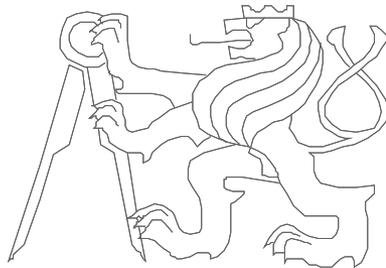# Advanced Computer Architectures

Parallel systems programming concepts, using Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) to create parallel programs.



Czech Technical University in Prague, Faculty of Electrical Engineering
Slides authors: Michal Štepanovský, update Pavel Píša

# Instruction-level parallelism (ILP)

- Parallelism on the lowest level – bit-level parallelism (word width; addition of 64-bit numbers on 32-bit microprocessor, buses, SIMD …)

- Instruction-level parallelism
  - Pipelining – temporal parallelism (<u>sequential</u> instruction flow)
  - Superscalar execution  (in a broader sense) – spatial parallelism

## Pipelining:

- Suppose that instruction execution can be divided to 5 stages

$$\longrightarrow \boxed{IF} \longrightarrow \boxed{ID} \longrightarrow \boxed{EX} \longrightarrow \boxed{MEM} \longrightarrow \boxed{WB} \longrightarrow$$
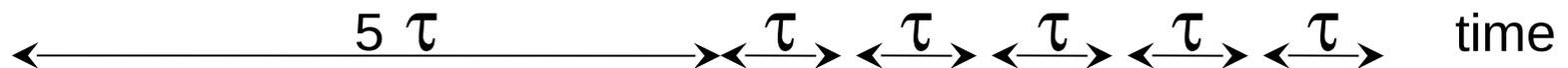
IF – Instruction Fetch, ID – Instr. decode (and Operand Fetch),
MEM – Memory Access, EX – Execute, WB – Write Back

let  $\tau = \max \{ \tau_i \}^k_{i=1}$,   where $\tau_i$ is *propagation delay* of *i*-the *pipeline stage*.

# Instruction-level parallelism – pipelining

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
| IF | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | I9 | I10 |
| ID | | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | I9 |
| EX | | | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 |
| MEM | | | | I1 | I2 | I3 | I4 | I5 | I6 | I7 |
| ST | | | | | I1 | I2 | I3 | I4 | I5 | I6 |
| | | | | | | | | | | |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

$$5\,\tau \qquad \tau \quad \tau \quad \tau \quad \tau \quad \tau \qquad \text{time}$$

- Execution time of $n$ instructions on a $k$-stage pipeline:

$$T_k = k.\tau + (n-1)\,\tau$$

Assumption: ideally-balanced pipeline

- Speedup: $\qquad S_k = \dfrac{T_1}{T_k} = \dfrac{nk\tau}{k\tau + (n-1)\tau} \qquad \lim_{n \to \infty} S_k = k$

# Instruction-level parallelism – pipelining

- Does not reduce execution time of a single instruction; it is usually longer due to interstage registers, etc. in path

- Hazards:
    Structural hazards (solved by duplication),
    Data hazards (the consequence of data dependencies)
    Control hazards (instructions modifying PC)...

- There are situations when it is necessary to stall or flush the pipeline to resolve some hazards which cannot be solved by forwarding or other non-blocking solution.

- Notice: A deeper pipeline (more stages) results in less gates in each stage, which allows to increase the clock frequency. But more stages mean more complex control and forwarding circuitry and higher cost of pipeline flush (instructions have to be better (re)ordered to reach the theoretical speedup)

# Instruction-level parallelism – pipelining + superscalar

Pipelined superscalar execution: N-ways/wide pipeline
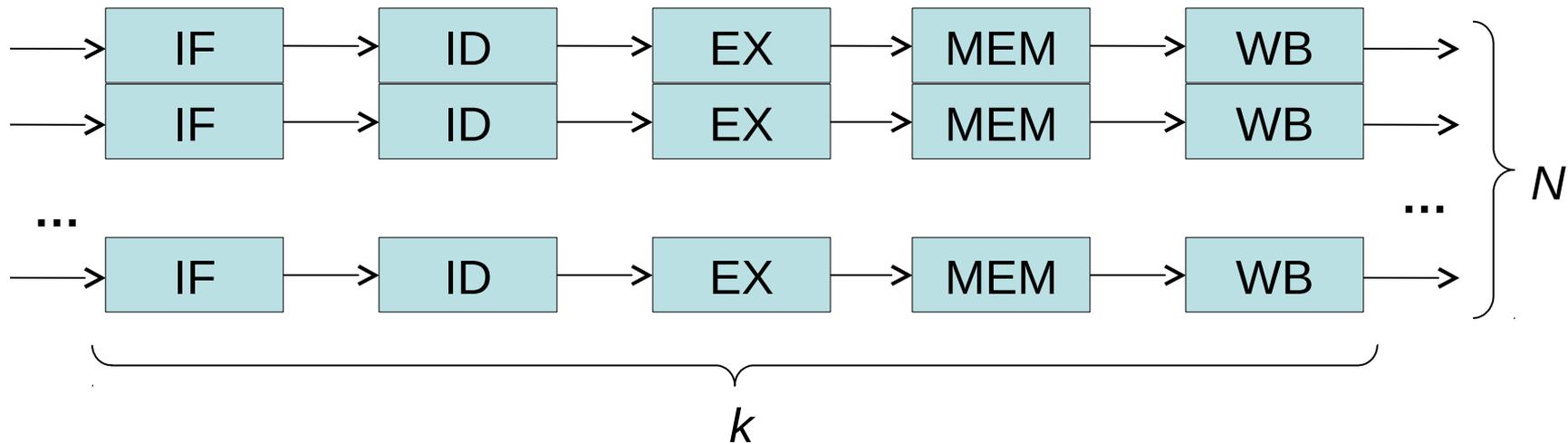


$$S_{k,N} = \frac{T_1}{T_{k,N}} = \frac{nNk\tau}{k\tau + (n-1)\tau} \qquad \lim_{n \to \infty} S_{k,N} = Nk$$

- Sequential instruction flow
- Data dependencies are dynamically identified by hardware (versus by software during compilation → static: WLIV)

# Instruction-level parallelism

Techniques used to achieve and utilize a higher degree of instruction-level parallelism:

- Propagation of results within the pipeline (**forwarding**)
- Instructions **out-of-order execution**
- **Register renaming**
- **Speculative execution**
- **Branch prediction**
- VLIW (Very Long Instruction Word) and EPIC – MIMD on the lowest level
- Details in lectures 02 to 06…

# Thread-level parallelism (TLP)

- Multithreading (MT) – more threads of execution share functional units of processor (cores) – attempt to utilize the units which are not fully loaded by multiprocessing
- Increases throughput of the whole system, not of individual threads
- Processor (core) is required to maintain the state of each thread in a group – context switching (copies of working registers – RF, GPR, PC, …)
- Virtual memory support
- Ability to switch threads much faster than classical process switching/scheduling
- Multithreading:
    - temporal (or interleaved multithreading)
        - fine-grained
        - coarse-grained
    - simultaneous (or hyperthreading – Intel) – always fine-grained
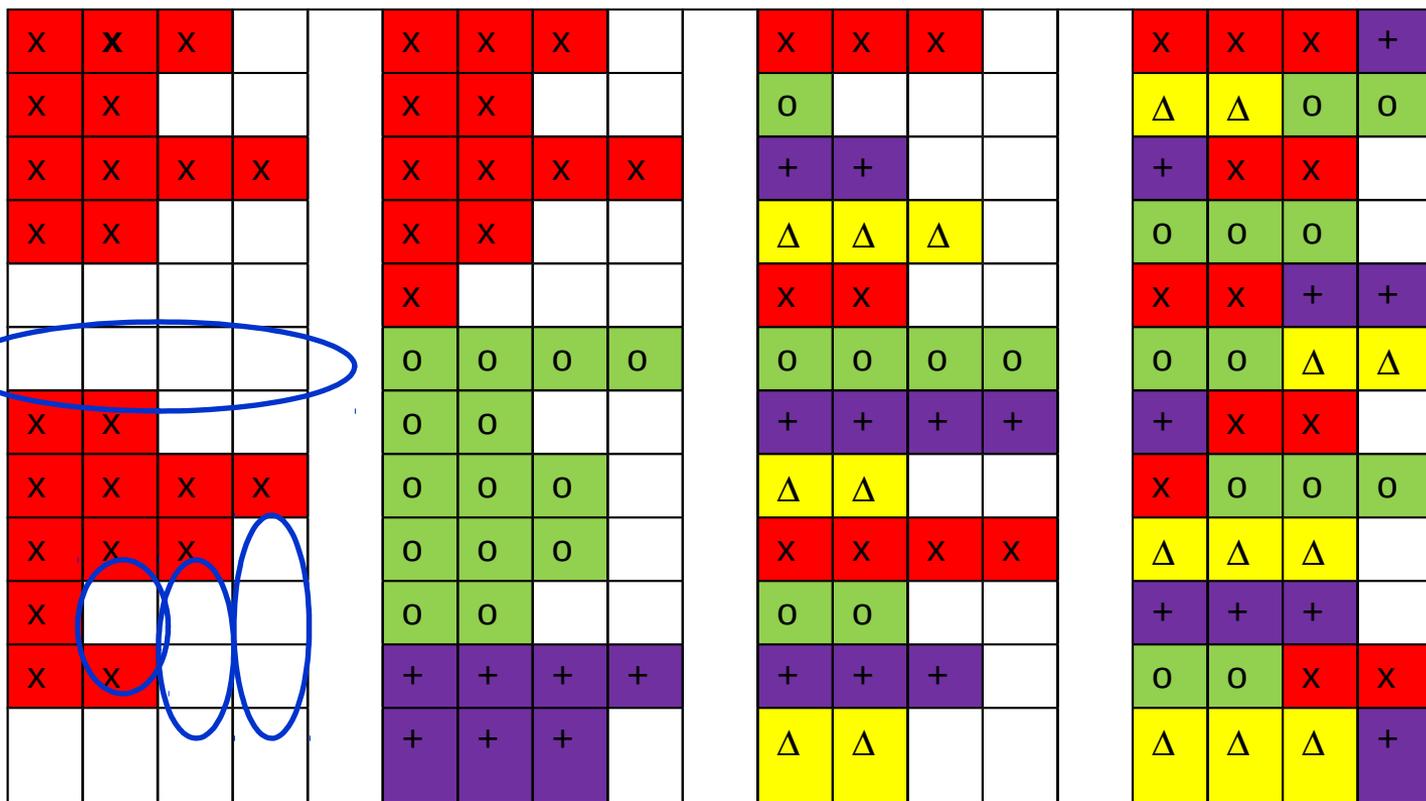
4-way superscalar processor (TLP -> ILP):

slots usage →

time

**vertical waste – bubble**

**horizontal waste – some units unused**

|  | superscalar only |  |  | | coarse-grained multithreading |  |  | | fine-grained multithreading |  |  | | simultaneous multithreading |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x | x | x |  | | x | x | x |  | | x | x | x |  | | x | x | x | + |
| x | x |  |  | | x | x |  |  | | o |  |  |  | | Δ | Δ | o | o |
| x | x | x | x | | x | x | x | x | | + | + |  |  | | + | x | x |  |
| x | x |  |  | | x | x |  |  | | Δ | Δ | Δ |  | | o | o | o |  |
|  |  |  |  | | x |  |  |  | | x | x |  |  | | x | x | + | + |
|  |  |  |  | | o | o | o | o | | o | o | o | o | | o | o | Δ | Δ |
| x | x |  |  | | o | o |  |  | | + | + | + | + | | + | x | x |  |
| x | x | x | x | | o | o | o |  | | Δ | Δ |  |  | | x | o | o | o |
| x | x | x |  | | o | o | o |  | | x | x | x | x | | Δ | Δ | Δ |  |
| x |  |  |  | | o | o |  |  | | o | o |  |  | | + | + | + |  |
| x | x |  |  | | + | + | + | + | | + | + | + |  | | o | o | x | x |
|  |  |  |  | | + | + | + |  | | Δ | Δ |  |  | | Δ | Δ | Δ | + |

# Thread-level parallelism

- Superscalar only – limited by resources for ILP, long latencies to fetch instructions, resolve branch miss-prediction and waiting for data read from memory (instruction L2 cache miss, data cache miss, etc.)

- Coarse-grained MT – long delays are eliminated by thread switching; but still empty cycles (start-up period) and low utilization of resources (execution units);

- Fine-grained MT – thread switch in each cycle; but still not all resources are utilized; blocked threads are ignored;

- Simultaneous MT – switch/schedule in each cycle; simultaneously executes more than one thread; use of all resources depends on threads demand (consider an ideal combination of a thread computing FP and a thread moving data)

# Task-level parallelism (TLP)

- Task-level parallelism (TLP) – also function parallelism or control parallelism – software/OS level/controlled

- multiprocessing considered as support for TLP – software (multitasking) and hardware view (symmetric / asymmetric; tightly / loosely coupled,..) – HW resources are assigned to threads by SW, possibly combined with SMT (processor Intel Core i7-980X: 6 cores, 2 threads/core simultaneously, SPARC T3 8 threads/core, POWER{8,9} 8 threads/core)

- TLP:  SPMD program:
      **if** CPU_ID == 0
        **then do** task "A"
      **else if** CPU_ID == 1
        **then do** task "B"
      **end if**

Each processor (core, SMT virtual core) has own ID. The program recognizes on which processor it runs and executes only the part of the program assigned to the given processor

# Task-level parallelism

- TLP  MPMD program: program is divided to modules (which communicate together!) – demanding scientific applications, but also client-server applications;

- Key components:
  - Communication between nodes (from HW point of view) or between processes/threads (from SW point of view)
  - Mutual synchronization

- According to the communication demands, HPC programs are executed on:
  - tightly-coupled multiprocessor systems (MPP)
  - loosely-coupled multiprocessor systems (Cluster, Grid)

- Execution of independent programs

# Data-level parallelism

- identical operations executed on data set/vector (SIMD) on hardware level

- distribute chunks of data to individual nodes (processes):

**for** i **from** lower_limit **to** upper_limit
    **do** a[i] = b[i] + c[i]

Each processor (core) uses different lower and upper limit $\rightarrow$ works with different data

- parallelism – explicitly programmed (OpenMP), implicit (on compilel level)

- support of programming languages for parallel computing

# Dependencies in programs

- Prerequisite for parallel execution of program segments – independence on other segments (at least for some/fundamental part of the algorithm)
- Expression of dependency relations – graph theory
  - Nodes – operations (segments)
  - Edges (always oriented) – relations between nodes
  Graph analysis – finding the existence of parallelism

Three types of dependencies:

- <u>Data</u> – defines succession relationships between commands
- <u>Resources</u> – resources of given system (conflict of shared resources – registers, memory, ALU, FPU, processors, …)
- <u>Control</u> – the order of operations execution cannot be determined before the program is started (conditional branches, iterations, achieving required precision, …)

# Data dependencies

- Data dependency:
  - Flow dependency (true dependency)
  - Anti-dependency (name/store dependency)
  - Output dependency
  - Input-output dependency
  - Unknown dependency

On instruction level when pipeline is realized

When parallel program is developed

- Flow dependency (Read-after-Write: RAW) S1 $\rightarrow$ S2:
  S2 flow dependent on S1 if $\exists$ execution path from S1 to S2, and at least one output of S1 is routed to S2. Symbolically: O(S1) ∩ I(S2) , S1-> S2

- (Flow) Anti-dependency (Write-After-Read: WAR) S1 $\xrightarrow{+}$ S2:
  I(S1) ∩ O(S2) , S1-> S2

- Output dependency (Write-after-Write: WAW) S1 $\xrightarrow{o}$ S2:
  O(S1) ∩ O(S2), S1->S2 (produce the same output variable)

# Data dependencies

- Input output dependency S1 $\xrightarrow{I/O}$ S2
  when both I/O commands (read, write) are referencing the same file (not variable)

- Unknown dependency – dependence relation cannot be determined
  - Index of a variable is itself indexed
  - A variable appears more than once with indexes containing the loop variable with different coefficients
  - Index defined by the loop variable is nonlinear
  - Etc.

# Data dependency

P1:   C = D*E

P2:   M = G+C

P3:   A = B+C

P4:   M = A+M

P5:   F = G/E

Alternatively, it is possible to write types (WAW, RAW, WAR, I/O) to arrows instead of the symbols
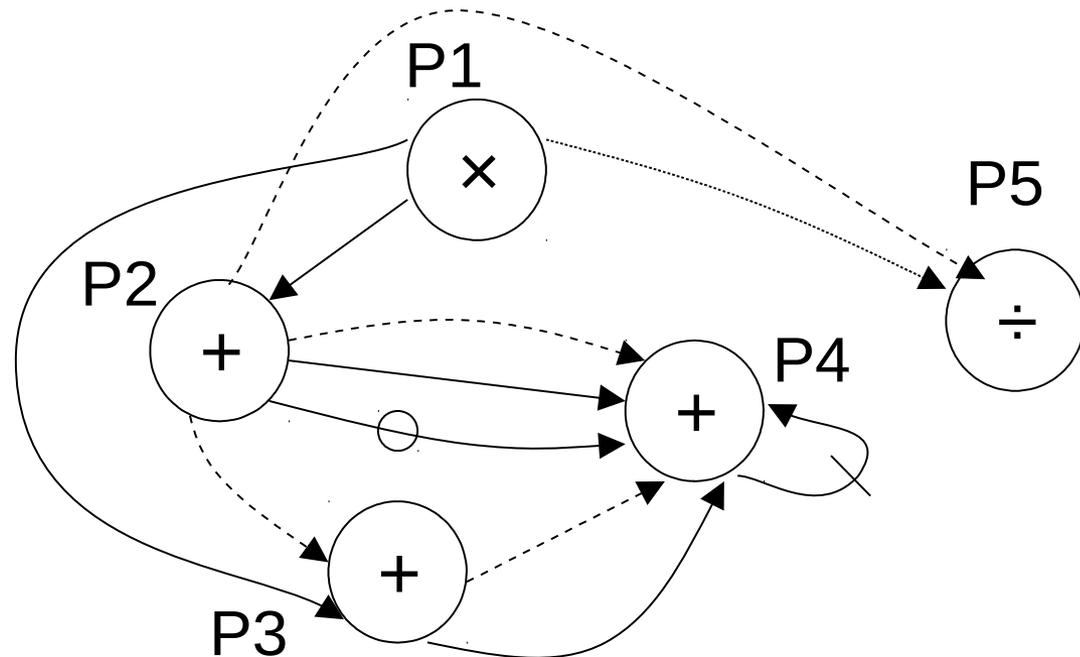


Solid line – data dependency

Dashed line – resource dependency

# Data dependency

P1:   C = D*E

P2:   M = G+C

P3:   A = B+C

P4:   M = A+M

P5:   F = G/E
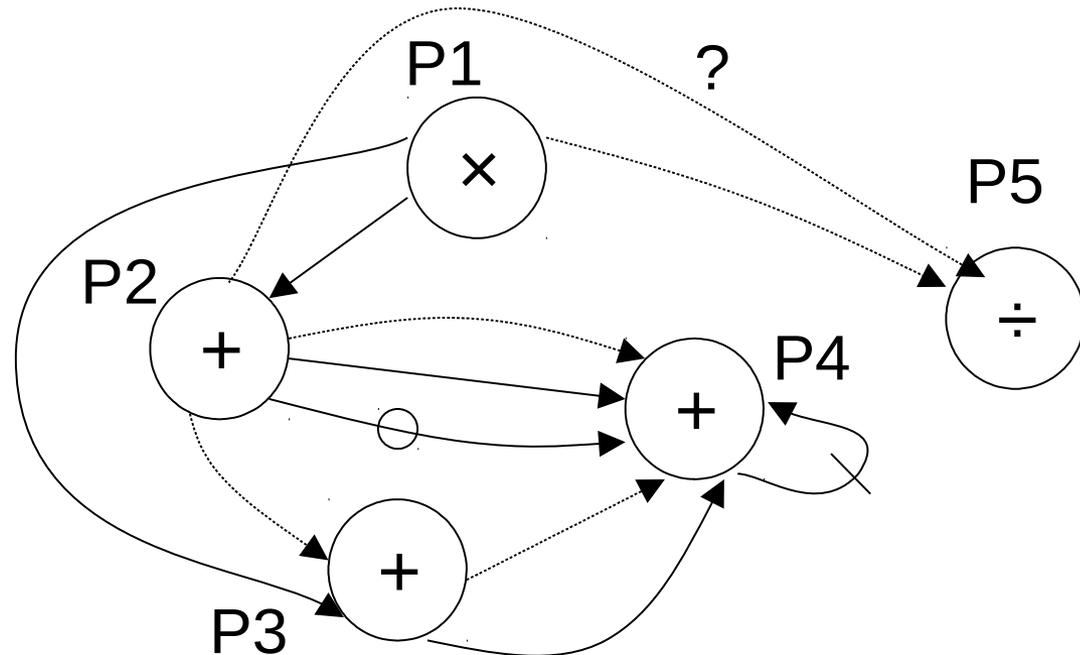
Remember, this does not need to be a single operation only... Use generalized way

Alternatively, it is possible to write types (WAW, RAW, WAR, I/O) to arrows instead of the symbols



Solid line – data dependency

Dashed line – resource dependency

# Bernstein's conditions of parallelism

- They determine when two processes can be performed in parallel in terms of spatial parallelism
  (process – software entity corresponding to program fragment abstraction on different levels of processing, instruction, source lines, matrix operations, …)

- I – input set of process ($\forall$ variables required to execute process)
- O – output set of process (variables generated by process)

- Processes $P_i$ and $P_j$ can be executed in parallel ($P_i \| P_j$) if:
  **$[I(P_i) \cap O(P_j)] \cup [O(P_i) \cap I(P_j)] \cup [O(P_i) \cap O(P_j)] = \emptyset$**

- $P_1 \| P_2 \| \ldots \| P_k$  if and only if $P_i \| P_j$ for $\forall i \neq j$

- Commutativity applies ($P_i \| P_j = P_j \| P_i$)

- Transitivity <u>does not apply</u> ($P_i \| P_j \wedge P_j \| P_k$ does not imply $P_i \| P_k$)

- Associativity applies ($[P_i \| P_j] \| P_k = P_i \| [P_j \| P_k]$)

# Bernstein's conditions of parallelism

| Program fragment | All pairs | All triplets |
|---|---|---|
| P1:   C = D*E | P1 \|\| P4, P1 \|\| P5 | P1 \|\| P4 \|\| P5 |
| P2:   M = G+C | P2 \|\| P3, P2 \|\| P5 | P2 \|\| P3 \|\| P5 |
| P3:   A = B+C | P3 \|\| P5 | x |
| P4:   M = A+M | P4 \|\| P5 | x |
| P5:   F = G/E | x | x |

Bernstein's conditions are necessary conditions of parallelization, but not sufficient …
All source (even indirect) dependencies $P_i$ (i<j) of $P_j$ have to be  executed!

If   $P_i \| P_j$   $\Rightarrow$ can be executed simultaneously or arbitrarily ordered

| This sequence cannot be executed: | This is allowed: | This also: | This as well: |
|---|---|---|---|
| 1.  P1 \|\| P4 \|\| P5 | 1.  P1 | 1.  P1 | 1.  P1 \|\| P5 |
| 2.  P2 \|\| P3 | 2.  P2 \|\| P3 | 2.  P2 \|\| P3 \|\| P5 | 2.  P2 \|\| P3 |
|  | 3.  P4 \|\| P5 | 3.  P4 | 3.  P4 |

# Bernstein's conditions of parallelism

Program fragment

P1:  C = D*E

P2:  M = G+C

P3:  A = B+C

P4:  M = A+M

P5:  F = G/E



time
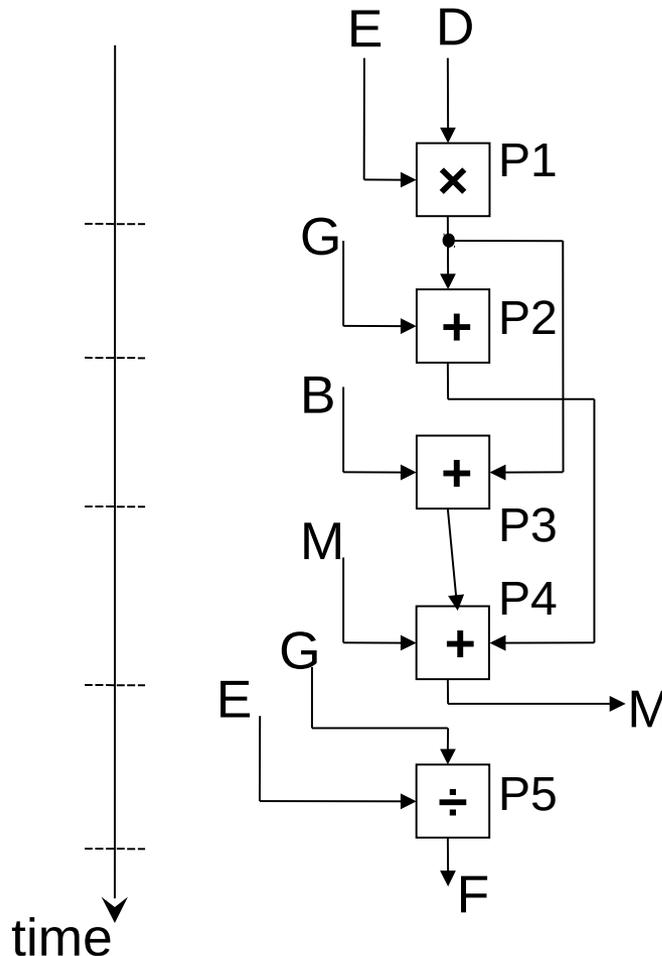
sequential: 5 steps
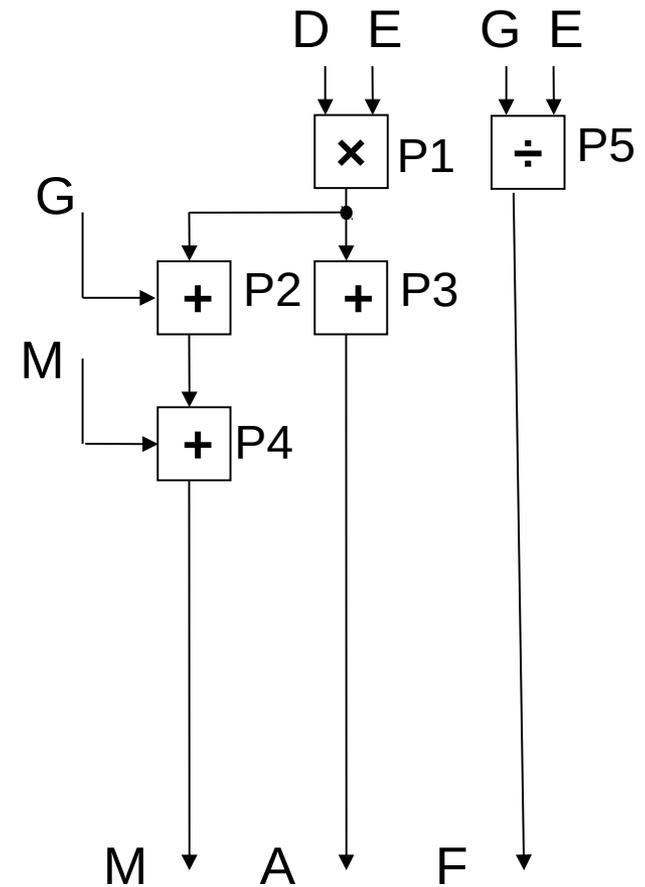
parallel: 3 steps
Two adders required

1. a = 1
2. b = 2
3. c = 3
4. d = 4
5. e = 5
6. f = a*b
7. g = c*d
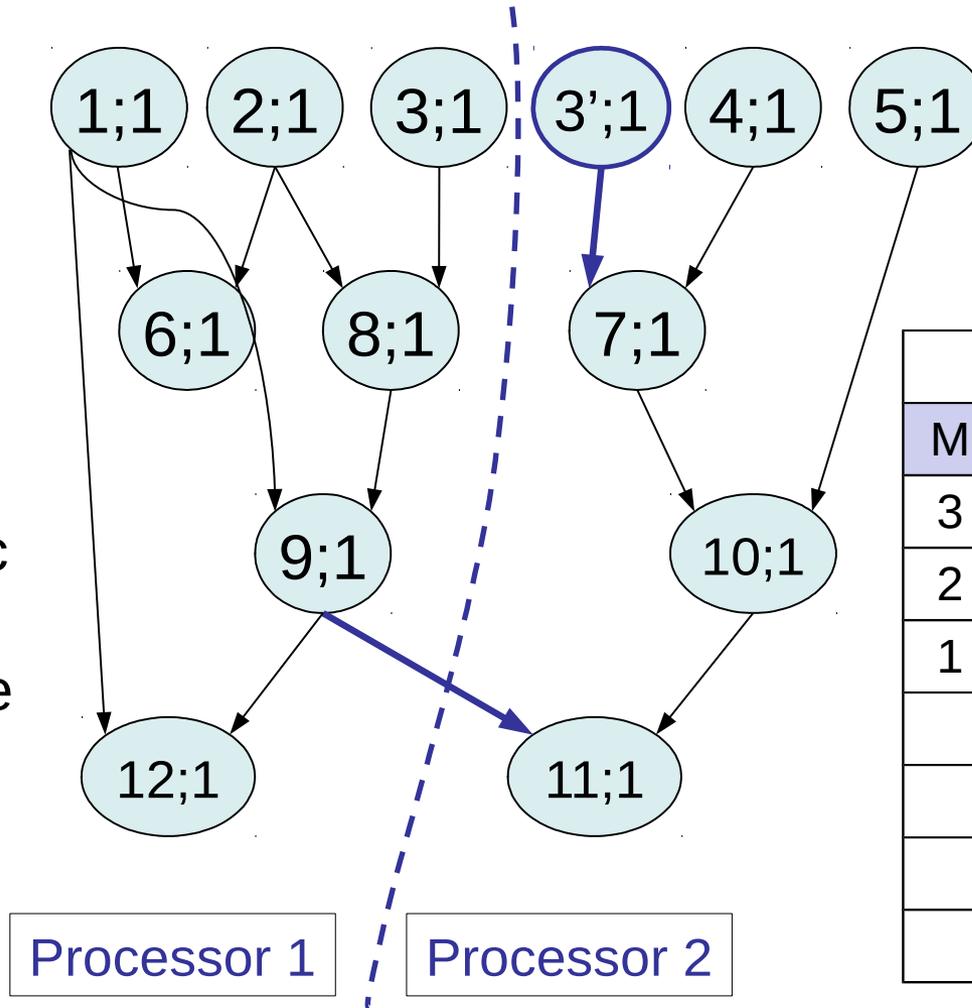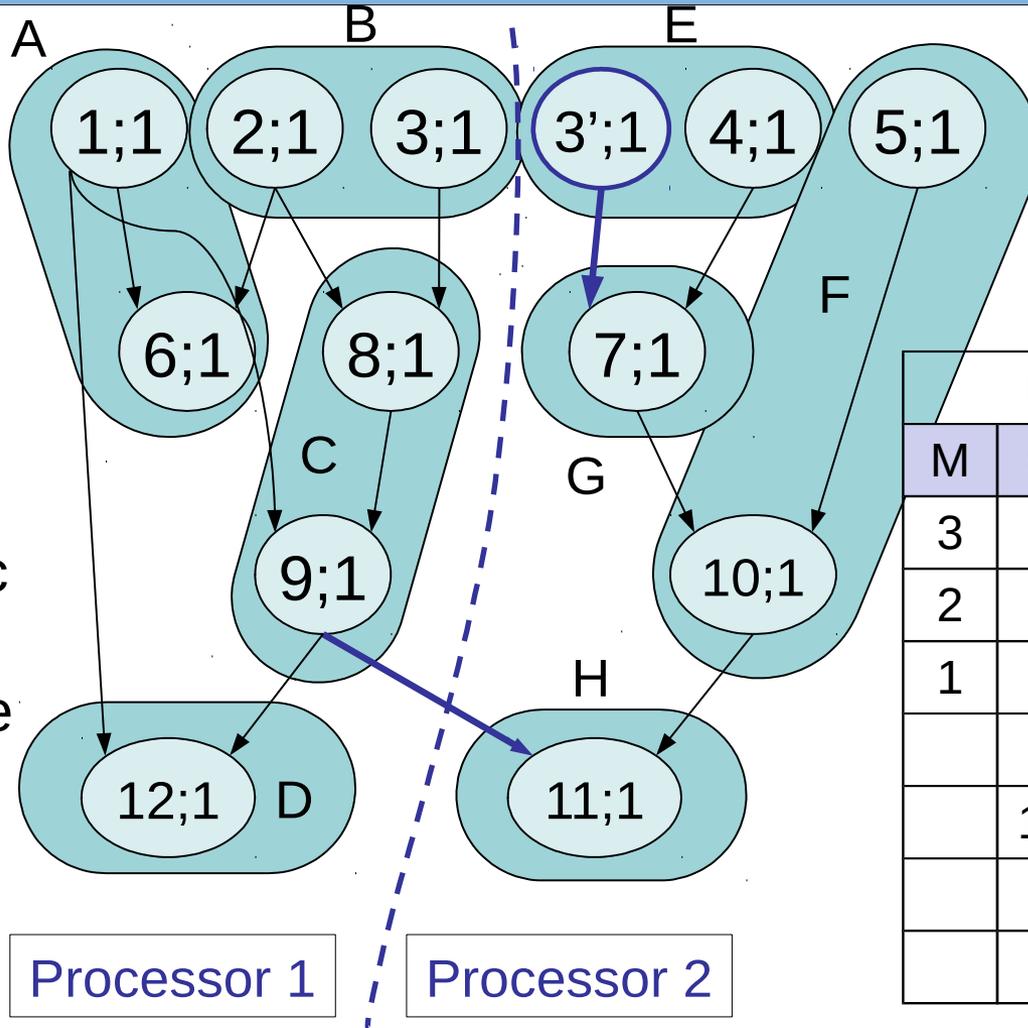8. h = b–c
9. i = a+h
10. b = g+e
11. c =b*i
12. j = a*i

Implement a program on a two-processor system that includes two-way processors capable of executing one memory access instruction and one arithmetic operation per cycle. Let the latency of the communication between the processors be L = 2 cycles. Communication is non-blocking.

1. a = 1
2. b = 2
3. c = 3
4. d = 4
5. e = 5
6. f = a*b
7. g = c*d
8. h = b–c
9. i = a+h
10. b = g+e
11. c =b*i
12. j = a*i



V;d

Node label

delay
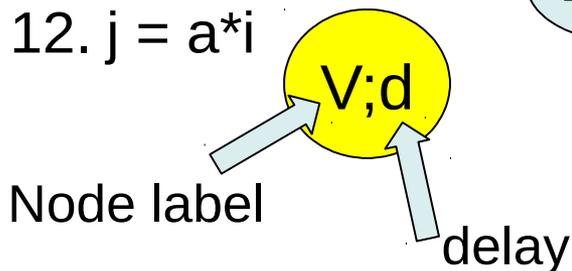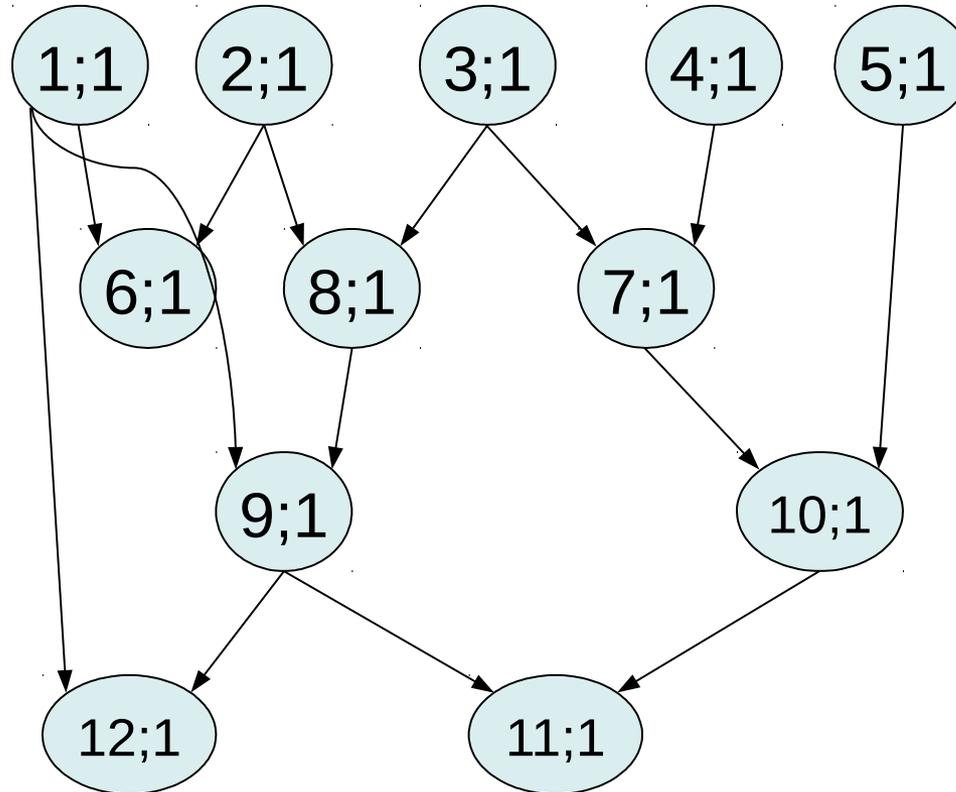
The node weight measures the amount of work assigned to that node. The simplest measure is the number of instructions (or the execution time of the node – the number of cycles).

1. a = 1
2. b = 2
3. c = 3
4. d = 4
5. e = 5
6. f = a*b
7. g = c*d
8. h = b–c
9. i = a+h
10. b = g+e
11. c =b*i
12. j = a*i
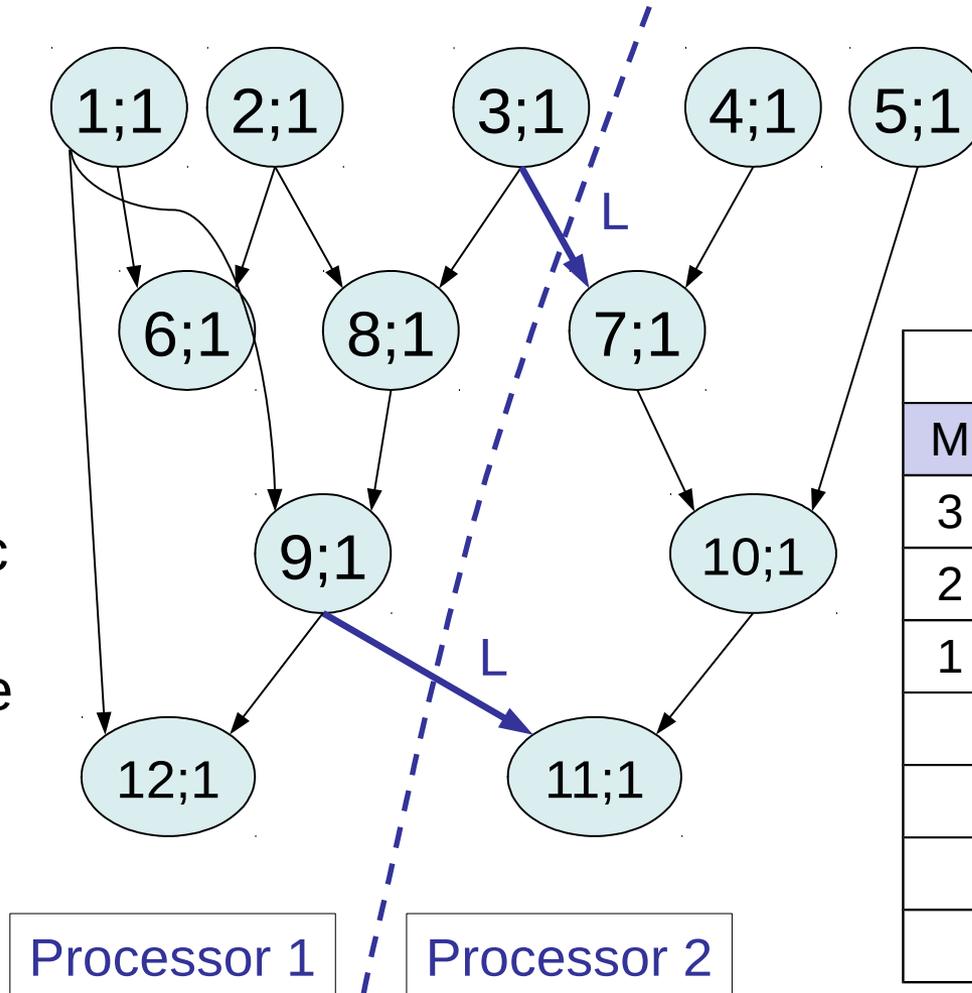


M – memory
C – compute
S – send
R – receive

| P1 | | | | | P2 | | | |
|---|---|---|---|---|---|---|---|---|
| M | C | S | R | | M | C | S | R |
| 3 | | | | | 4 | | | |
| 2 | | 3 | | | 5 | | | 3 |
| 1 | 8 | 3 | | | | | | 3 |
| | 9 | | | | | 7 | | |
| | 12 | 9 | | | | 10 | | 9 |
| | 6 | 9 | | | | | | 9 |
| | | | | | | 11 | | |

Processor 1   Processor 2

1. a = 1
2. b = 2
3. c = 3
4. d = 4
5. e = 5
6. f = a*b
7. g = c*d
8. h = b–c
9. i = a+h
10. b = g+e
11. c =b*i
12. j = a*i



Significant speedup can be achieved by node duplication

| P1 | | | | | P2 | | | |
|---|---|---|---|---|---|---|---|---|
| M | C | S | R | | M | C | S | R |
| 3 | | | | | 4 | | | |
| 2 | | | | | 3 | | | |
| 1 | 8 | | | | 5 | 7 | | |
| | 9 | | | | | 10 | | |
| | 12 | 9 | | | | | | 9 |
| | 6 | 9 | | | | | | 9 |
| | | | | | | 11 | | |

Processor 1    Processor 2

1. a = 1
2. b = 2
3. c = 3
4. d = 4
5. e = 5
6. f = a*b
7. g = c*d
8. h = b–c
9. i = a+h
10. b = g+e
11. c =b*i
12. j = a*i

| | P1 | | | | | P2 | | |
|---|---|---|---|---|---|---|---|---|
| M | C | S | R | | M | C | S | R |
| 3 | | | | | 4 | | | |
| 2 | | | | | 3 | | | |
| 1 | 8 | | | | 5 | 7 | | |
| | 9 | | | | | 10 | | |
| | 12 | 9 | | | | | | 9 |
| | 6 | 9 | | | | | | 9 |
| | | | | | | 11 | | |

Processor 1    Processor 2

1. a = 1
2. b = 2
3. c = 3
4. d = 4
5. e = 5
6. f = a*b
7. g = c*d
8. h = b–c
9. i = a+h
10. b = g+e
11. c =b*i
12. j = a*i

B;2

E;2

A;1

G;1

C;2

F;2

D;1

H;1

Processor 1
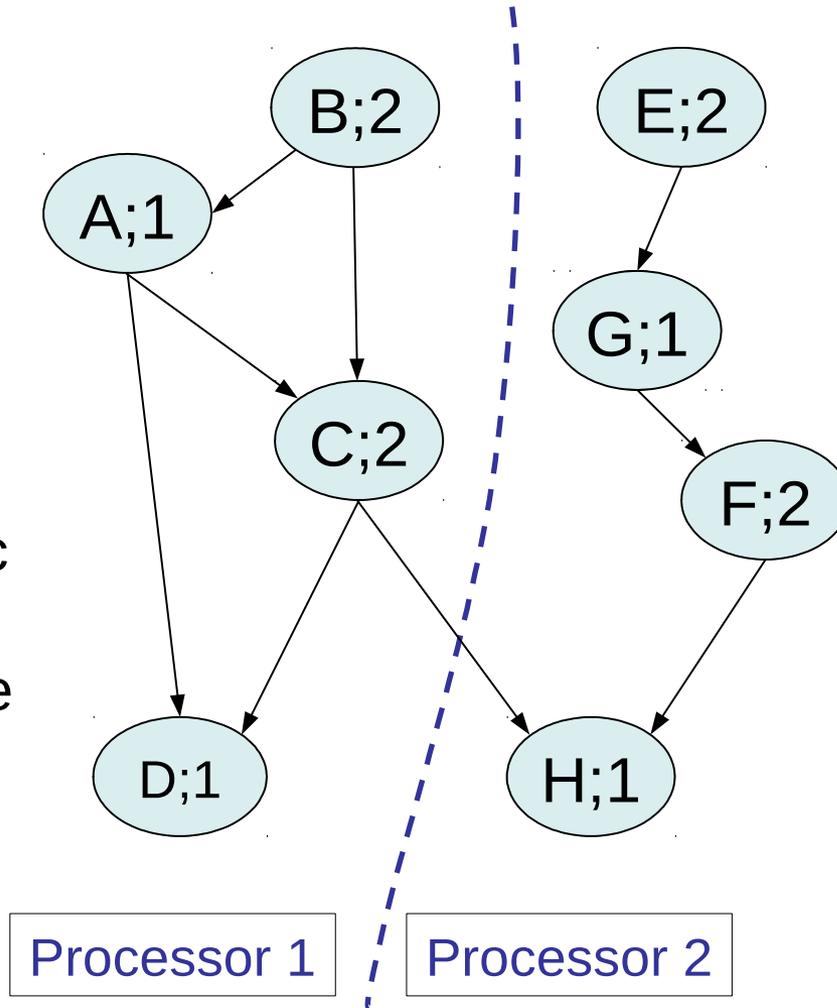
Processor 2

Grain packing can provide a significant simplification of scheduling while maintaining the same speedup.

| P1 | | | | P2 | | |
|----|----|----|----|----|----|----|
| C | S | R | | C | S | R |
| B | | | | E | | |
| B | | | | E | | |
| A | | | | G | | |
| C | | | | F | | |
| C | | | | F | | |
| D | C | | | | | C |
| | C | | | | | C |
| | | | | H | | |

Let there be three equivalent processors available.
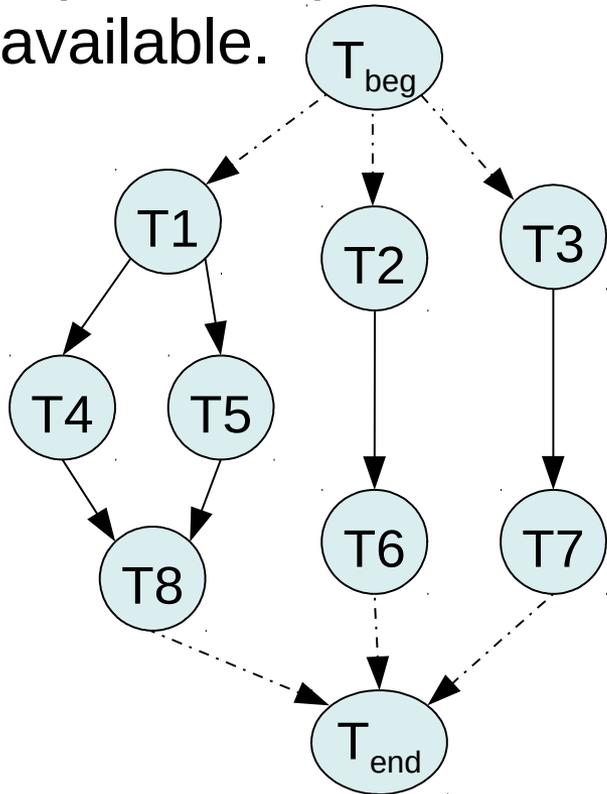
Execution time of each task

| T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|----|----|----|----|----|----|----|----|
| 3  | 5  | 7  | 3  | 6  | 8  | 7  | 5  |

Communication times (amount of data to deliver) if source and destination tasks are run on different processors

| T1,T4 | T1,T5 | T2,T6 | T3,T7 | T4,T8 | T5,T8 |
|-------|-------|-------|-------|-------|-------|
| 2     | 6     | 2     | 5     | 3     | 1     |

How to divide individual tasks among them ???

Let there be three equivalent processors available.

| T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|----|----|----|----|----|----|----|----|
| 3  | 5  | 7  | 3  | 6  | 8  | 7  | 5  |

| T1,T4 | T1,T5 | T2,T6 | T3,T7 | T4,T8 | T5,T8 |
|-------|-------|-------|-------|-------|-------|
| 2     | 6     | 2     | 5     | 3     | 1     |



Gantt chart:
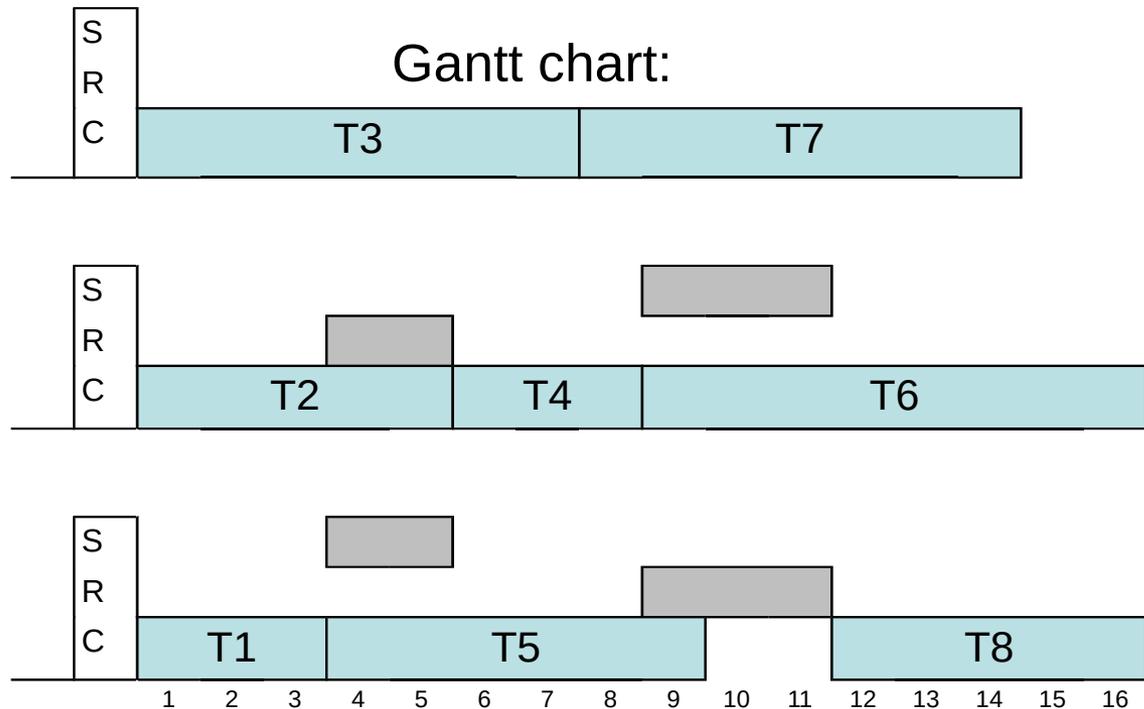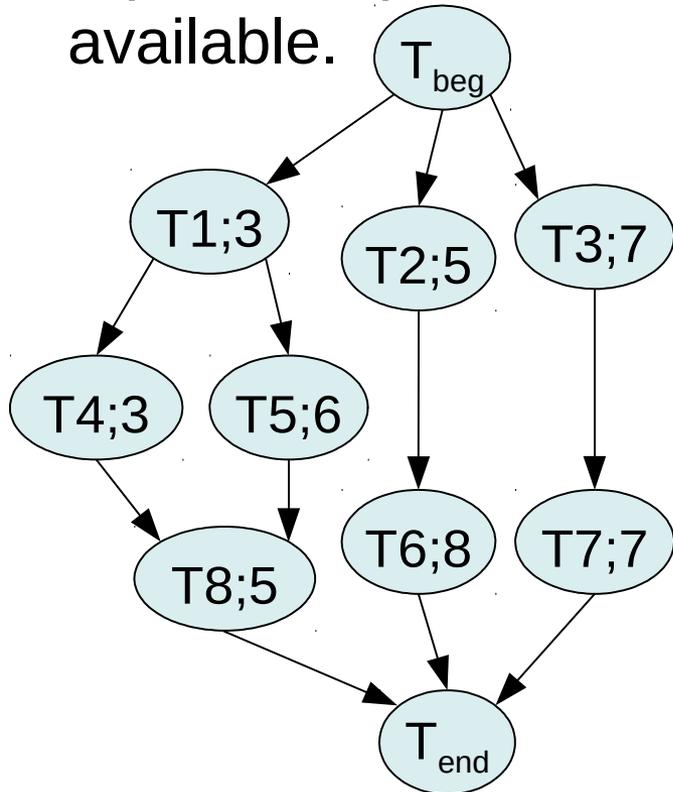
Resources/processors utilization

$S = 44 / 16 = 2{,}75$

# Parallel computers' memory architectures

- **Shared memory systems (SMS)** – access to the whole memory possible for each processor (global address space), memory resources are shared, complexity of memory-CPU communication geometrically increases when increasing CPU counts, same to maintain memory coherence…
  - **UMA** (Uniform Memory Access) – identical memory access time, SMP (Symmetric Multiprocessor), CC-UMA (Cache Coherent UMA)
  - **NUMA** (Non-Uniform) – variable access time – depends on CPU and address; can be build as interconnection of multiple SMP – where a SMP node can access the memory of another node; if Cache Coherency is preserved, then CC-NUMA

| CPU | CPU | |
|-----|-----|--------|
| CPU | CPU | Memory |

UMA

| CPU | CPU | |
|-----|-----|--------|
| CPU | CPU | Memory |

| CPU | CPU | |
|-----|-----|--------|
| CPU | CPU | Memory |
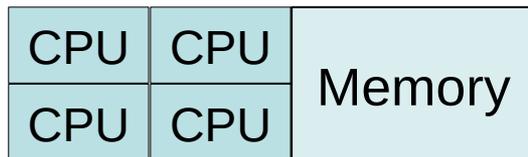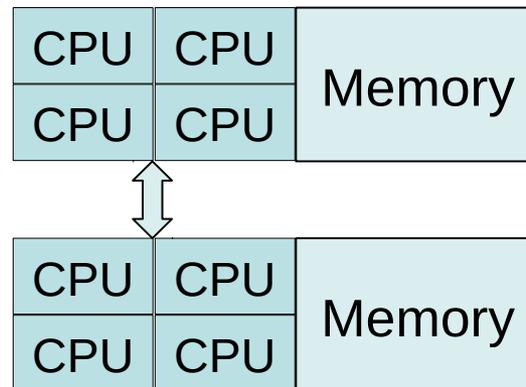
NUMA, RMA (Remote Memory Access)

DSM (Distributed Shared Memory) – **DGAS** (D. Global Address Space)

- **Distributed memory systems (DMS)** – separated local address spaces, node-local physical memory; communication and synchronization solved by programmer/SW; easier scalability when CPU count increases; NORMA (No Direct Remote Memory Access)

| CPU | Memory |
|-----|--------|

| CPU | Memory |
|-----|--------|

| CPU | Memory |
|-----|--------|

**Interconnection network**
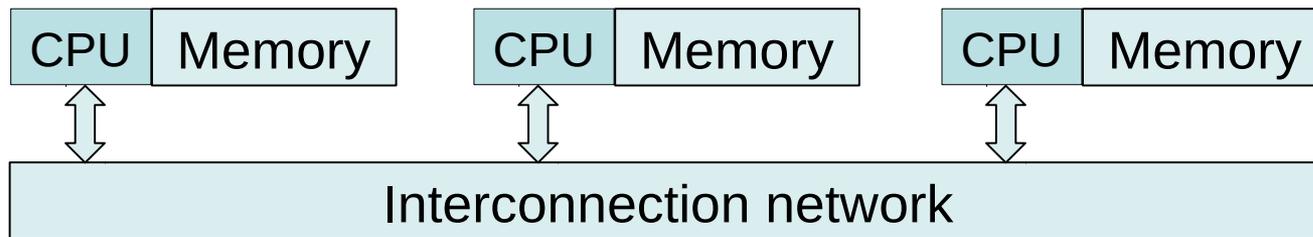
- **Hybrid** (distributed + shared)

| CPU | CPU | Memory |
|-----|-----|--------|
| CPU | CPU | |

| CPU | CPU | Memory |
|-----|-----|--------|
| CPU | CPU | |

| CPU | CPU | Memory |
|-----|-----|--------|
| CPU | CPU | |

**Interconnection network**
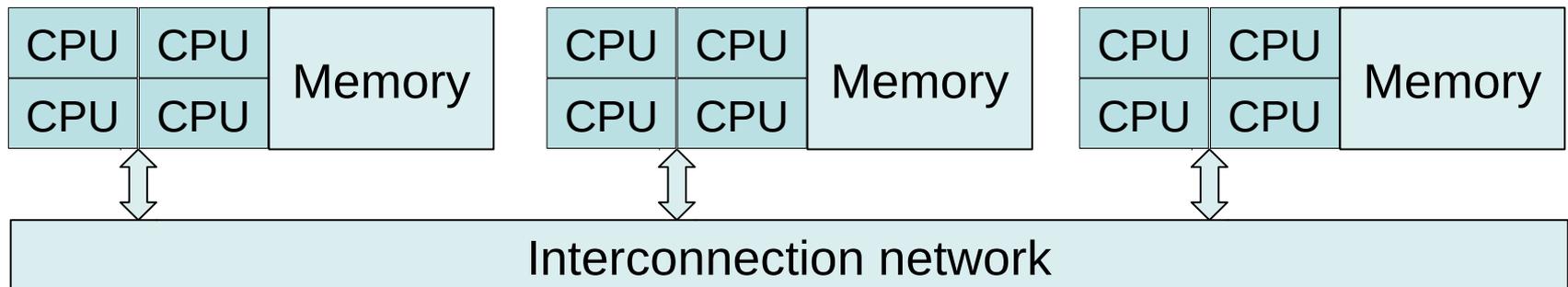
# Programming models

Abstraction of hardware and memory architecture;

Not necessarily tied to a particular architecture.

- **Shared memory** – Tasks share global address space, asynchronous read and write; locks, semaphores, …; explicit communication is not needed when exchanging data; Where are stored the data that the processor works with?

- **Threads** – POSIX Threads (Pthreads) – very explicit parallelism – the program must be designed to run tasks "in parallel"; OpenMP – parallelization expressed in directives, more automatic with the help of compiler.

# Programming models

- **Message passing** – exchange of data and events by sending and receiving of messages; typical for DMS but usable/used on SMS as well.
  What is the maximum communication latency as not to degrade performance?

- **Data-parallel** – focuses on the parallel execution of operations over data sets; suitable for both SMS and DMS; support in both languages (HPF – High Performance Fortran), and compiler directives (OpenMP).

- **Hybrid** – combination of already described models with use of SPMD (Single Program Multiple Data), or for complex systems MPMD (Multiple Programs Multiple Data).

## Scheduling



Zdroj: http://shmu.sk
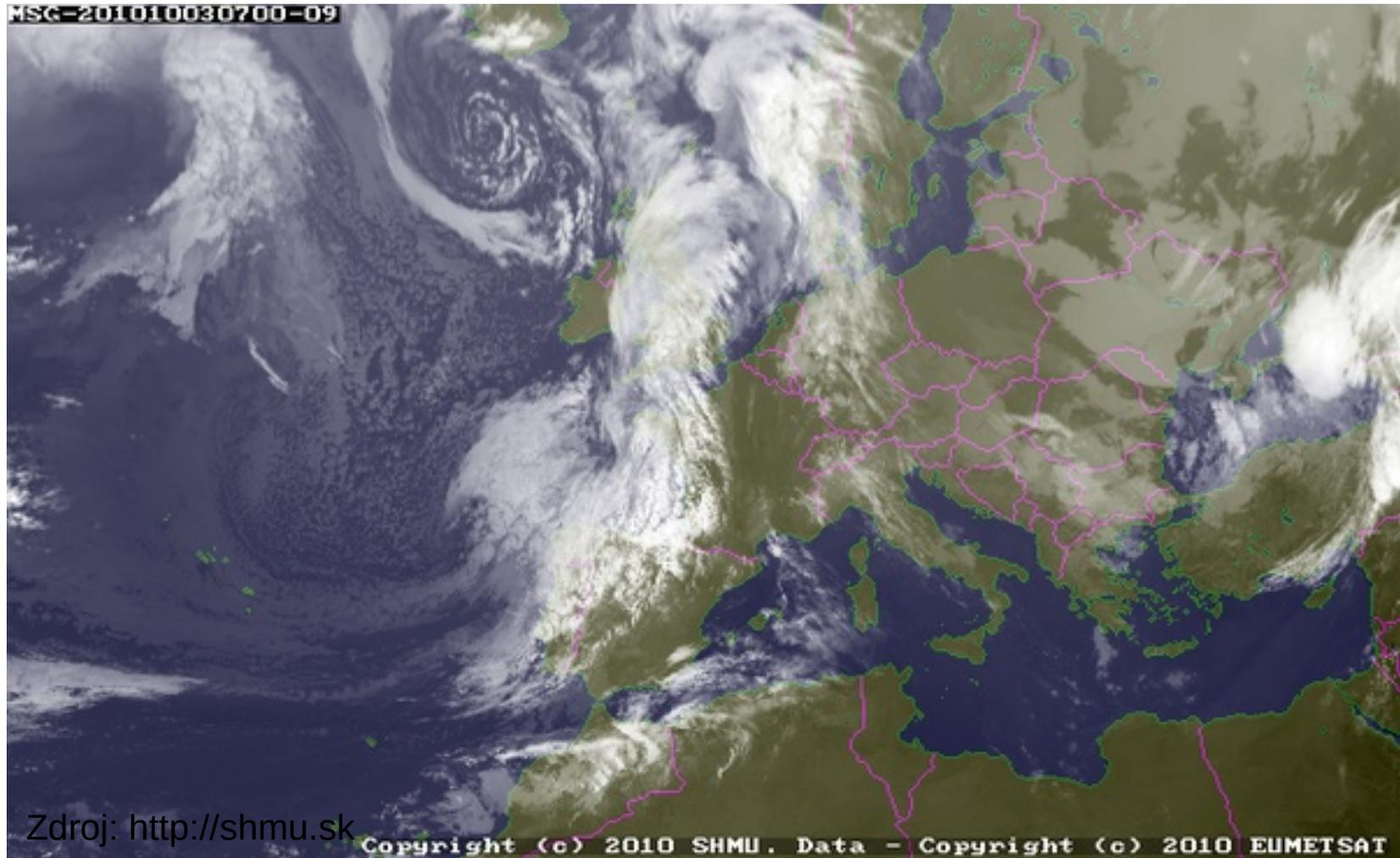
## Scheduling

- Top down view (<u>functional decomposition</u>): The aim is to divide the program into a set of tasks which can be executed in parallel with respect to mutual communication; can be applied recurrently

Climate model

Atmosphere model

Hydrological model

Continental model
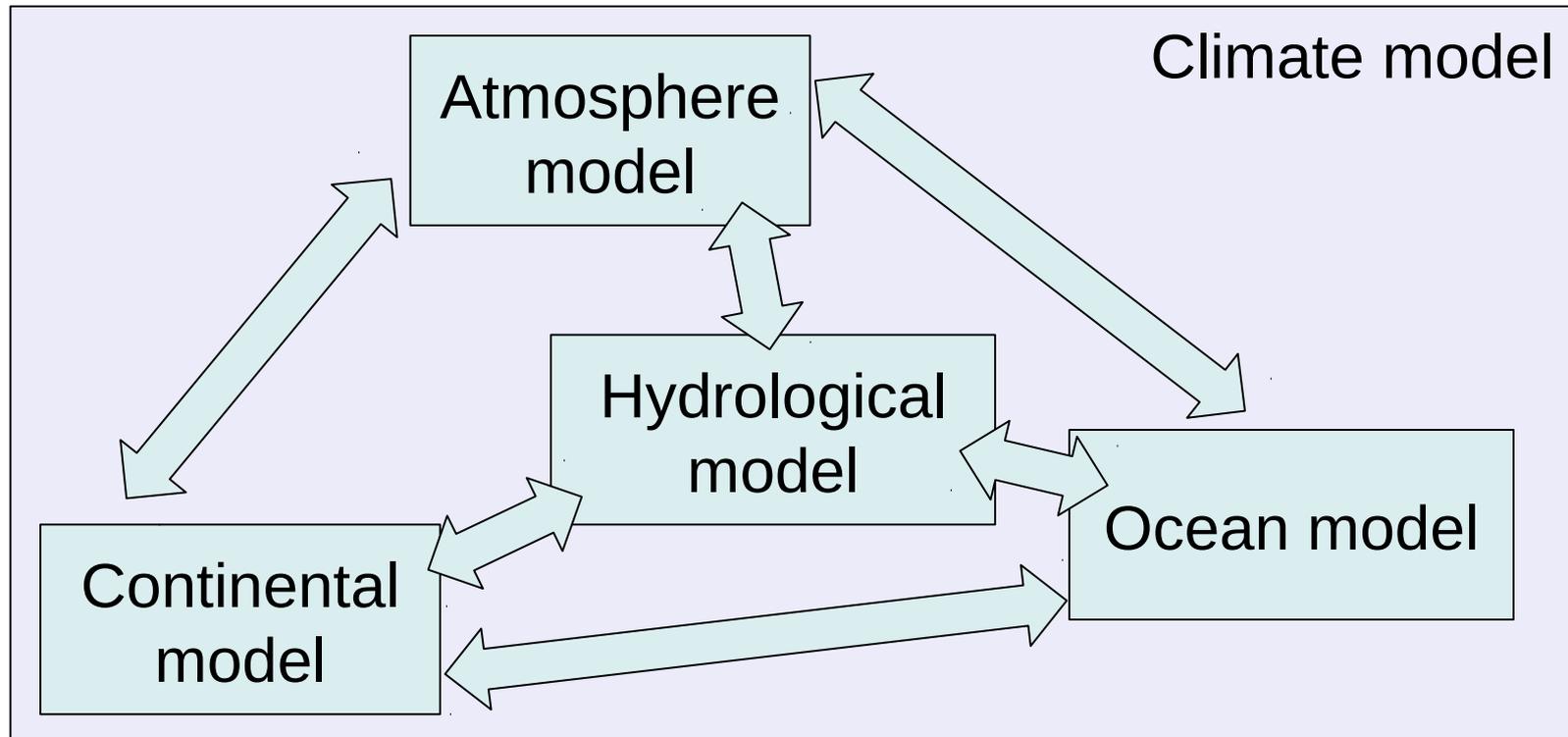
Ocean model

## Scheduling

- Speculative decomposition



Condition
initial
computation

time T1

Yes        No

IF

time T2

Branch  A        Branch  B
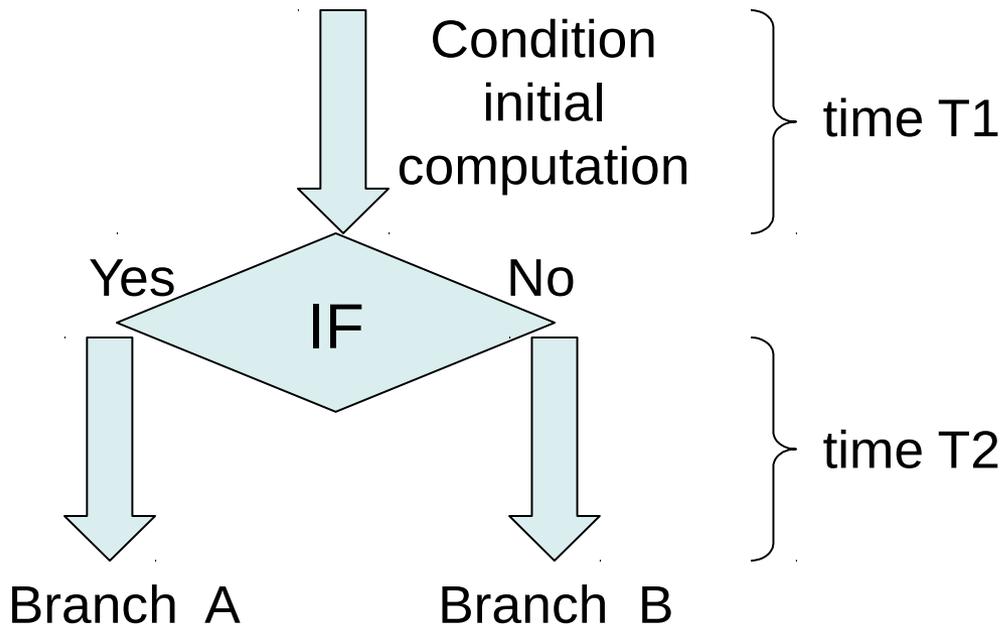
## Scheduling

- Speculative decomposition
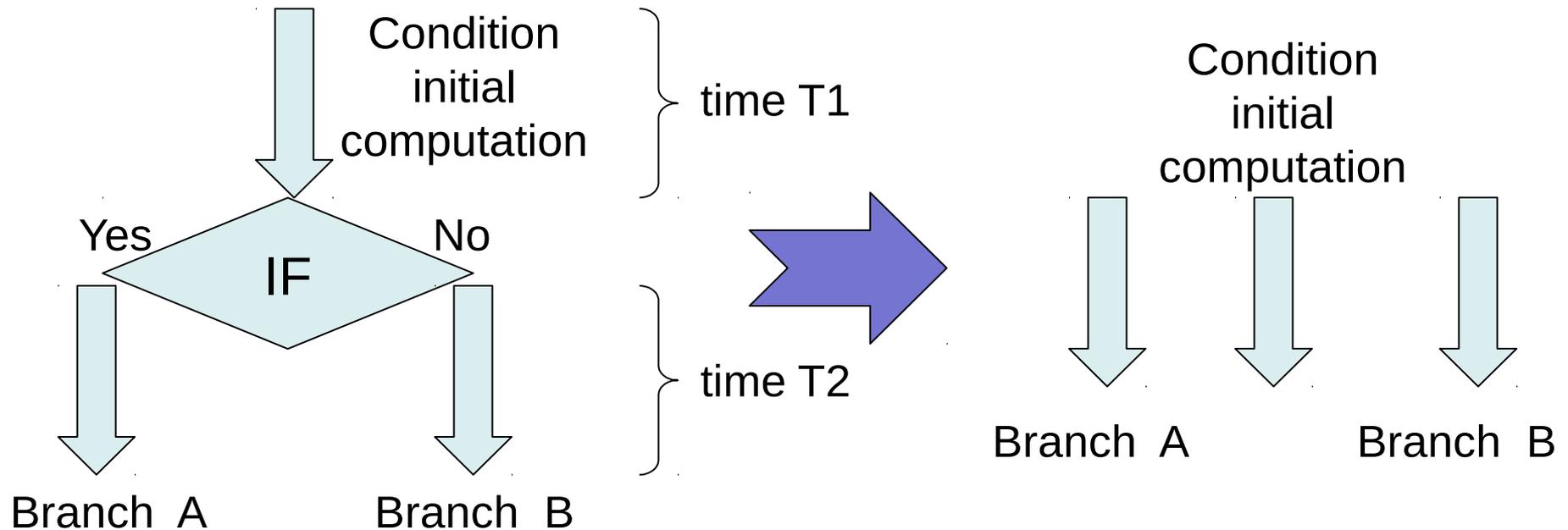
**Scheduling**

- Bottom-up: The goal is to group sequentially executed instructions, commands, program fragments without a link to another (one line of instruction flow) – the grain packing at the lowest level, possibly continue according to a specific strategy in the grain packing with respect to communication (see introductory examples).

- The aim is to have the greatest possible compactness and the minimal possible mutual coupling.

- Compiler vs. programmer.

- Take into account memory architecture.

- Homogeneous vs. heterogeneous computer system.

- Scheduling works also as a system resource allocation algorithm (many tasks and less CPUs..).

Let there be three equivalent processors available.

| T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|----|----|----|----|----|----|----|----|
| 3  | 5  | 7  | 3  | 6  | 8  | 7  | 5  |

| T1,T4 | T1,T5 | T2,T6 | T3,T7 | T4,T8 | T5,T8 |
|-------|-------|-------|-------|-------|-------|
| 2     | 6     | 2     | 5     | 3     | 1     |

Gantt chart:



Resources/processors utilization

$S = 44 / 16 = 2{,}75$

Let there be three equivalent processors available.

| T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|----|----|----|----|----|----|----|----|
| 3  | 5  | 7  | 3  | 6  | 8  | 7  | 5  |

| T1,T4 | T1,T5 | T2,T6 | T3,T7 | T4,T8 | T5,T8 |
|-------|-------|-------|-------|-------|-------|
| 2     | 6     | 2     | 5     | 3     | 1     |

$T_{beg}$

T1;3  T2;5  T3;7

T4;3  T5;6

T6;8  T7;7

T8;5

$T_{end}$

Gantt chart:

S R C — Job 1

S R C — Job 2

S R C — Job 3

1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
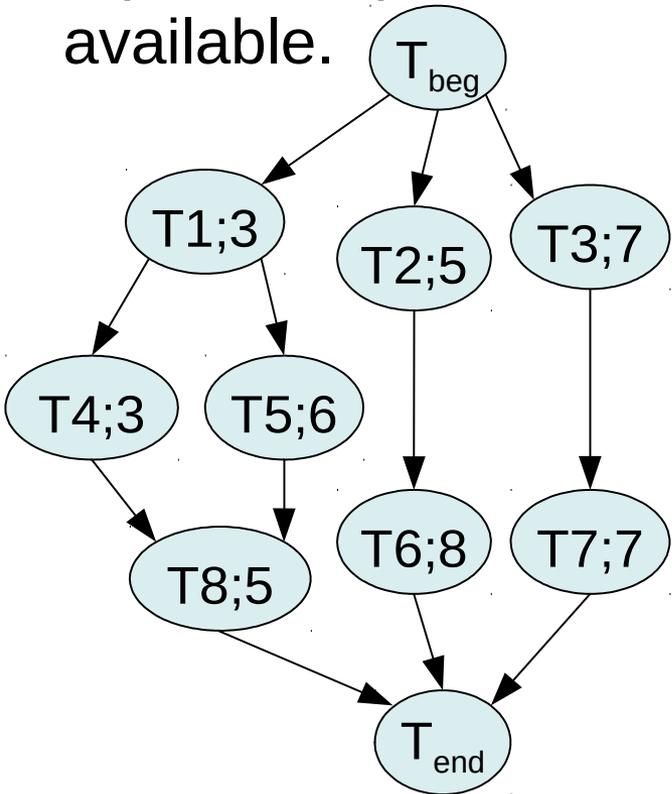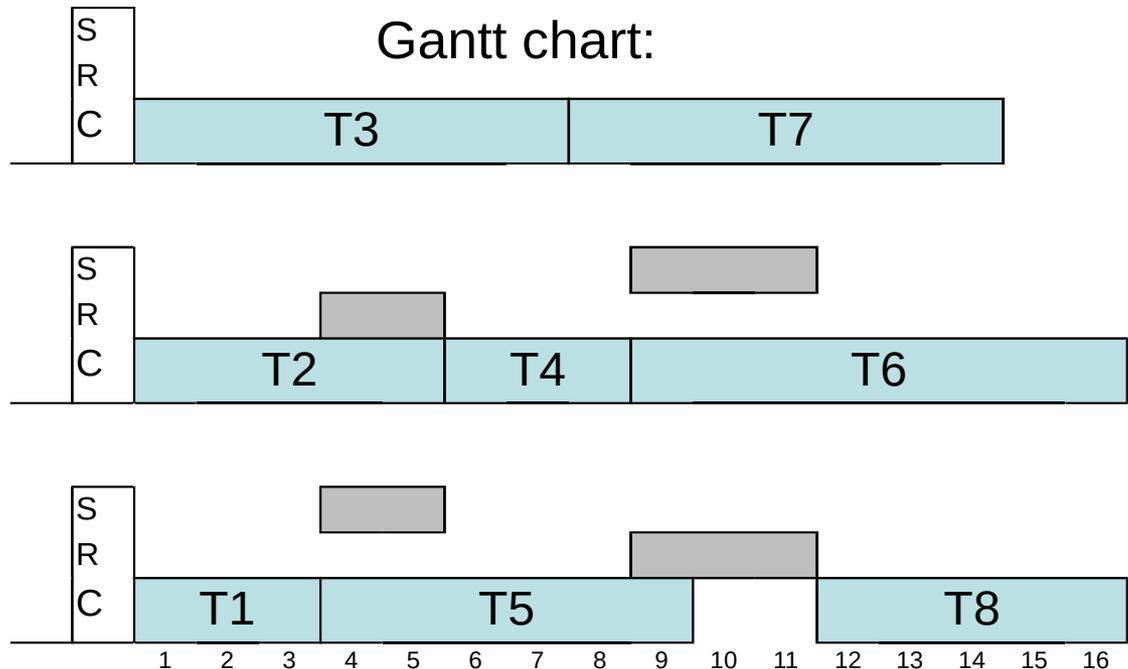
Resources/processors utilization

S = 44 / 16 = 2,75

- Scheduling as a system resource allocation algorithm – decides which task should run on which CPU and when
  - First-come-first-serve (waiting for others causes delays),
  - Gang scheduling (problem are I/O and blocking communication),
  - Paired gang scheduling.

CPU count < tasks count
=> all cannot run simultaneously

I/O

**Partitioning** – Domain decomposition



**Sharpening**

**How to do that?**

**Partitioning** – Domain decomposition

How to sharpen an image? ✉ Convolution

How to utilize more CPUs in a parallel program?



Thread 1

Thread 2

Thread 3

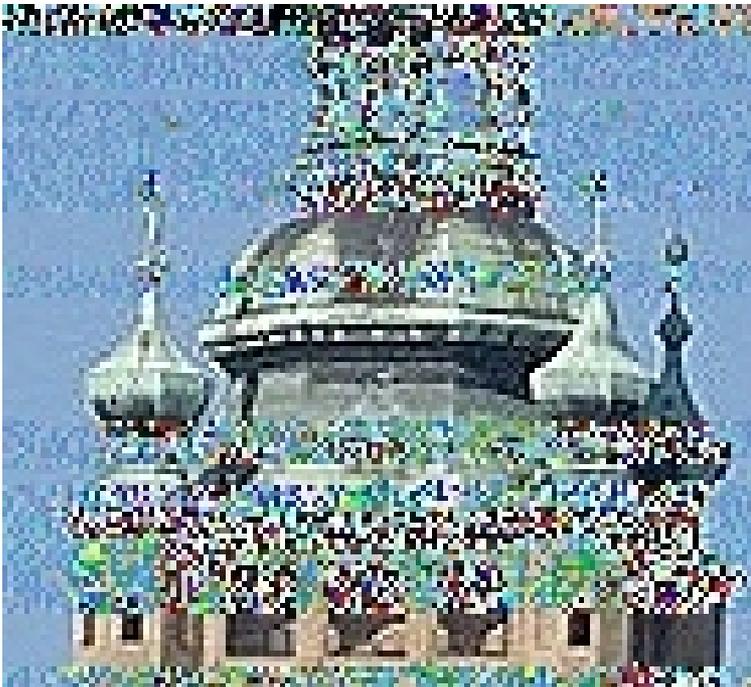Pixels on the strips border are accessed by two threads

**What about memory access conflicts?**
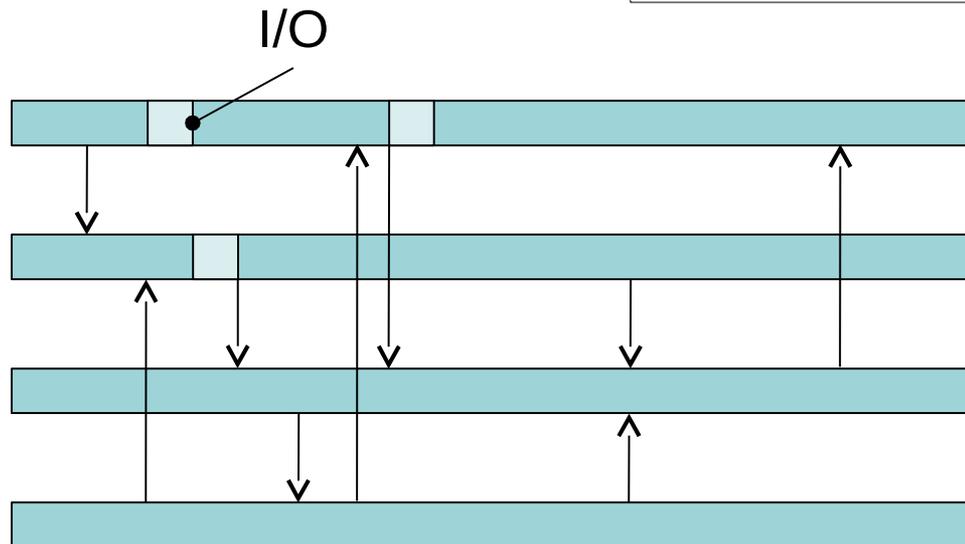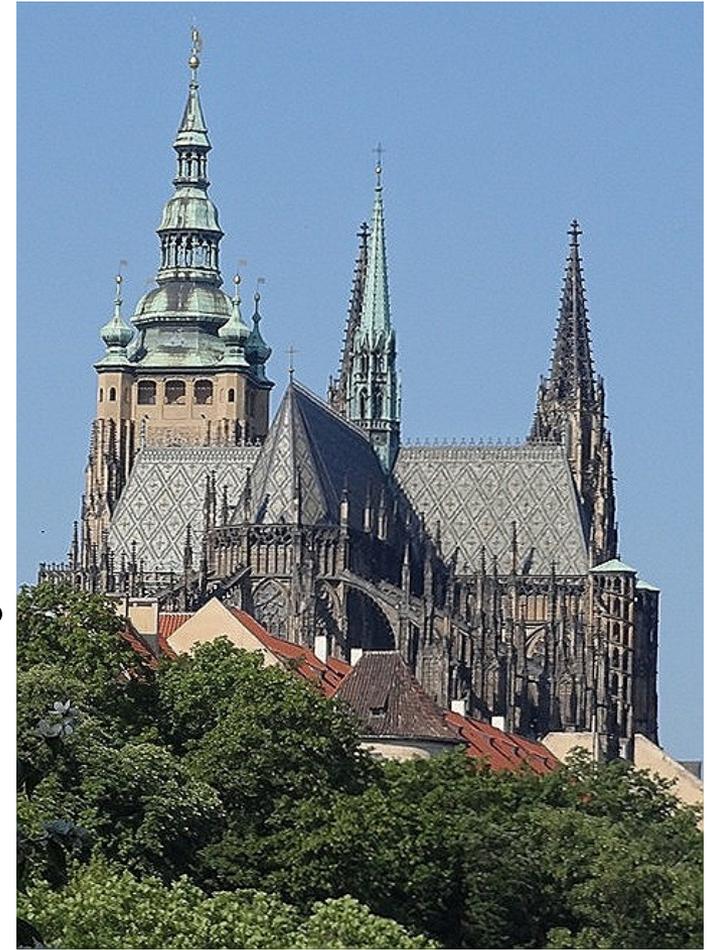
# Design of parallel program – partitioning

**Partitioning** – Domain decomposition

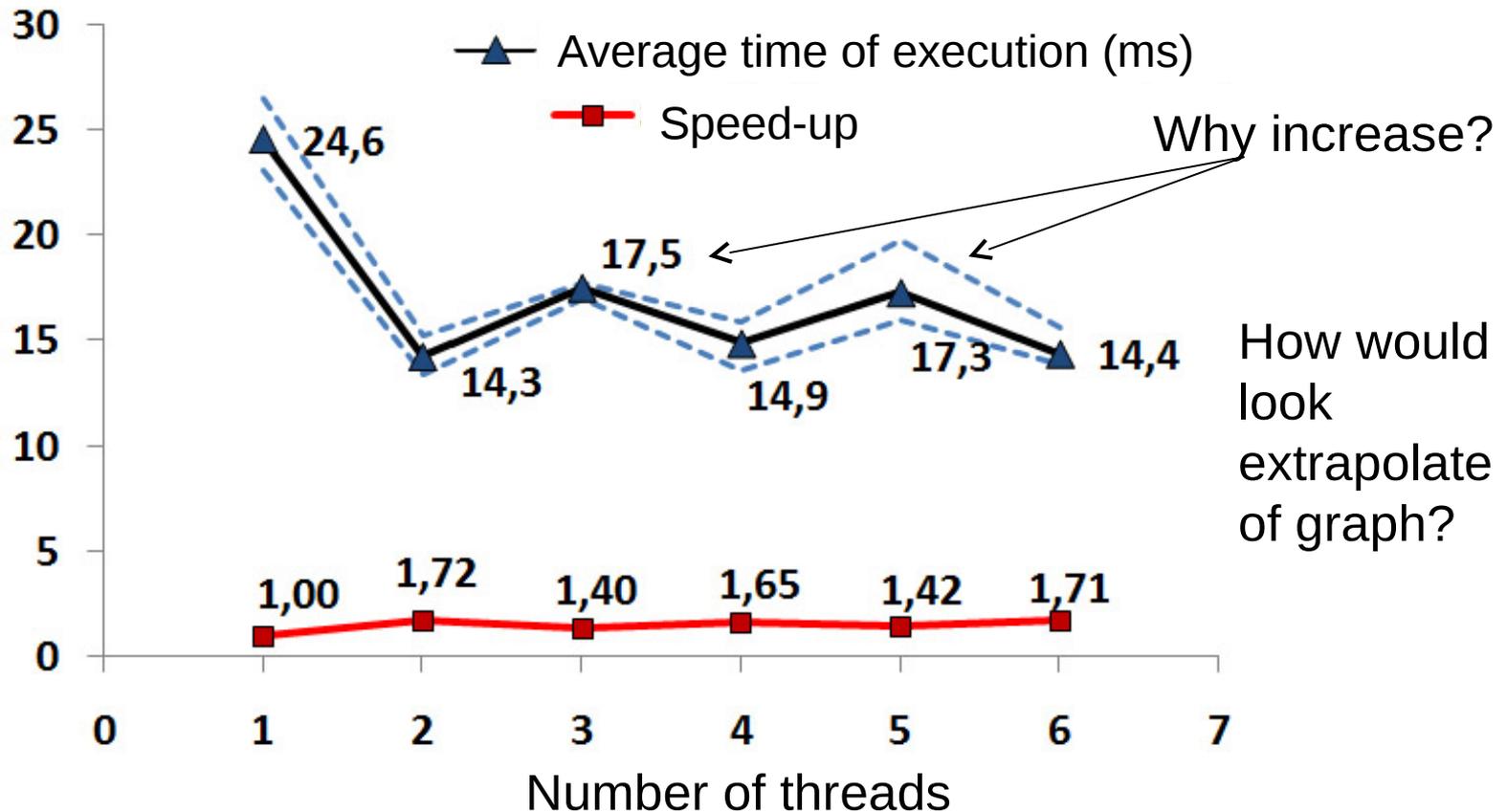Parallel program result can look even as seen in the picture:



What is wrong?

**Partitioning** – Domain decomposition

Results from St. Vitus cathedral sharpening on a two-cores CPU?



Why increase?

How would look extrapolate of graph?

Number of threads

**Partitioning** – Domain decomposition
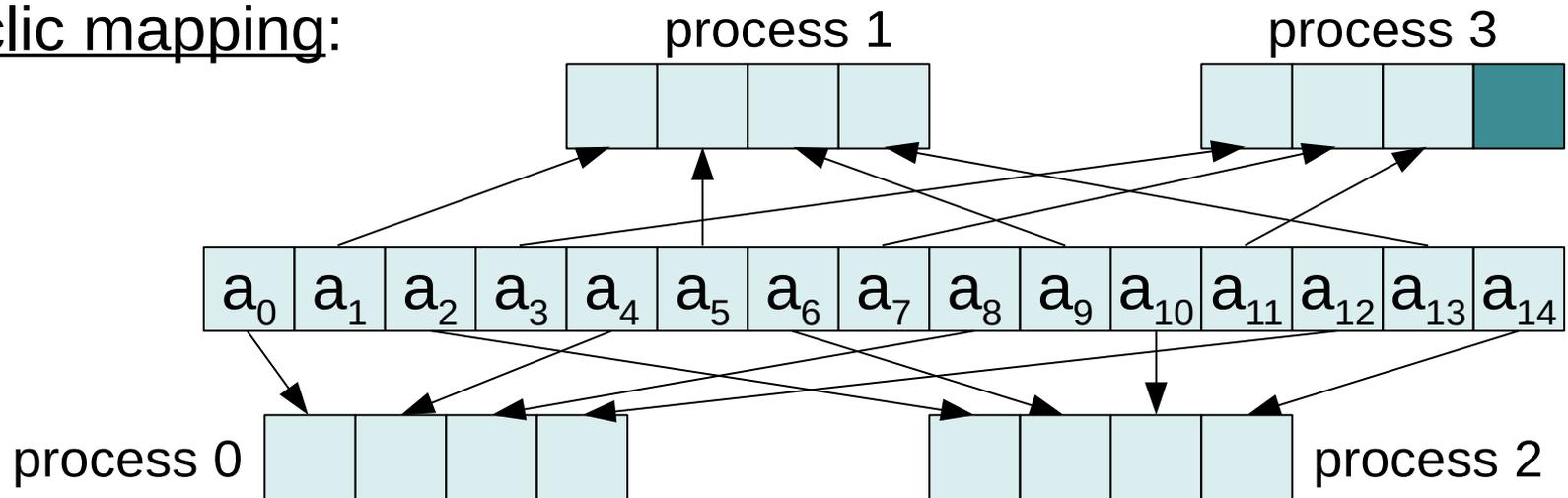
- Data set is distributed to individual processes

- $A = (a_0, a_1, \ldots, a_{n-1})$    n elements
  $P = (q_0, q_1, \ldots, q_{p-1})$    p processes
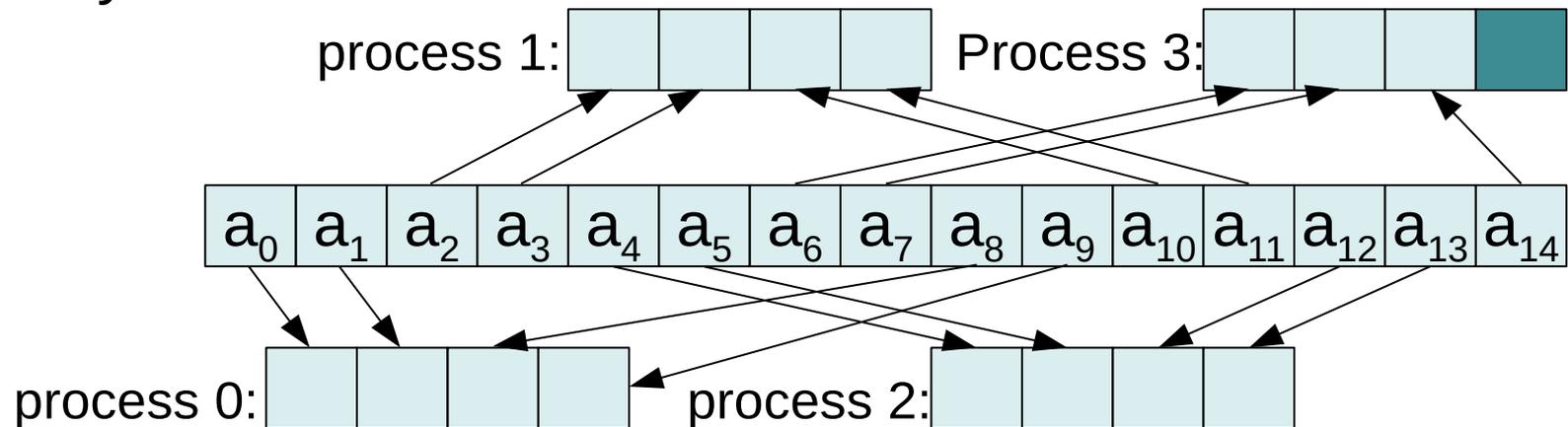
Block mapping:
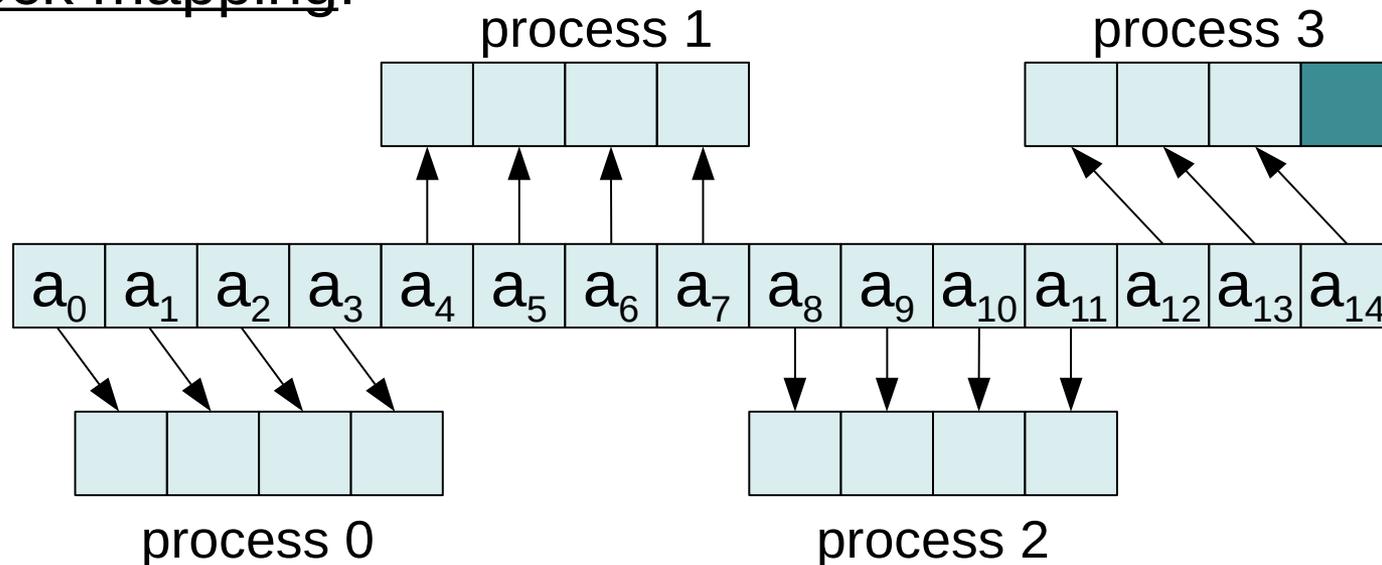
# Design of parallel program – partitioning

- <u>Cyclic mapping</u>:



- <u>Block-cyclic</u>:

- Which mapping is better? Depends on solved task properties.. (it can influence precision of result or execution time)

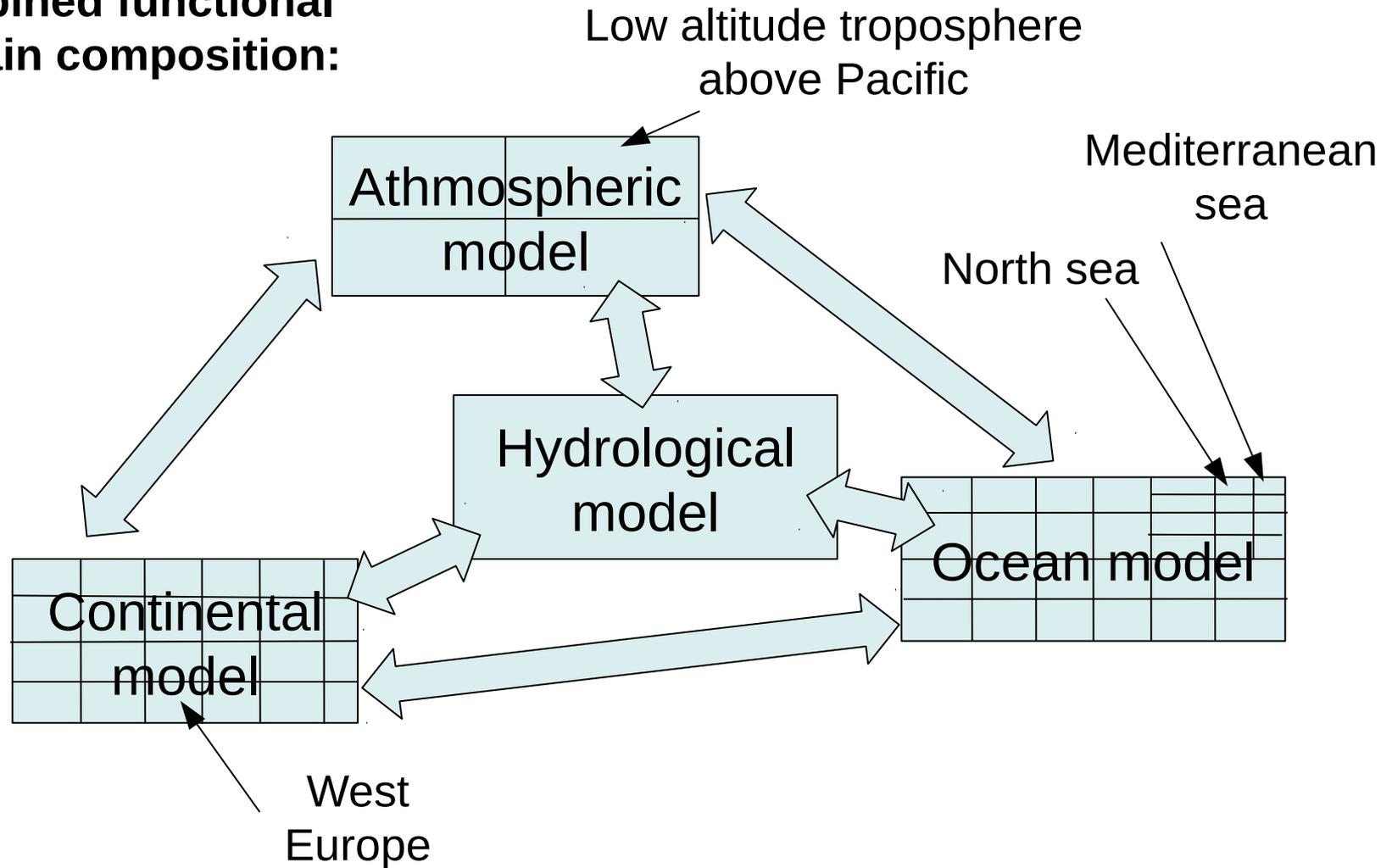$$\sum_{i=1}^{N} \frac{1}{i^2} = \sum_{i=N}^{1} \frac{1}{i^2} = \sum_{i=1}^{N} \frac{1}{(N - i + 1)^2}$$

$$\sum_{i=1}^{10^{10}} \frac{1}{i^2} \approx 1.6449340578301865,$$

$$\sum_{i=10^{10}}^{1} \frac{1}{i^2} \approx 1.6449340667482264$$

Result would depends on chosen mapping and would be somwhere between (double type and precision used)

Execution time – Computation can require higher number of iterations to achieve convergence for some elements groups/mapping…

**Combined functional
domain composition:**

Low altitude troposphere
above Pacific

Athmospheric
model

Mediterranean
sea

North sea

Hydrological
model

Ocean model

Continental
model

West
Europe

# Design of parallel program – communication

- Direct communication between processes (threads) can be hidden to programmer (depends on the model: shared memory, data-parallel model, threads, message passing…).

- Communication cost.

- **Latency** and **bandwidth** – many short messages – latency domination…, few huge messages – bandwidth is more important..

- Synchronous and asynchronous communication.

- Point-to-point (Unicast) and collective communication; Collective:
  - Broadcast (one-to-all) – one node sends its data to all nodes
  - Multicast (one-to-many)
  - Scatter – distribution – different (part of) data from one node to all nodes
  - Gather – complementary to scatter, collect data from nodes in one node
  - Reduction – collect some aspect of data into one node
  - And others.. (Allreduce, Allgather, AlltoAll) -> Collective communication.: always blocking.

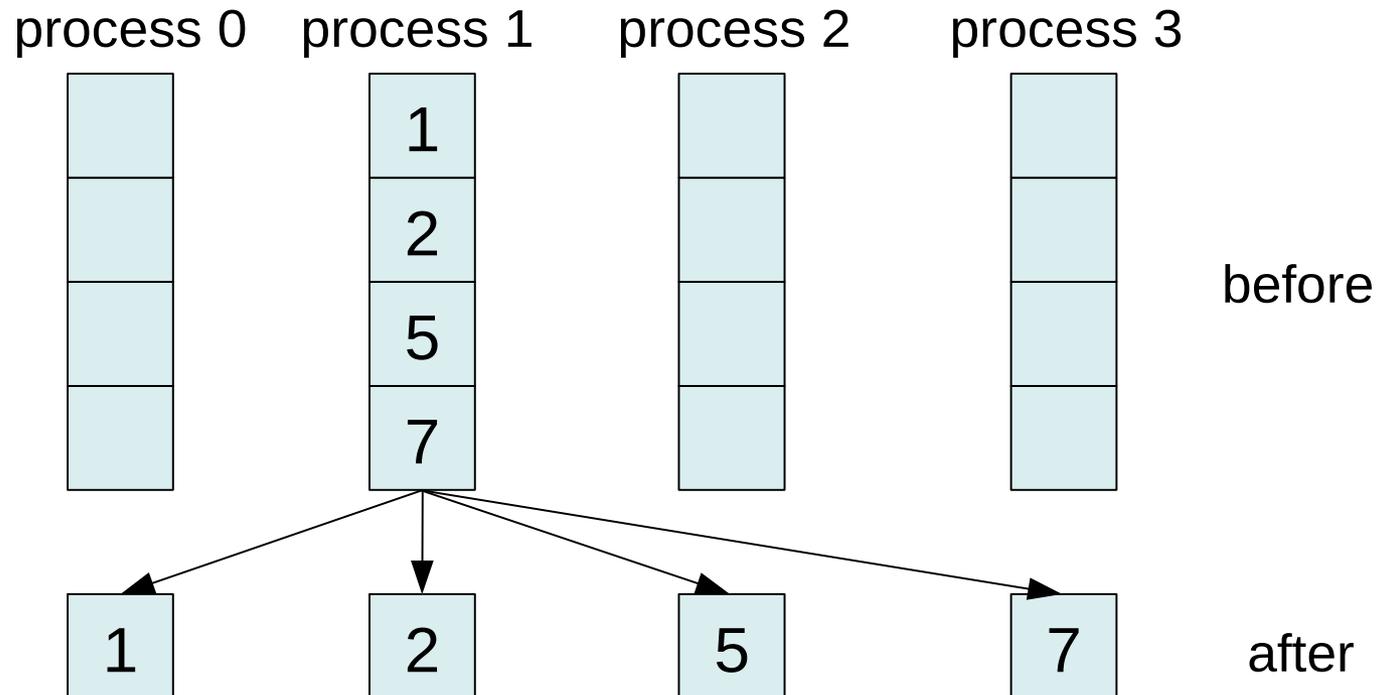# Design of parallel program – communication

Broadcast (source = 1):
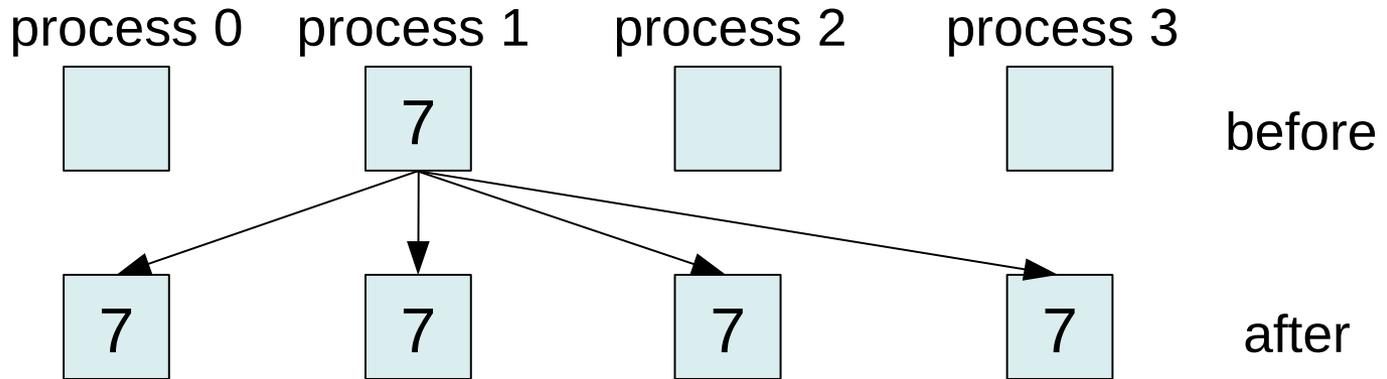


Reduce (destination = 1, operation +):
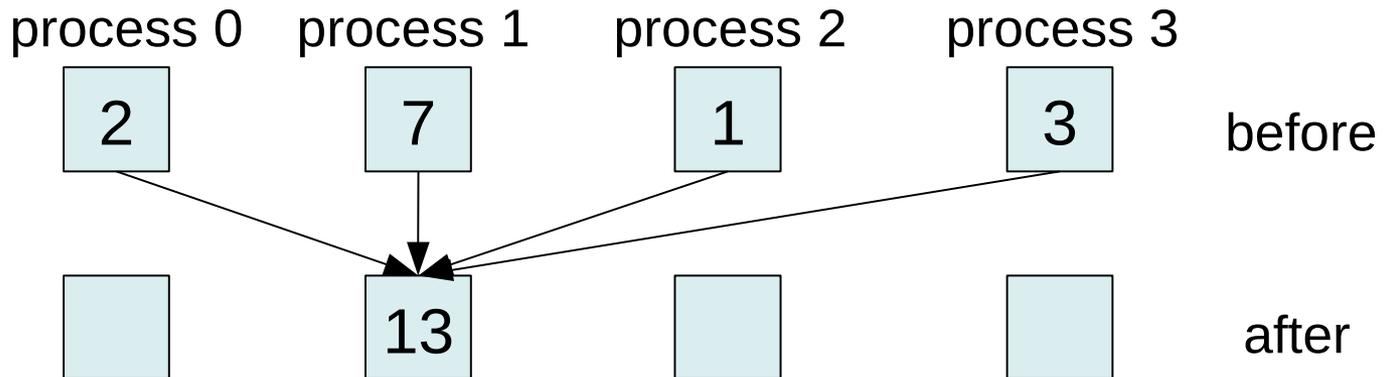
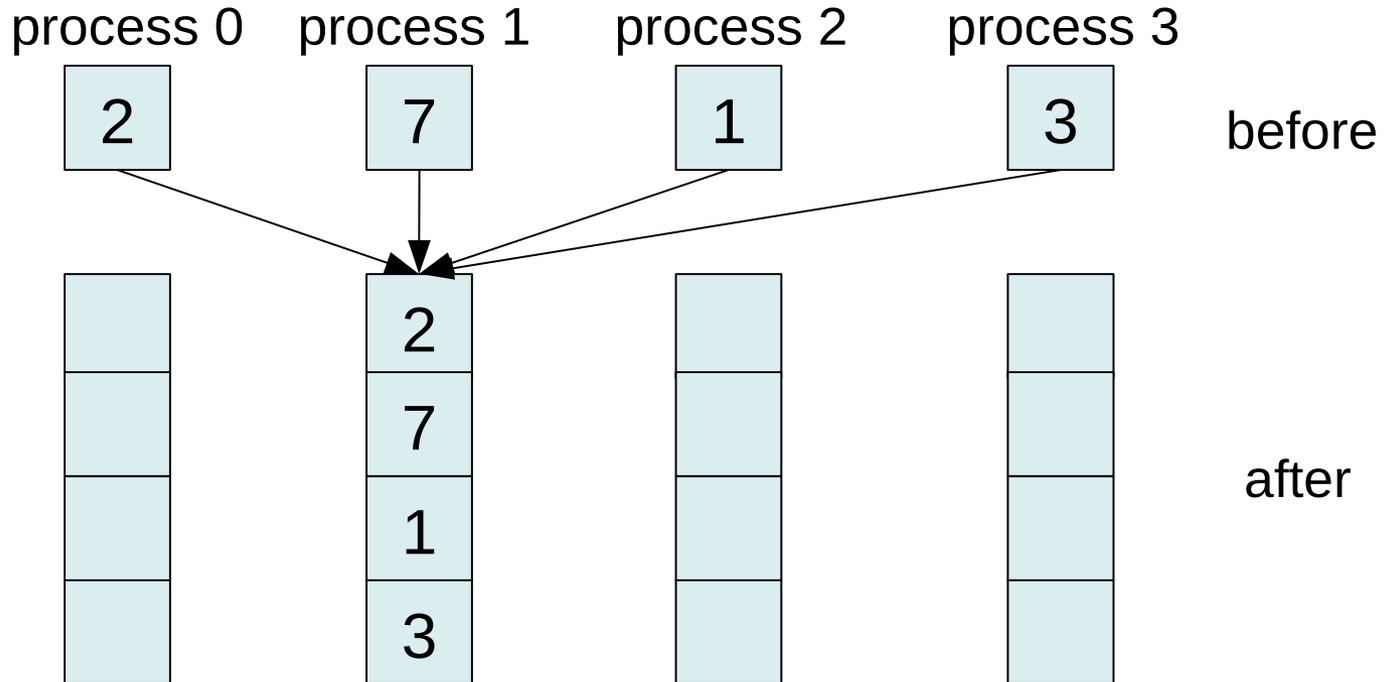# Design of parallel program – communication

Scatter (source = 1):

# Design of parallel program – communication

## Gather (destination = 1)

All to All:

process 0    process 1    process 2    process 3

| 1 | 5 | 9 | 13 |
| 2 | 6 | 10 | 14 |
| 3 | 7 | 11 | 15 |
| 4 | 8 | 12 | 16 |

before

| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

after
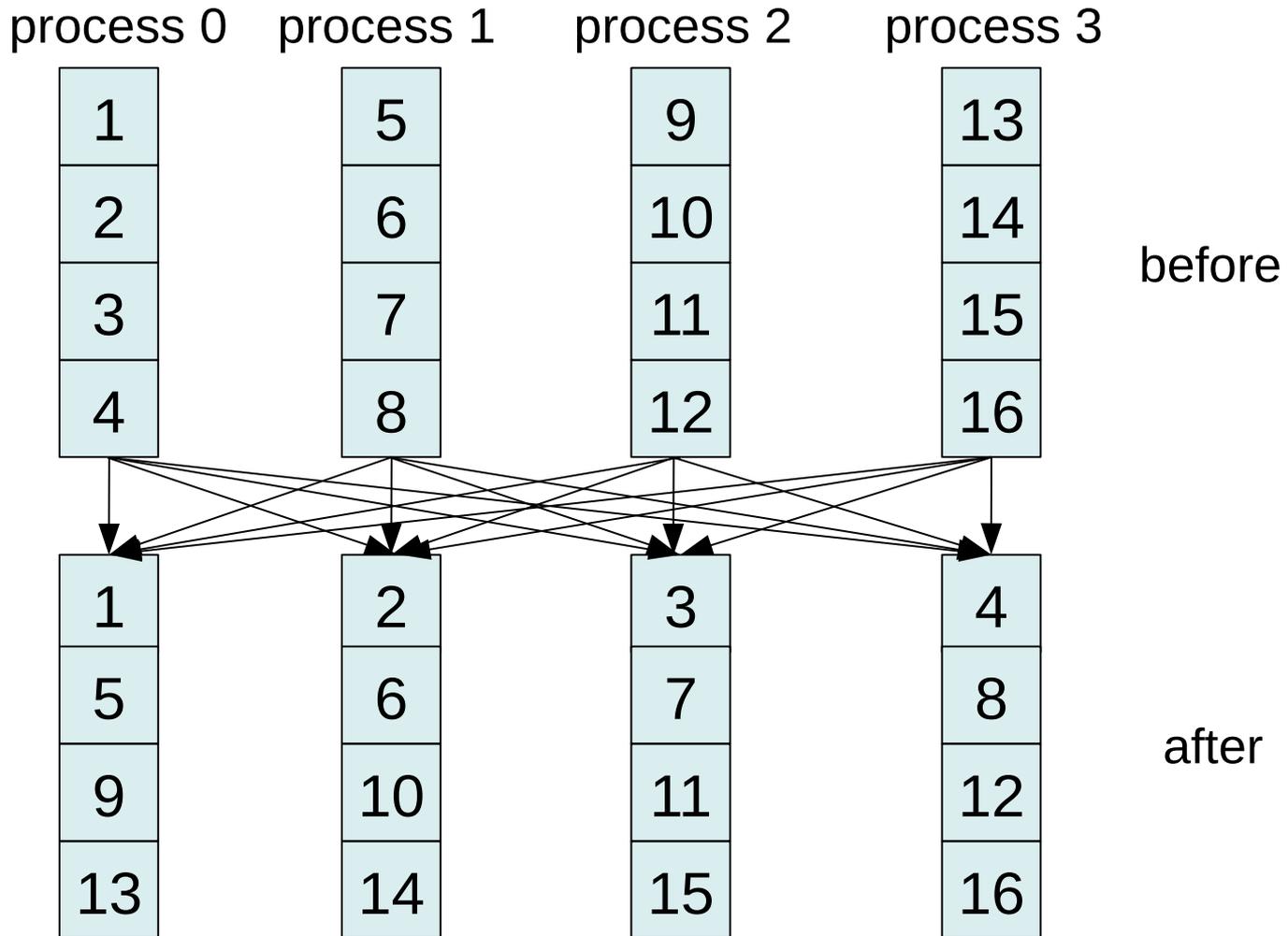
# Design of parallel program – communication

- Blocking point-to-point communication  <u>unbuffered</u>



Sending process is waiting until receiving process is ready

Receiving process waits for data – waste of its time

„Optimal" situation; but communication period usually lefts processors "unused"
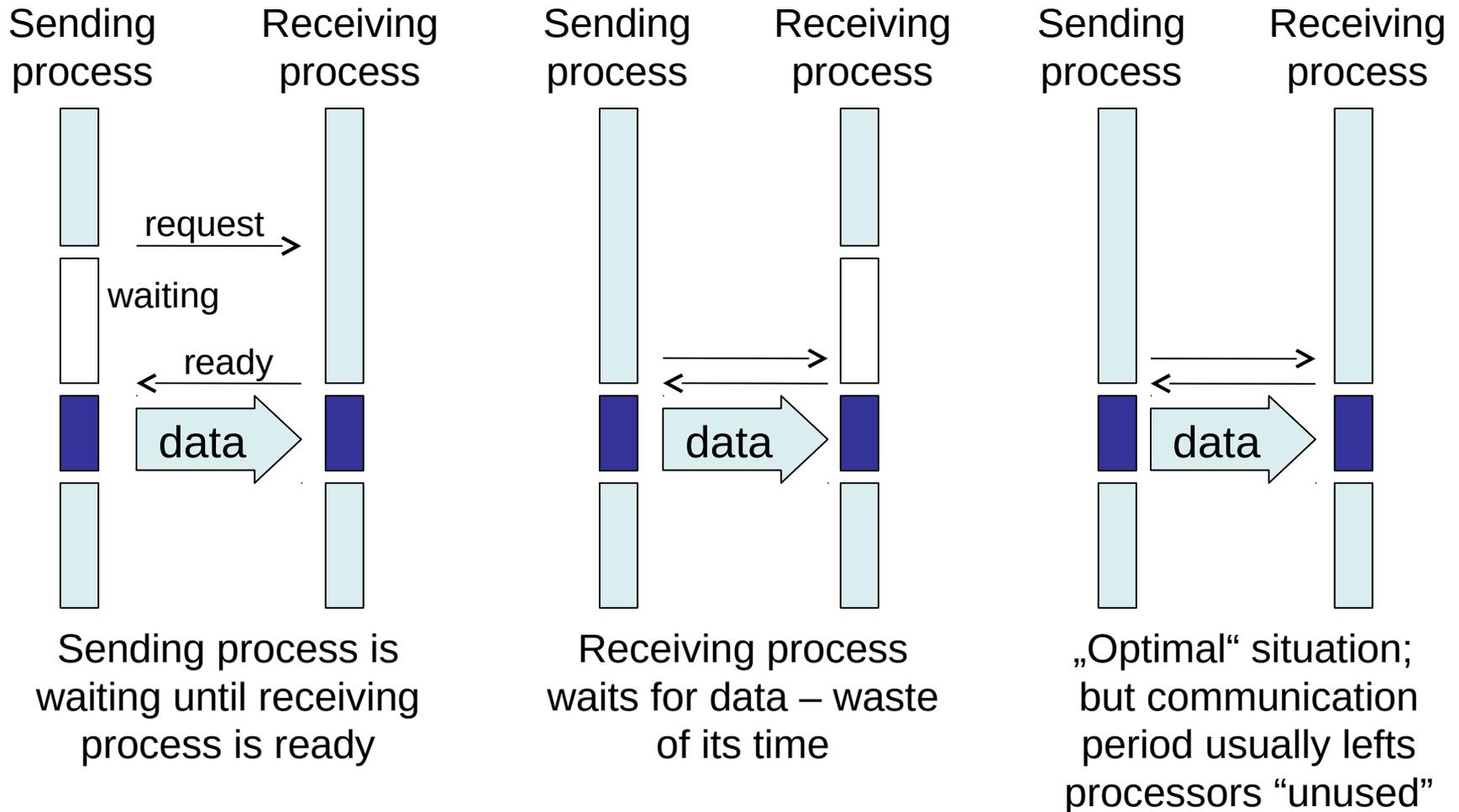
# Design of parallel program – communication

- **Dead-lock** caused by point-to-point blocking unbuffered communication.
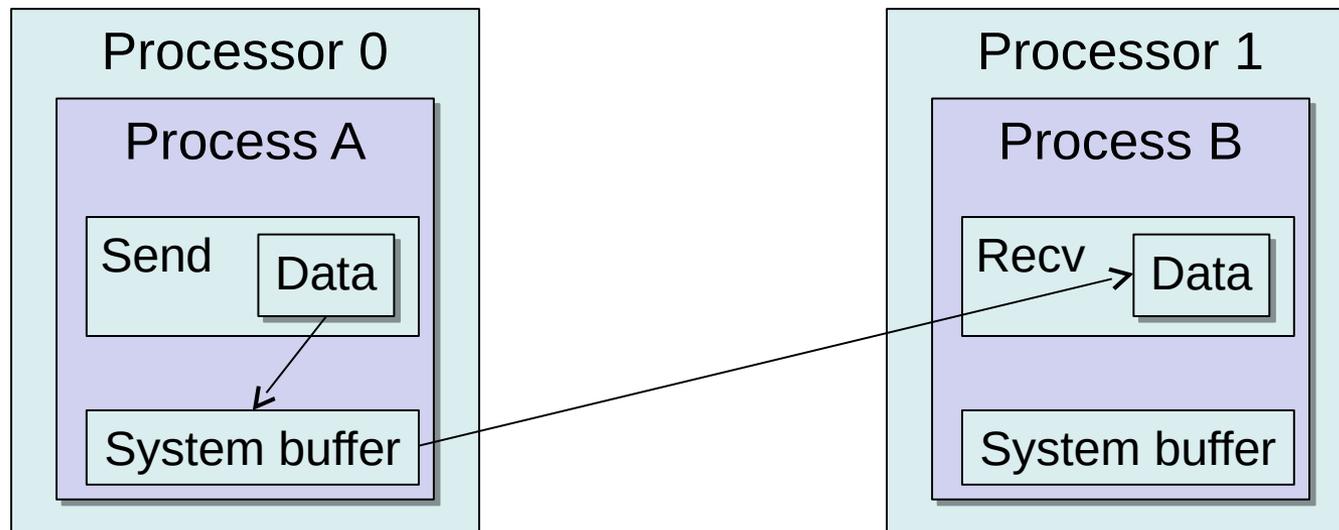
  P0:       P1:
  send()    send()
  receive() receive()

- Solution: use communication buffers may be application buffer (application memory) or system buffer (hidden to programmer).

# Design of parallel program – communication

**Blocking** communication

- <u>Sending</u> is finished (return from routine) only when application buffer can be used again freely (use of system buffer is not necessary – in such a case sending is implemented synchronously).
- Synchronous <u>sending</u> – same as above + receive is finished as well.
- Buffered <u>sending</u> – data are copied into sending buffer – used if there is not enough space in system buffer.
- <u>Receiving</u> – blocks until data are received into application buffer

**Non-blocking** communication

- <u>Sending</u> – program continues without waiting; application buffer can be used again only when sending is finished – test for release required…!!!
- Synchronous <u>sending</u> – test successful only when data are received
- Buffered <u>sending</u> – test for success required…
- <u>Receiving</u> – program continues without waiting; – test required…

# Design of parallel program – Virtual topology

Virtual topology – method of making a collection of processes act as if they are in a particular shape (i.e. MPI_Cart_create)

- Advantageous for specific communication requirements.

- Define neighborhoodness of processes (nodes) – neighboring processes can communicate directly.

- Implementation may optimize the decision of mapping processes to physical nodes…
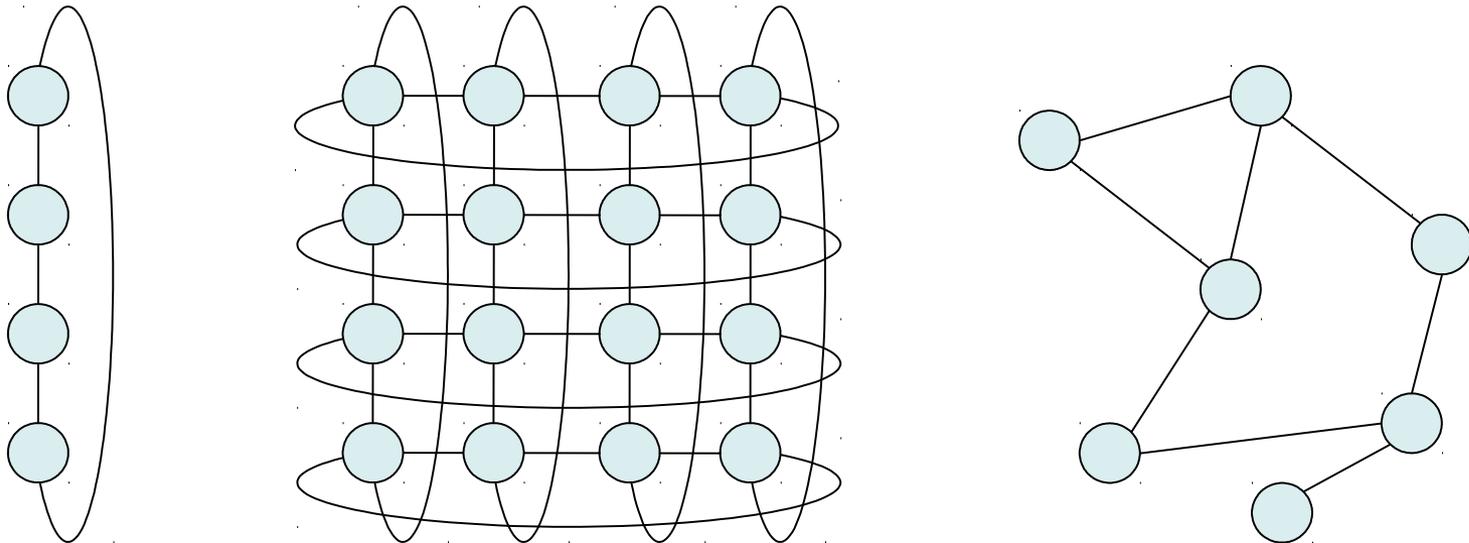
# Design of parallel program – Synchronization

- Barrier
    - each task stops on the barrier, execution continues when all tasks reach the barrier
    - for example new iteration in data processing loop

- Mutex / Lock / Semaphore
    - to solve conflict of accesses into shared memory

- Operations of synchronous communication.

# Task to schedule for parallel execution

Analyze the following program. Find maximum degree of parallelism between its 16 instructions; suppose that there are no conflicts between resources and functional units. All instructions are executed in single machine cycle. All other overhead is not accounted.

a)  Draw a 16-node program graph to visualize the relationships between these 16 instructions.

b)  Use a three-way superscalar processor to execute this program for a minimum amount of time. For one machine cycle, the processor can issue one memory access instruction (*Load* or *Store*, but not both), one *Add*/*Sub* instruction and one *Mul* instruction.

c)  Implement the program on a dual-processor system, each processor being the above-defined three-way superscalar processor.  Partition the program into two balanced halves. Find the optimal schedule of the split parallel program for two processors, achieving minimum time.

# Task to schedule for parallel execution

```
 1:  Load R1, A           /R1 ← Mem(A)/
 2:  Load R2, B           /R2 ← Mem(B)/
 3:  Mul R3, R1, R2       /R3 ← (R1) x (R2)/
 4:  Load R4, D           /R4 ← Mem(D)/
 5:  Mul R5, R1, R4       /R5 ← (R1) x (R4)/
 6:  Add R6, R3, R5       /R6 ← (R3) + (R5)/
 7:  Store X, R6          /Mem(X) ← (R6)/
 8:  Load R7, C           /R7 ← Mem(C)/
 9:  Mul R8, R7, R4       /R8 ← (R7) x (R4)/
10:  Load R9, E           /R9 ← Mem(E)/
11:  Add R10, R8, R9      /R10 ← (R8) + (R9)/
12:  Store Y, R10         /Mem(Y) ← (R10)/
13:  Add R11, R6, R10     /R11 ← (R6) + (R10)/
14:  Store U, R11         /Mem(U) ← (R11)/
15:  Sub R12, R6, R10     /R12 ← (R6) – (R10)/
16:  Store V, R12         /Mem(V) ← (R12)/
```

# Resources and links

-  John L. Hennessy; David A. Patterson (2003). Computer Architecture: a quantitative approach (3rd ed.). Morgan Kaufmann. ISBN 1-55860-724-2.

- https://computing.llnl.gov/tutorials/openMP/

- https://www.open-mpi.org/doc/v2.1/

- http://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture09-multithreading.pdf

- http://meseec.ce.rit.edu/eecc756-spring2011/756-3-17-2011.ppt

- http://www.umsl.edu/~siegelj/CS4740_5740/MPIandOpenMP/vtopologies.ppt