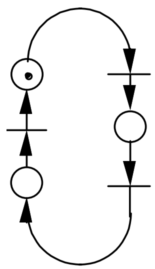
Petri Nets

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

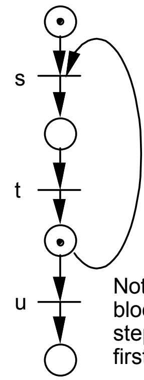
Alive: A Petri-Net is alive (not blocked at any time), if at least one of its transitions is activated and this is also the case for every subsequent state.

Petri Nets

Examples:

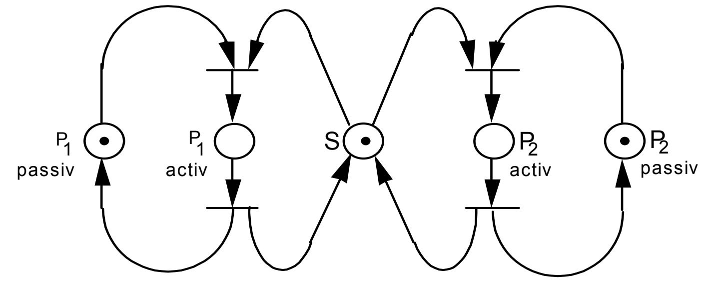


alive



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

2.1 Petri Nets for Process Synchronization



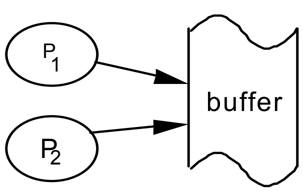
S = Semaphor

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

Petri Nets for Process Synchronization

Example: P1 generates data independent of P2 and writes them into a buffer area, then continues in parallel without waiting. P2 reads data from buffer and processes in parallel with P1.

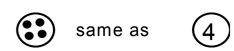
- Simultaneous access must be prevented.
- Synchronization: one process may have to wait for a short time



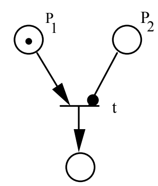
Queue or stack
(unlimited size)

2.2 Extended Petri Nets

i.) Multiple labels for one position

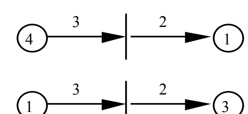


ii.) Negation



t can fire if P1 is labeled
and P2 is not labeled

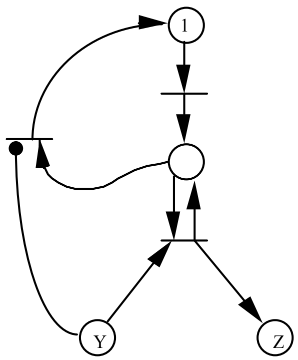
iii.) Edge weighting



When a transition fires:
Number of labels is decremented / incremented

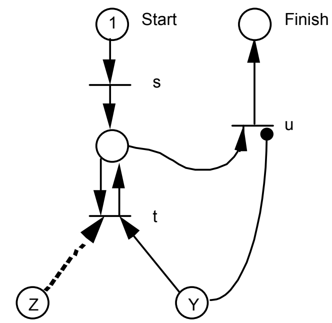
With this extension Petri-Nets are as powerful as the Turing-Machine.

Extended Petri Nets: Adder

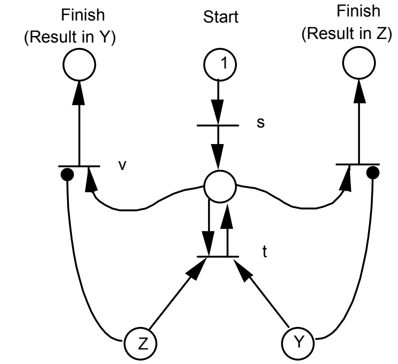


Z := Z + Y

Extended Petri Nets: Subtractor



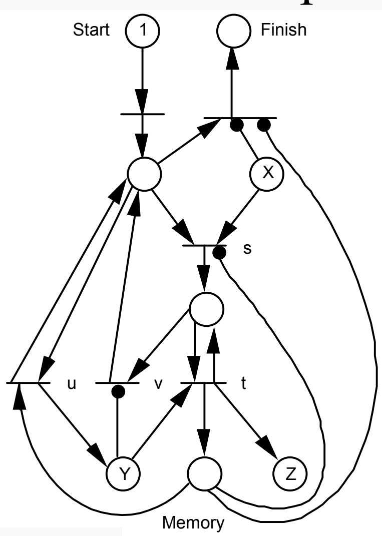
if Z>=Y then Z :=Z-Y
else *undefined*



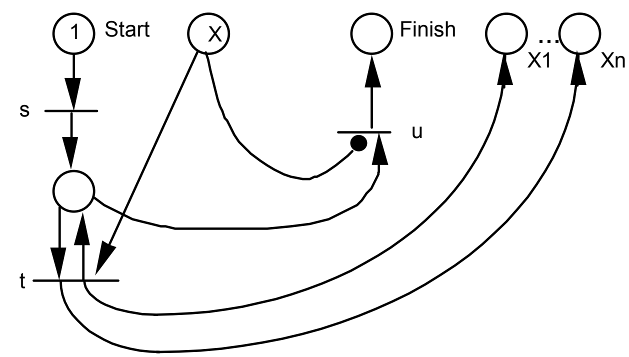
if Z>=Y then Z := Z-Y
if Y>=Z then Y:= Y-Z

Extended Petri Nets: Multiplier

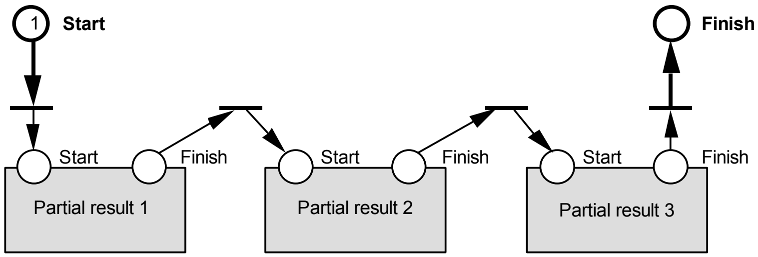
Z := Z + X * Y



Extended Petri Nets: Copying Variables



Sequential Processing



Parallel Processing