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

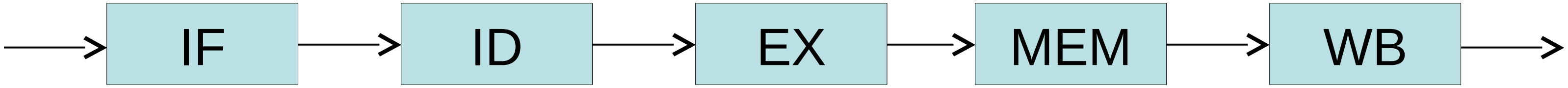
Instruction level parallelism (ILP)

- 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 (squential instructions flow)
  - Superscalar execution (in a broader sense) – spatial parallelism

Pipelining:

- Suppose that instruction execution can be divided to 5 stages



IF → ID → EX → MEM → WB

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 τ_i is *propagation delay* of *i*-the *pipeline stage*.

Instruction level parallelism – pipelining

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

$5\ \tau$ τ τ τ τ τ čas

- Execution time of *n* instructions *k*-stages pipeline:

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

Assumption: ideally balance pipeline

- Speedup:

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

Instruction level parallelism – pipelining

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

- Hazards:

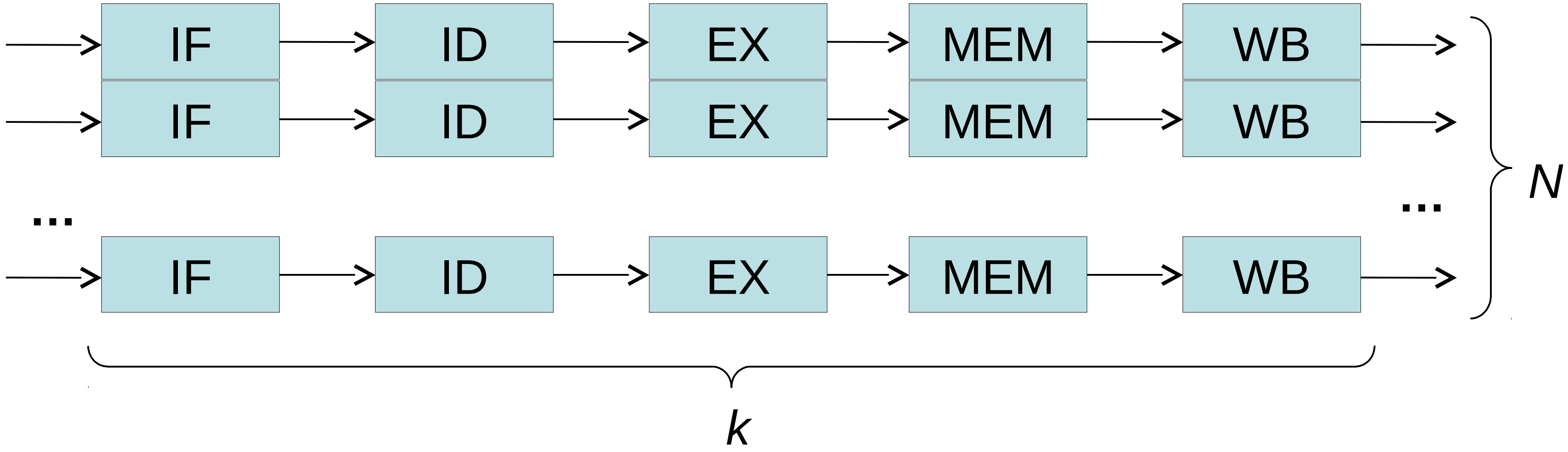
  Structural hazards (solved by duplication),
  Data hazards (the consequence of data dependencies)
  Control hazards (instructions modifying PC)...

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

- Notice: Deeper pipeline (more stages) results in less gates in each stage which allows to increase clock frequency. But more stages means more complex control and forwarding circuitry and higher cost of pipeline flush (instructions has to be better (re)ordered to utilize 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 instructions flow
- Data dependencies are dynamically identified by hardware (versus by software during compilation → static: WLIV)

Instruction level parallelism

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

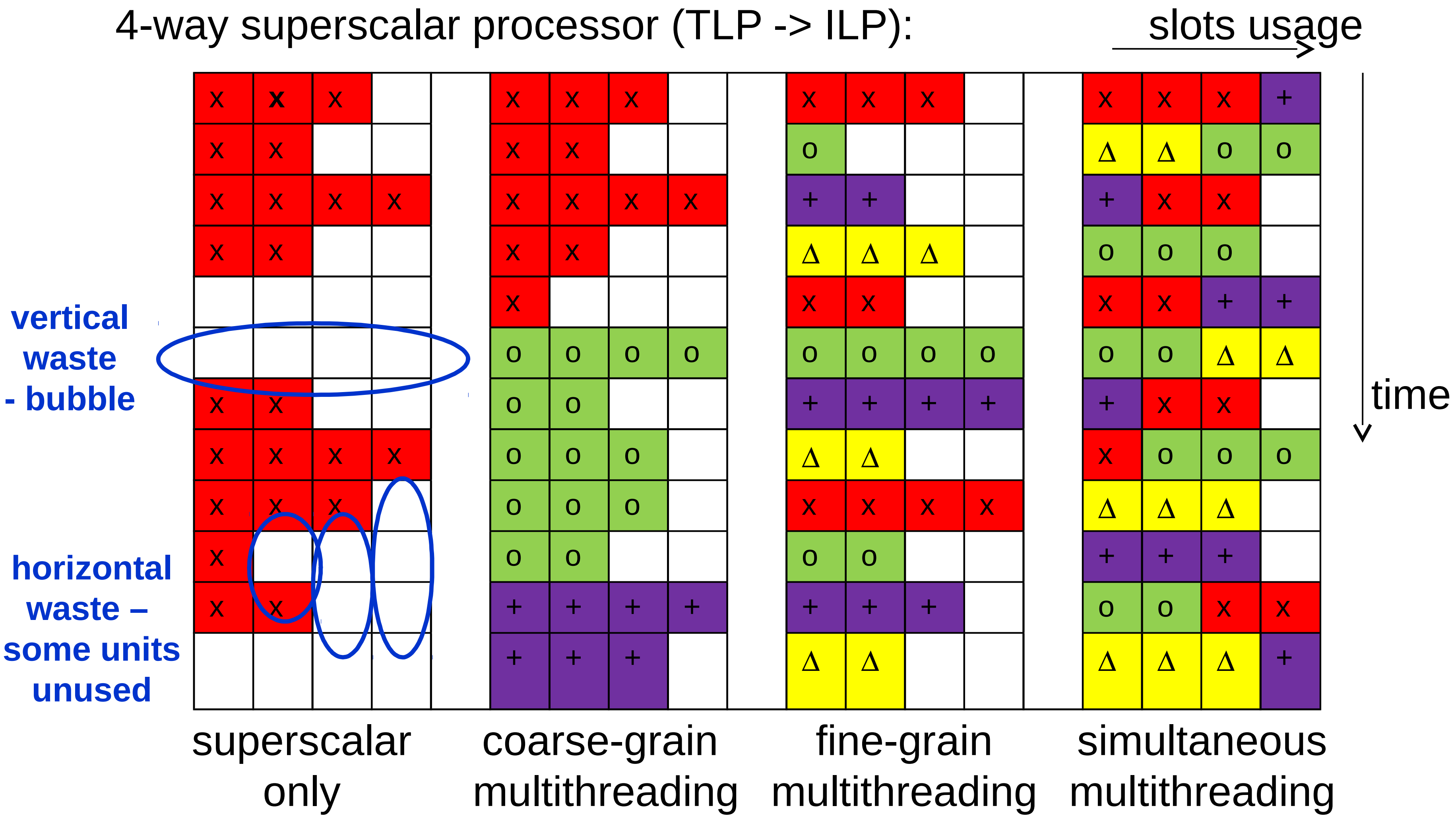
- Propagation results within the pipeline (**forwarding**)
- Instructions **out-of-order execution**
- **Register renaming**
- **Speculative execution**
- **Branch prediction**
- VLIW (Very Long Instruction Word) and EPIC – MIMD on the lowest level
- Details in lectures from 02 until 06…

Thread level parallelism (TLP)

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

Thread level parallelism

4-way superscalar processor (TLP -> ILP):


slots usage
time
vertical waste - bubble
horizontal waste – some units unused
superscalar only
coarse-grain multithreading
fine-grain multithreading
simultaneous multithreading

superscalar only

x	x	x	
x	x		
x	x	x	x
x	x		
x	x		
x	x	x	x
x	x	x	
x			
x	x		

coarse-grain multithreading

x	x	x	
x	x		
x	x	x	x
x	x		
x			
o	o	o	o
o	o		
o	o	o	
o	o	o	
o	o		
+	+	+	+
+	+	+	

fine-grain multithreading

x	x	x	
o			
+	+		
Δ	Δ	Δ	
x	x		
o	o	o	o
+	+	+	+
Δ	Δ		
x	x	x	x
o	o		
+	+	+	
Δ	Δ		

simultaneous multithreading

x	x	x	+
Δ	Δ	o	o
+	x	x	
o	o	o	
x	x	+	+
o	o	Δ	Δ
+	x	x	
x	o	o	o
Δ	Δ	Δ	
+	+	+	
o	o	x	x
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 MT – long delays are eliminated by thread; but still empty cycles (start-up period) and low utilization of resources (execution units);
- Finegrain MT – thread switch in each cycle; but still not all resources 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 as ideal combination thread computing FP and thread moving data)

Task level parallelism (TLP)

- Task-level parallelism (TLP) – also function parallelism or control parallelism – software/OS level/controlled
- multiprocessing taken 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 therads/core, POWER8 and 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. Program recognizes on which processor runs and executes only part of program assigned to given processor

Task level parallelism

- TLP MPMD program: program is divided to modules (which communicate together!) – demanding 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 demans are HPC programs 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 → works with different data

- parallelism – explicitly programmed (OpenMP), implicit (on compilators level)
- support of programming languages for parallel computing

Dependencies in programs

- Prerequisite for parallel execution of prrogram segments – independence on other segments (at least for some/fundamental part of 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:

- Data – defines succession relationships between commands
- Resources – resources of given system (conflict of shared resources – registers, memory, ALU, FPU, processors…)
- Control – order of operations execution cannot be determined before program is started (condtional 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 ∃ 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 $\overset{+}{\rightarrow}$ S2:
  I(S1) ∩ O(S2) , S1-> S2
- Output dependency (Write-after-Write: WAW) S1 $\overset{\ominus}{\rightarrow}$ 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 variable is indexed
  - Variable appears more than once with indexes which have multiply loop variable by different coefficients
  - Index defined by loop variable is nonlinear
  - Etc.

Data dependency

P1: C = D*E

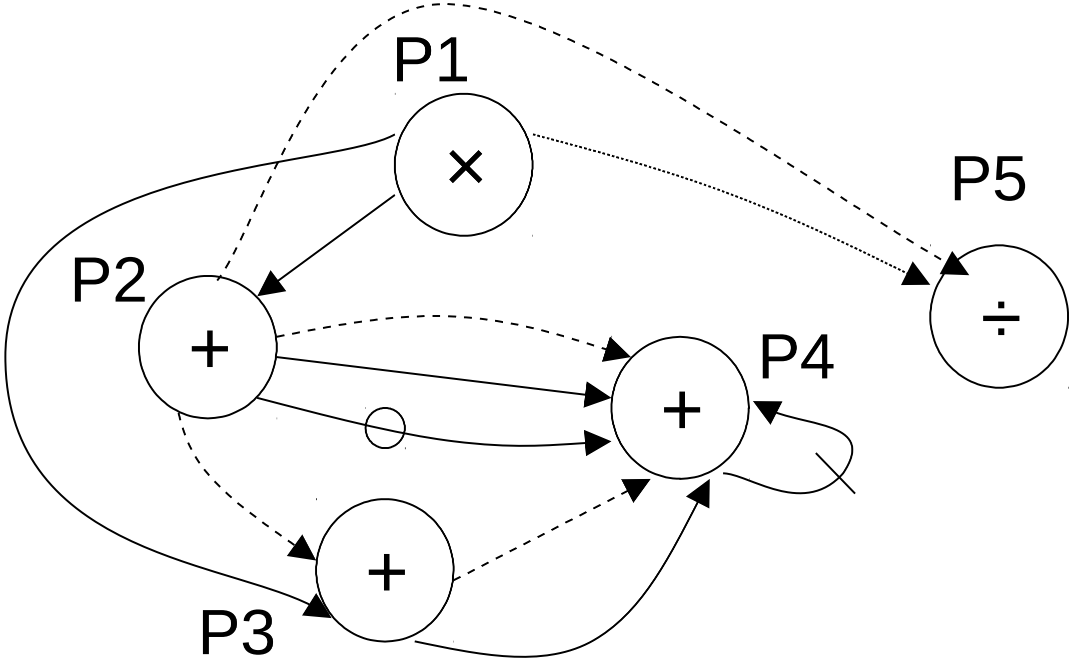
P2: M = G+C

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

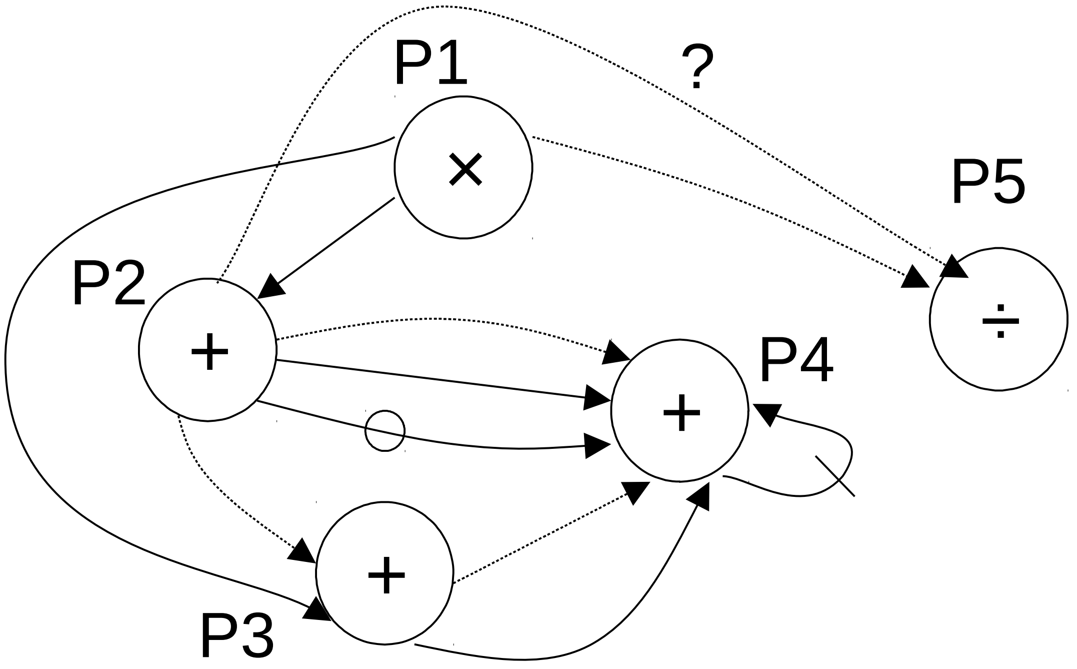
P2: M = G+C

P3: A = B+C

P4: M = A+M

P5: F = G/E

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



Remember, this need not to be single operation only... Use generalized way

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 ($P_i \parallel P_j$) if:
  $$[I(P_i) \cap O(P_j)] \cup [O(P_i) \cap I(P_j)] \cup [O(P_i) \cap O(P_j)] = \emptyset$$

- $P_1 \parallel P_2 \parallel \ldots \parallel P_k$ if and only if $P_i \parallel P_j$ for $\forall i \neq j$
- Commutativity applies ($P_i \parallel P_j = P_j \parallel P_i$)
- Transitivity doe not apply ($P_i \parallel P_j \wedge P_j \parallel P_k$ does not imply $P_i \parallel P_k$)
- Associativity applies ($[P_i \parallel P_j] \parallel P_k = P_i \parallel [P_j \parallel P_k]$)

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 ⇒ can be executed simultaneously or arbitrarily ordered

Next sequnce cannot be executed:

1. P1 || P4 || P5
2. P2 || P3

This is allowed:

1. P1
2. P2 || P3
3. P4 || P5

This also:

1. P1
2. P2 || P3 || P5
3. P4

This as well:

1. P1 || P5
2. P2 || P3
3. P4

Bernstein's conditions of parallelism

Program fragment

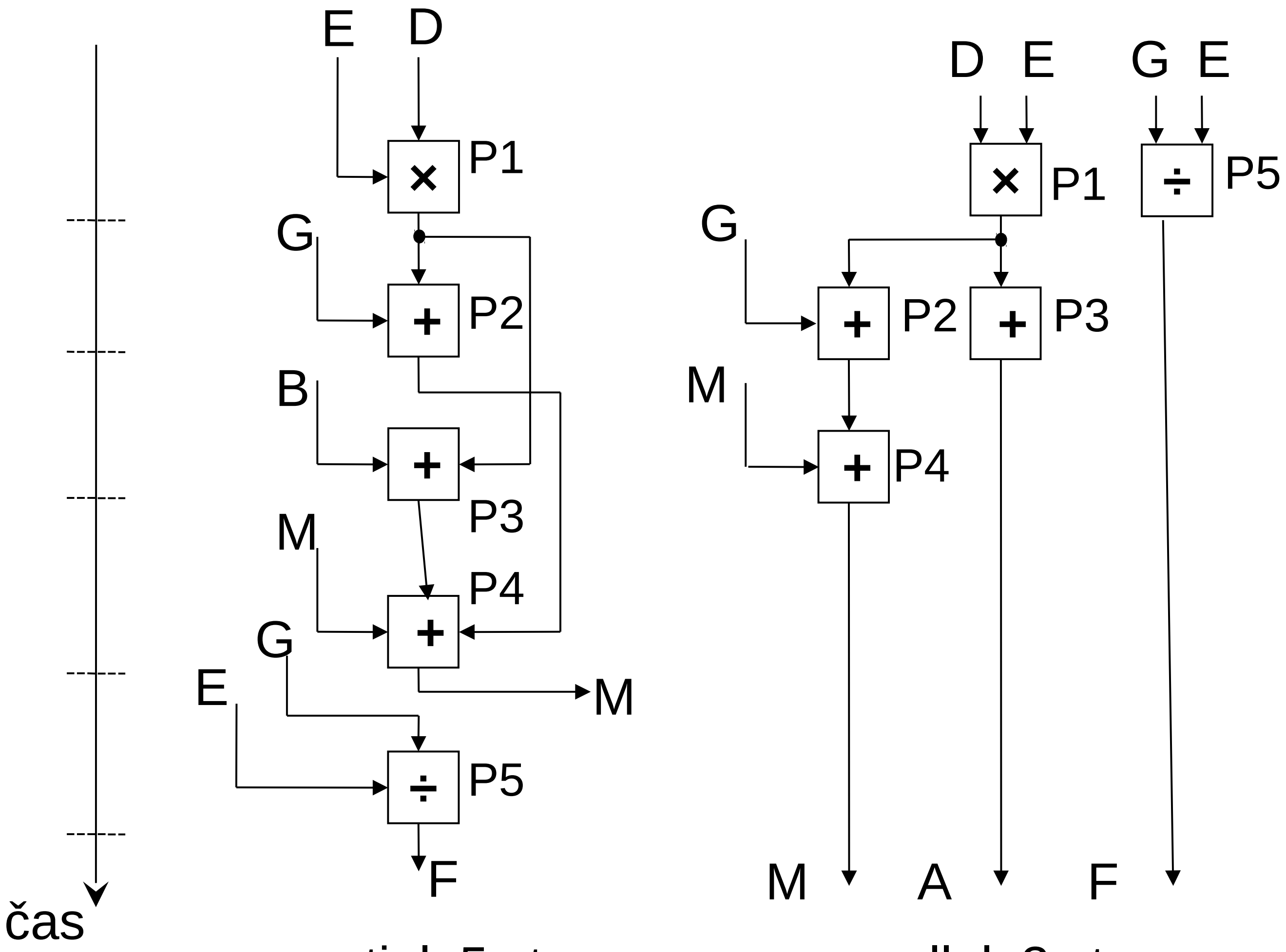
P1: C = D*E

P2: M = G+C

P3: A = B+C

P4: M = A+M

P5: F = G/E



sequential: 5 steps

parallel: 3 steps
Two adders required

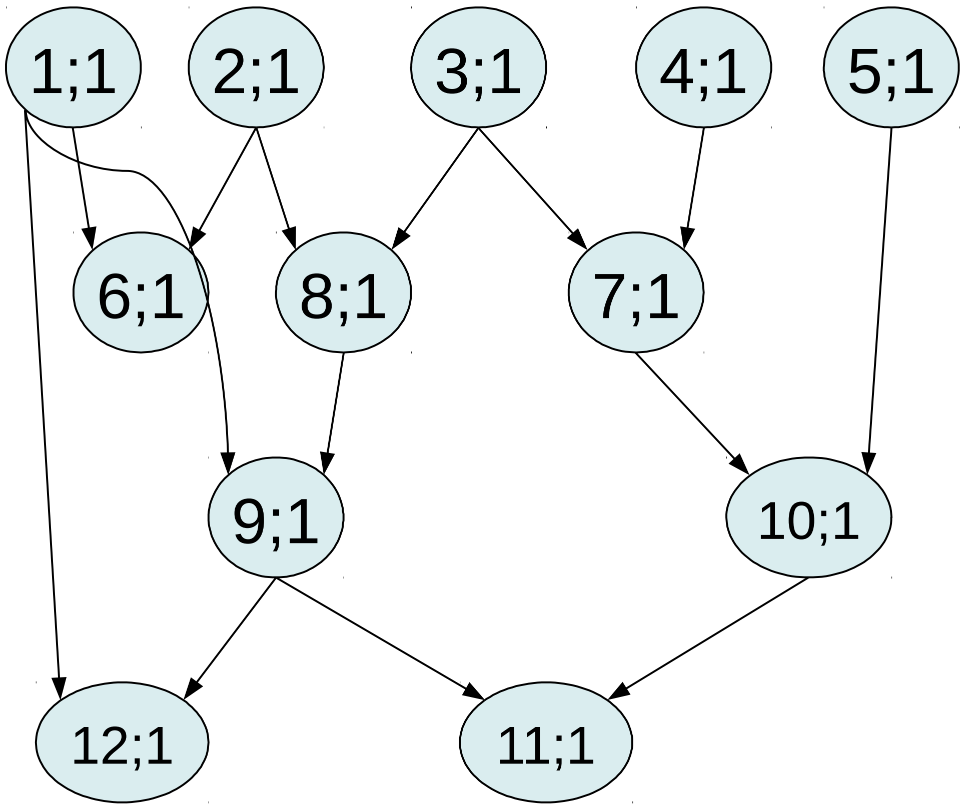
Multiprocessor illustrative example No 1

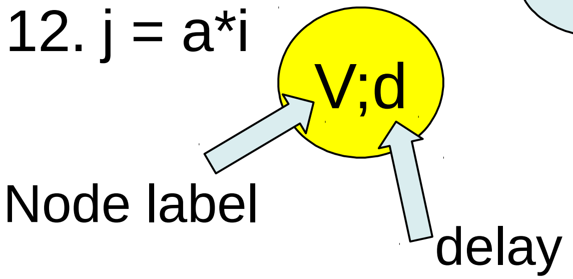
1. a = 1
2. b = 2
3. c = 3
4. d = 4
5. e = 5
6. f = a*b
7. g = c*d
8. h = b–c
9. i = a+h
10. b = g+e
11. c =b*i
12. j = a*i

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

Multiprocessor illustrative example No 1

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

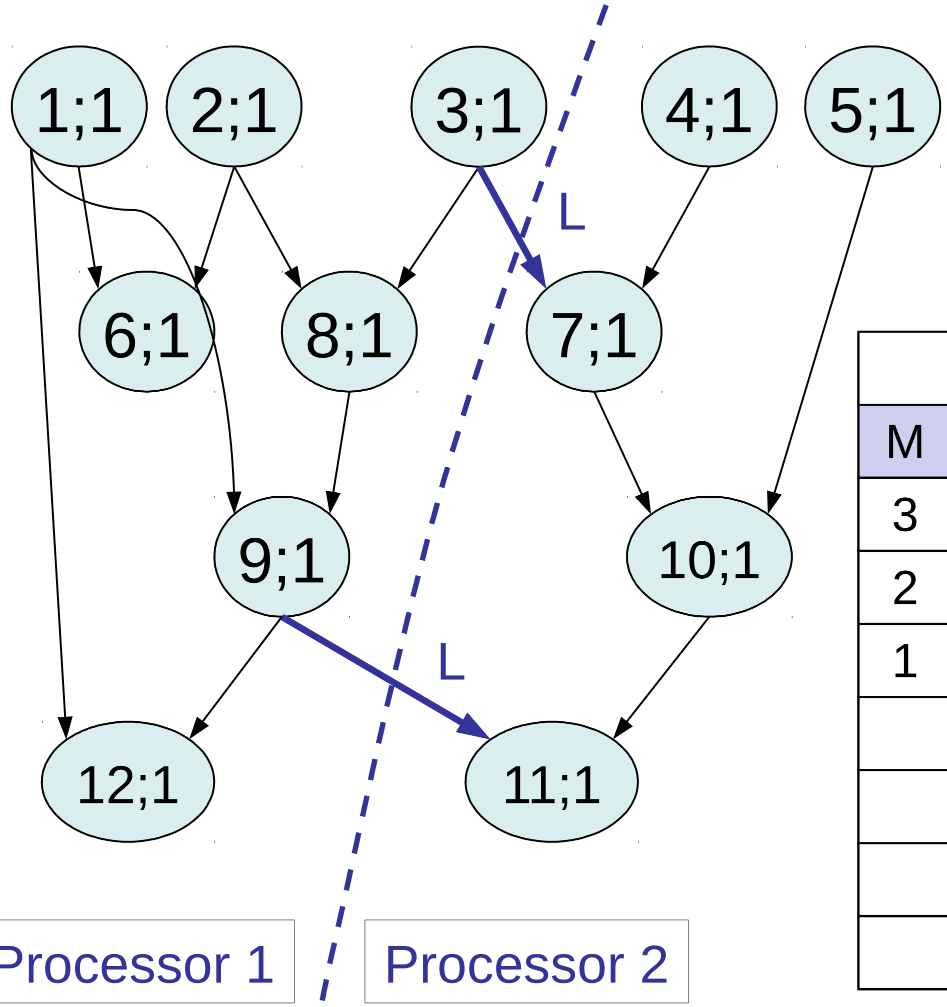




The node weight measures of the amount of work assigned to that node. The simplest measure is the number of instructions (or the execution time of the node - the number of cycles).

Multiprocessor illustrative example No 1

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

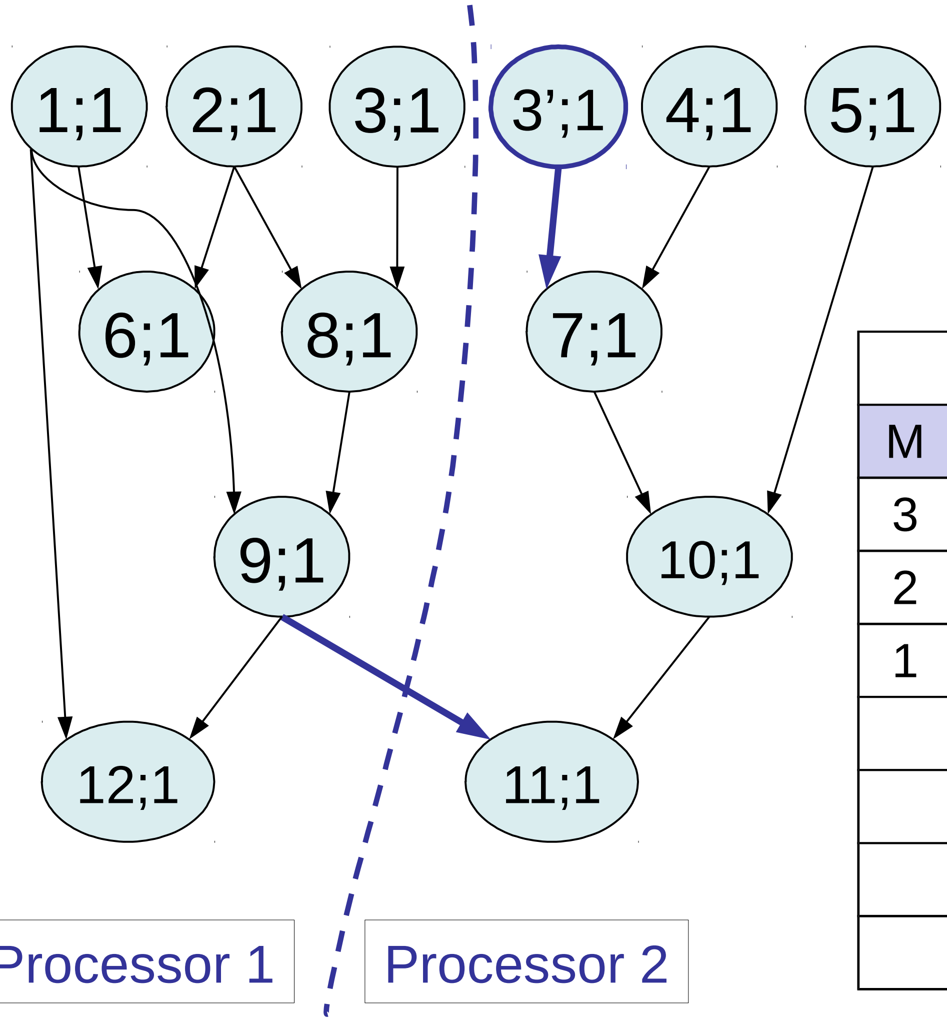


M – memory
C – compute
S – send
R – receive

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		

Multiprocessor illustrative example No 1

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

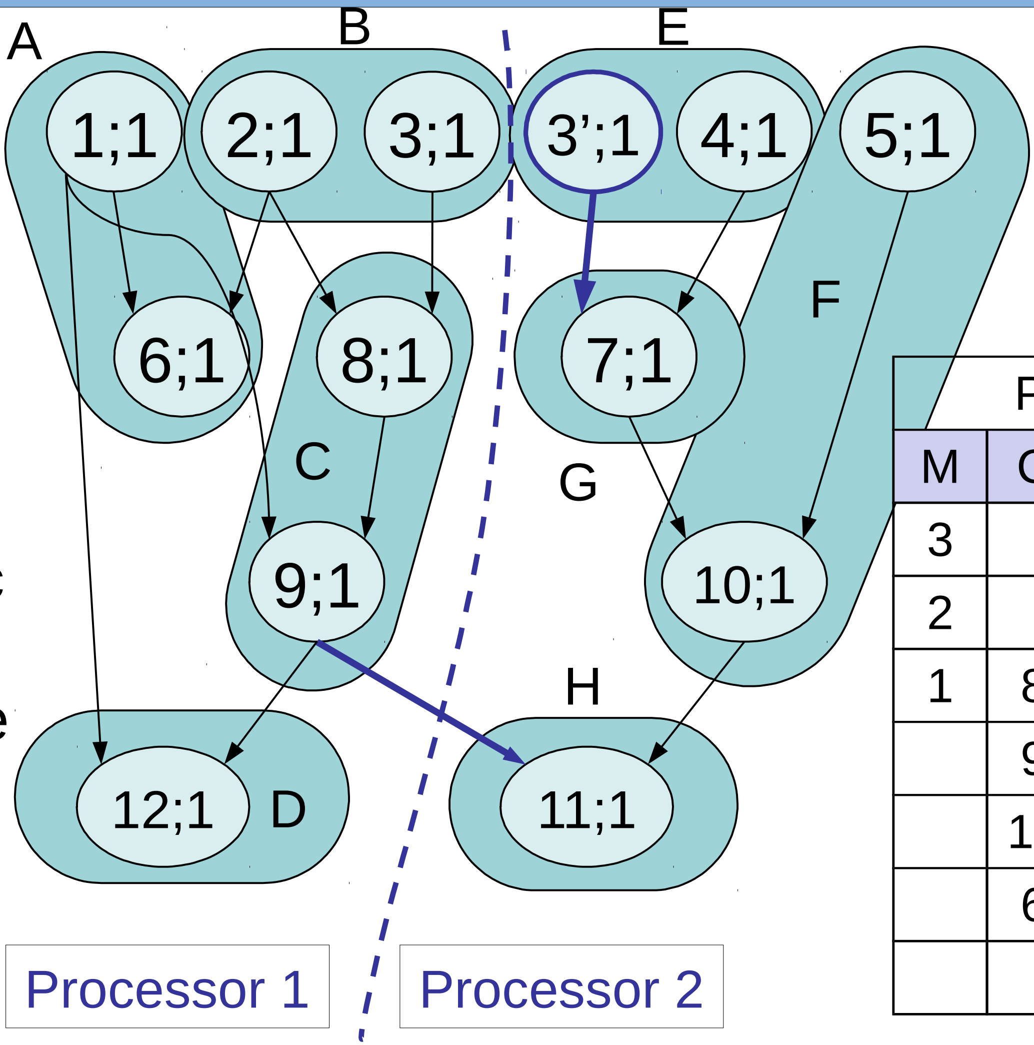


Significant speedup can be achieved by node duplication

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		

Multiprocessor illustrative example No 1

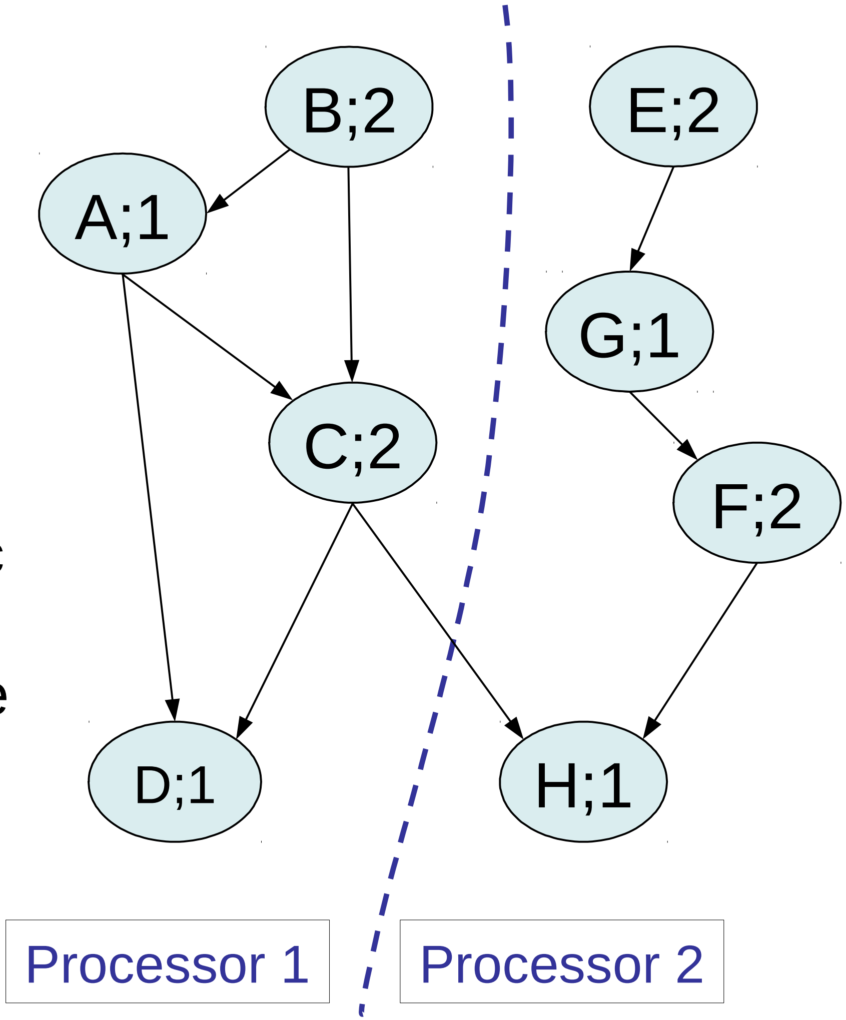
1. a = 1
2. b = 2
3. c = 3
4. d = 4
5. e = 5
6. f = a*b
7. g = c*d
8. h = b–c
9. i = a+h
10. b = g+e
11. c =b*i
12. j = a*i



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		

Multiprocessor illustrative example No 1

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

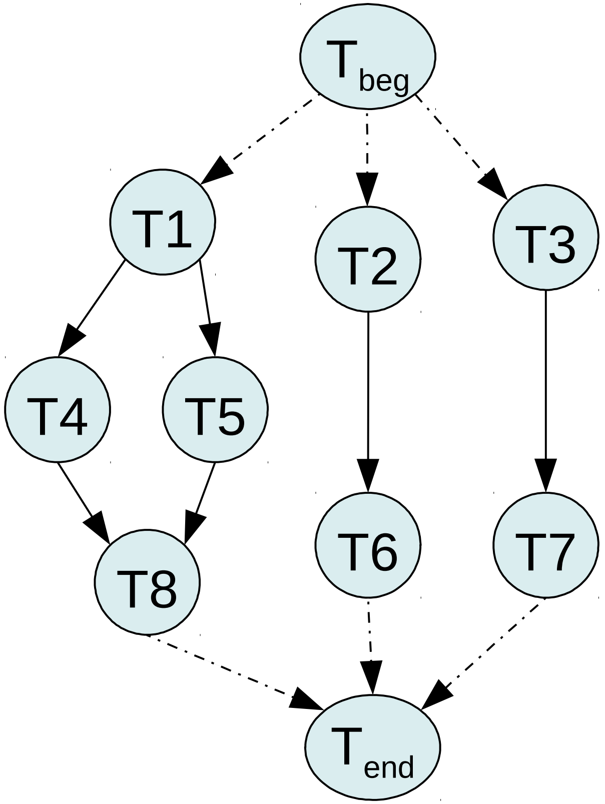


Grain packing can provide a significant simplification of scheduling while maintaining same speedup.

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

Multiprocessor illustrative example No 2

Lets are 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 processor

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

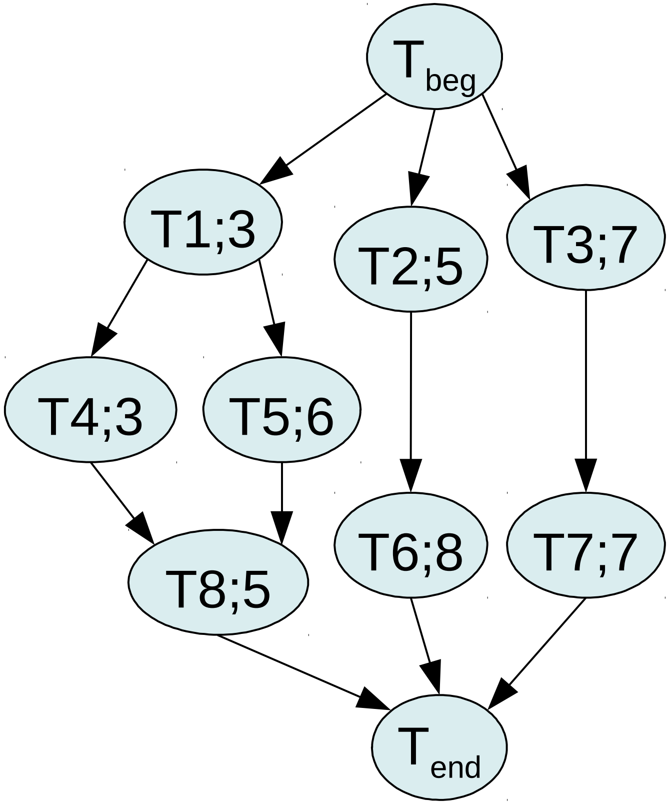
How to divide individual tasks between them ???

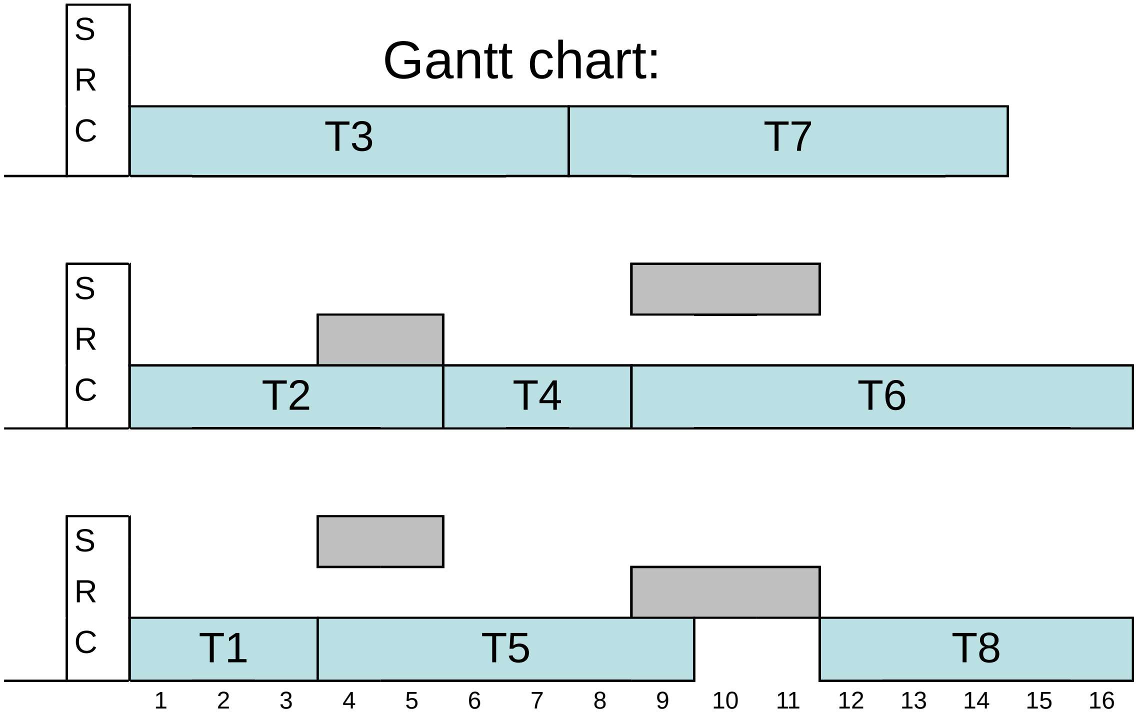
Multiprocessor illustrative example No 2

Lets are 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



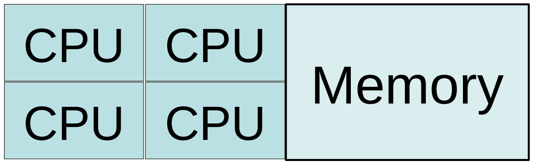


Resources/processors utilization

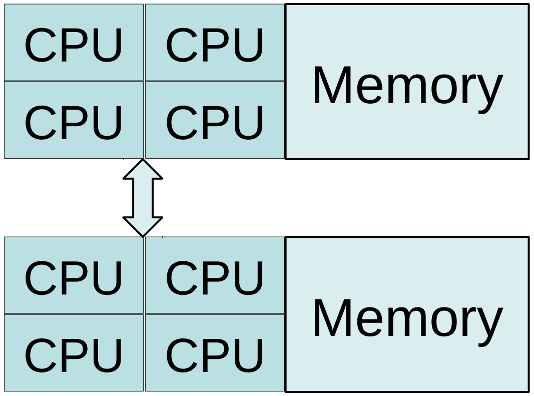
S = 44 / 16 = 2,75

Parallel computers' memory architectures

- **Shared memory systems (SMS)** – access to 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) – same memory access time, SMP (Symetric 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 – when SMP node can access into memory of other node; when Cache Coherency preserved then CC-NUMA



UMA

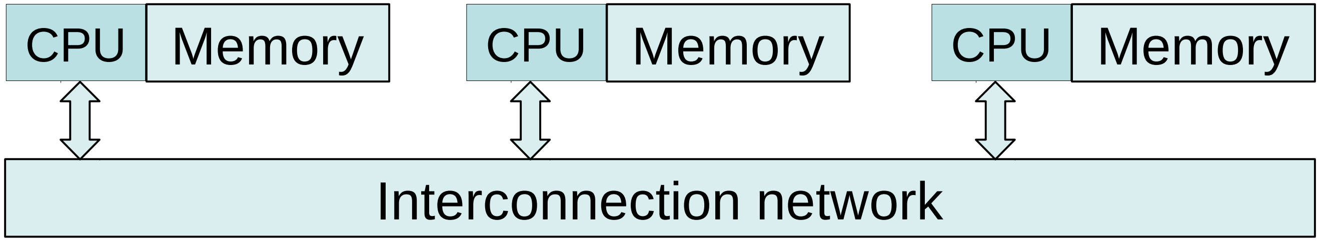


NUMA, RMA (Remote Memory Access)

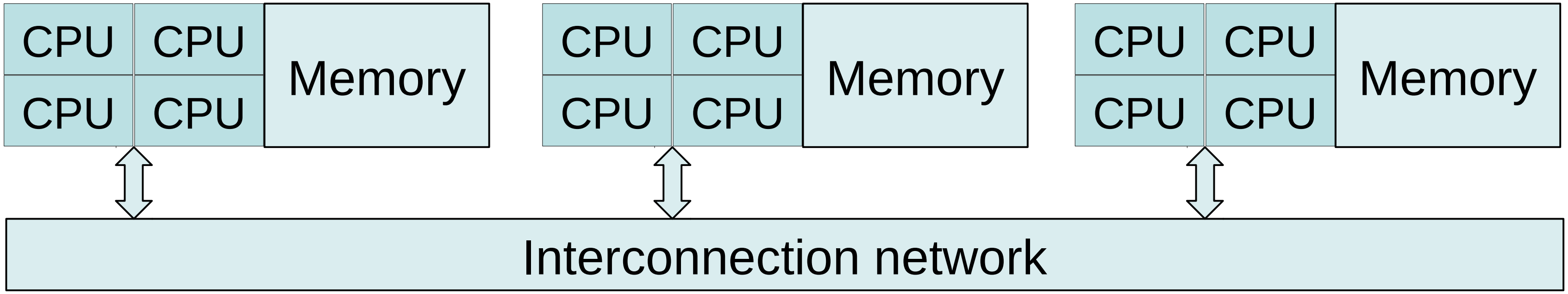
DSM (Distributed Shared Memory) – **DGAS** (D. Global Address Space)

Parallel computers' memory architectures

- **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)



- **Hybrid** (distributed + shared)



Programming models

Abstraction of hardware and memory architecture;

Not necessarily tied to a particular architecture.

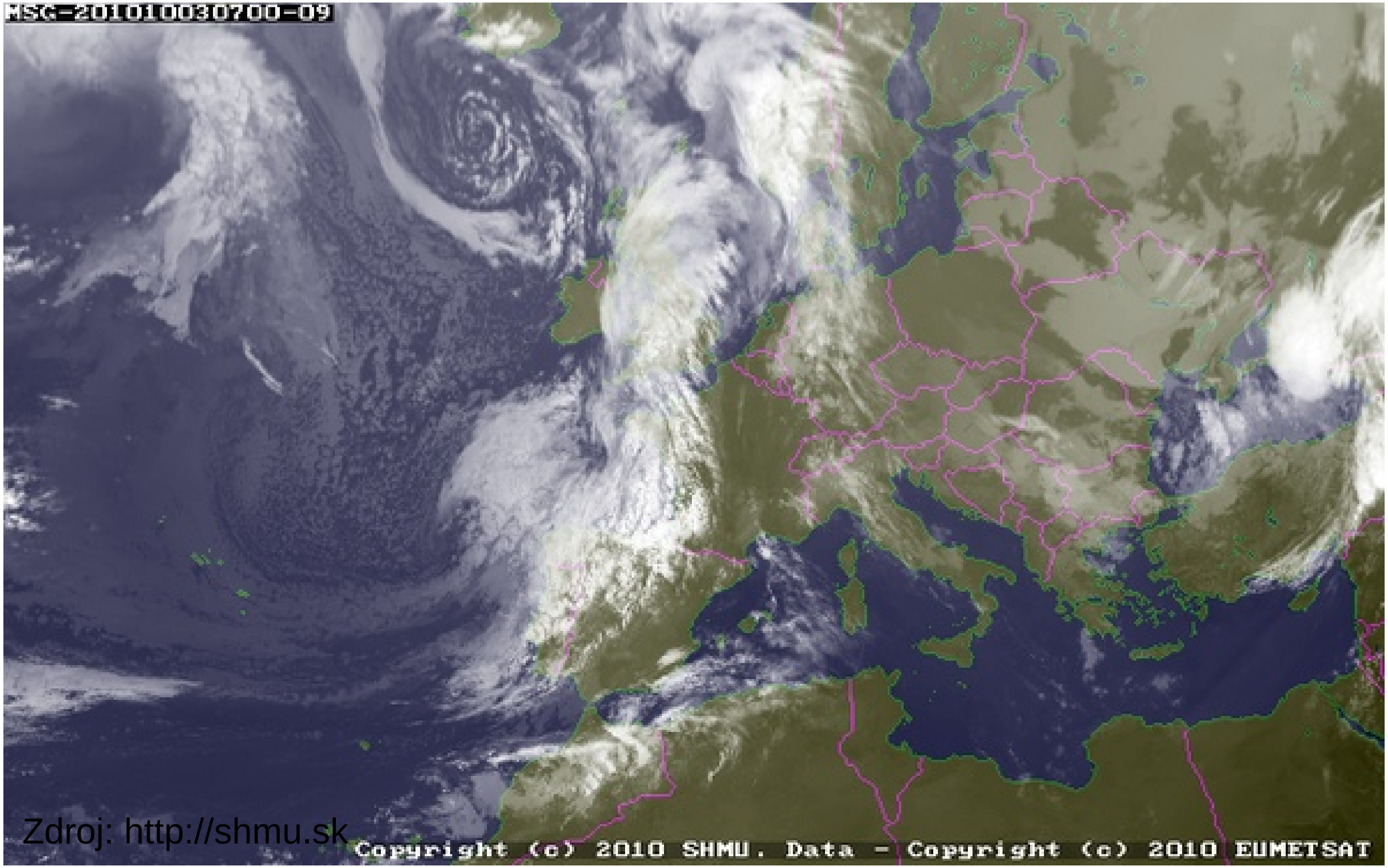
- **Shared memory** – Tasks share global address space, asynchronous read and write; locks, semaphores, ..; explicit communication is not needed when exchanging data; Where are stored the data that the processor works with?
- **Threads** - POSIX Threads (Pthreads) - very explicit parallelism – the program must be designed to run tasks "in parallel"; OpenMP – parallelization expressed in directives, more automatic with help of compiler.

Programming models

- **Messages passing** – dtata and events exchange by sending and receiving of messages; typical for DMS but usable/used on SMS as well;
  What is maximal communication latency to not 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).

Parallel program development – scheduling is fundamental

Scheduling

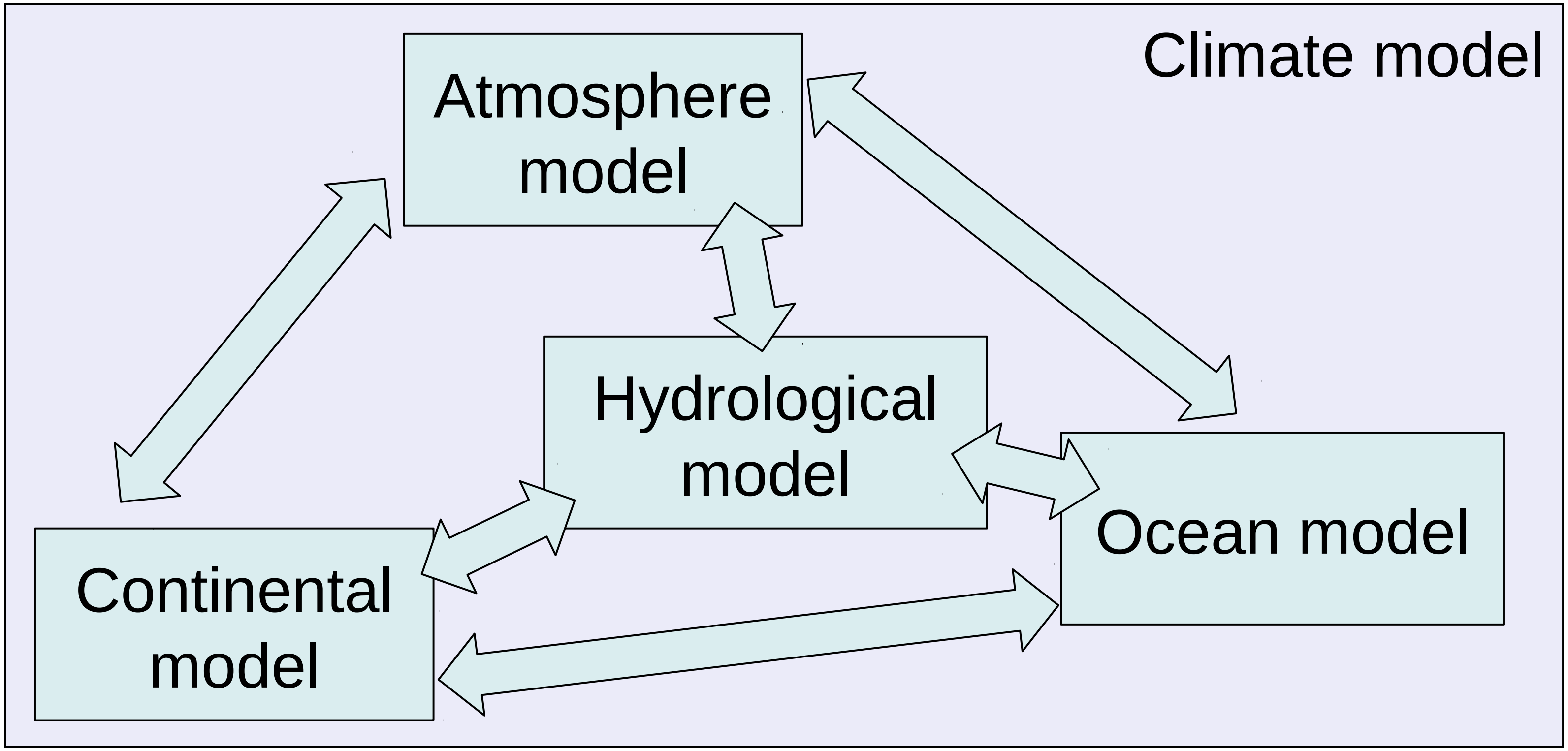


Zdroj: http://shmu.sk

Parallel program development – scheduling is fundamental

Scheduling

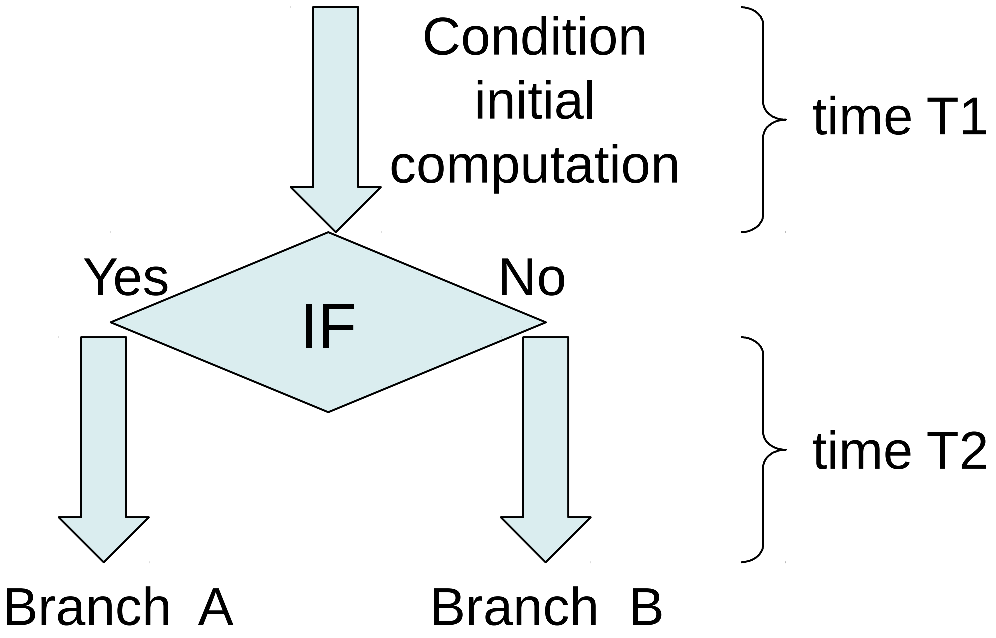
- Top down view (functional decomposition): The aim is to divide the program into a set of tasks which can be executed in parallel with respect to mutual communication; can be applied recurrently



Parallel program development – scheduling is fundamental

Scheduling

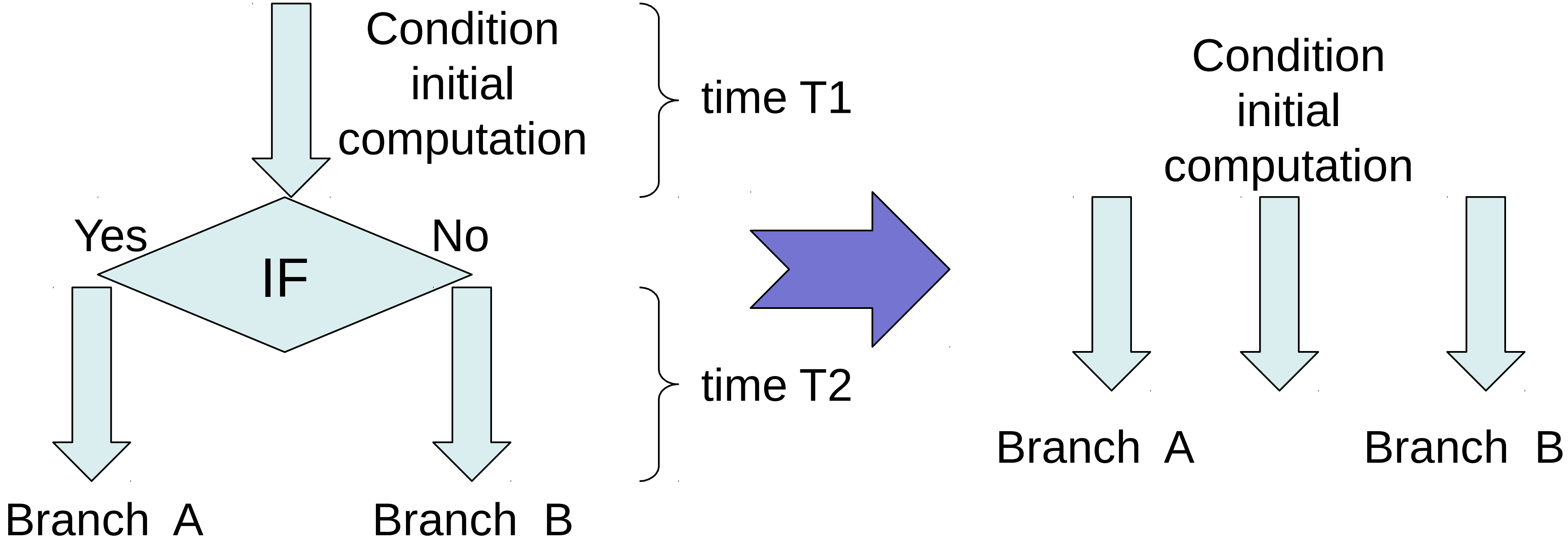
- Speculative decomposition



Parallel program development – scheduling is fundamental

Scheduling

- Speculative decomposition



Parallel program development – scheduling is fundamental

Scheduling

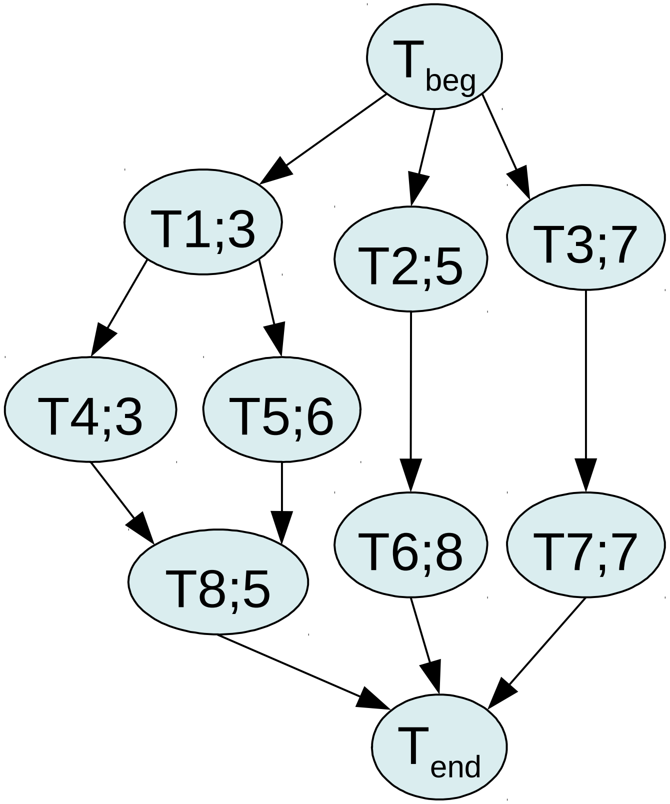
- Bottom-up: The goal is to group sequentially executed instructions, commands, program fragments without linking to another (one line of instruction flow) - the grain packing at the lowest level, possibly continue according to a specific strategy in the grain packing with respect to communication (see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

Example No 2 – scheduling / mapping / load balance

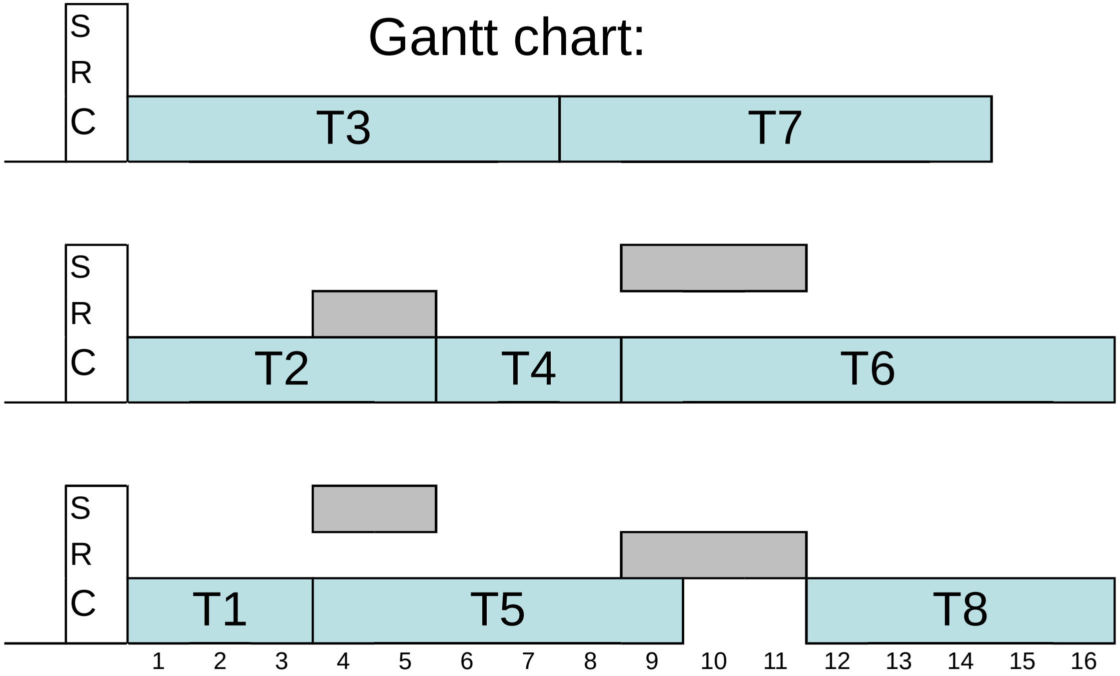
Lets are three equivalent processors available.

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

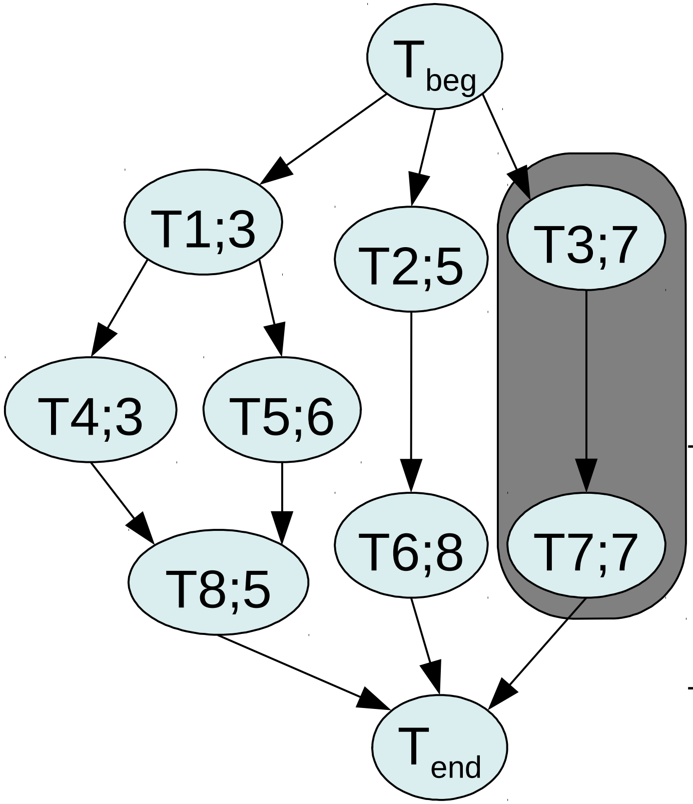
S = 44 / 16 = 2,75

Example No 2 – scheduling / mapping / load balance

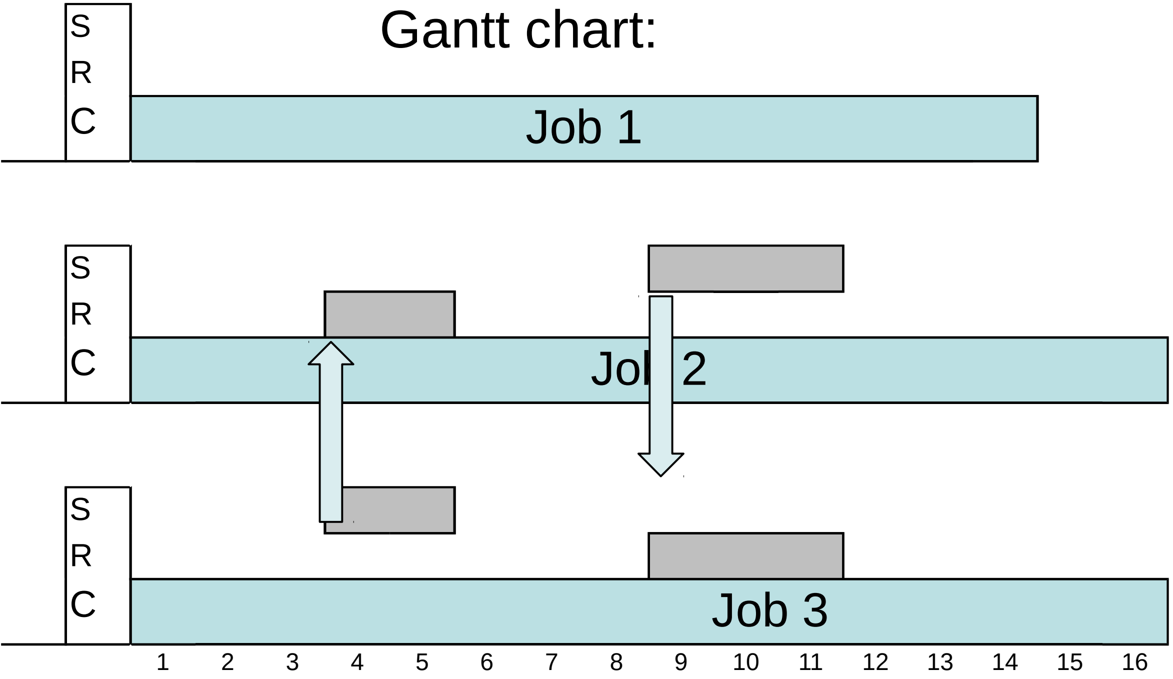
Lets are three equivalent processors available.

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:



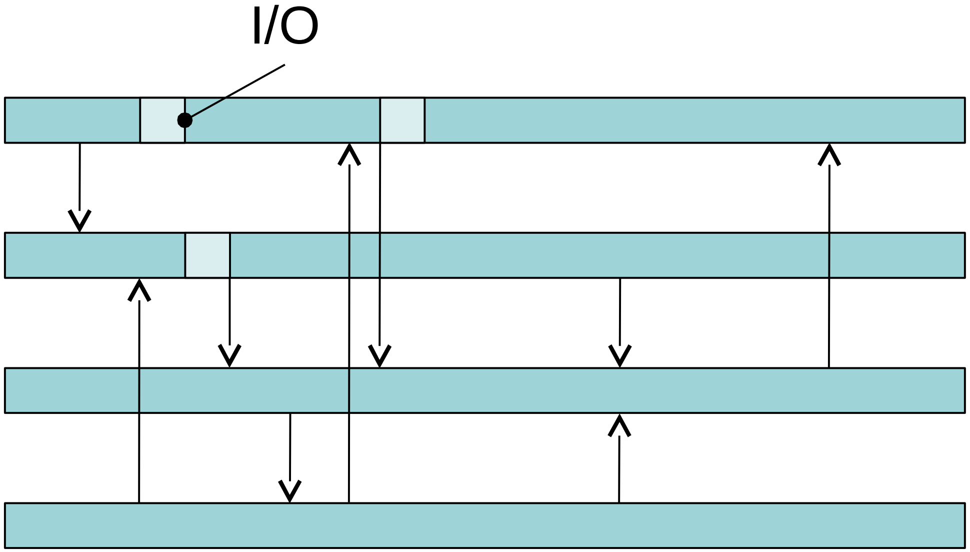
Resources/processors utilization

S = 44 / 16 = 2,75

Scheduling

- 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



Design of parallel program – partitioning

Partitioning – Domain decomposition

Sharping

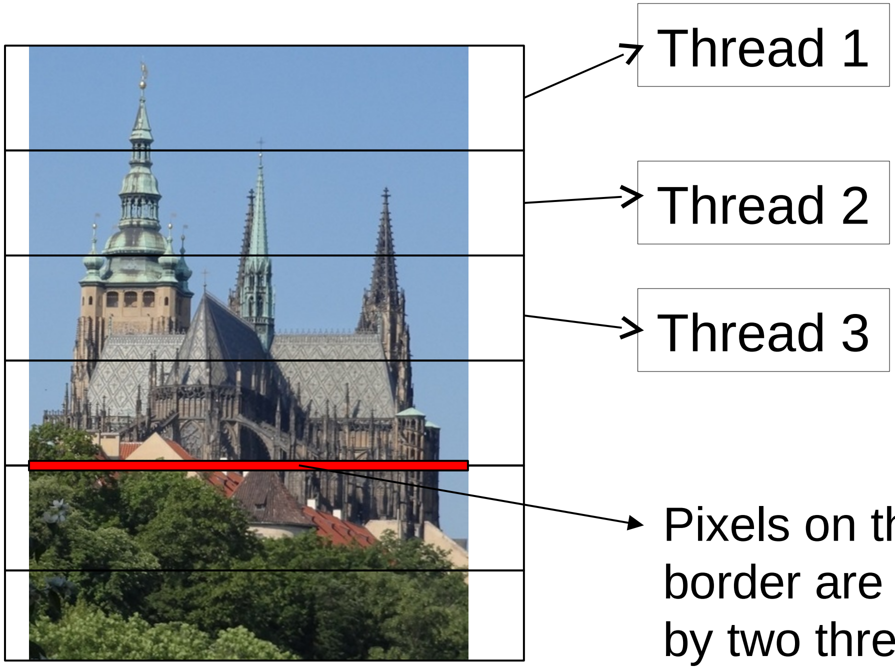
How to do that?

Design of parallel program – partitioning

Partitioning – Domain decomposition

How to sharp image? ✉ Convolution

How to utilize more CPUs in parallel program?


Thread 1

Thread 2

Thread 3

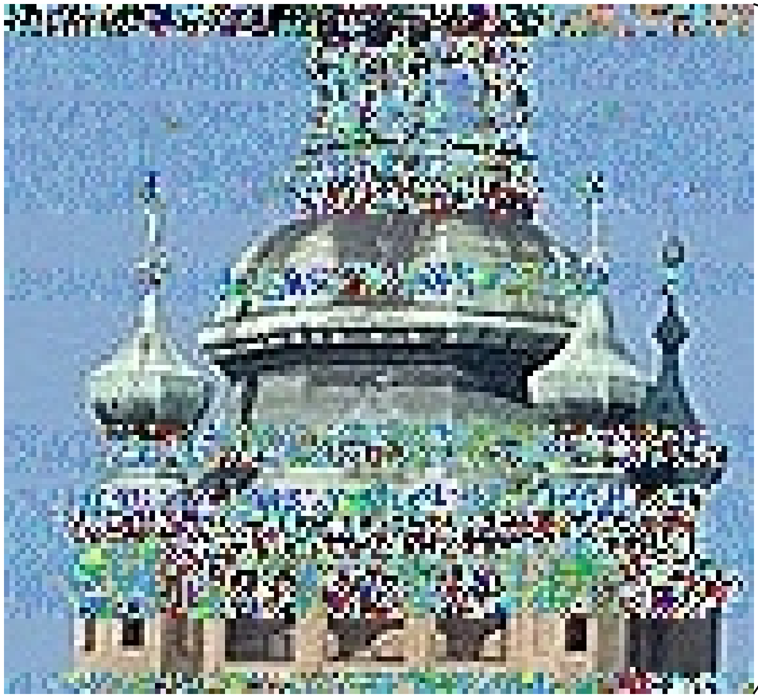
Pixels on the strips border are accessed by two threads

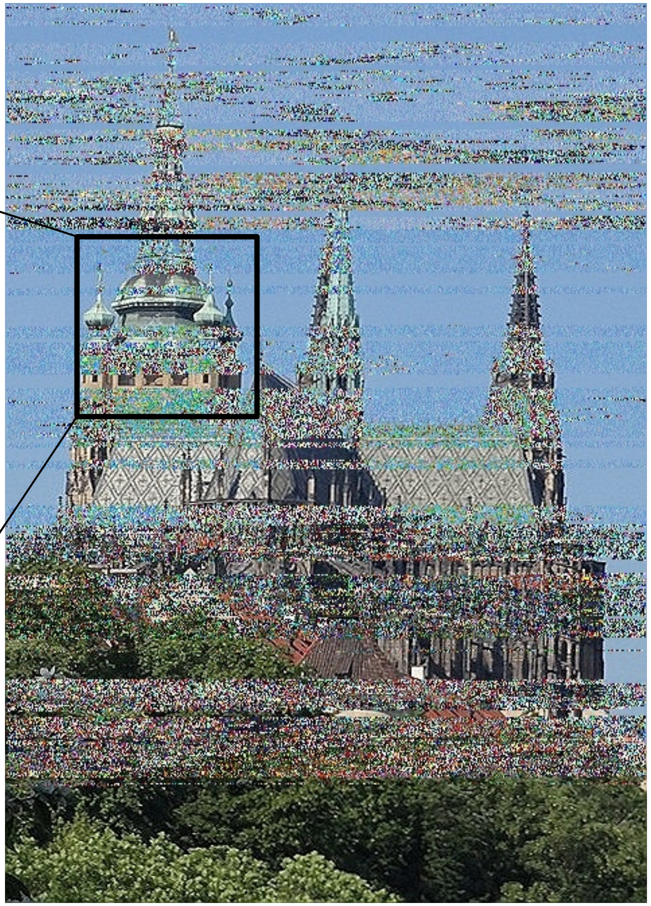
What about memory access conflicts?

Design of parallel program – partitioning

Partitioning – Domain decomposition

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



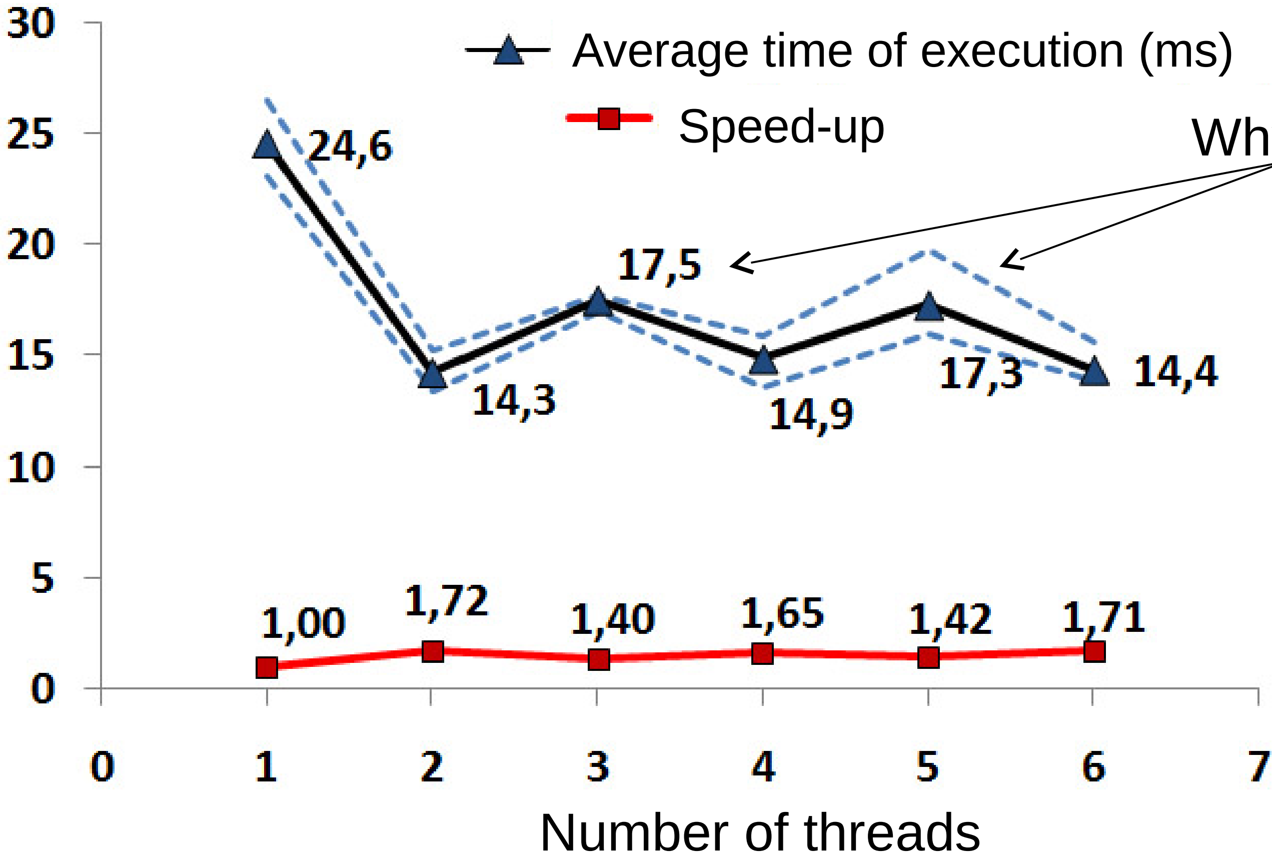


What is wrong?

Design of parallel program – partitioning

Partitioning – Domain decomposition

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



Why increase?

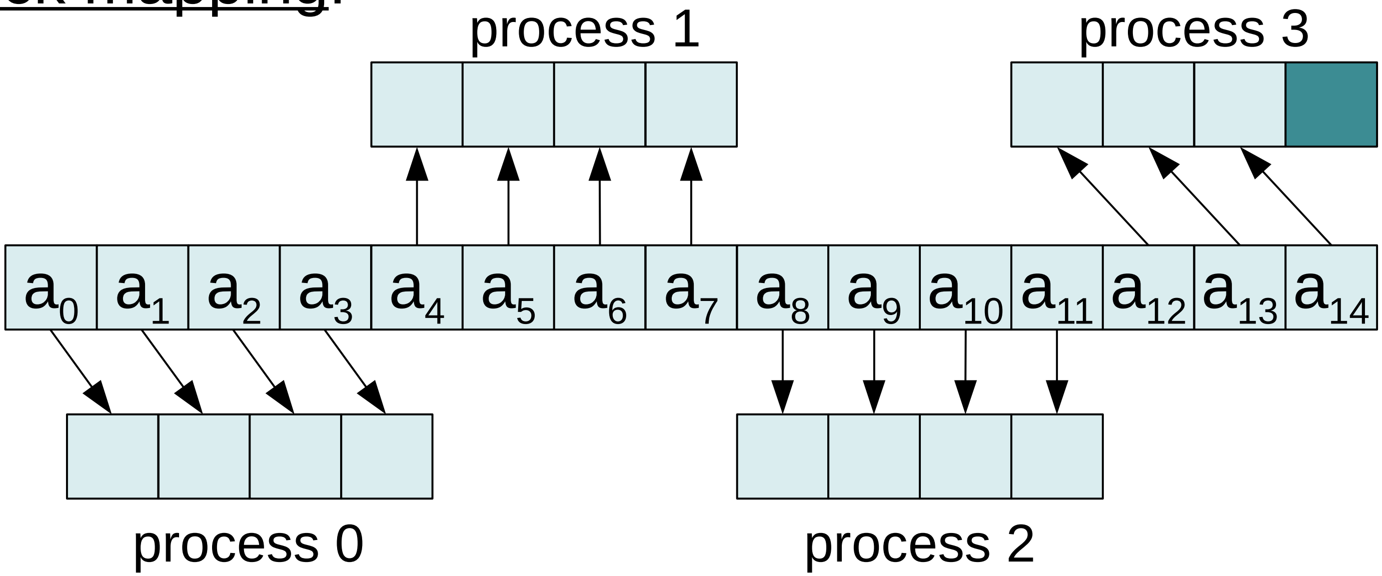
How would look extrapolate of graph?

Design of parallel program – partitioning

Partitioning – Domain decomposition

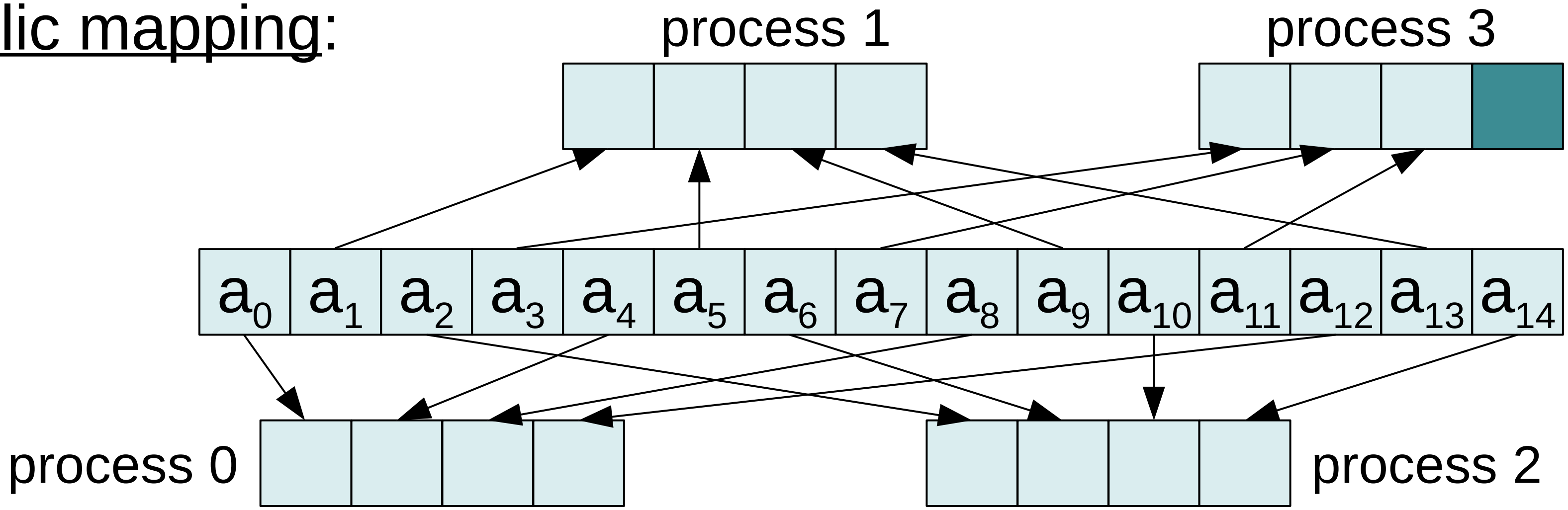
- Data set is distributed to individual processes
- $A = (a_0, a_1, \ldots, a_{n-1})$ n elements
  $P = (q_0, q_1, \ldots, q_{p-1})$ p processes

<u>Block mapping</u>:



Design of parallel program – partitioning

- Cyclic mapping:



process 1

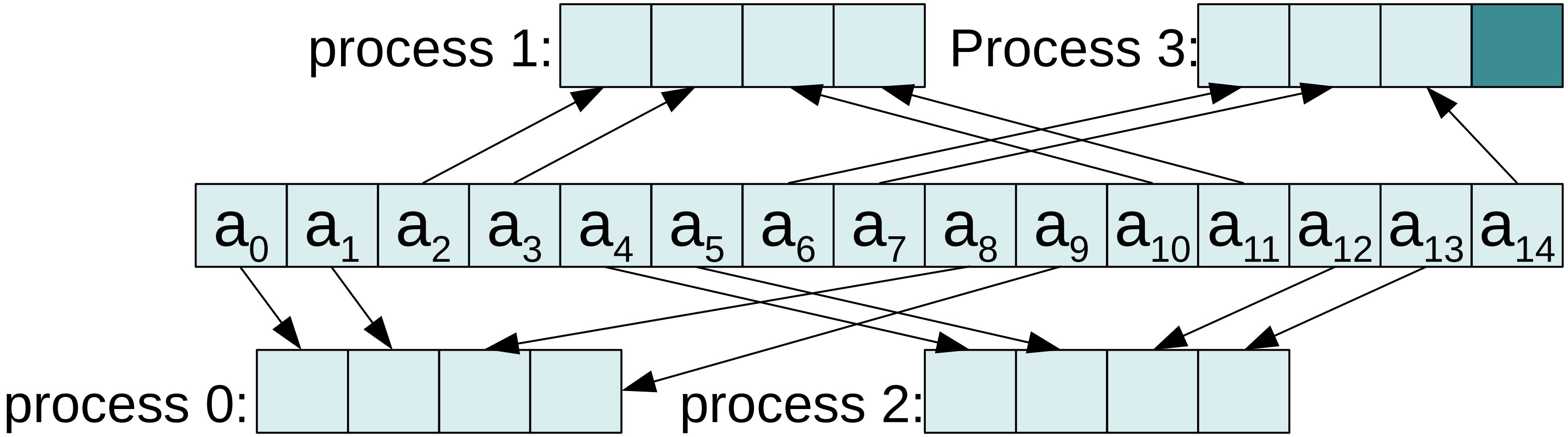
process 3

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14}

process 0

process 2

- Block-cyclic:



process 1:

Process 3:

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14}

process 0:

process 2:

Design of parallel program – partitioning

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

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

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

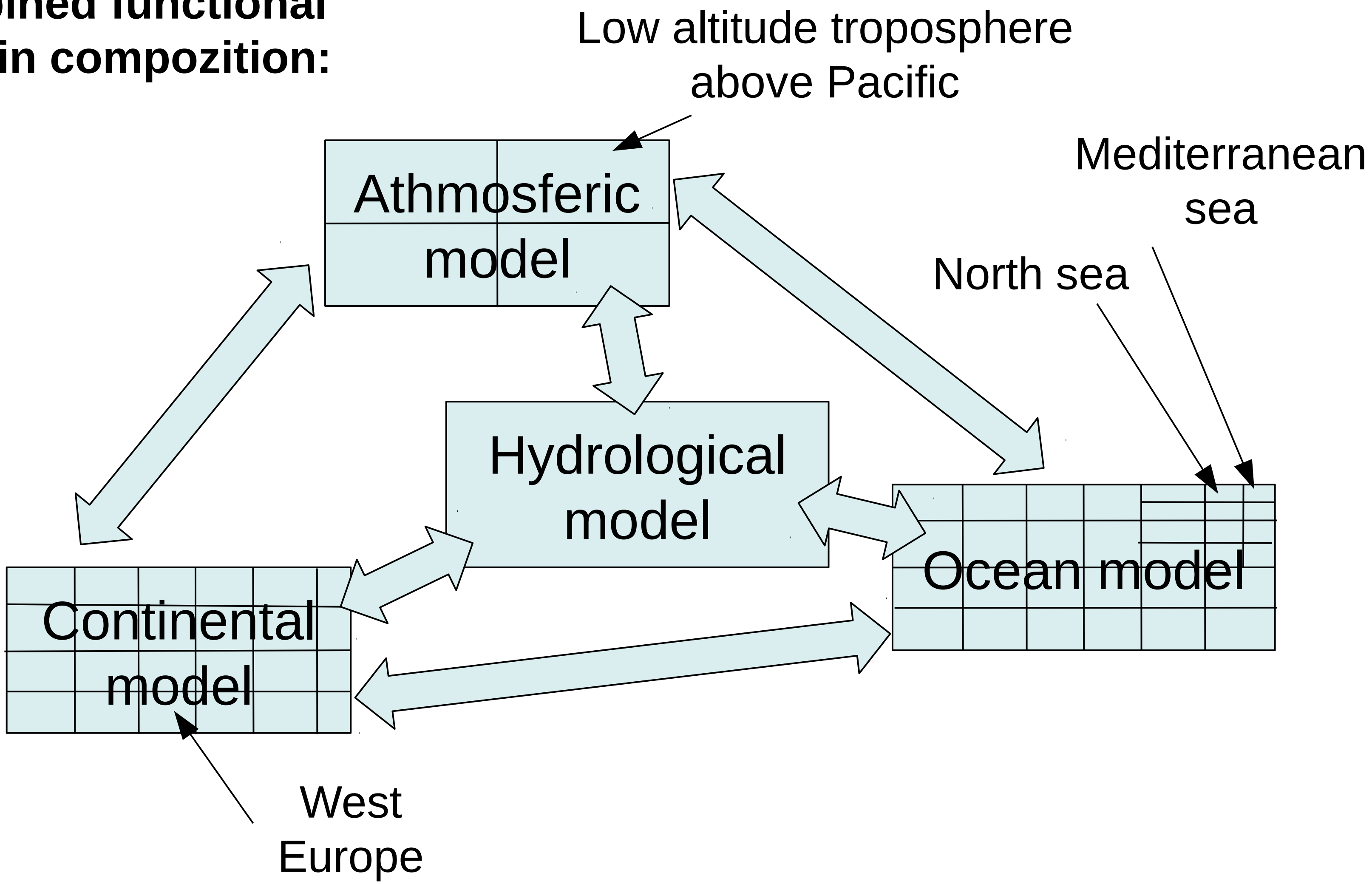
$$\sum_{i=10^{10}}^{1}\frac{1}{i^2}\approx 1.6449340667482264$$

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 achieven convergence for some elements groups/mapping…

Design of parallel program – partitioning

Combined functional domain compozition:

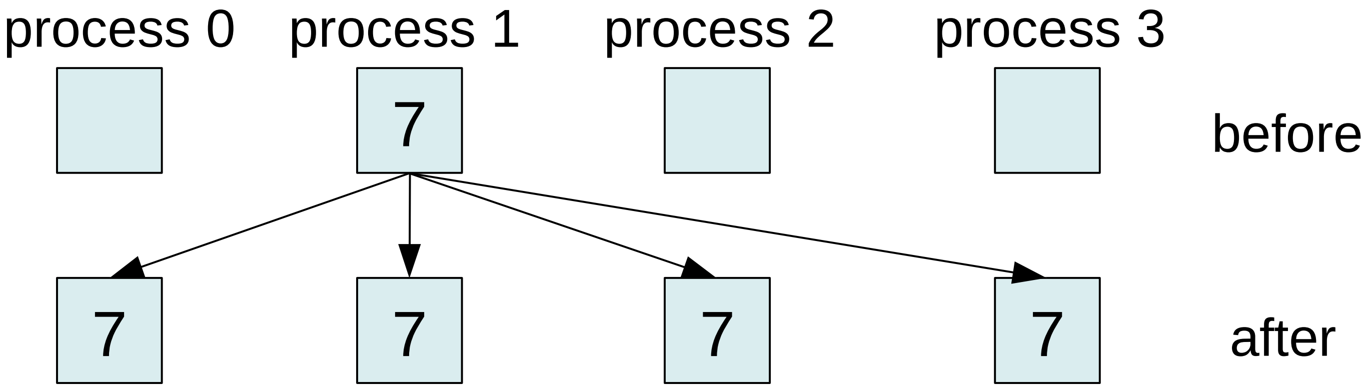


Design of parallel program – communication

- Direct communication between processes (threads) can be hidded to programmer (depends on model: shared memory, data-parallel model, threads, message passing…).
- Communication price.
- **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 – contrary 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.: allways blocking.

Design of parallel program – communication

Broadcast (source = 1):



process 0 process 1 process 2 process 3

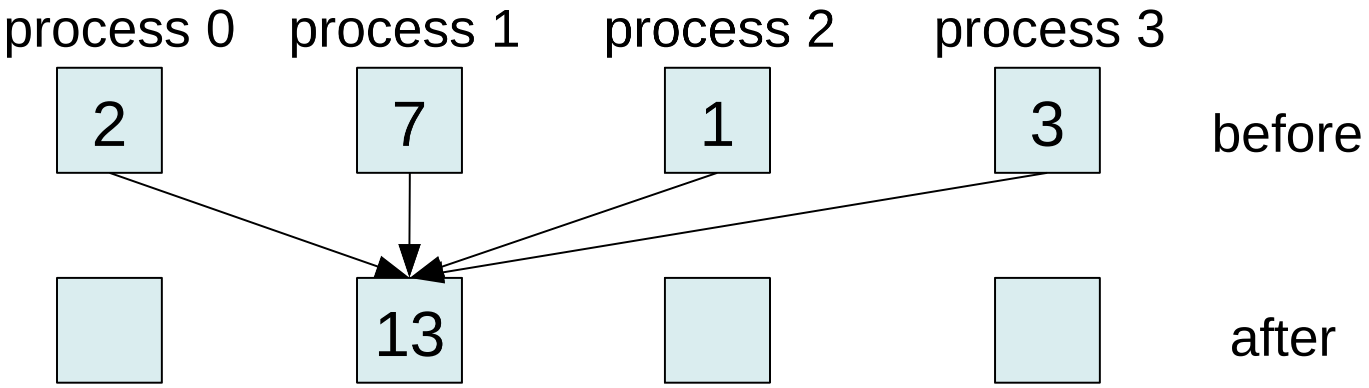
7

before

7 7 7 7

after

Reduce (destination = 1, operation +):



process 0 process 1 process 2 process 3

2 7 1 3

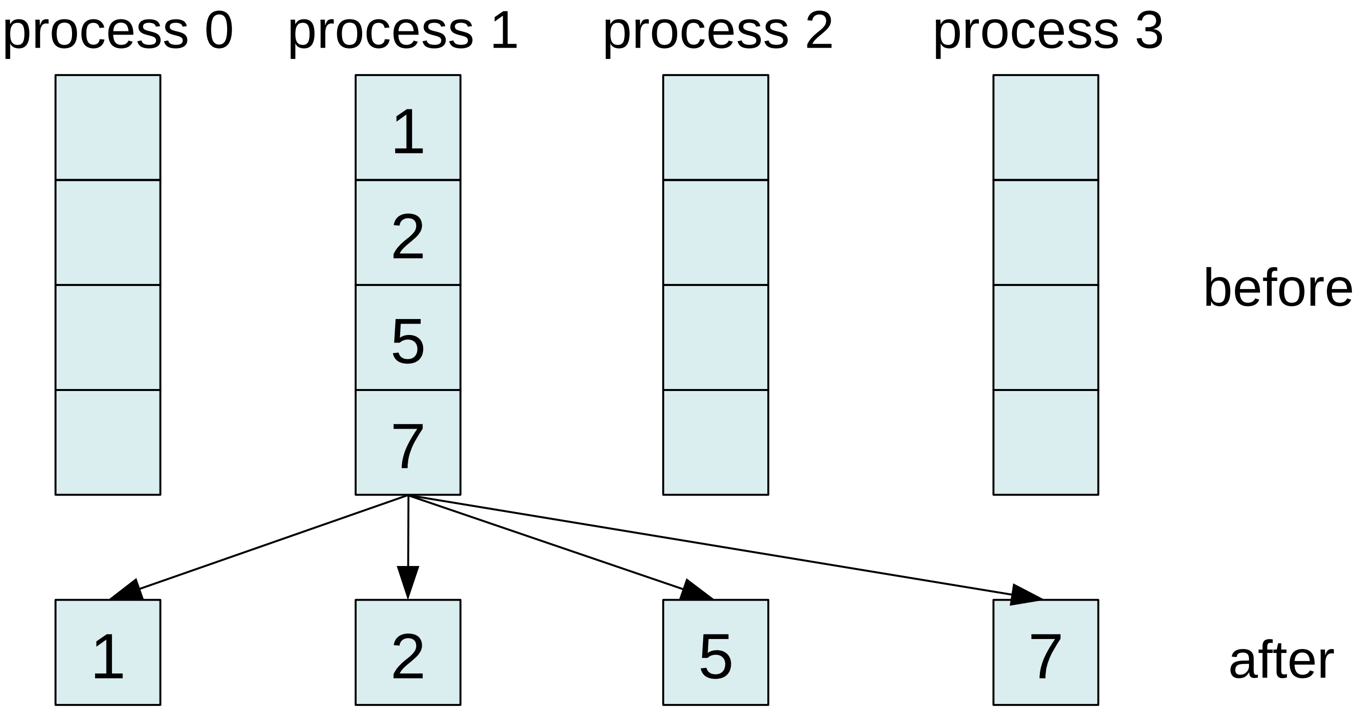
before

13

after

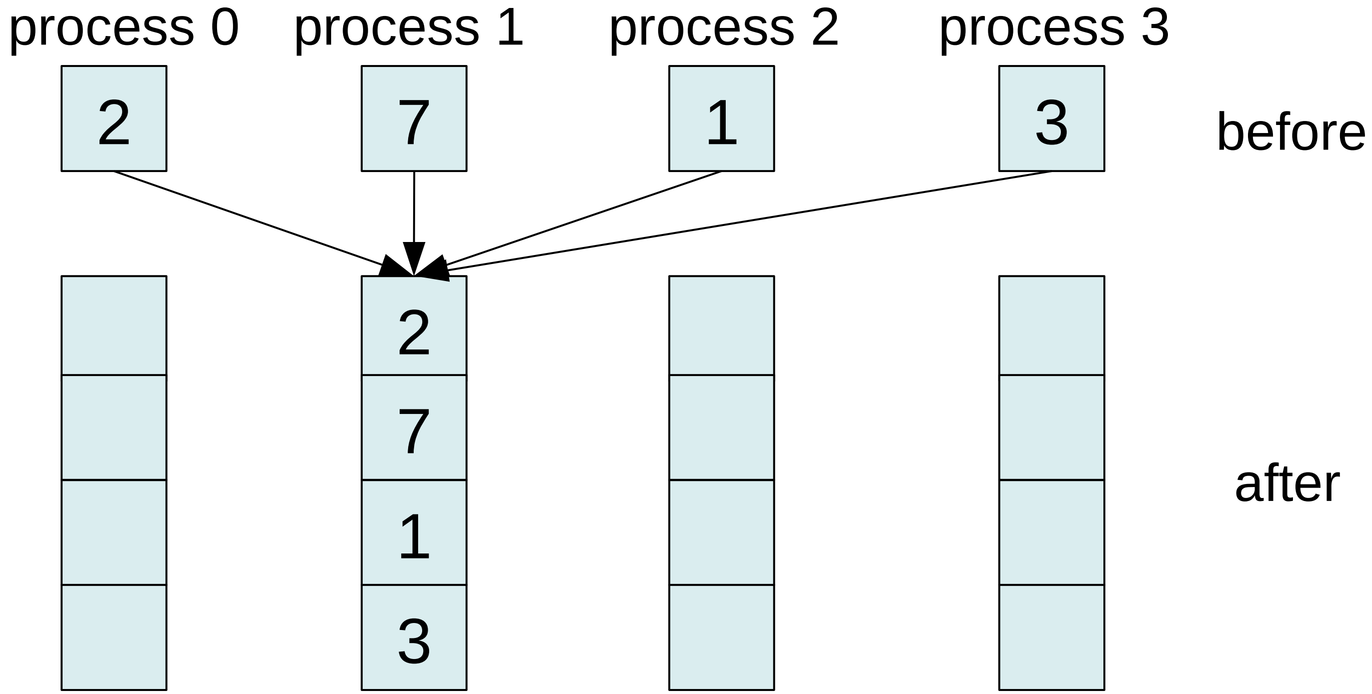
Design of parallel program – communication

Scatter (source = 1):



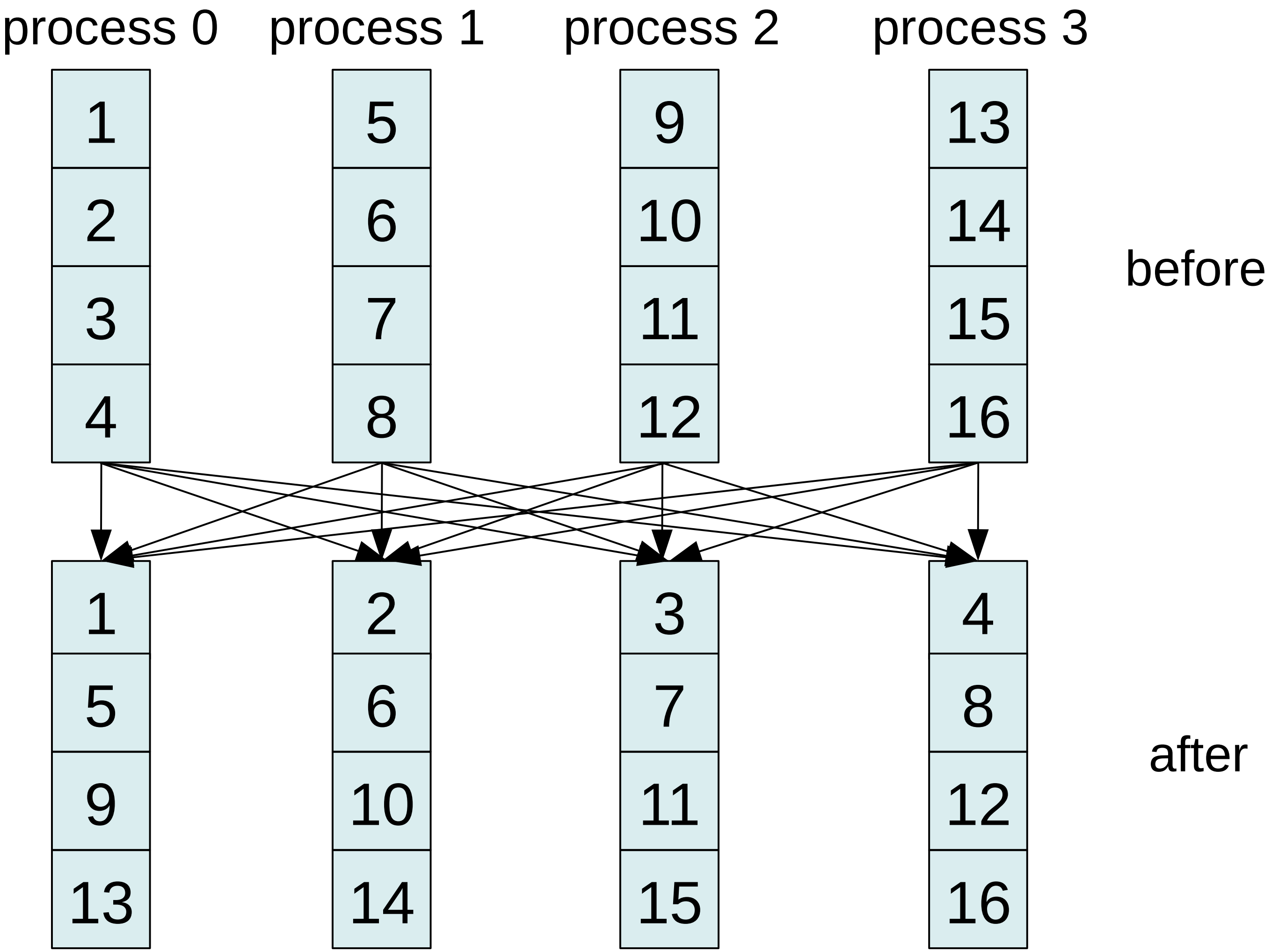
Design of parallel program – communication

Gather (destination = 1)



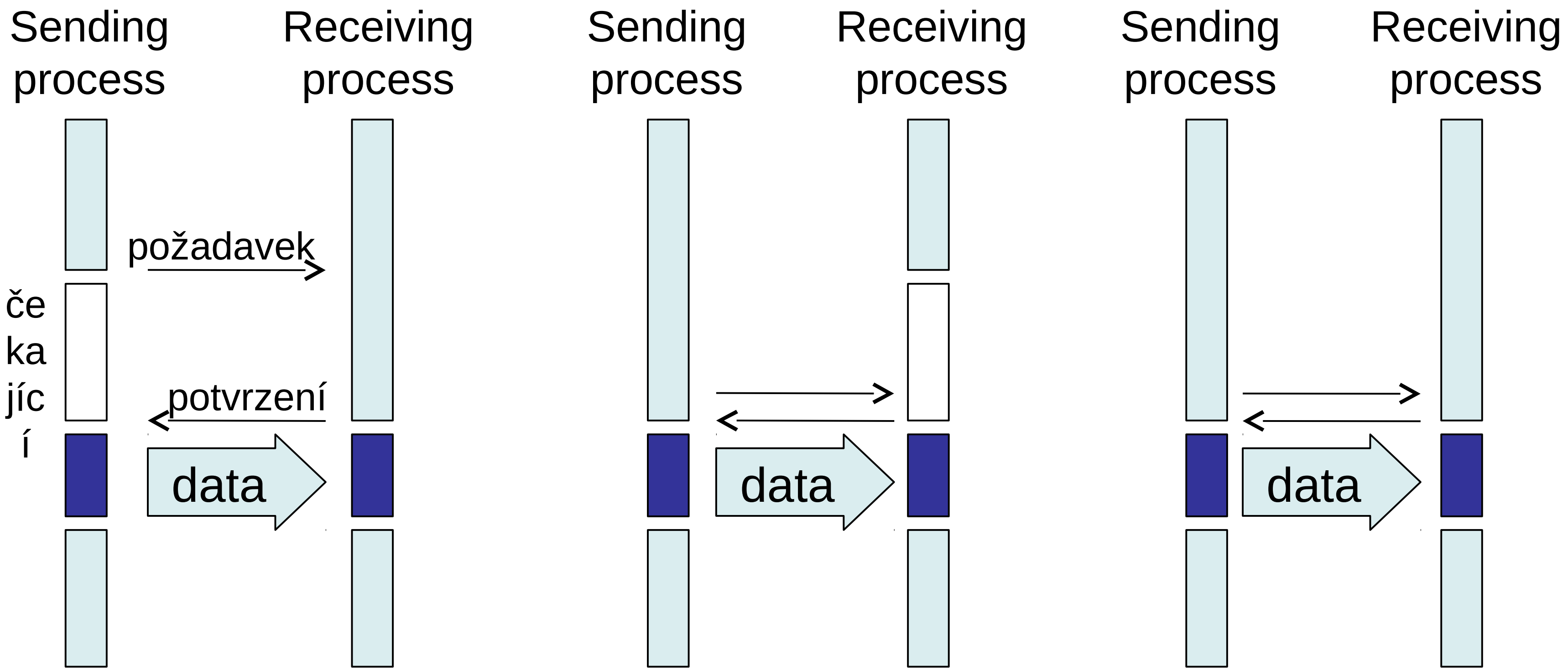
Design of parallel program – communication

All to All:



Design of parallel program – communication

- Blocking point-to-point communication unbuffered



Sending process is waiting until receiving process is ready

Receiving process waits for data – waste of its time

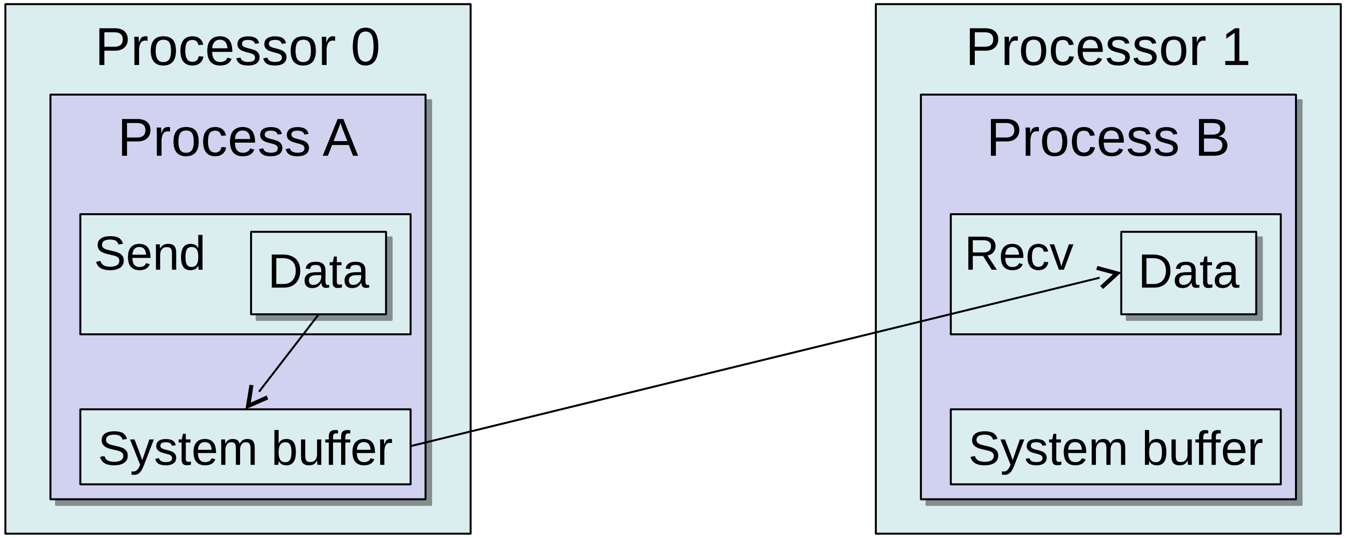
„Optimal“ situation; bud communication period usually lefts processors “unused”

Design of parallel program – communication

- **Dead-lock** caused by point-to-point blocking unbuffered communication.

P0:	P1:
send()	send()
recieve()	recieve()

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



Design of parallel program – communication

Blocking communication

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

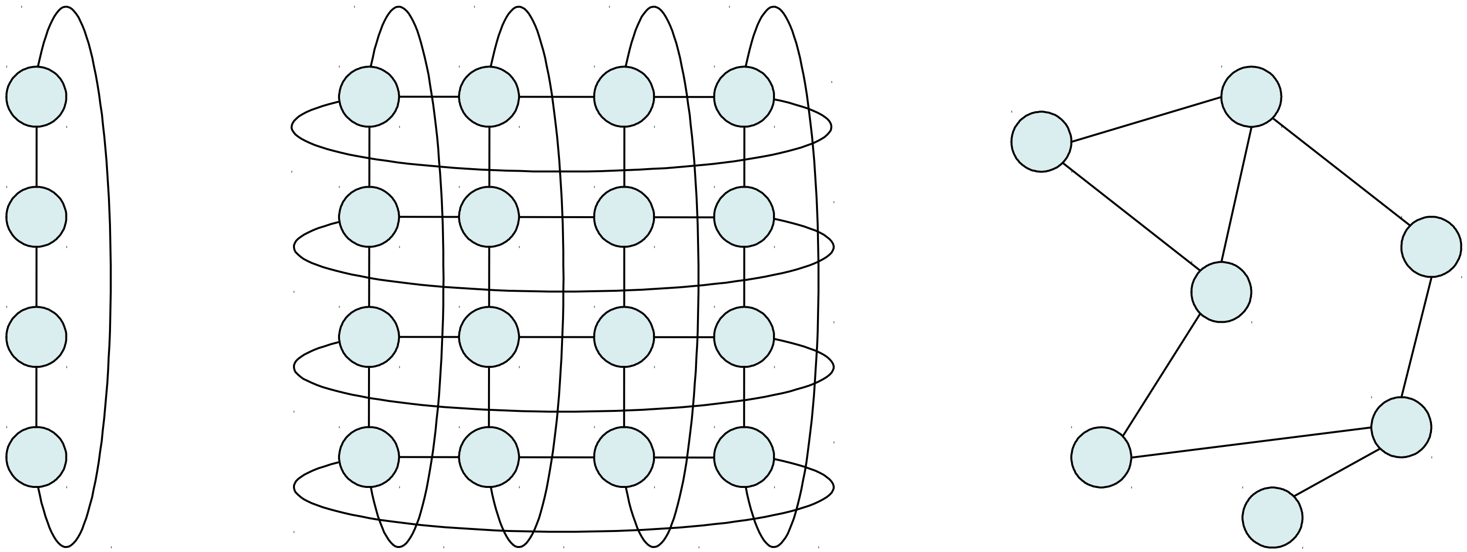
Non-blocking communication

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

Design of parallel program – Virtual topology

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

- Advantageous for specific communication requirements.
- Define neighbourhoodness of processes (nodes) – neighbourhood processes can communicate directly.
- Implementation can use to optimize decision of mapping processes to physical nodes…



Design of parallel program – Synchronization

- Barrier
  - each task stops on barrier, execution continues when all tasks reach 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 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 a above defined three-way superscalar processor. Partition program into two balanced halves. Find the optimal schedule of split parallel program by two processors to achieve 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