Chapter 10

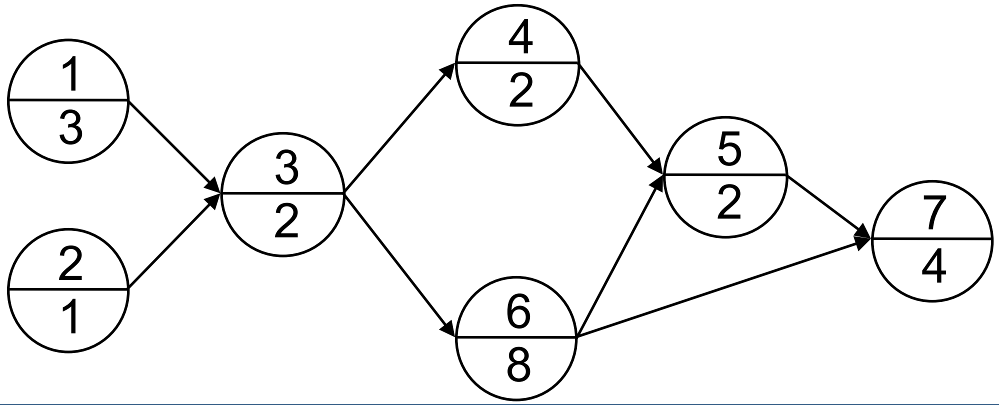
Scheduling of dependent threads

10.1 Introduction

- In the previous chapters we assumed that the threads to be executed were either independent on each other or described by a TIG.
- Now we consider the dependencies between the threads in that way that an order of the thread execution must be observed.
- On the set of threads a partial order is defined that specifies which thread has to be executed before which other thread.
- The threads are then modeled as a Task Precedence Graph (TPG), i.e. as directed acyclic graph (DAG). The thread weights indicate the execution times of the threads.

Example of a DAG

The upper numbers are the thread numbers, the lower ones are the execution times.



Scheduling of dependent threads

- Threads as parts of a parallel program that is to be executed on a cluster computer.

Goal

- For a given number of processors minimize the schedule length (makespan) under the restriction of the order dependencies.

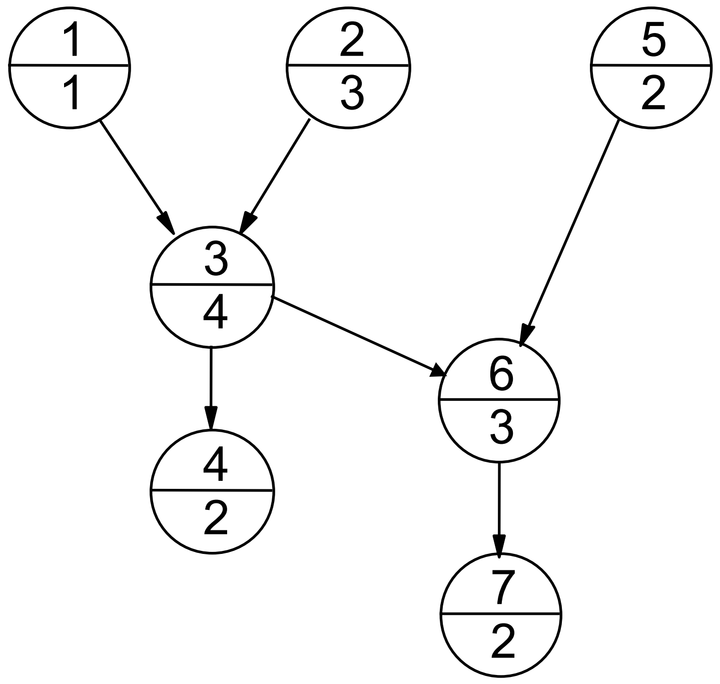
Variants:

  - specific forms of DAGs (e.g. tree, forest,...)
  - Additional consideration of communication times (delays along the edges)
  - Additional consideration of deadlines
  - Additional consideration of the communication network‘s topology

Example

How many processors should we use?

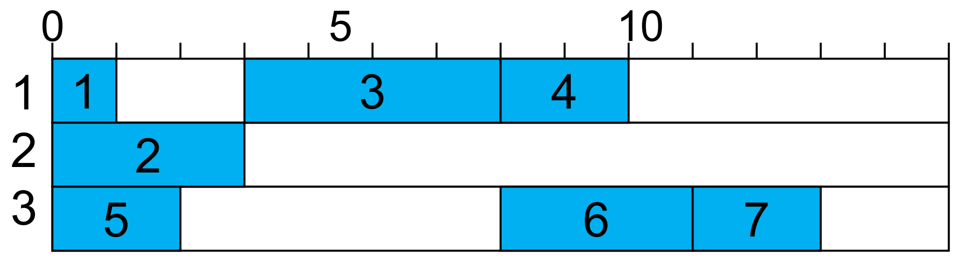
Precedence graph



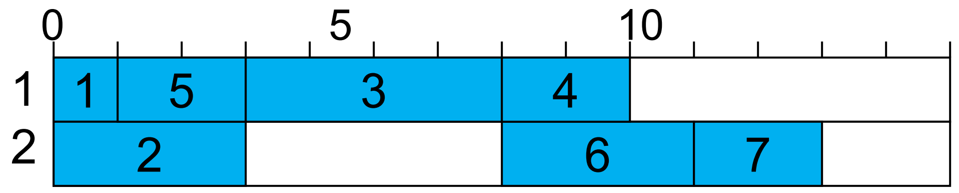
Example

Gantt-Chart

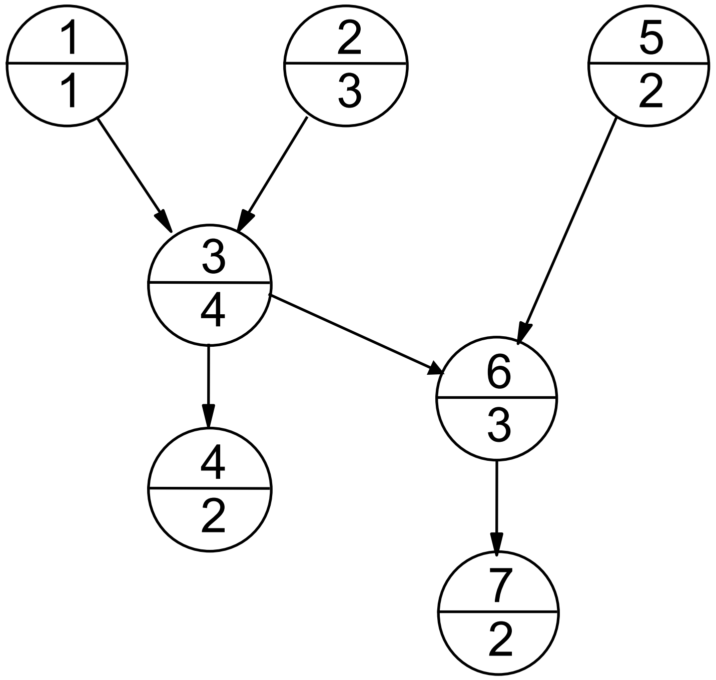
for 3 processors



for 2 processors



Precedence graph



Model

For the sake of simplicity we assume that

- the processors are homogeneous (same speed) and
- the latencies between each two processors are the same.

Realistic?

Model

For the sake of simplicity we assume that

- the processors are homogeneous (same speed) and
- the latencies between each two processors are the same.

That means the execution and message latencies depend only on the threads, not on the processors.

In addition, intermediate results are to be transferred at synchronization points.

That leads to the following model:

P	Set of processors
T	Set of threads (vertices)
$n = \lvert T \rvert$	number of threads
$E^T \subseteq T \times T$	precedence relation (edge set)
α_{ij}	amount of information exchanged (edge weight)
β_i	execution time of thread *i* (vertex weight)

Other quantities

ms	schedule length (makespan)
f_i	finishing time of thread T_i
s_i	starting time of thread T_i

where the following holds: $ms := \max \{f_i \mid T_i \in T\}$

$\pi: T \rightarrow P$ mapping of the threads to the processors

Considering communication cost

Let $\pi(T_i)$ be the processor to which T_i is assigned.

Model A

Total cost = $ms + cc$

where $cc = \left|\{ (T_i, T_j) \in E^T \mid \pi(T_i) \neq \pi(T_j)\}\right| \cdot \alpha$

The message latencies α_{ij} are assumed to be constant: $\alpha_{ij} = \alpha$.

Model B

Total cost = $ms + cc$

where $cc = \left|\{ (P_k, T_i) \mid \exists k : P_k = \pi(T_j) \wedge (T_i, T_j) \in E^T\}\right| \cdot \alpha$

If two successors of T_i are executed on the same, but different from $\pi(T_i)$ processor, then the result of T_i needs to be shipped only once to the processor on which the successors reside. This is considered in model B (in contrast to model A).

Considering communication cost

Model C

Model C explicitly integrates the communication overhead in the calculation of the schedule.

Communication costs between threads on the same processor are zero.

Let $(T_i, T_j) \in E^T$, $\pi(T_i) = P_k$. Then the following must hold for T_j:

$s_j >= f_i$, if $\pi(T_j) = P_k$ (T_j starts after T_i finished)

$s_j >= f_i + \alpha_{ij}$, *otherwise* (T_j starts after T_i finished and transmitted the results)

Total cost = *ms*

NP-completeness

The NP-completeness of the scheduling problem is given for the following problem instances:

Graph	# Processors	Execution time	Communica-tion time	Communica-tion model
arbitrary	*m*	1	0	-
forest	*m*	1	0	-
tree	*m*	1	1	Model A
tree	*m*	1	1	Model B
arbitrary	2	1	1	Model B
arbitrary	unlimited	1	>1	Model C

Optimal Algorithms

For the following problem instances optimal efficient algorithms are known (unit execution time, communication model C):

Graph	# Pro-cessors	comm.-time	Authors	Complexity
arbitrary	2	0	Coffman&Graham, 1972	$O(n^2)$
tree	m	0	Hu, 1961	$O(n)$
interval order	m	0	Papadimitriou & Yanakakis, 1979	$O(n+e)$
arbitrary	m	c	Jung et al.	$O(n^{c+1})$
interval order	m	1	Ali & El-Rewini, 1993	$O(e + n\ p)$
tree	2	1	El-Rewini & Ali, 1994	$O(n^2)$

10.2 List Scheduling

- For general scheduling problems of dependent threads, usually the **List Scheduling** is used.
- It is a *heuristic* off-line-algorithm that not necessarily produces optimal schedules (such with minimal length).
- Only for trees it is optimal.

Algorithm schema:

Given:	Precedence graph as DAG, vertices weighted with priorities
Goal:	makespan of minimal length for p processors
Initialization:	insert all source vertices (vertices without predecessor) into the list.

Loop:

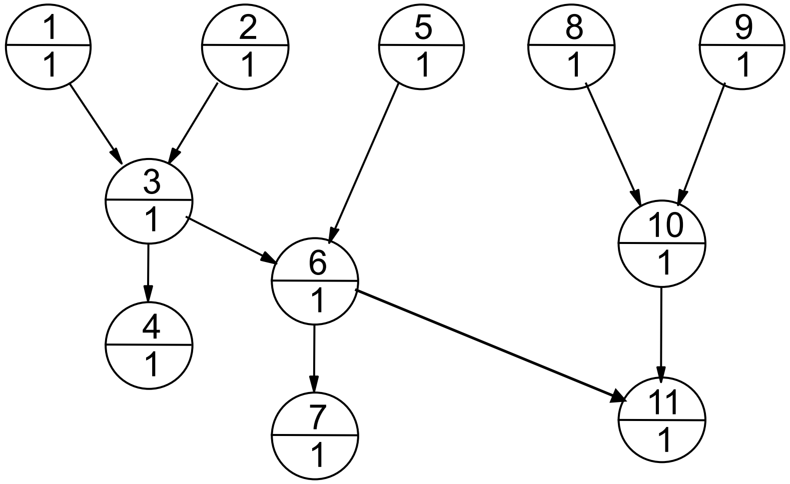
```
while list not empty do
    (i)   take vertex with highest priority from the list.
    (ii)  select an idle processor to execute this vertex.
    (iii) check all vertices that are not yet assigned to a processor and
          not yet in the list, whether all direct predecessors are executed.
          If yes, insert that vertex into the list.
```

Example

We use the graph of slide 4 and give all threads the same priority.

The list can then organized as a FIFO queue.

Which plans does the list algorithm produce?



time	P1	P2	P3
1			
2			
3			
4			

For three processors

time	P1	P2	P3	P4
1				
2				
3				
4				

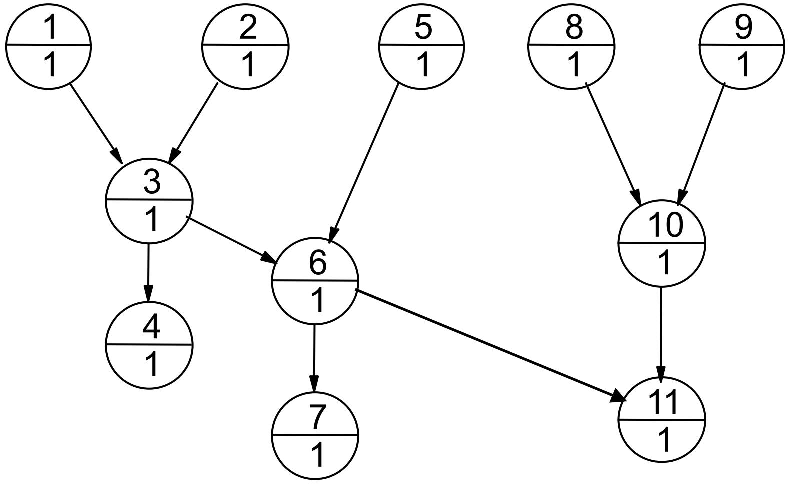
For four processors

Example

We use the graph of slide 4 and give all threads the same priority.

The list can then organized as a FIFO queue.

The list algorithm produces the following plans:



time	P1	P2	P3
1	1	2	5
2	8	9	3
3	10	4	6
4	7	11	-

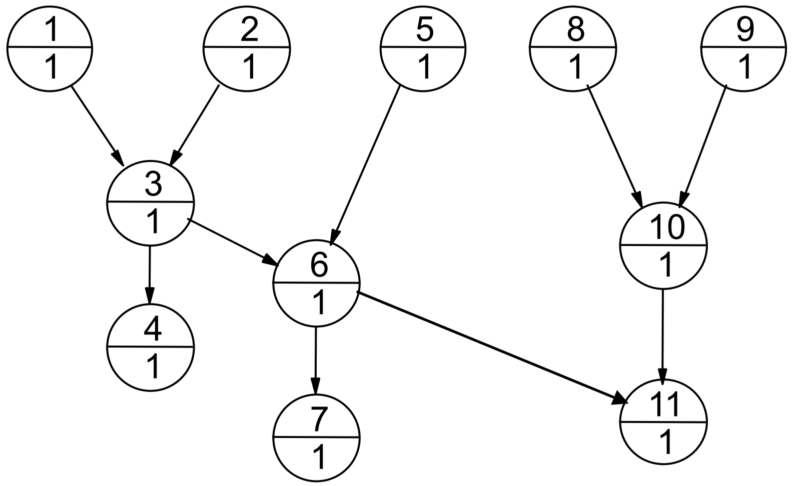
For three processors

time	P1	P2	P3	P4
1	1	2	5	8
2	9	3	-	-
3	10	4	6	-
4	7	11	-	-

For four processors

Example

Is the schedule for 4 processors optimal?



time	P1	P2	P3
1	1	2	5
2	8	9	3
3	10	4	6
4	7	11	-

For three processors

time	P1	P2	P3	P4
1	1	2	5	8
2	9	3	-	-
3	10	4	6	-
4	7	11	-	-

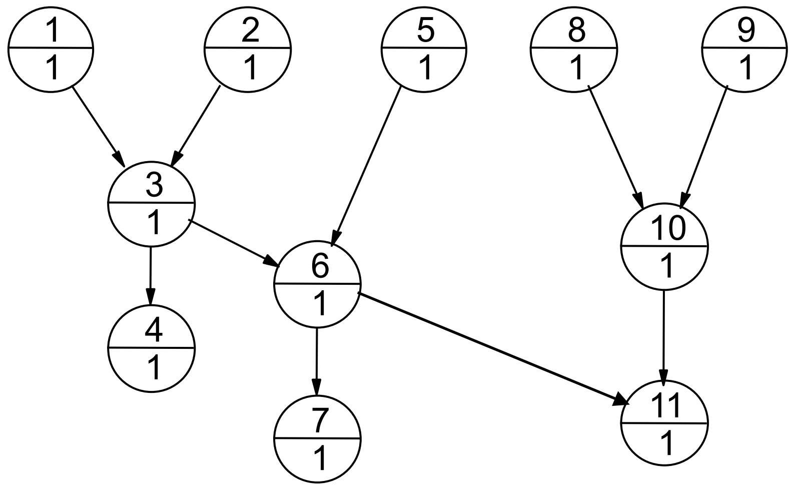
For four processors

Variants of List Scheduling

- If more than one thread is ready to execute, i.e. it is in the list, then the priority decides which to take next. If not externally given, the priority can be set according to different strategies targeting different goals.

- **Def**. The **path length** in a dependency graph is defined as the sum of all vertex weights (execution times) along a path including the first and the last vertex.
- **Def**. The **Level** (static b-Level) of a vertex x is the length of the longest path from x to a sink, i.e. to a vertex without successor (bottom).
- **Def**. The **Co-level** (static t-Level) of a vertex x is the length of the longest path from x to a source, i.e. to a vertex without predecessor (top).
- **Def**. The **critical path** (cp) is the length of the longest path from a source to a sink.

Example

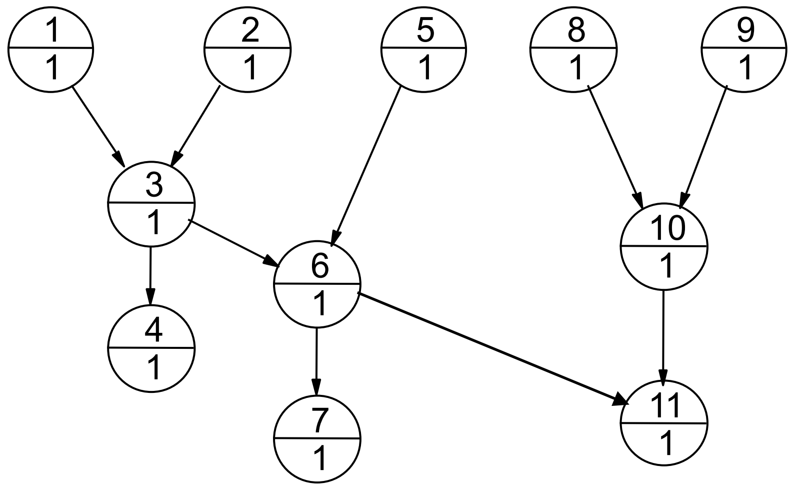


No.	1	2	3	4	5	6	7	8	9	10	11
Level											
Co-level											

Def. The **Level** (static b-Level) of vertex x is the length of the longest path from x to the sink.

Def. The **Co-level** (static t-Level) of vertex x is the length of the longest path from x to a source

Example



No.	1	2	3	4	5	6	7	8	9	10	11
Level	4	4	3	1	3	2	1	3	3	2	1
Co-level	1	1	2	3	1	3	4	1	1	2	4

Def. The **Level** (static b-Level) of vertex x is the length of the longest path from x to the sink.

Def. The **Co-level** (static t-Level) of vertex x is the length of the longest path from x to a source

Variants of List Scheduling

HLF Highest Level First:

- The vertex with the longest chain of successors receives the highest priority. The strategy is sometimes also called CP (critical path), since the chosen vertex lies on the critical path.
- Because the critical path determines the whole schedule length, this strategy is promising.
- HLF is optimal for tree-like graphs with unit execution times and generally good in all practical cases. (with random graphs at most 5% worse than the optimal solution in 90% of cases.)
- HLF takes the height of the tree that depends on vertex x as a priority criterion.
- Other variants choose the breadth or cardinality of the sub-tree as priority instead of the height.
- The aim is to deblock (put in the list) as many threads as possible.

10.3 Algorithms considering communication

- The general approach of List Scheduling can be extended to graphs with edge weights.
- The calculation of levels as priorities is somewhat more complicated since, in contrast to the static levels of HLF, the values now depend on the placement of the predecessors.

  - WHY?

10.3 Algorithms considering communication

- The general approach of List Scheduling can be extended to graphs with edge weights.
- The calculation of levels as priorities is somewhat more complicated since, in contrast to the static levels of HLF, the values now depend on the placement of the predecessors.
- We distinguish (for each vertex x)
  - b-level: length of longest path from vertex x to some leaf vertex or sink, respectively. (b like bottom)
  - t-level: length of longest path from a source to vertex x (t like top)
- The path lengths contain not only the execution times (vertex weights) but also the communication times (edge weights).
- Reference point for the calculation of the levels is the starting point of a vertex.

Calculation of vertex attributes

Reminder: α_{ij} = edge weight, β_i = vertex weight

```
**t_level:**
construct list of nodes in topological order (TopList)
for each node $T_i \in$TopList do
    max $\leftarrow$ 0
    for each parent $T_x$ of $T_i$ do
        if (t_level($T_x$) + $\beta_x$ + $\alpha_{xi}$ > max)
            max $\leftarrow$ t_level($T_x$) + $\beta_x$ + $\alpha_{xi}$
    t_level ($T_i$) $\leftarrow$ max
```

```
**b_level:**
construct list of nodes in reverse topological order (RevTopList)
for each node $T_i \in$ RevTopList do
    max $\leftarrow$ 0
    for each child $T_y$ of $T_i$ do
        if (b_level($T_y$) + $\alpha_{iy}$ > max)
            max $\leftarrow$ b_level($T_y$) + $\alpha_{iy}$
    b_level ($T_i$) $\leftarrow$ $\beta_i$ + max
```

Calculation of vertex attributes

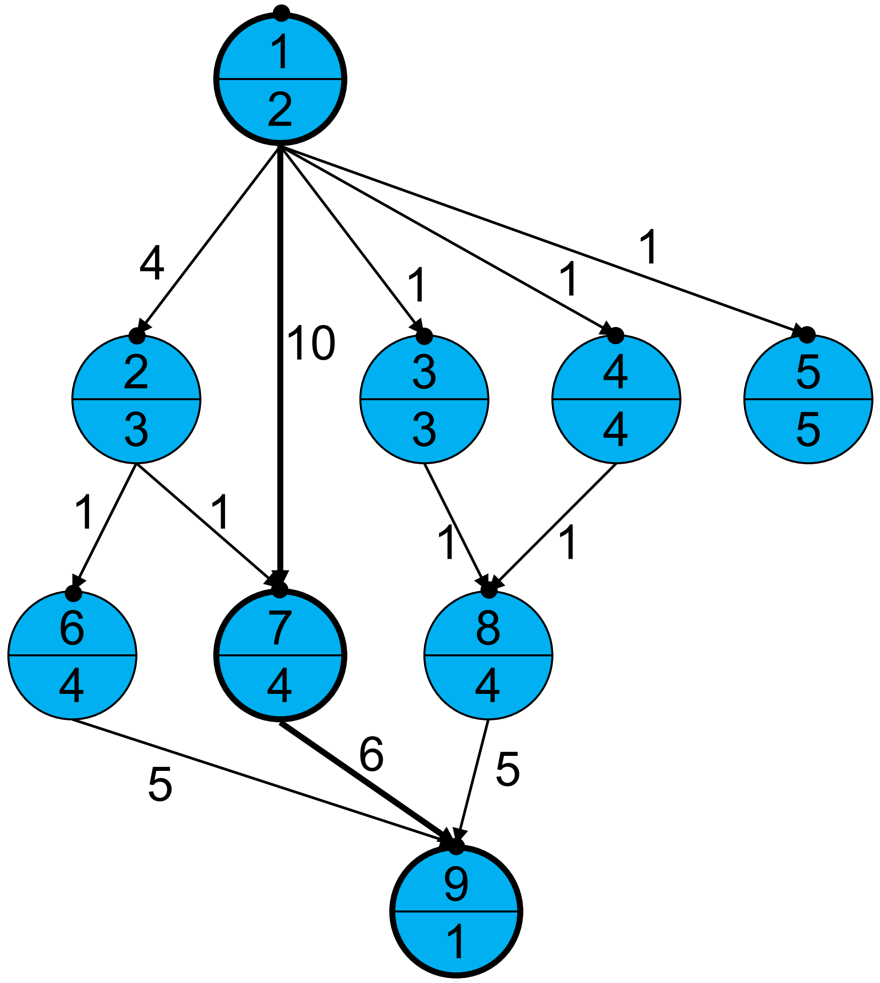
- Some scheduling algorithms use the attribute „as late as possible“ (ALAP). The ALAP value indicates the latest start time of the thread, that doesn't lead to an in increased schedule length (makespan) *ms*.

Calculation ALAP:

```
construct list of nodes in reverse topological order (RevTopList)
for each node $T_i \in$ RevTopList do
    minft $\leftarrow$ length of critical path
    for each child $T_y$ of $T_i$ do
        if (alap($T_y$) - $\alpha_{iy}$ < minft)
            minft $\leftarrow$ alap($T_y$) - $\alpha_{iy}$
    alap($T_i$) $\leftarrow$ minft - $\beta_i$
```

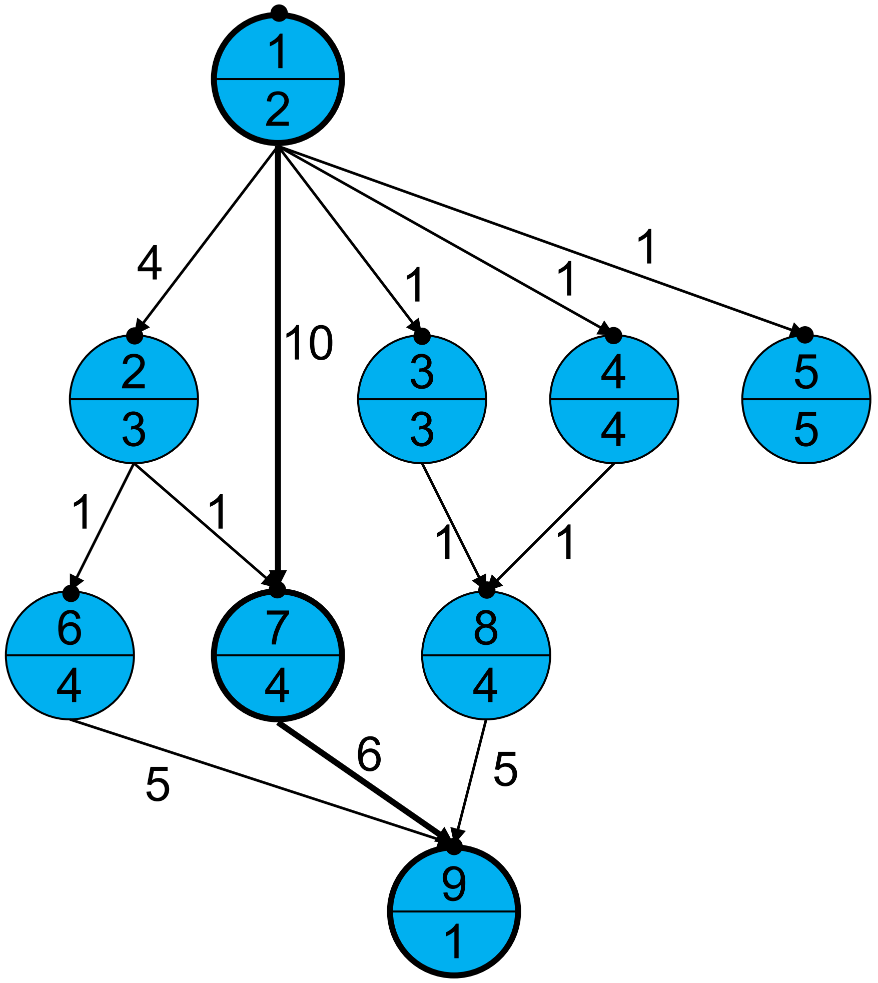
Example

thread	level	b-level	t-level	ALAP
T_1				
T_2				
T_3				
T_4				
T_5				
T_6				
T_7				
T_8				
T_9				



Example

thread	level	b-level	t-level	ALAP
T_1	11	23	0	0
T_2	8	15	6	8
T_3	8	14	3	9
T_4	9	15	3	8
T_5	5	5	3	18
T_6	5	10	10	13
T_7	5	11	12	12
T_8	5	10	8	13
T_9	1	1	22	22



Assumptions

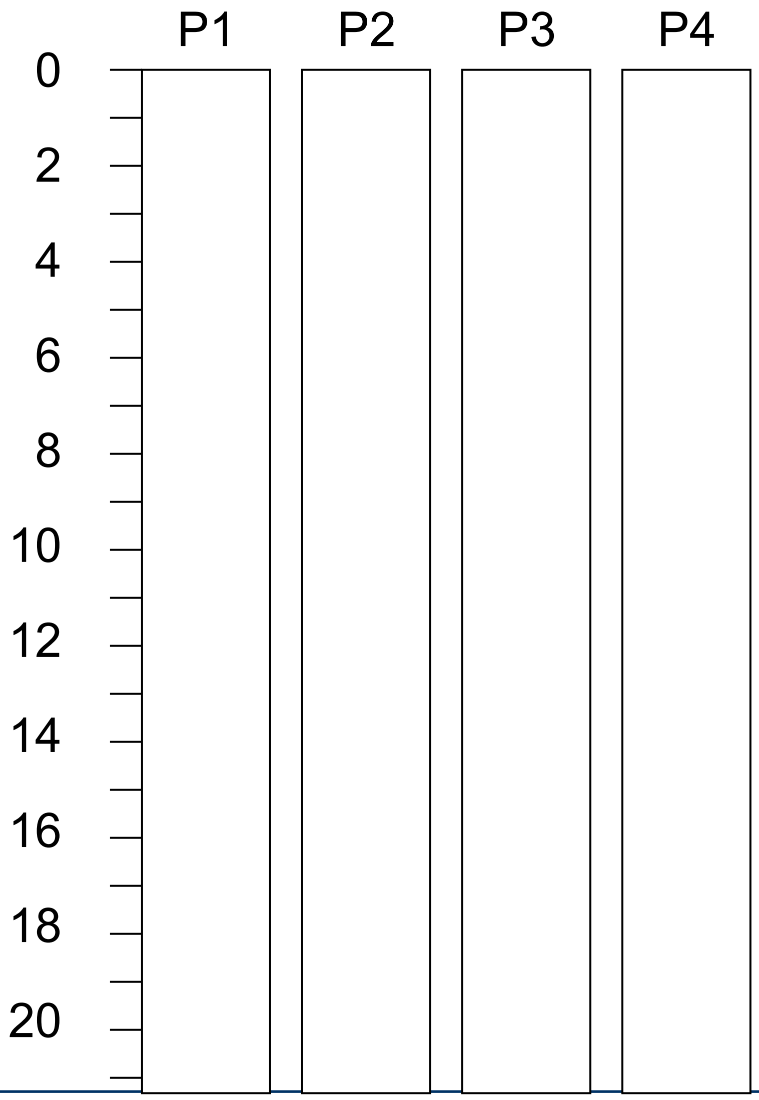
- The communication times between arbitrary processors are constant, i.e. the network topology does not matter.
- No contention for bandwidth, i.e. many messages can be sent over the same link without additional delay.
- The processor elements are able to execute a thread and simultaneously send messages.

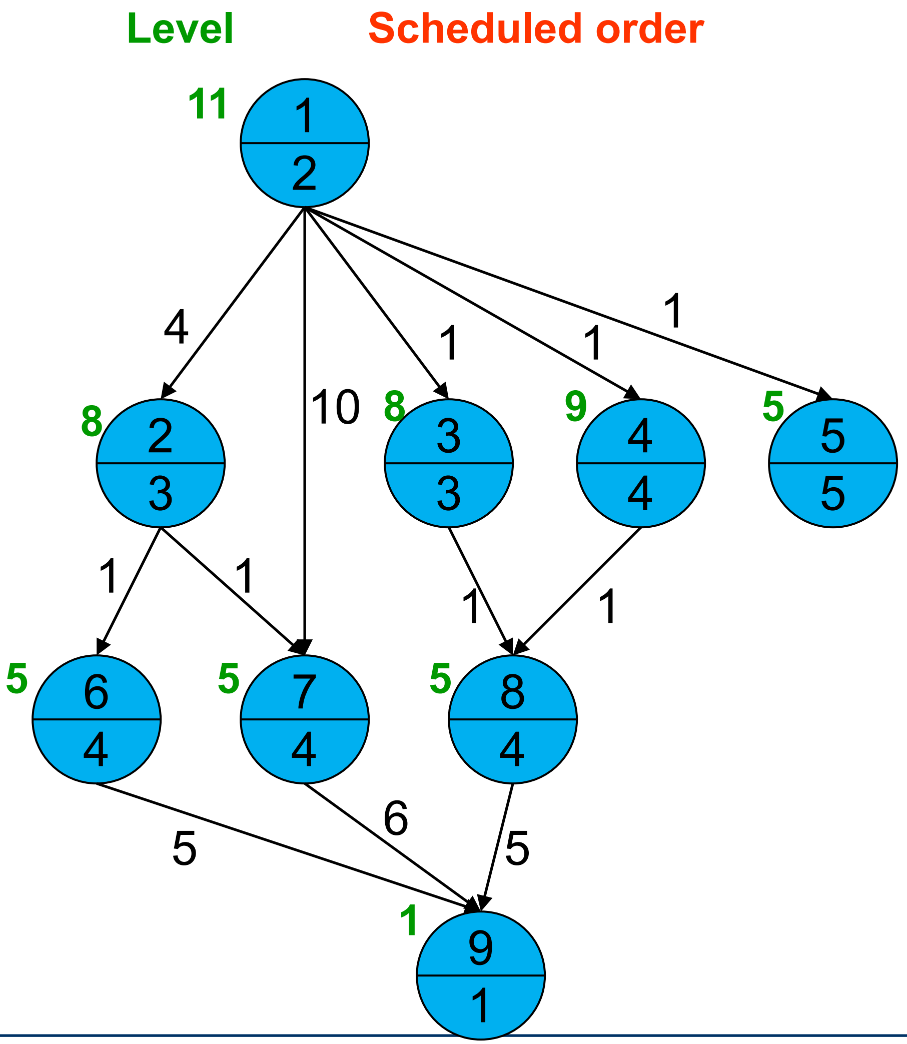
Insertion Scheduling Heuristic (ISH)

- Calculate Level (static b-Level) for each vertex
- Build the ready list in descending level-order from root vertices (entry nodes)
- **While** (ready-list not empty) **do**
  - Place the first thread of the ready list to the processor that allows the earliest execution.
  - If this placement induces idle times, find as many threads from ready list as possible that can be scheduled into this idle time, unless they can start on an another processor earlier.
  - update ready list.

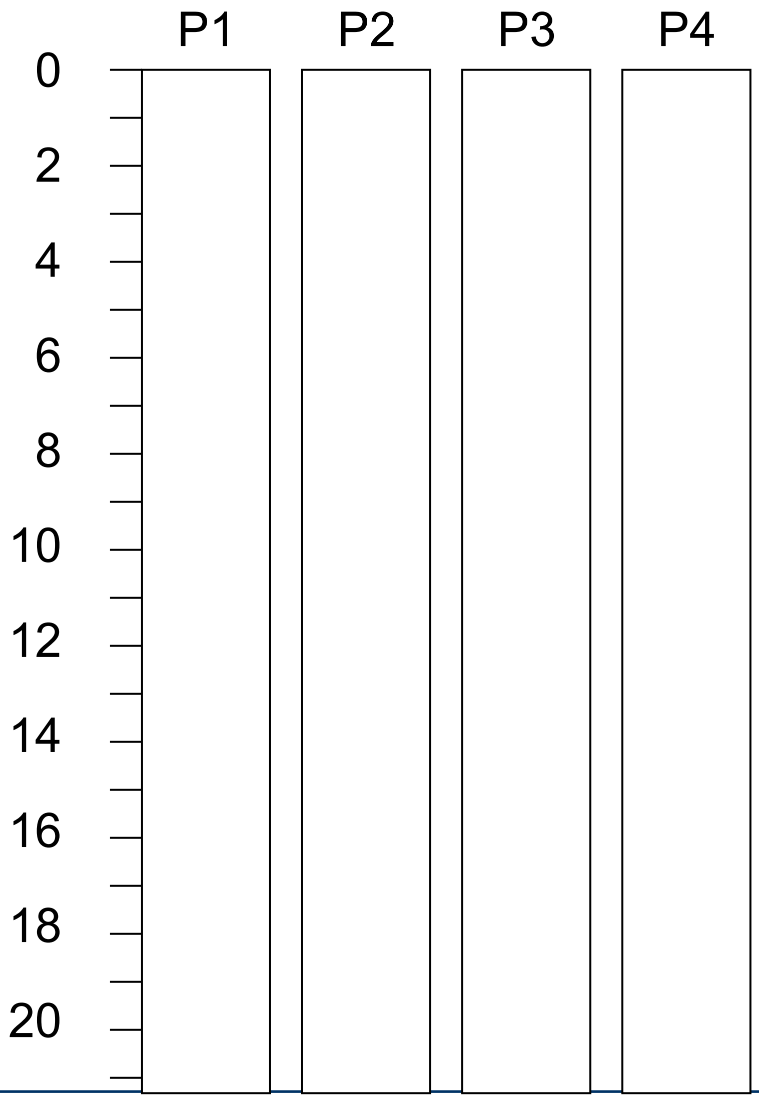
- Complexity: O(n^2)

ISH Example

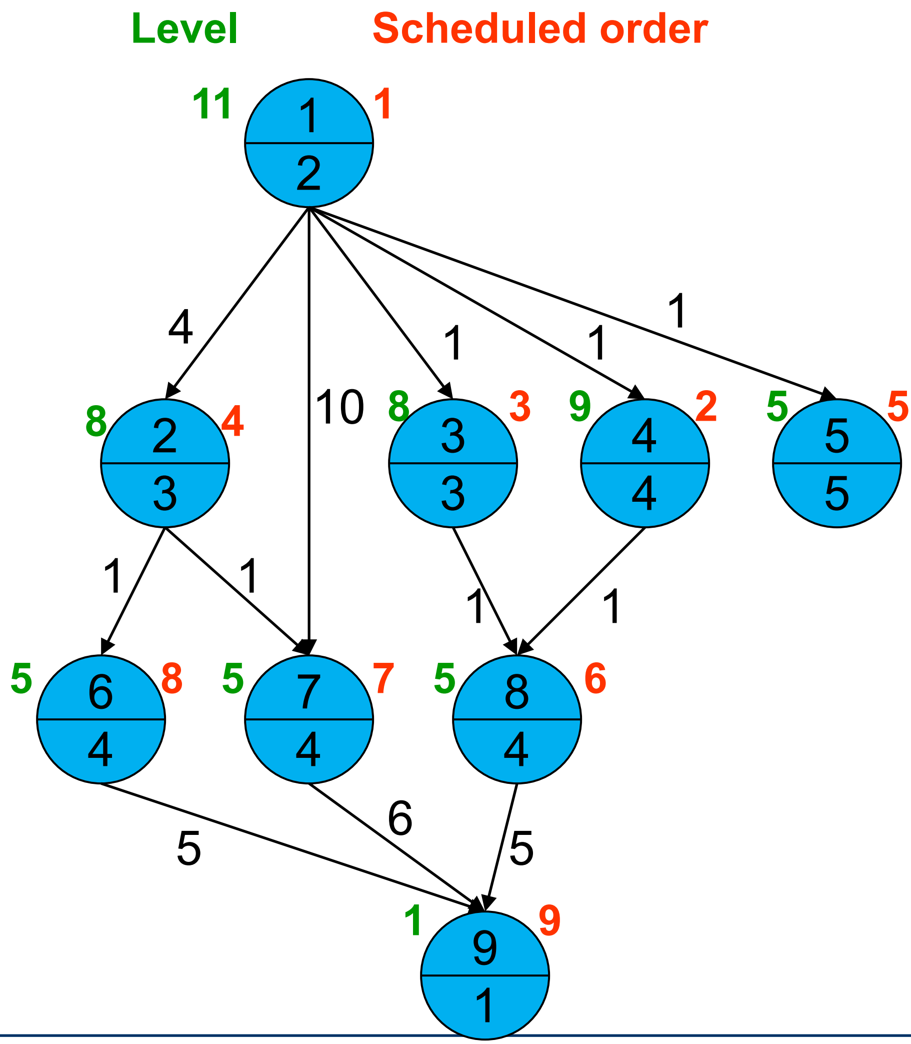

P1
P2
P3
P4
0
2
4
6
8
10
12
14
16
18
20


Level
Scheduled order
11
1
2
4
10
1
1
1
8
2
3
8
3
3
9
4
4
5
5
5
1
1
1
1
5
6
4
5
7
4
5
8
4
5
6
5
1
9
1

ISH Example

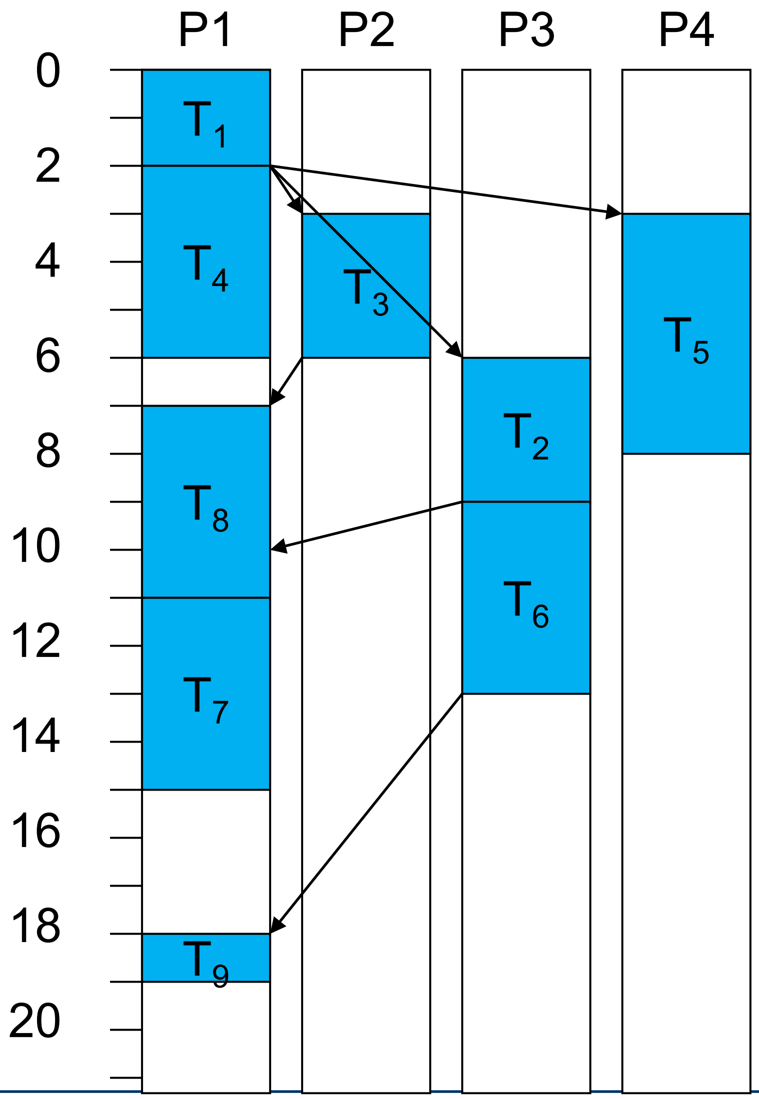


P1
P2
P3
P4
0
2
4
6
8
10
12
14
16
18
20

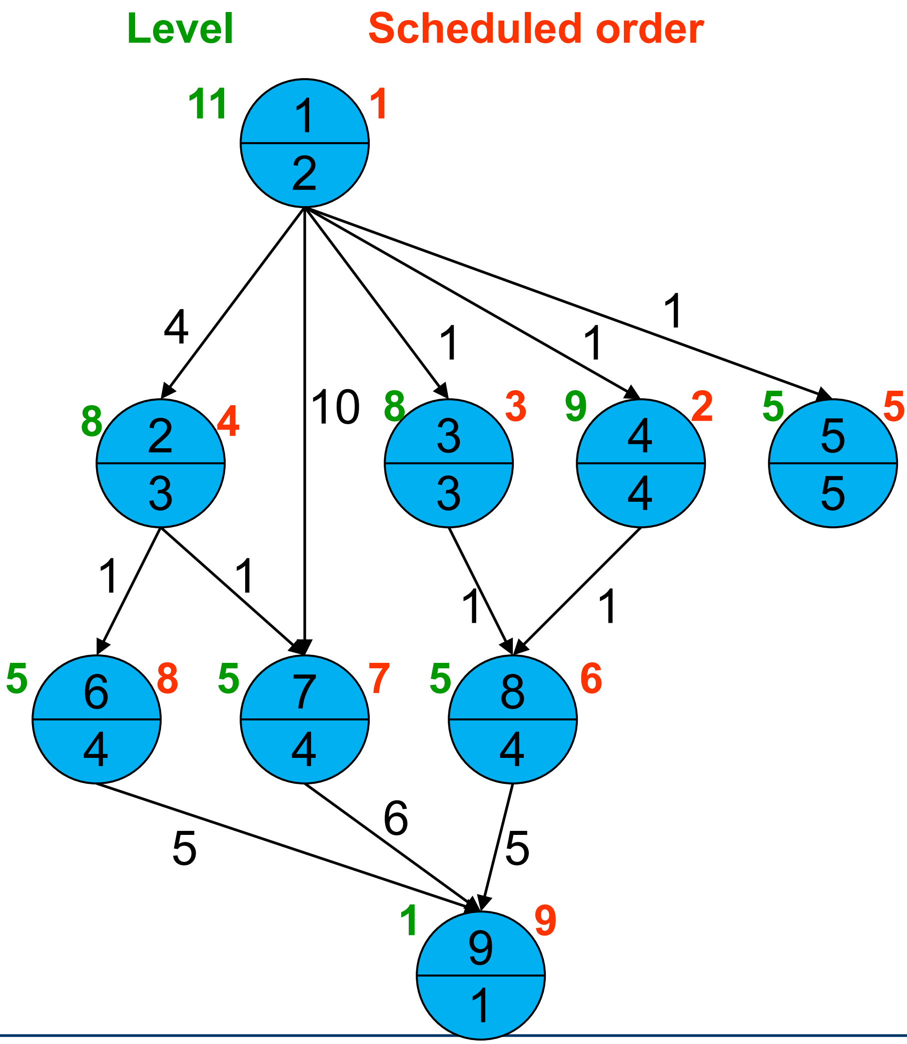


Level
Scheduled order
11
1
1
2
4
10
1
1
1
8
4
2
3
8
3
3
3
9
2
4
4
5
5
5
5
1
1
1
1
5
8
6
4
5
7
7
4
5
6
8
4
5
6
5
1
9
9
1

ISH Example



P1
P2
P3
P4
0
2
4
6
8
10
12
14
16
18
20
T_1
T_4
T_3
T_5
T_2
T_8
T_6
T_7
T_9



Level
Scheduled order
11
1
1
2
4
10
1
1
1
8
2
3
4
8
3
3
3
9
4
4
2
5
5
5
5
1
1
1
1
5
6
4
8
5
7
4
7
5
8
4
6
5
6
5
1
9
1
9

Modified Critical Path Method (MCP)

Initialization:

Calculate ALAP times of each vertex.

Build for each vertex a list that contains the ALAP times of itself and its successors in descending order.

Sort these lists in ascending lexicographic order.

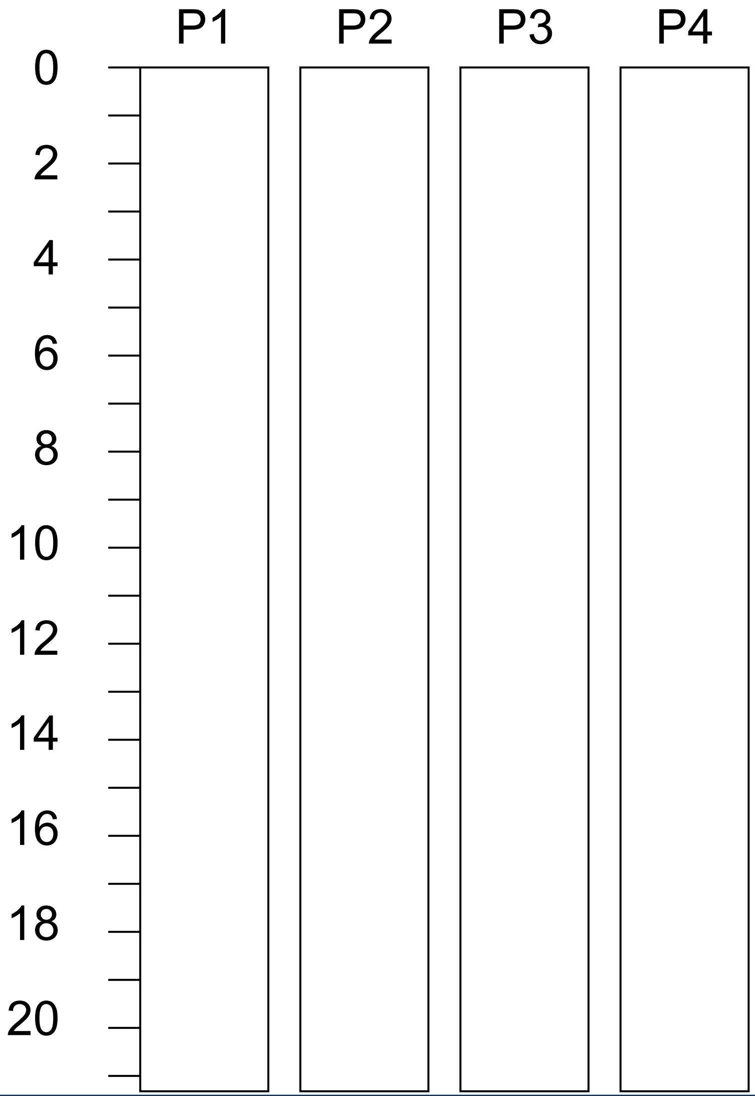
Create a vertex list in that order.

Loop:

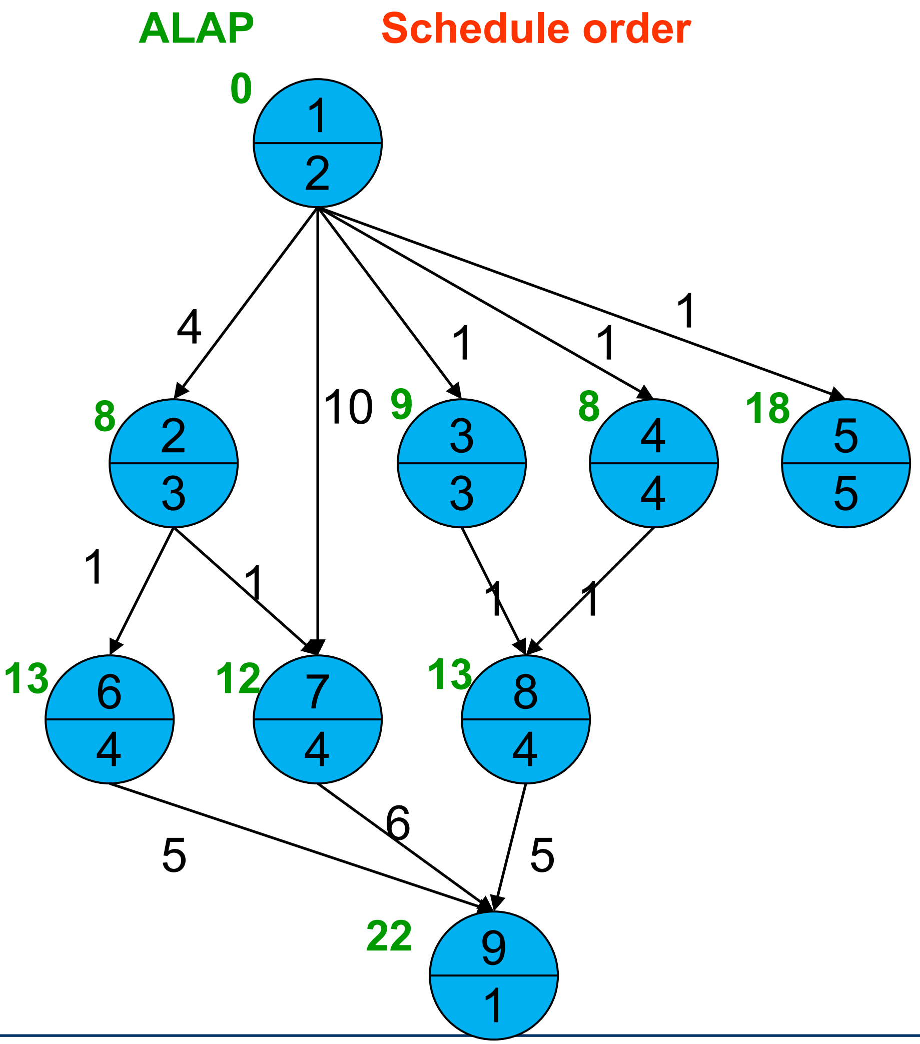
```
while vertex list not empty do
    Place the first vertex of the list to that processor that allows the
    earliest execution.
    Remove that vertex from list.
```

Complexity $O(n^2 \log n)$

MCP Example

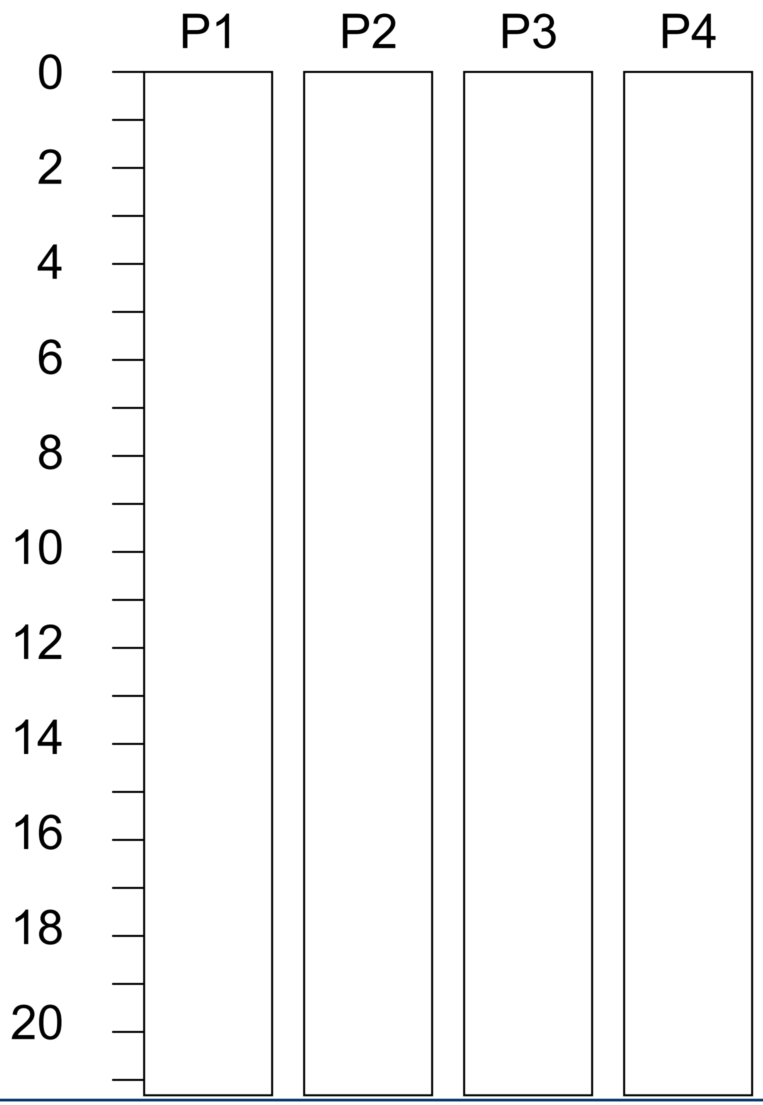


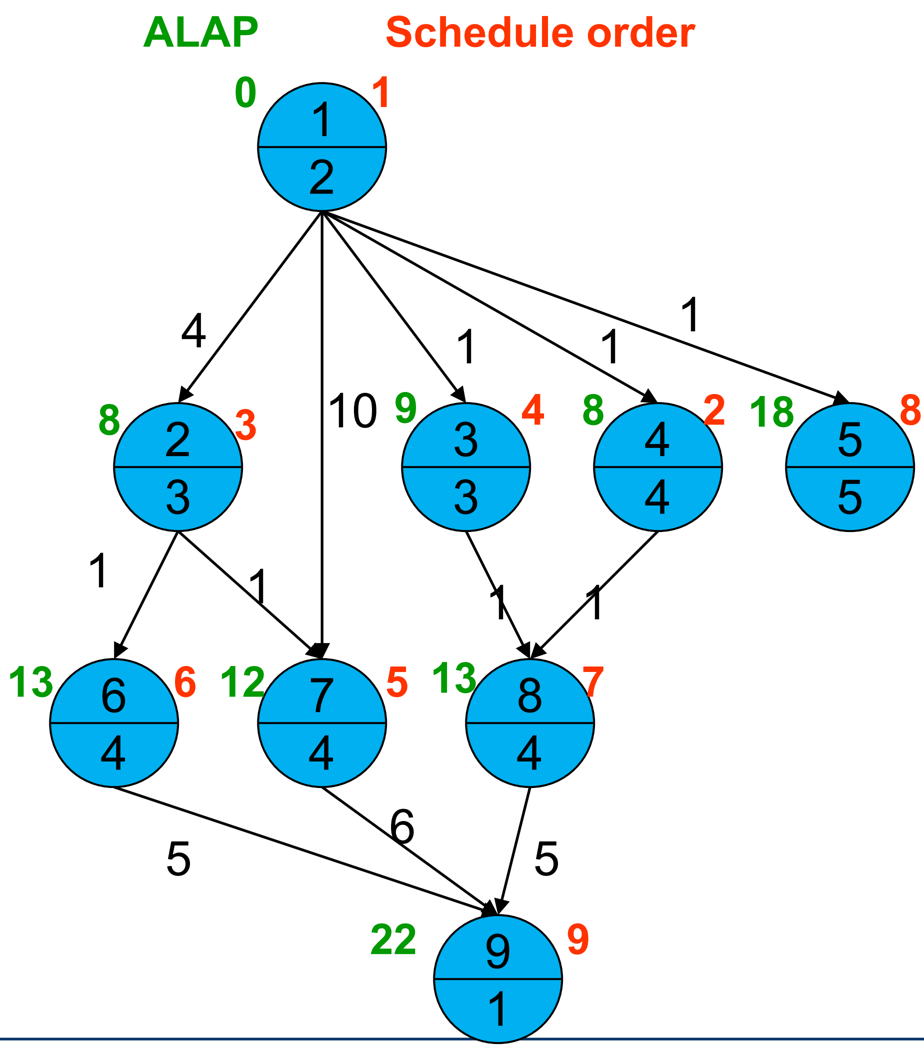
P1
P2
P3
P4
0
2
4
6
8
10
12
14
16
18
20



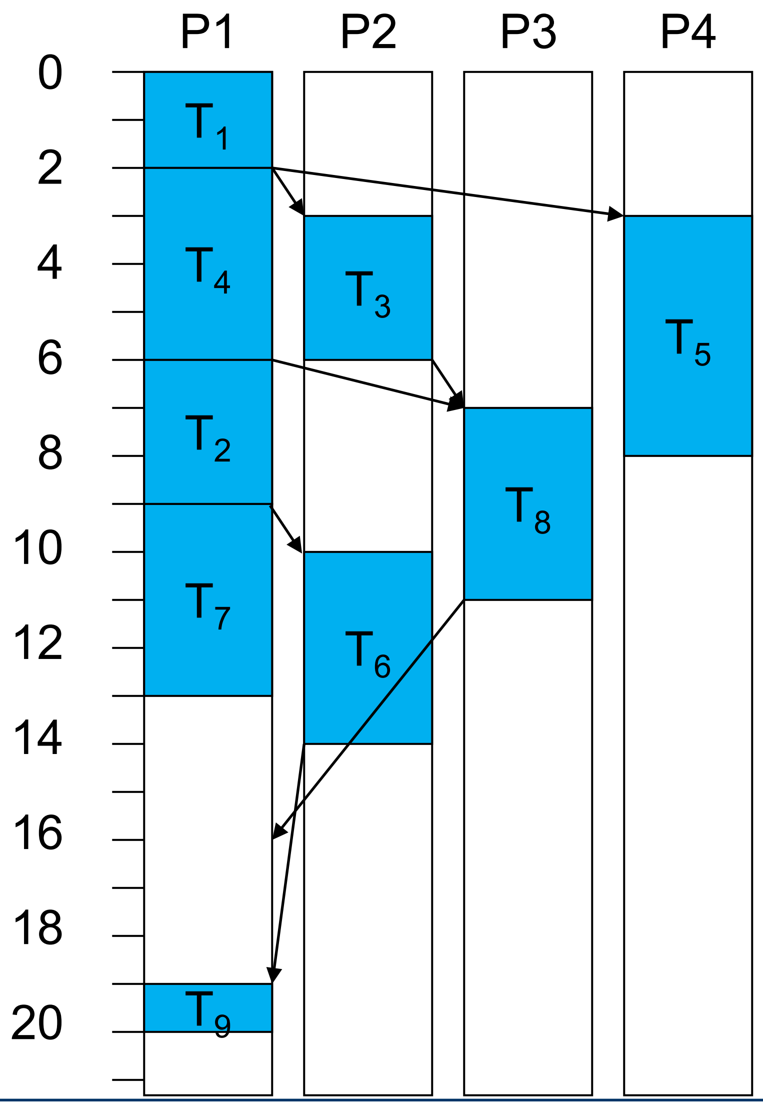
ALAP
Schedule order
0
1
2
4
10
1
1
1
8
2
3
9
3
3
8
4
4
18
5
5
1
1
1
1
13
6
4
12
7
4
13
8
4
5
6
5
22
9
1

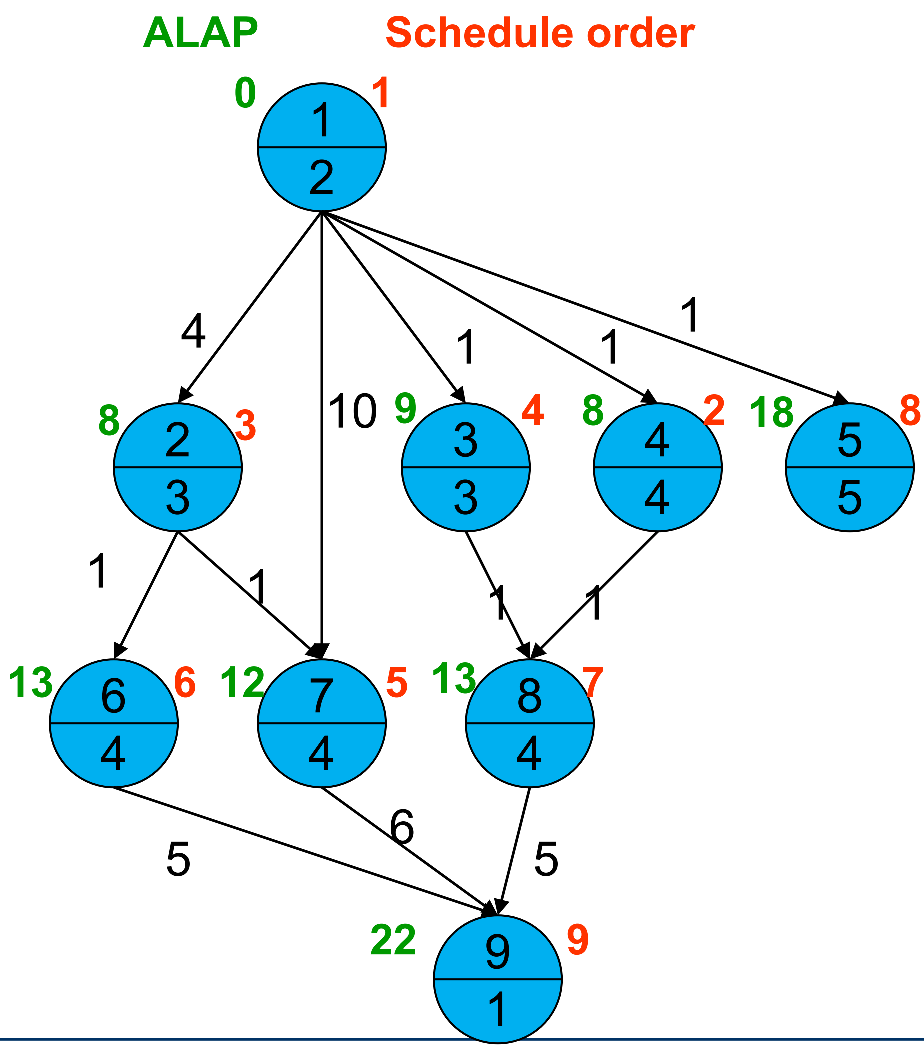
MCP Example


P1
P2
P3
P4
0
2
4
6
8
10
12
14
16
18
20


ALAP
Schedule order
0
1
1
2
4
10
1
1
1
8
2
3
3
9
3
3
4
8
4
4
2
18
5
5
8
1
1
1
1
13
6
4
6
12
7
4
5
13
8
4
7
5
6
5
22
9
1
9

MCP Example


P1
P2
P3
P4
0
2
4
6
8
10
12
14
16
18
20
T_1
T_4
T_2
T_7
T_9
T_3
T_6
T_8
T_5

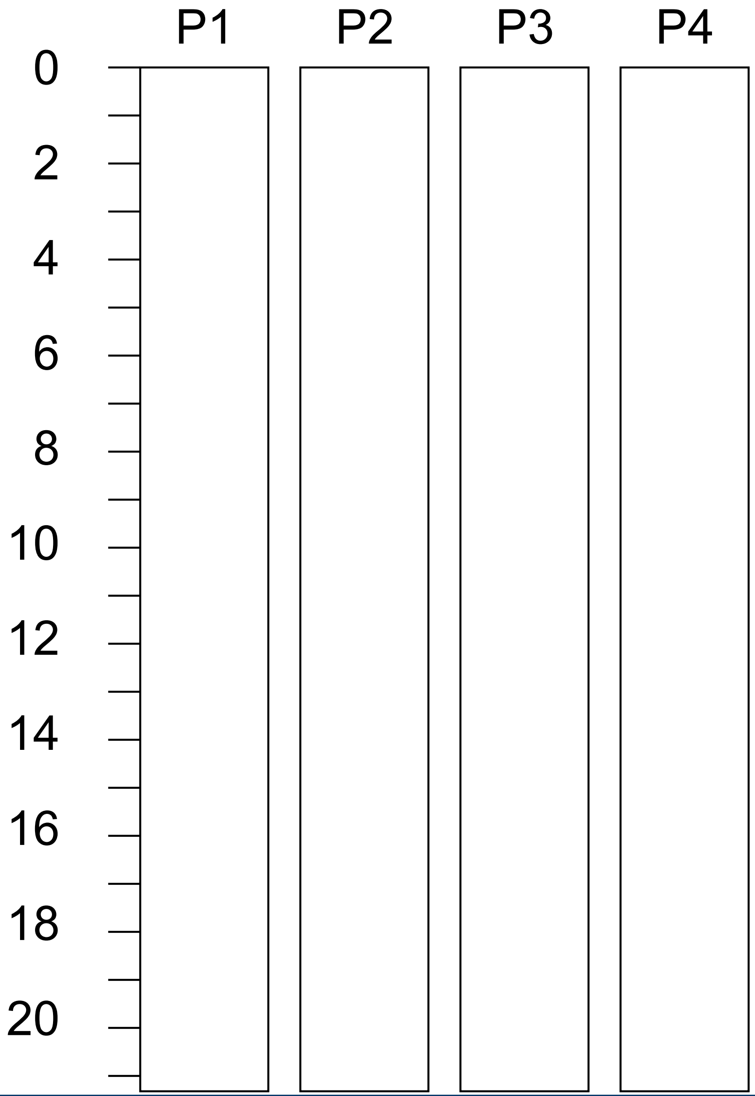

ALAP
Schedule order
0
1
1
2
4
10
1
1
1
8
2
3
3
9
3
3
4
8
4
4
2
18
5
5
8
1
1
1
1
13
6
4
6
12
7
4
5
13
8
4
7
5
6
5
22
9
1
9

Earliest Time First Method (ETF)

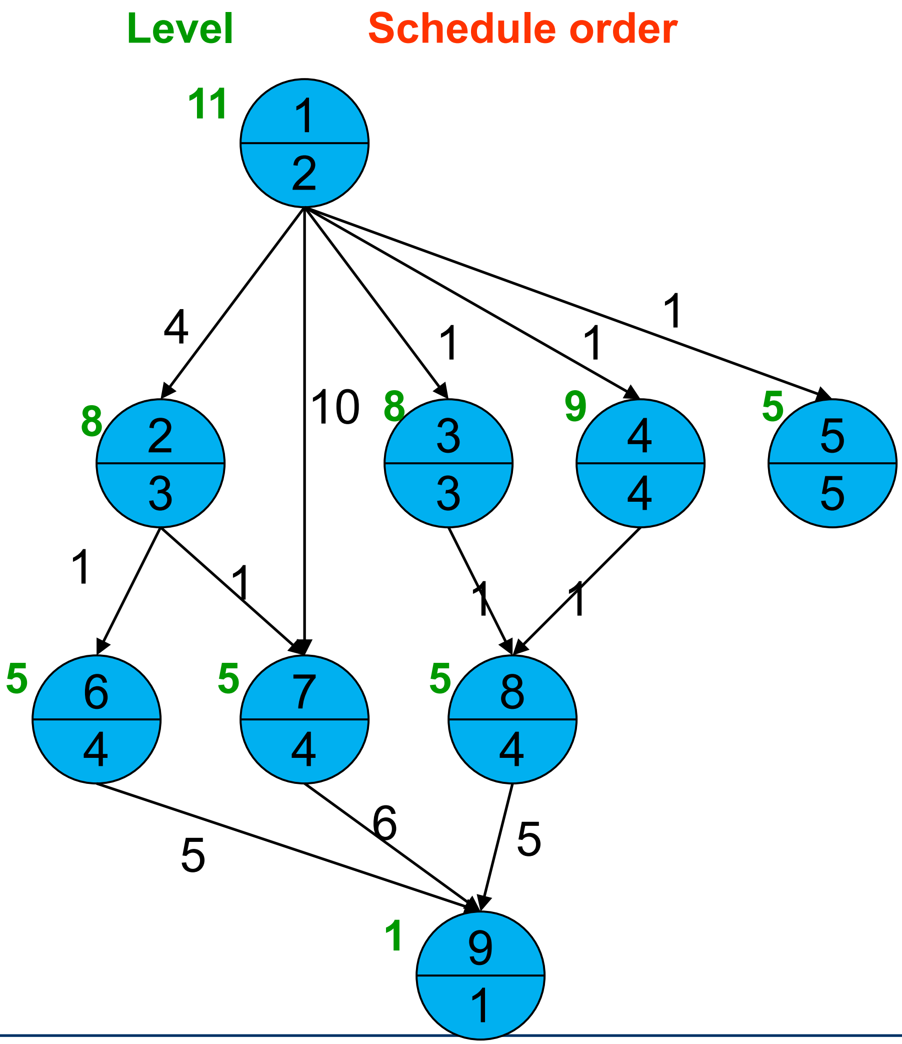
```
Initialization:
    Calculate the static b-Level of each vertex.
    Insert source vertices into ready list.
Loop:
    while vertex list not empty do
        Calculate earliest starting time on each processor for each
        vertex in the ready list.
        Select the vertex-processor-pair with the smallest start time.
        Place the vertex accordingly.
        Insert new ready vertices into ready list.
```

Complexity $O(n^2)$

ETF Example

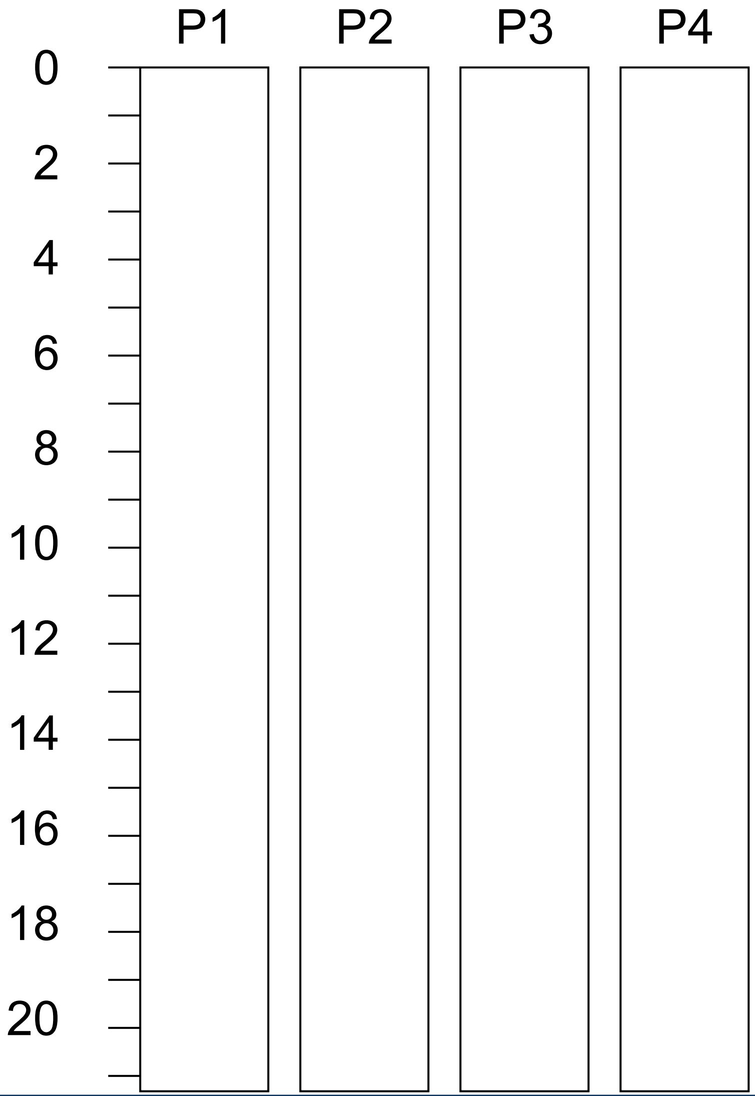


P1
P2
P3
P4
0
2
4
6
8
10
12
14
16
18
20

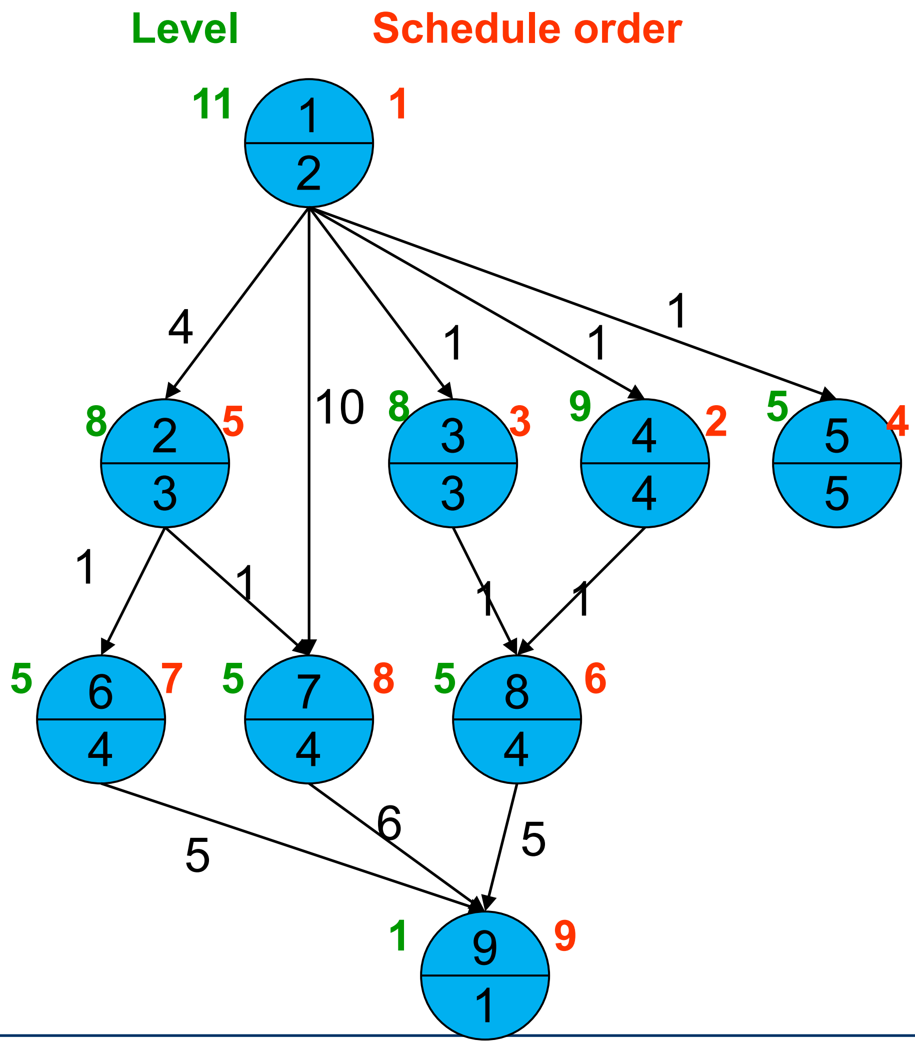


Level
Schedule order
11
1
2
4
10
1
1
1
8
2
3
8
3
3
9
4
4
5
5
5
1
1
1
1
5
6
4
5
7
4
5
8
4
5
6
5
1
9
1

ETF Example

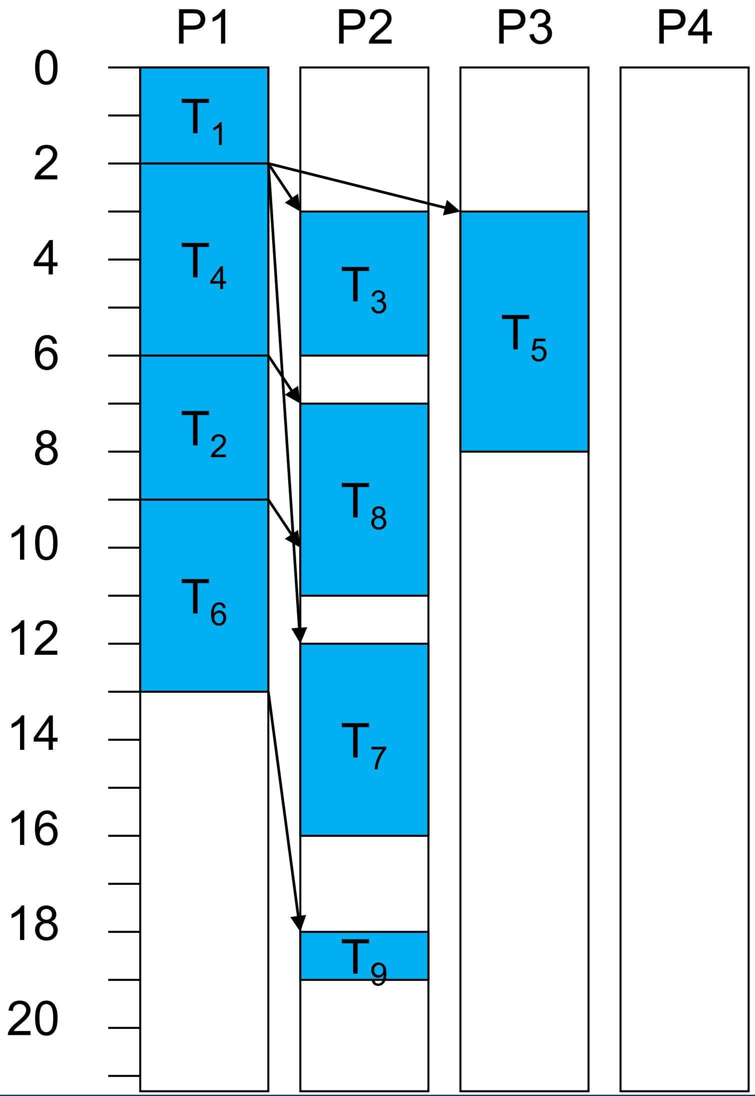


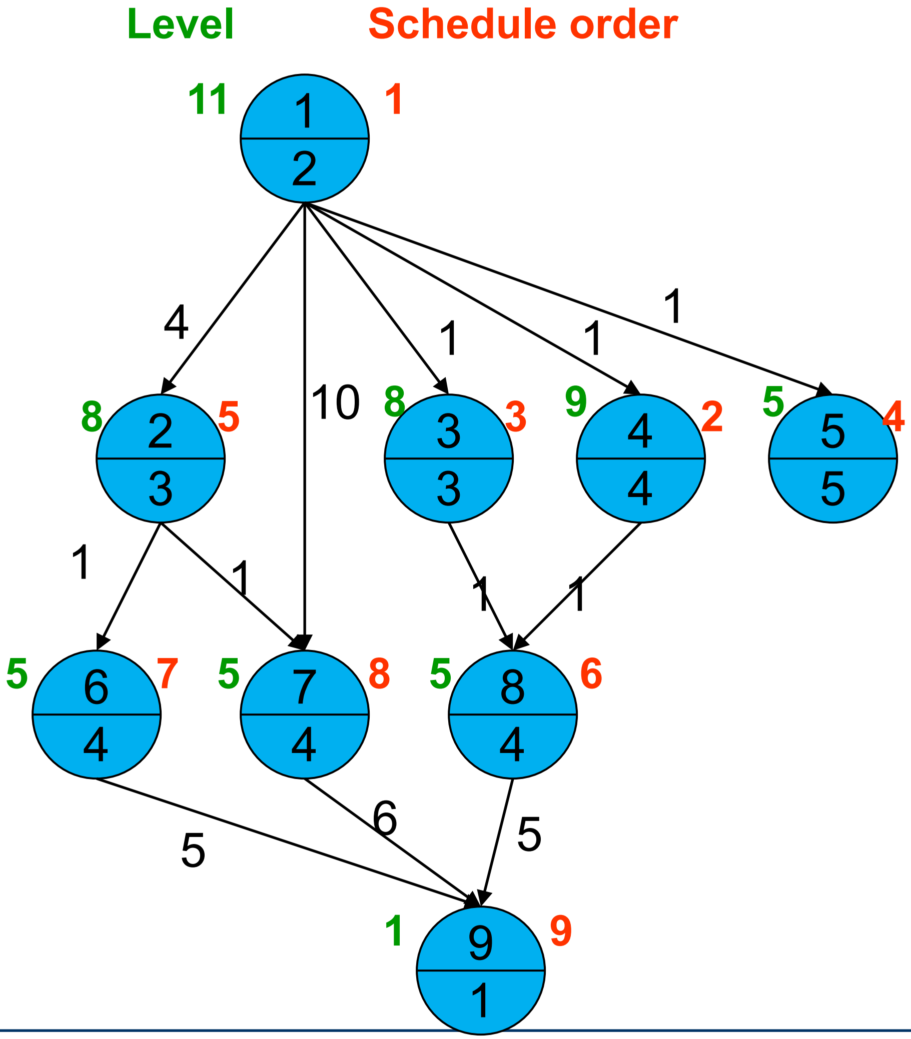
P1
P2
P3
P4
0
2
4
6
8
10
12
14
16
18
20



Level
Schedule order
11 1 2 1
8 2 3 5
8 3 3 3
9 4 4 2
5 5 5 4
5 6 4 7
5 7 4 8
5 8 4 6
1 9 1 9
4
10
1
1
1
1
1
1
1
5
6
5

ETF Example


P1
P2
P3
P4
0
2
4
6
8
10
12
14
16
18
20
T1
T4
T2
T6
T3
T8
T7
T9
T5


Level
Schedule order
11
1
4
10
1
1
1
8
5
8
3
9
2
5
4
1
1
1
1
5
7
5
8
5
6
5
6
5
1
9

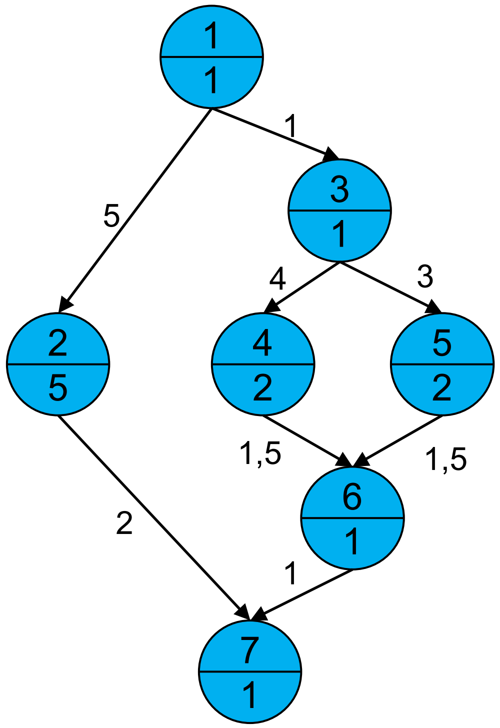
10.4 Scheduling with Clustering

- Approaches using clustering decompose the allocation process into two phases:
  - Phase 1: Clustering, i.e. combining all threads that are allocated to the same processor.
  - Phase 2: Ordering; i.e. determining at which times the threads start execution.

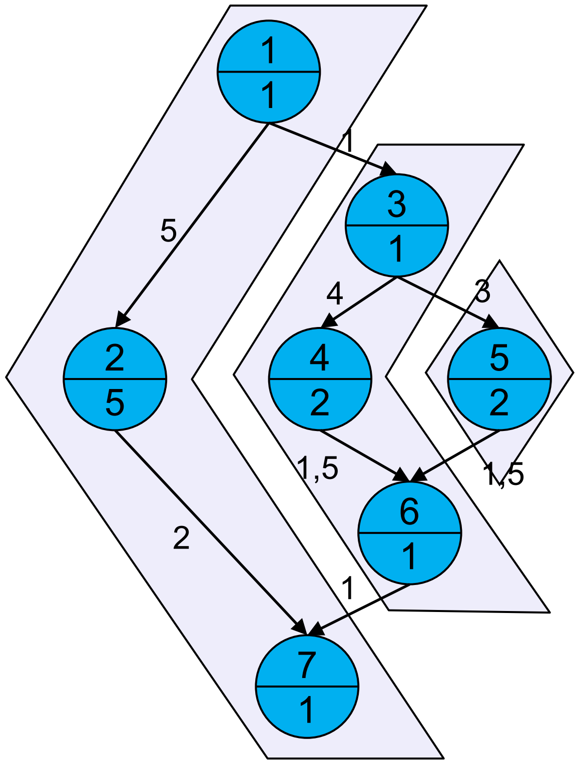
- There are as many clusters formed as processors are available
- The clustering process starts with single element clusters and stepwise merges clusters to larger ones.
- Threads of the same cluster run on the same processor. Communication between those threads is free of cost.

Linear Clustering

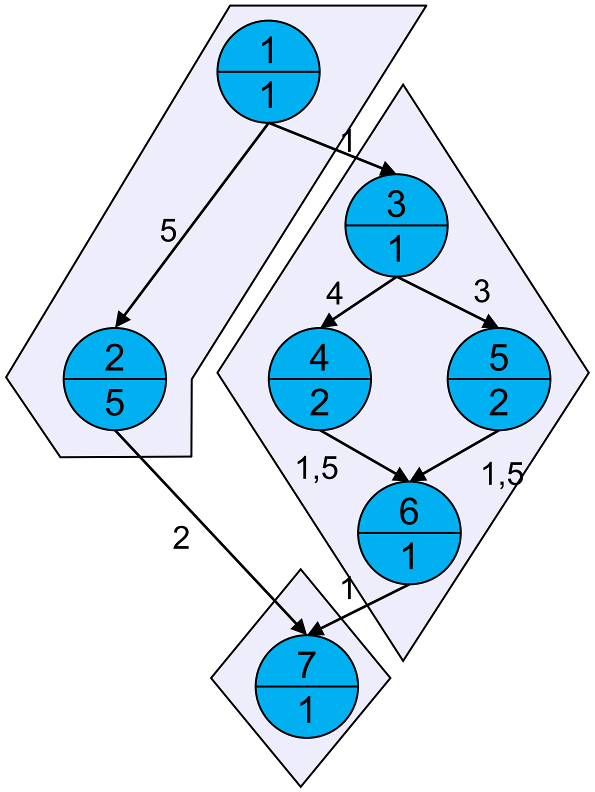
A clustering is called non**linear**, if two independent threads are assigned to the same cluster, otherwise it is called linear.



DAG



linear clustering

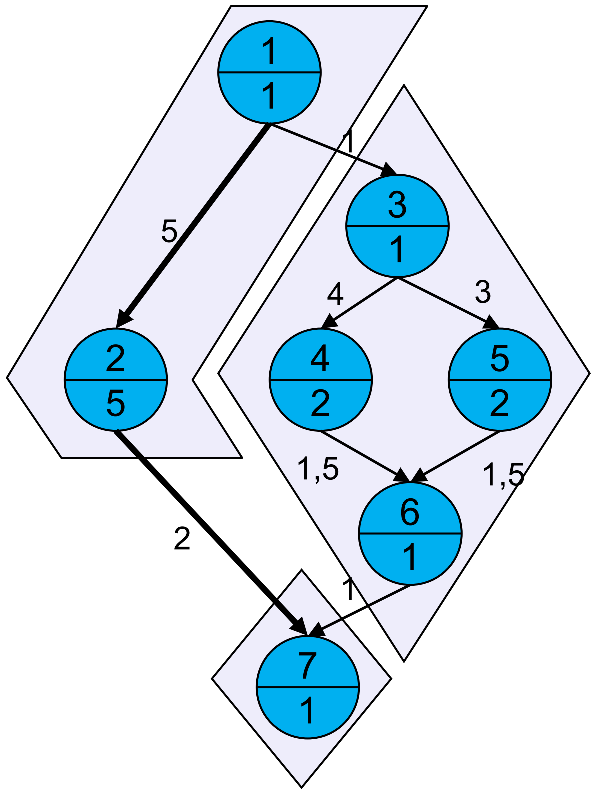


nonlinear clustering

Clustering

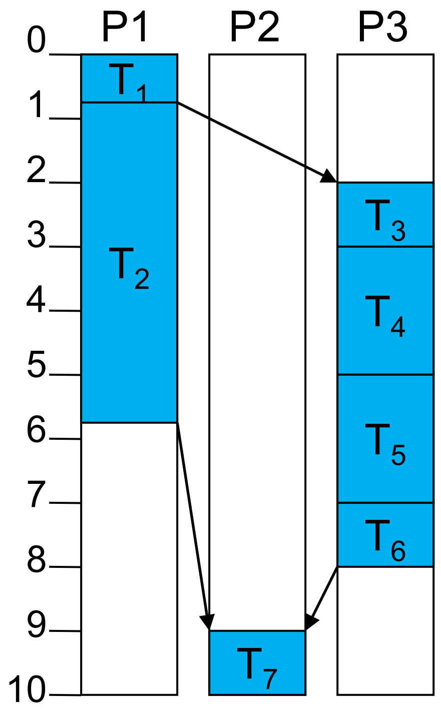
- The clustering modifies the DAG by zeroing some edge weights
- By doing so, the length of the critical path may also be affected.
- We make the distinction:
  - The (original) DAG
  - the DAG after clustering (clustered DAG, CDAG)
  - the DAG after ordering (scheduled DAG, SDAG)
- The critical path of the SDAG is called **dominant sequence** (DS) of the CDAG.

Critical Path and Dominant Sequence

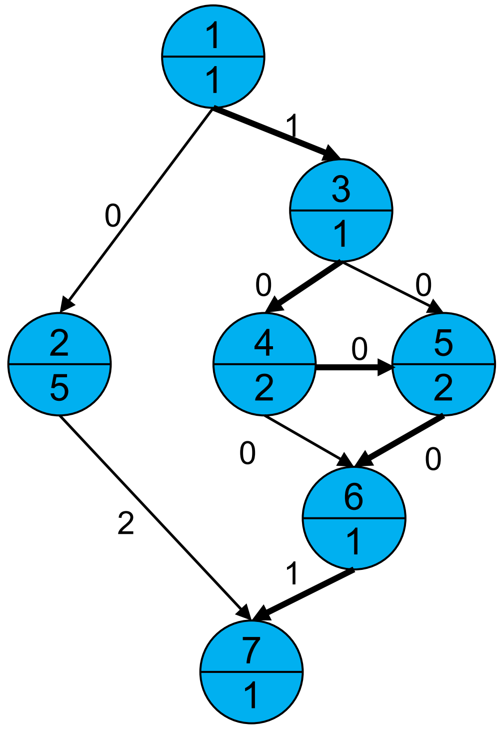


CDAG

CP: T_1, T_2, T_7
CP length: 9



Gantt-Chart



SDAG

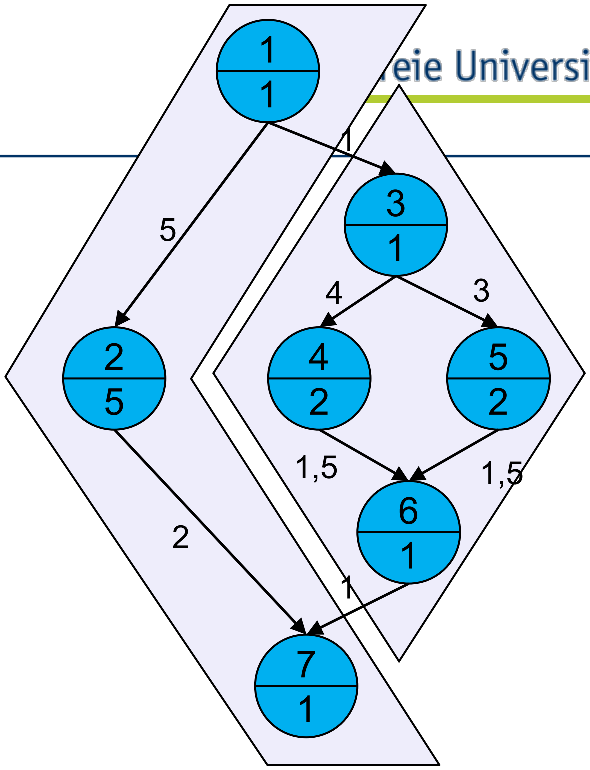
DS: T_1, T_3, T_4, T_5, T_6, T_7

DS length: 10

Sarkar's Algorithm

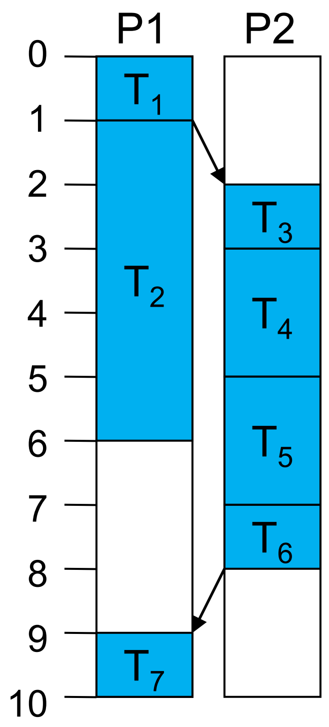
- Initialization:
  - All vertices are forming single element clusters
  - All edges are unmarked
  - All edges are being sorted according to descending communication cost.
- Repeat
  - Set unmarked edge with highest weight to 0 (merge clusters), if the makespan is not increased by that.
  - Mark that edge
  - If two clusters are merged, the edges are ordered according to HLF-rule (highest b-level first.).

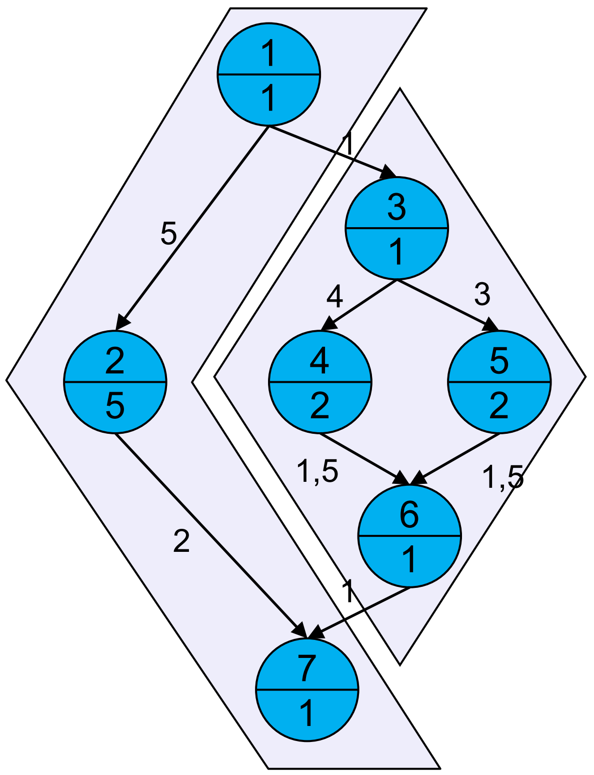
  until all edges marked



Step	edge	Zeroing	makespan
0			14
1	(T_1,T_2)	(T_1,T_2)	13,5
2	(T_3,T_4)	(T_3,T_4)	12,5
3	(T_3,T_5)	(T_3,T_5)	11,5
4	(T_2,T_7)	(T_2,T_7)	11,5
5	(T_4,T_6)	(T_4,T_6)	11,5
6	(T_5,T_6)	(T_5,T_6)	10
7	(T_1,T_3)	$\varnothing$	10
8	(T_6,T_7)	$\varnothing$	10

Sarkar's Algorithm





Complexity: O(e (n+e))

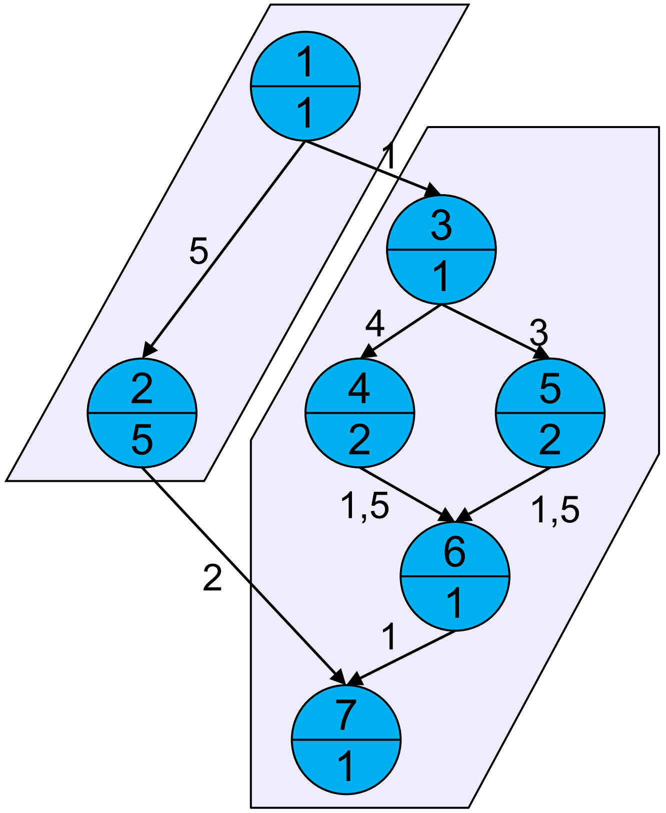
Dominant Sequence Clustering (DSC)

- Initialization:
  - All vertices are single element clusters
  - All edges are unmarked
  - r=0
  - Calculate DS_0
  - Initialize Ready List with source vertices
- While not all edges marked
  - Let (Ti,Tj) be the topmost unmarked edge in DSr
  - Mark this edge
  - Delay zeroing of edge until Tj becomes ready.
  - Select ready vertex T_k as the one which runs to the longest path through ready vertices of the SDAG
  - Zero those incoming edges of T_k, that minimize the t-level of T_k.
  - Schedule T_k after the last already scheduled vertex of its cluster.
  - Insert successors of T_k into ready list that just became ready, if available
  - If all edges in DSr are marked: increment r and find new DSr

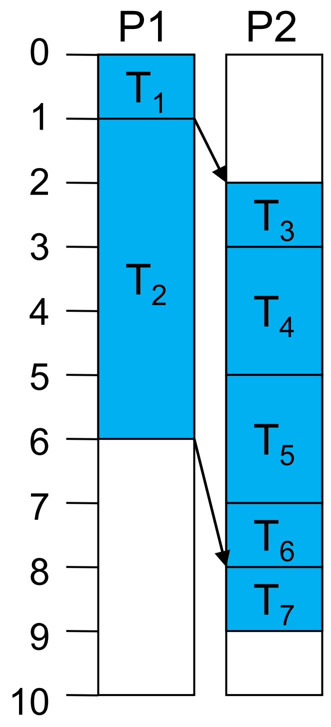
  end while

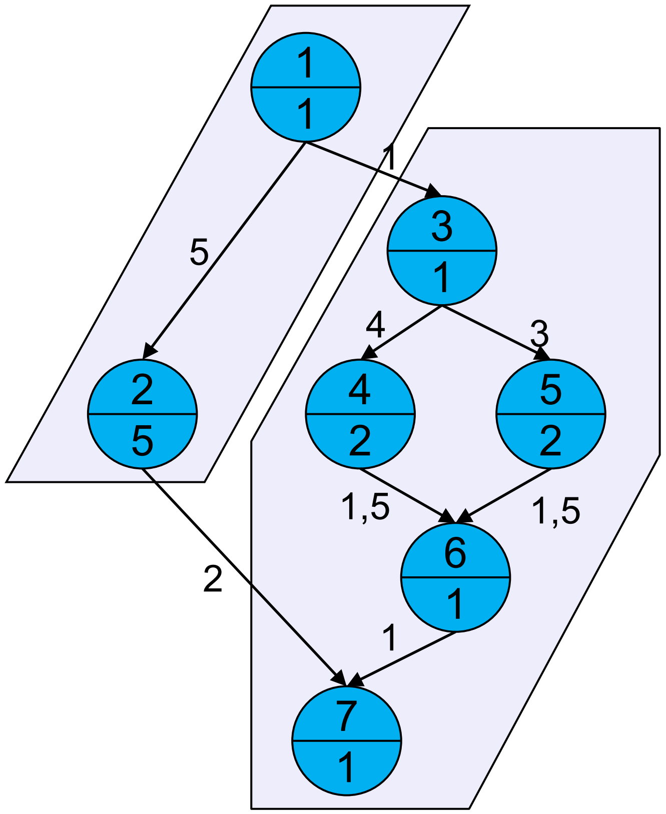
DSC-Algorithm

Step	Edge	ready	Zeroing	makespan
0				14
1	∅	T_1	∅	14
2	(T_1,T_2)	T_2	(T_1,T_2)	13,5
3	(T_1,T_3)	T_3	∅	13,5
4	(T_3,T_5)	T_4	(T_3,T_4)	12,5
5	(T_2,T_7)	T_5	(T_3,T_5)	11,5
6	(T_4,T_5)	T_6	(T_4,T_6) (T_5,T_6)	10
7	(T_5,T_6)	T_7	(T_6,T_7)	9



DSC-Algorithm





Complexity: O((n+e) log n)

Further references

- Buyya,R.: *High Performance Cluster Computing*, Vol. 1, Prentice Hall, 1999, chapter 24
- Zomaya,A.: *Parallel and Distributed Computing Handbook*, McGraw Hill, 1995, chapter 9
- Gerasoulis, A.; Yang,T.: *A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Task Graphs on Multiprocessors*, J. of Parallel and Distributed Computing 16 (1992), pp. 276-291
- Kwok, Y-K.; Ahmad, I.: *Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors*, ACM Computing Surveys 31(4), December 1999, pp. 406-471