CR02 - Resilient and energy-aware scheduling algorithms

Anne Benoit

ENS Lyon

Anne.Benoit@ens-lyon.fr

http://graal.ens-lyon.fr/~abenoit/

CR02 - 2016/2017

Course Outline

- Scheduling
- Resilience
- Energy

Task graph scheduling (Scheduling part 1)

Anne Benoit

ENS Lyon

Anne.Benoit@ens-lyon.fr

http://graal.ens-lyon.fr/~abenoit

CR02 - 2016/2017

Outline

What is scheduling?

- Scheduling is studied in Computer Science and Operations Research
- Broad definition: the temporal allocation of activities to resources to achieve some desirable objective
- Examples:
  - Assign workers to machines in an factory to increase productivity
  - Pick classrooms for classes at a university to maximize the number of free classrooms on Fridays
  - Assign users to a pay-per-hour telescope to maximize profit
  - **Assign computations to processors and communications to network links so as to minimize application execution time**

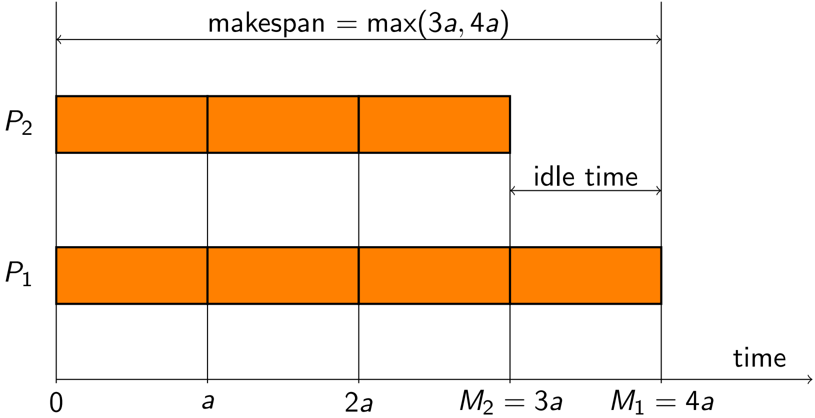
A simple scheduling problem

- A scheduling problem is defined by three components:
  1. A description of a set of resources
  2. A description of a set of tasks
  3. A description of a desired objective
- Let us get started with a simple problem: INDEP(2)
  1. Two identical processors, P_1 and P_2
     - Each processor can run only one task at a time
  2. Application: n compute tasks
     - Task i can run on P_1 or P_2 in a_i seconds
     - Tasks are independent: can be computed in any order
  3. Objective: minimize $\max(M_1, M_2)$
     - M_i is the time at which processor P_i finishes computing

The easy case

- If all tasks are identical, i.e., take the same amount of compute time, then the solution is obvious: Assign $\lceil n/2 \rceil$ tasks to $P1$ and $\lfloor n/2 \rfloor$ tasks to P_2
  - Rule of thumb: try to have both processors finish at the same time
- Problem size is $O(n)$ (not $O(\log n)$ because each task needs to be identified)
- The "scheduling algorithm" is $O(n)$, therefore we have a polynomial time (in fact linear) algorithm
  - For each task, pick one of the two processors by comparing the index of the task with $n/2$
- We declare the problem "solved"

Gantt chart for INDEP(2) with 7 identical tasks



Non-identical tasks

- Task T_i (for $i = 1, \ldots, n$) takes time $a_i \geq 0$
- Problem size:
  $Size = O(n + \sum_{i=1}^{n} \log a_i)$ or $Size = O(n^3 \times \max_{i=1}^{n} \log a_i))$ or ...
- We say a problem is "easy" when we have a polynomial-time (p-time) algorithm:
  - Number of elementary operations is $O(f(Size))$, where f is a polynomial and $Size$ is the problem size
- $\mathcal{P}$ is the set of problems that can be solved with a p-time algorithm
- Question: is there a p-time algorithm to solve $\mathrm{INDEP}(2)$?
- Disclaimer: Some of you may be familiar with algorithms and computational complexity, so bear with me while I review some fundamental background

Decision vs. optimization problem

- Complexity theory is for decision problems, i.e., problems that have a yes/no answer
- Scheduling problems are optimization problems
- Decision version of $\text{INDEP}(2)$: for an integer bound k, is there a schedule whose makespan is lower than k?
- If we have a p-time algorithm for the optimization problem, then we have p-time algorithm for the decision problem
  - Run the optimization algorithm, and check whether the makespan is lower than k

Decision vs. optimization problem

- If the decision problem is in $\mathcal{P}$, then there is often (not always!) a p-time algorithm to solve the optimization problem
  - Binary search for the lowest k ($k \leq n \times \max_i a_i$)
  - Adds a $\log(n \times \max_i a_i)$ complexity factor, still p-time
- Almost always the case in scheduling, and decision and optimization problems are often thought of as interchangeable

Problem size?

- One has to be careful when defining the problem size
- For INDEP(2):
  - We need to enumerate n integers (the a_i's), so the size is at least polynomial in n
  - Each a_i must be encoded (in binary) in $\lceil \log(a_i) \rceil$ bits
  - The data is $O(f(n) + \sum_{i=1}^{n} \lceil \log(a_i) \rceil)$, where f is a polynomial
- A problem is in $\mathcal{P}$ only if an algorithm exist that is polynomial in the data size as defined above

Pseudo-polynomial algorithm

- It is often possible to find algorithms polynomial in a quantity that is exponential in the (real) problem size
- For instance, to solve INDEP(2), one can resort to dynamic programming to obtain an algorithm with complexity $O(n \times \sum_{i=1}^{n} a_i)$ EXERCISE
- This is a polynomial algorithm if the a_i's are encoded in unary, i.e., polynomial in the numerical values of the a_i's
- But with the a_i's encoded in binary, $\sum_{i=1}^{n} a_i$ is exponential in the problem size!
  - To a log, linear is exponential ☺
- We say that this algorithm is pseudopolynomial

So, is INDEP(2) difficult?

- Nobody knows a p-time algorithm for solving INDEP(2)
- We define a new complexity class, $\mathcal{NP}$
  - Problems for which we can verify a certificate in p-time.
  - "Given a possible solution, can we check that the problem's answer is Yes in p-time?"
- There are problems not in $\mathcal{NP}$, but not frequent
- Obviously $\mathcal{P} \subseteq \mathcal{NP}$
  - empty certificate, just solve the problem
- Big question: is $\mathcal{P} \neq \mathcal{NP}$?
  - Most people believe so, but we have no proof
  - For all the following, "unless $\mathcal{P} = \mathcal{NP}$" is implied

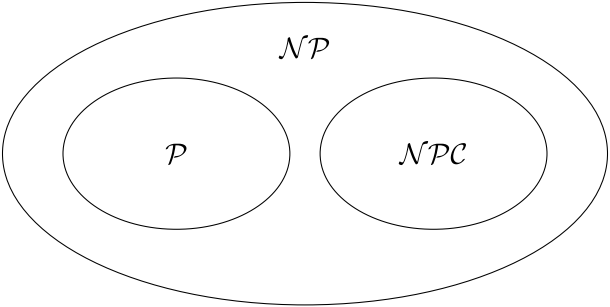
$\mathcal{NP}$-complete problems

- Some problems in $\mathcal{NP}$ are at least as difficult as all other problems in $\mathcal{NP}$
- They are called $\mathcal{NP}$-complete, and their set is $\mathcal{NPC}$
- Cook's theorem: The SAT problem is in $\mathcal{NPC}$
  - Satisfiability of a boolean conjunction of disjunctions
- How to prove that a problem, P, is $\mathcal{NP}$-complete:
  - Prove that $P \in \mathcal{NP}$ (typically easy)
  - Prove that P reduces to Q, where $Q \in \mathcal{NPC}$ (can be hard)
    - For an instance I_Q, construct in p-time an instance I_P
    - Prove that I_P has a solution if and only if I_Q has a solution
- By now, we know many problems in $\mathcal{NPC}$
- Goal: pick $Q \in \mathcal{NPC}$ so that the reduction is easy

$\mathcal{NP}$-complete problems

- Some problems in $\mathcal{NP}$ are at least as difficult as all other problems in $\mathcal{NP}$
- They are called $\mathcal{NP}$-complete, and their set is $\mathcal{NPC}$
- Cook's theorem: The SAT problem is in $\mathcal{NPC}$
  - Satisfiability of a boolean conjunction of disjunctions
- How to prove that a problem, P, is $\mathcal{NP}$-complete:
  - Prove that $P \in \mathcal{NP}$ (typically easy)
  - Prove that P reduces to Q, where $Q \in \mathcal{NPC}$ (can be hard)
    - For an instance I_Q, construct in p-time an instance I_P
    - Prove that I_P has a solution if and only if I_Q has a solution
- By now, we know many problems in $\mathcal{NPC}$
- Goal: pick $Q \in \mathcal{NPC}$ so that the reduction is easy

Well-known complexity classes



INDEP(2) is $\mathcal{NP}$-complete

- INDEP(2) (decision version) is in $\mathcal{NP}$
  - Certificate: for each a_i whether it is scheduled on P_1 or P_2
  - In linear time, compute the makespan on both processors, and compare to k to answer "Yes"
- Let us consider an instance of 2-PARTITION $\in \mathcal{NPC}$:
  - Given n integers x_i, is there a subset I of $\{1, \dots, n\}$ such that $\sum_{i \in I} x_i = \sum_{i \notin I} x_i$?
- Let us construct an instance of INDEP(2):
  - Let $k = \frac{1}{2} \sum x_i$, let $a_i = x_i$
- The proof is trivial
  - If k is non-integer, neither instance has a solution
  - Otherwise, each processor corresponds to one subset
- In fact, INDEP(2) is essentially identical to 2-PARTITION

So what?

- This $\mathcal{NP}$-completeness proof is probably the most trivial in the world ☺
- But now we are thus pretty sure that there is no p-time algorithm to solve $\mathrm{INDEP}(2)$
- What we look for now are approximation algorithms...

Approximation algorithms

- Consider an optimization problem
- A p-time algorithm is a λ-approximation algorithm if it returns a solution that is at most a factor λ from the optimal solution (the closer λ to 1, the better)
  - λ is called the approximation ratio
- Polynomial Time Approximation Scheme (PTAS): for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm (may be non-polynomial in $1/\epsilon$)
- Fully Polynomial Time Approximation Scheme (FPTAS): for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm polynomial in $1/\epsilon$
- Typical goal: find a FPTAS, if not find a PTAS, if not find a λ-approximation for a low value of λ

Approximation algorithms

- Consider an optimization problem
- A p-time algorithm is a λ-approximation algorithm if it returns a solution that is at most a factor λ from the optimal solution (the closer λ to 1, the better)
  - λ is called the approximation ratio
- Polynomial Time Approximation Scheme (PTAS): for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm (may be non-polynomial in $1/\epsilon$)
- Fully Polynomial Time Approximation Scheme (FPTAS): for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm polynomial in $1/\epsilon$
- Typical goal: find a FPTAS, if not find a PTAS, if not find a λ-approximation for a low value of λ

Approximation algorithms

- Consider an optimization problem
- A p-time algorithm is a λ-approximation algorithm if it returns a solution that is at most a factor λ from the optimal solution (the closer λ to 1, the better)
  - λ is called the approximation ratio
- Polynomial Time Approximation Scheme (PTAS): for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm (may be non-polynomial in $1/\epsilon$)
- Fully Polynomial Time Approximation Scheme (FPTAS): for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm polynomial in $1/\epsilon$
- Typical goal: find a FPTAS, if not find a PTAS, if not find a λ-approximation for a low value of λ

Approximation algorithms

- Consider an optimization problem
- A p-time algorithm is a λ-approximation algorithm if it returns a solution that is at most a factor λ from the optimal solution (the closer λ to 1, the better)
  - λ is called the approximation ratio
- Polynomial Time Approximation Scheme (PTAS): for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm (may be non-polynomial in $1/\epsilon$)
- Fully Polynomial Time Approximation Scheme (FPTAS): for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm polynomial in $1/\epsilon$
- Typical goal: find a FPTAS, if not find a PTAS, if not find a λ-approximation for a low value of λ

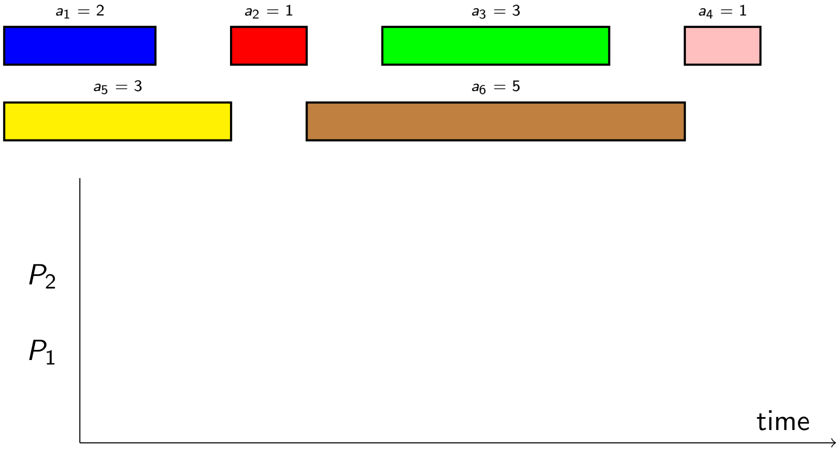
Greedy algorithms

- A greedy algorithm is one that builds a solution step-by-step, via local incremental decisions
- It turns out that several greedy scheduling algorithms are approximation algorithms
  - Informally, they are not as "bad" as one may think
- Two natural greedy algorithms for INDEP(2):
  - **greedy-online**: take the tasks in arbitrary order and assign each task to the least loaded processor
    - We don't know which tasks are coming
  - **greedy-offline**: sort the tasks by decreasing a_i, and assign each task in that order to the least loaded processor
    - We know all the tasks ahead of time

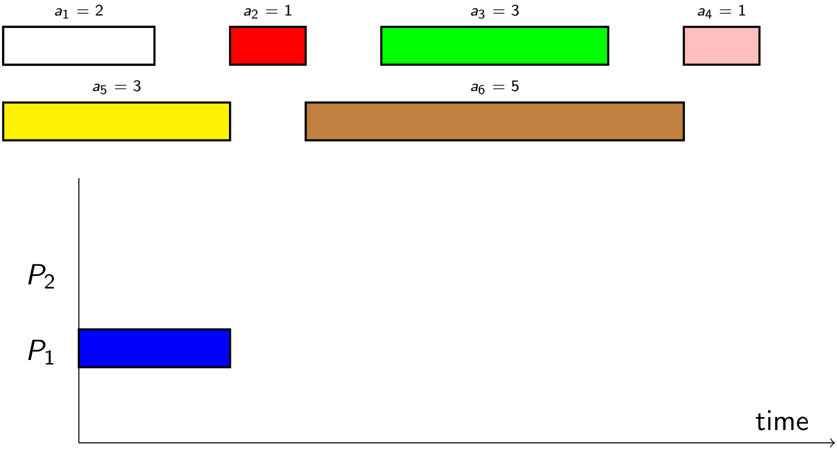
Greedy algorithms

- A greedy algorithm is one that builds a solution step-by-step, via local incremental decisions
- It turns out that several greedy scheduling algorithms are approximation algorithms
  - Informally, they are not as "bad" as one may think
- Two natural greedy algorithms for INDEP(2):
  - **greedy-online**: take the tasks in arbitrary order and assign each task to the least loaded processor
    - We don't know which tasks are coming
  - **greedy-offline**: sort the tasks by decreasing a_i, and assign each task in that order to the least loaded processor
    - We know all the tasks ahead of time

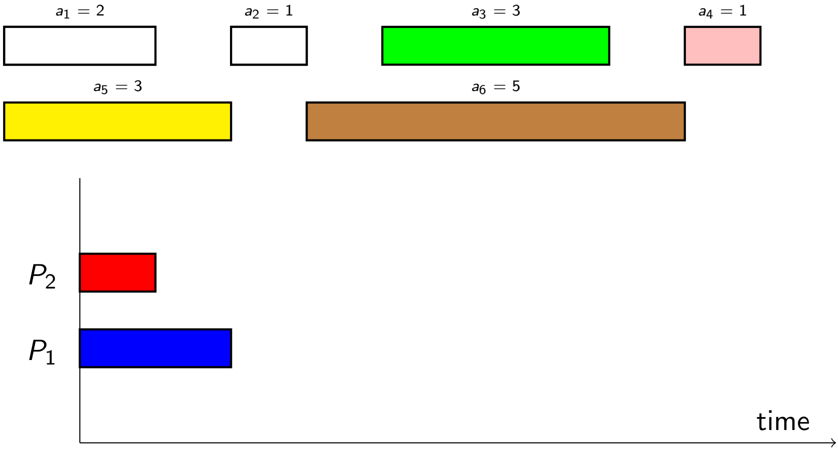
Example with 6 tasks: Online



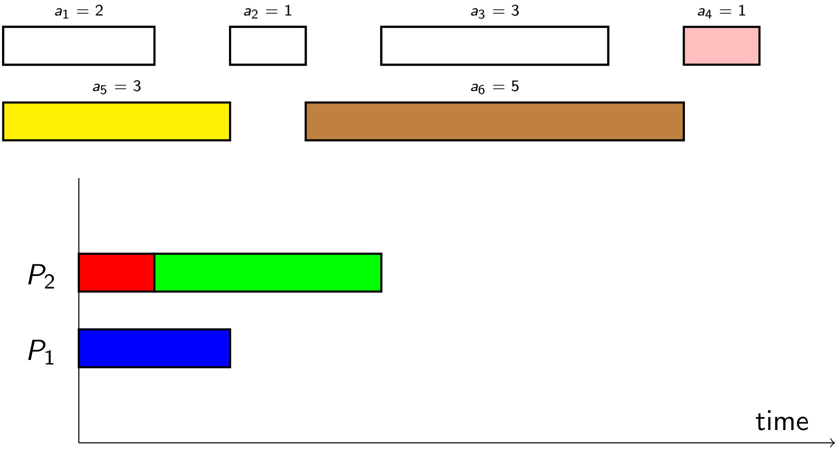
Example with 6 tasks: Online



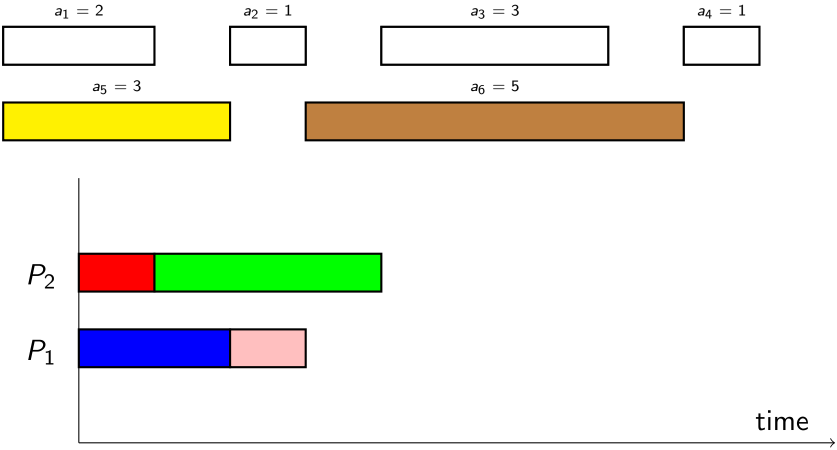
Example with 6 tasks: Online



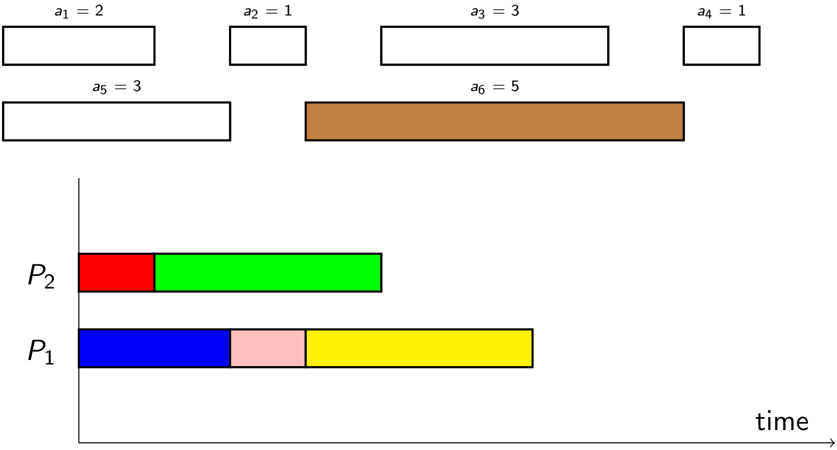
Example with 6 tasks: Online



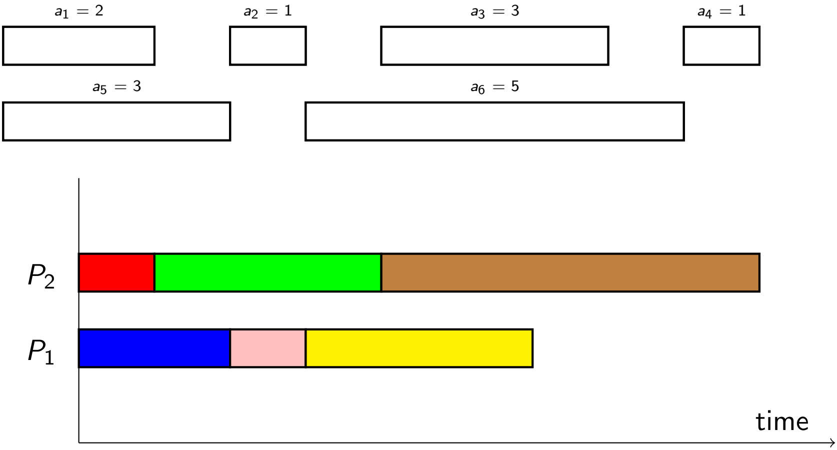
Example with 6 tasks: Online



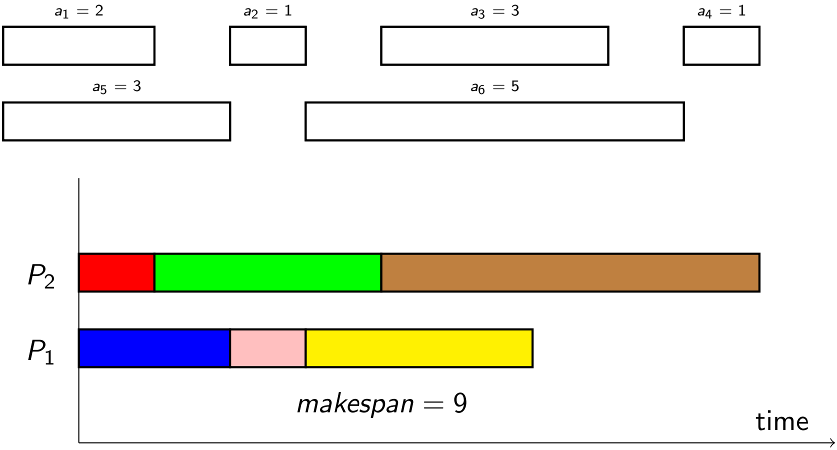
Example with 6 tasks: Online



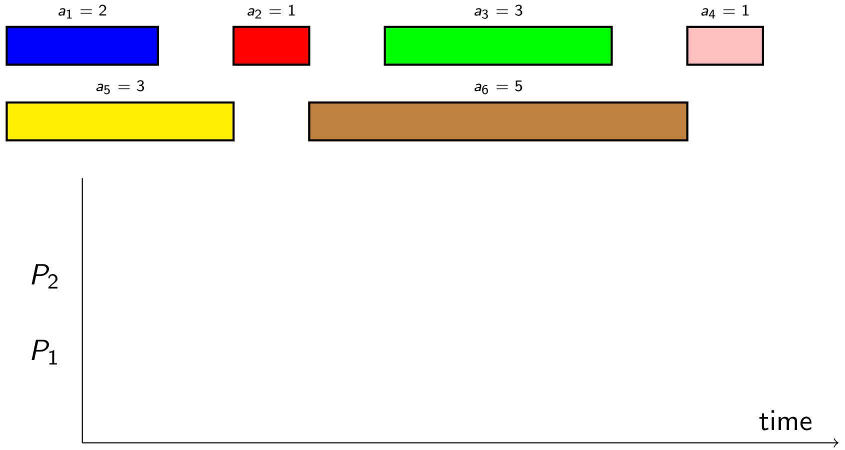
Example with 6 tasks: Online



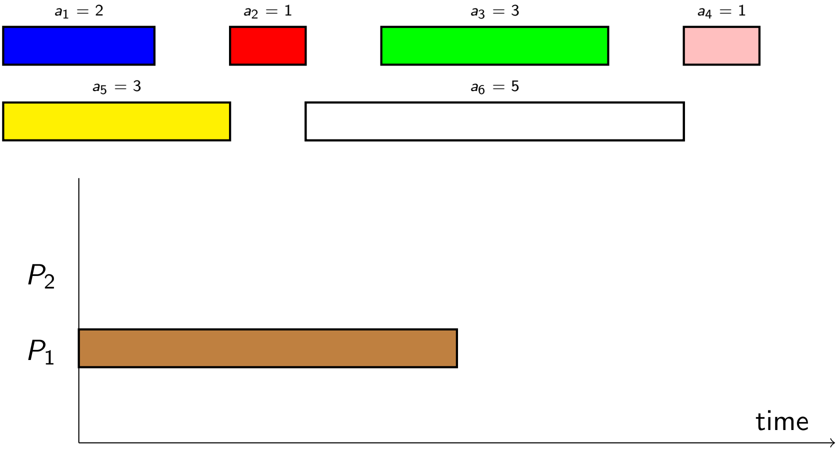
Example with 6 tasks: Online



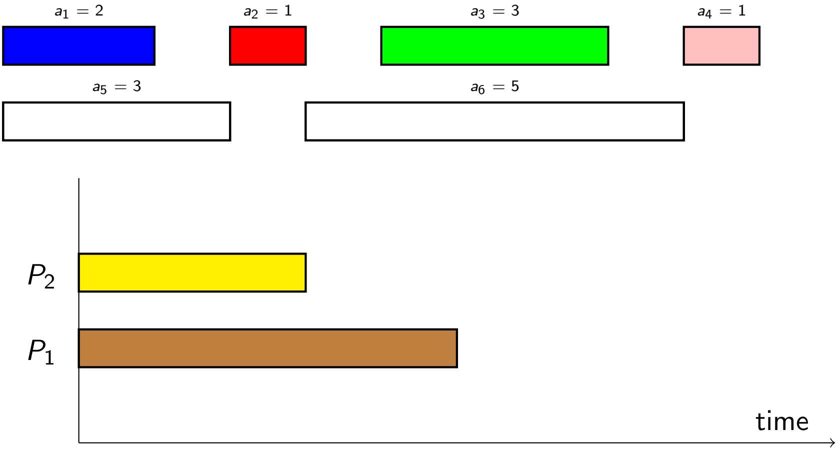
Example with 6 tasks: Offline



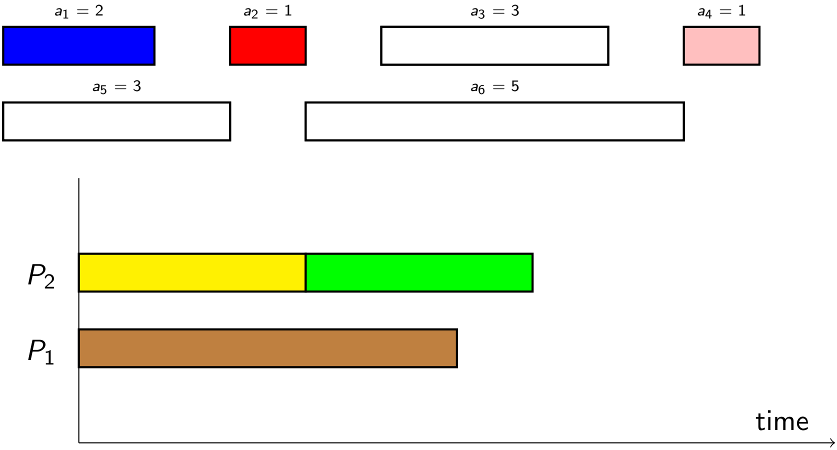
Example with 6 tasks: Offline



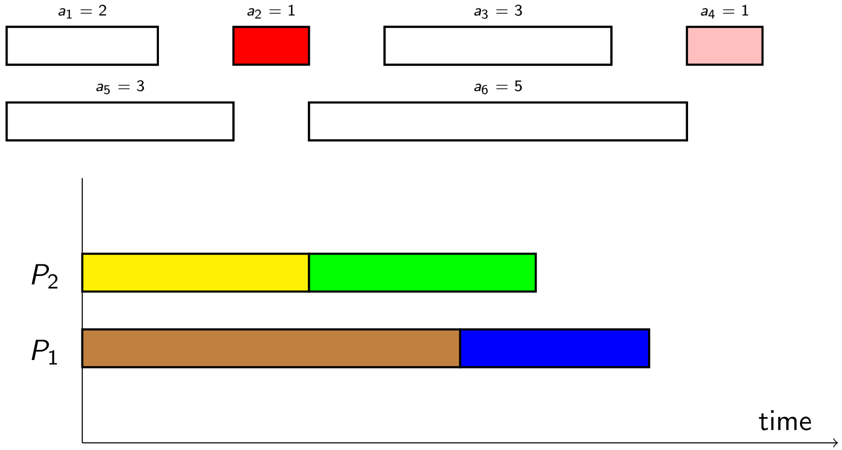
Example with 6 tasks: Offline



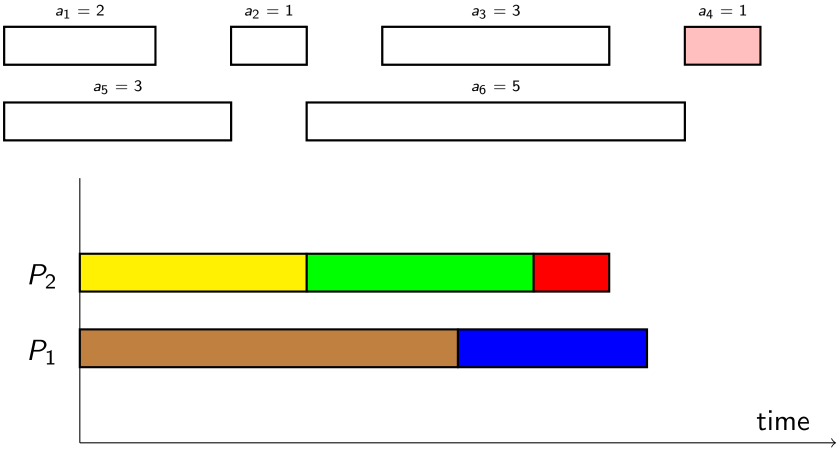
Example with 6 tasks: Offline



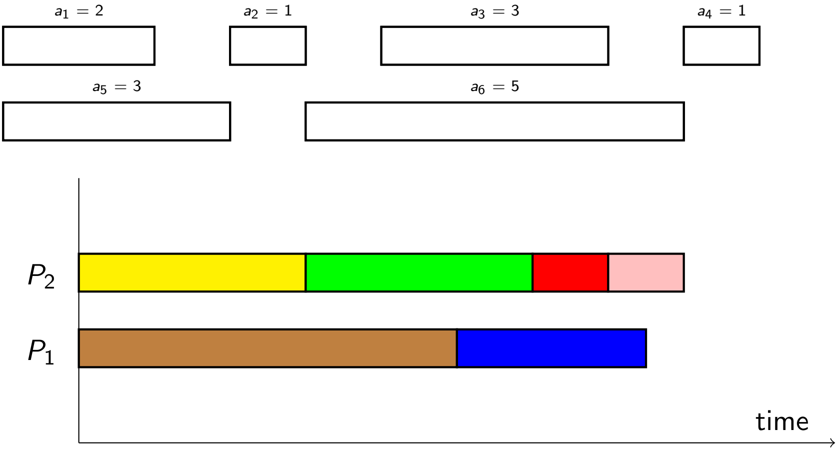
Example with 6 tasks: Offline



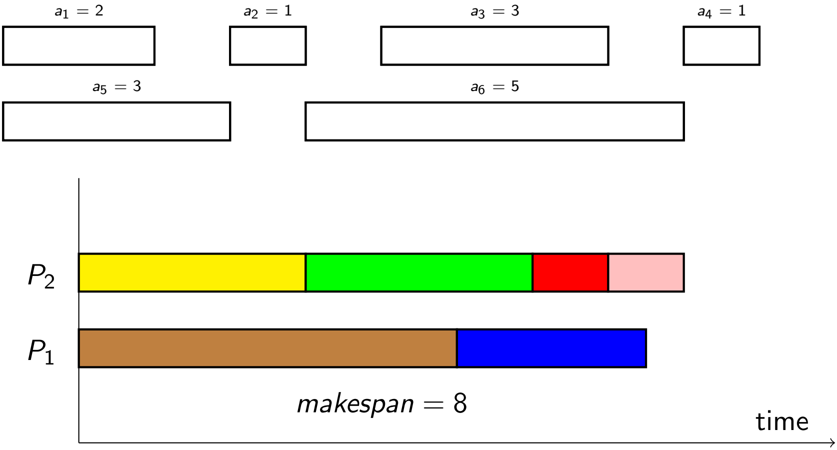
Example with 6 tasks: Offline



Example with 6 tasks: Offline



Example with 6 tasks: Offline



Approximation results for INDEP(2)

Theorem

Greedy-online is a $\frac{3}{2}$-approximation

Theorem

Greedy-offline is a $\frac{7}{6}$-approximation

... and the bounds are tight:

- Greedy-online:
  - a_i's $= \{1,1,2\}$
  - $M_{greedy} = 3$; $M_{opt} = 2$
  - $ratio = \frac{3}{2}$
- Greedy-offline:
  - a_i's $= \{3, 3, 2, 2, 2\}$
  - $M_{greedy} = 7$; $M_{opt} = 6$
  - $ratio = \frac{7}{6}$

PTAS and FPTAS for INDEP(2)

Theorem

There is a PTAS ($(1+\epsilon)$-approximation) for INDEP(2)

- Proof sketch:
  - Classify tasks as either "small" or "large"
    - Very common technique
  - Replace all small tasks by same-size tasks
  - Compute an optimal schedule of the modified problem in p-time (not polynomial in $1/\epsilon$)
  - Show that the cost is $\leq 1+\epsilon$ away from the optimal cost
  - The proof is a couple of pages, but not terribly difficult

Theorem

There is an FPTAS ($(1+\epsilon)$-approx pol. in $1/\epsilon$) for INDEP(2)

We know a lot about INDEP(2)

- INDEP(2) is NP-complete
- We have simple greedy algorithms with guarantees on result quality
- We have a simple PTAS
- We even have a (less simple) FPTAS
- INDEP(2) is basically "solved"

- Sadly, not many scheduling problems are this well understood...

Outline

INDEP(P) is much harder

- INDEP(P) is $\mathcal{NP}$-complete by trivial reduction to 3-PARTITION:
  - Give $3n$ integers $a_1, \dots, a_{3n}$ and an integer B, can we partition the $3n$ integers into n sets, each of sum B? (assuming that $\sum_i a_i = nB$)
- 3-PARTITION is $\mathcal{NP}$-complete "in the strong sense", unlike 2-PARTITION
  - Even when encoding the input in unary (i.e., no logarithmic numbers of bits), one cannot find an algorithm polynomial in the size of the input!
  - Informally, a problem is $\mathcal{NP}$-complete "in the weak sense" if it is hard only if the numbers in the input are unbounded
- INDEP(P) is thus fundamentally harder than INDEP(2)

Approximation algorithm for INDEP(P)

Theorem

Greedy-online is a $(2 - \frac{1}{p})$-approximation

- Proof (usual reasoning):
  - Let $M_{greedy} = \max_{1 \leq i \leq p} M_i$, and j be such that $M_j = M_{greedy}$
  - Let T_k be the last task assigned to processor P_j
  - $\forall i, \quad M_i \geq M_j - a_k$ (greedy algorithm)
  - $S = \sum_i^p M_i = M_j + \sum_{i \neq j} M_i \geq M_j + (p-1)(M_j - a_k) = pM_j - (p-1)a_k$
  - Therefore, $M_{greedy} = M_j \leq \frac{S}{p} + (1 - \frac{1}{p})a_k$
  - But $M_{opt} \geq a_k$ and $M_{opt} \geq S/p$
  - So $M_{greedy} \leq M_{opt} + (1 - \frac{1}{p})M_{opt} \quad \square$
- This ratio is "tight" (e.g., an instance with $p(p-1)$ tasks of size 1 and one task of size p has this ratio)

Approximation algorithm for INDEP(P)

Theorem

Greedy-offline is a $(\frac{4}{3} - \frac{1}{3p})$-approximation

- The proof is more involved, but follows the spirit of the proof for INDEP(2) EXERCISE
- This ratio is tight
- There is a PTAS for INDEP(P), a $(1+\epsilon)$-approximation (massively exponential in $1/\epsilon$)
- Unlike for INDEP(2), there cannot exist any FPTAS, unless $\mathcal{P} = \mathcal{NP}$

Approximation algorithm for INDEP(P)

Theorem

Greedy-offline is a $(\frac{4}{3} - \frac{1}{3p})$-approximation

- The proof is more involved, but follows the spirit of the proof for INDEP(2) EXERCISE
- This ratio is tight

- There is a PTAS for INDEP(P), a $(1+\epsilon)$-approximation (massively exponential in $1/\epsilon$)
- Unlike for INDEP(2), there cannot exist any FPTAS, unless $\mathcal{P} = \mathcal{NP}$

Outline

Where do task graphs come from?

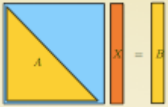
Solving $A.x = B$ where A is lower triangular matrix

for $i = 1$ **to** n **do**

  $\boxed{\text{Task } T_{i,i}:}$ $x(i) \leftarrow b(i)/a(i,i)$

  for $j = i+1$ **to** n **do**

    $\boxed{\text{Task } T_{i,j}:}$ $b(j) \leftarrow b(j) - a(j,i) \times x(i)$



For a given value $1 \leqslant i \leqslant n$, all tasks $T_{i,*}$ are computations done during the i^{th} iteration of the outer loop.

$<_{seq}$ is the sequential order :

$$T_{1,1} <_{seq} T_{1,2} <_{seq} T_{1,3} <_{seq} \ldots <_{seq} T_{1,n} <_{seq} T_{2,2} <_{seq} T_{2,3} <_{seq} \ldots <_{seq} T_{n,n}.$$

Independence

However, some independent tasks could be executed in parallel.
Independent tasks are the ones whose execution order can be changed without modifying the result of the program.
Two independent tasks may read the value but never write to the same memory location.

For a given task T, $In(T)$ denotes the set of input variables and $Out(T)$ the set of output variables.
In the previous example, we have :

$$\begin{cases} In(T_{i,i}) = \{b(i), a(i,i)\} \\ Out(T_{i,i}) = \{x(i)\} \text{ and} \end{cases}$$
$$\begin{cases} In(T_{i,j}) = \{b(j), a(j,i), x(i)\} \\ Out(T_{i,j}) = \{b(j)\} \text{ for } j > i. \end{cases}$$

for $i = 1$ **to** n **do**
  Task $T_{i,i}$: $x(i) \leftarrow b(i)/a(i,i)$
  for $j = i+1$ **to** n **do**
    Task $T_{i,j}$: $b(j) \leftarrow b(j) - a(j,i) \times x(i)$

Bernstein conditions

Definition.

Two tasks T and T' are not independent ($T \perp T'$) whenever they share a written variable:

$$T \perp T' \Leftrightarrow \begin{cases} & In(T) \cap Out(T') \neq \emptyset \\ \text{or} & Out(T) \cap In(T') \neq \emptyset \\ \text{or} & Out(T) \cap Out(T') \neq \emptyset \end{cases} .$$

Those conditions are known as Bernstein's conditions [Bernstein66].

We can check that:

- $Out(T_{1,1}) \cap In(T_{1,2}) = \{x(1)\}$ $\leadsto T_{1,1} \perp T_{1,2}$.
- $Out(T_{1,3}) \cap Out(T_{2,3}) = \{b(3)\}$ $\leadsto T_{1,3} \perp T_{2,3}$.

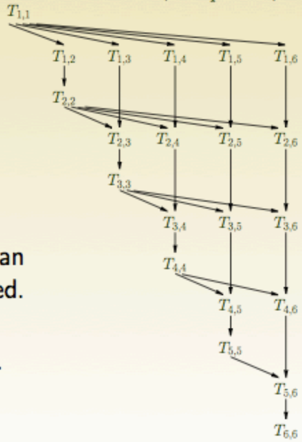
for $i = 1$ **to** n **do**
  Task $T_{i,i}$: $x(i) \leftarrow b(i)/a(i,i)$
  for $j = i+1$ **to** n **do**
    Task $T_{i,j}$: $b(j) \leftarrow b(j) - a(j,i) \times x(i)$

Precedences

If $T \perp T'$, then they should be ordered with the sequential execution order. $T \prec T'$ if $T \perp T'$ and $T <_{seq} T'$.

More precisely $\prec$ is defined as the transitive closure of $(<_{seq} \cap \perp)$.

for $i = 1$ **to** n **do**
  Task $T_{i,i}$: $x(i) \leftarrow b(i)/a(i,i)$
  for $j = i+1$ **to** n **do**
    Task $T_{i,j}$: $b(j) \leftarrow b(j) - a(j,i) \times x(i)$

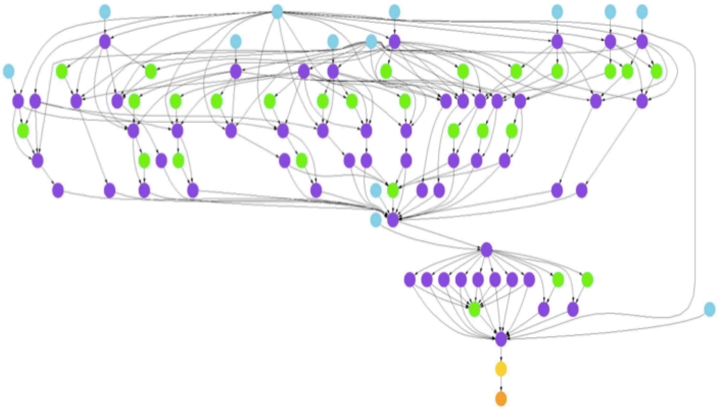


A dependence graph G is used.

$(e : T \rightarrow T') \in G$ means that T' can start only if T has already been finished. T is a predecessor of T'.

Transitivity arcs are generally omitted.

Scientific workflows



See Pegasus at `pegasus.isi.edu`

Task system

Definition: Task system.

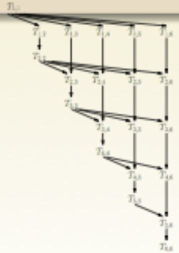
A task system is an directed graph $G = (V, E, w)$ where :

- V is the set of tasks (V is finite)
- E represent the dependence constraints:

$$e = (u, v) \in E \text{ iff } u \prec v$$

- $w : V \to \mathbb{N}^*$ is a time function that give the weight (or duration) of each task.

We could set $w(T_{i,j}) = 1$ but also decide that performing a division is more expensive than a multiplication followed by an addition.



Schedule and allocation

Definition: Schedule.

A schedule of a task system $G = (V, E, w)$ is a time function $\sigma : V \rightarrow \mathbb{N}^*$ such that:

$$\forall (u, v) \in E,\ \sigma(u) + w(u) \leqslant \sigma(v)$$

Let us denote by $\mathcal{P} = \{P_1, \ldots, P_p\}$ the set of processors.

Definition: Allocation.

An allocation of a task system $G = (V, E, w)$ is a function $\pi : V \rightarrow \mathcal{P}$ such that:

$$\pi(T) = \pi(T') \Leftrightarrow \begin{cases} \sigma(T) + w(T) \leqslant \sigma(T') \text{ or} \\ \sigma(T') + w(T') \leqslant \sigma(T) \end{cases}$$

Basic feasibility condition

Theorem 1.

Let $G = (V, E, w)$ be a task system. There exists a valid schedule of G iff G has no cycle.

Sketch of the proof.

$\boxed{\Rightarrow}$ Assume that G has a cycle $v_1 \to v_2 \to \dots \to v_k \to v_1$. Then $v_1 \prec v_1$ and a valid schedule σ should hold $\sigma(v_1) + w(v_1) \leqslant \sigma(v_1)$ true, which is impossible because $w(v_1) > 0$.

$\boxed{\Leftarrow}$ If G is acyclic, then some tasks have no predecessor. They can be scheduled first.
More precisely, we sort topologically the vertexes and schedule them one after the other on the same processor. Dependences are then fulfilled. □

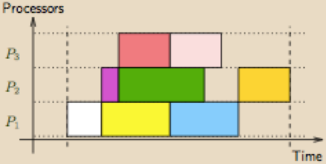
Therefore all task systems we will be considering in the following are Directed Acyclic Graphs.

Makespan

Definition: Makespan.

The makespan of a schedule is the total execution time :

$$MS(\sigma) = \max_{v \in V}\{\sigma(v) + w(v)\} - \min_{v \in V}\{\sigma(v)\} .$$



The makespan is also often referred as $C_{\max}$ in the literature.

$$C_{\max} = \max_{v \in V} C_v$$

- $Pb(p)$: find a schedule with the smallest possible makespan, using at most p processors. $MS_{opt}(p)$ denotes the optimal makespan using only p processors.
- $Pb(\infty)$: find a schedule with the smallest makespan when the number of processors that can be used is not bounded. We note $MS_{opt}(\infty)$ the corresponding makespan.

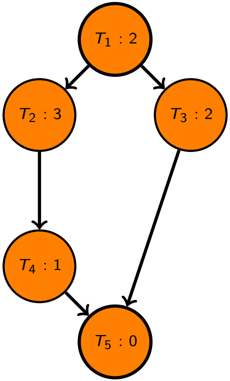
$Pb(\infty)$ is in $\mathcal{P}$

- If one has an infinite number of processors, obtaining the optimal schedule is actually very simple:
  - Assign each task to a different processor
  - Start a task whenever it is "ready", i.e., when its parent tasks have completed)
- Sketch of a proof
  - No (unnecessary) idle time occurs between tasks on any path
  - Consider the tasks on the critical path
  - The last tasks of the DAG is on the critical path
    - If not, add a "dummy" task
  - The makespan is equal to the length of the critical path
  - Therefore it's optimal
  - Another "obvious" proof that may look complex in its full formal version

EXERCISE

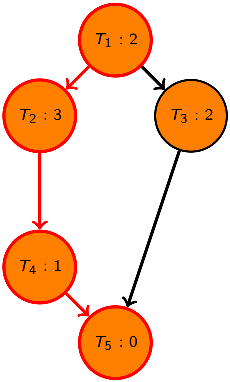
Critical path (1/2)

- In practice tasks often have dependencies
- A general model of computation is the Acyclic Directed Graph (DAG), $G = (V, E)$
- Each task has a weight (i.e., execution time in seconds), parents, and children
- The first task is the source, the last task the sink
- Topological (partial) order of the tasks



Critical path (2/2)

- Assume that the DAG executes on p processors
- The longest path (largest weight) is called the critical path
- The length of the critical path (CP) is a lower bound on M_{opt}, regardless of the number of processors
- In this example, the CP length is 6 (the other path has length 4)



Complexity

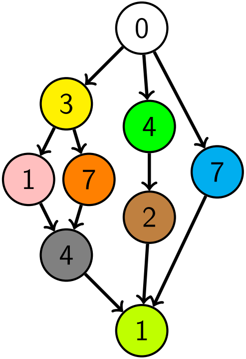
- Unsurprisingly, DAG scheduling $Pb(p)$ is $\mathcal{NP}$-complete
  - Independent tasks is a special case of DAG scheduling
- Typical greedy algorithm skeleton:
  - Maintain a list of ready tasks (with cleared dependencies)
  - Greedily assign a ready task to an available processor as early as possible (don't leave a processor idle unnecessarily)
  - Update the list of ready tasks
  - Repeat until all tasks have been scheduled
- This is called List Scheduling
- Many list scheduling algorithms are possible
  - Depending on how to select the ready task to schedule next

Complexity

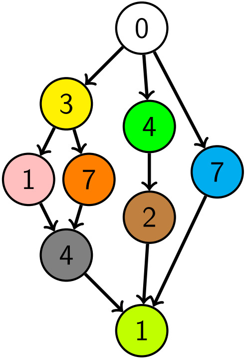
- Unsurprisingly, DAG scheduling $Pb(p)$ is $\mathcal{NP}$-complete
  - Independent tasks is a special case of DAG scheduling
- Typical greedy algorithm skeleton:
  - Maintain a list of ready tasks (with cleared dependencies)
  - Greedily assign a ready task to an available processor as early as possible (don't leave a processor idle unnecessarily)
  - Update the list of ready tasks
  - Repeat until all tasks have been scheduled
- This is called List Scheduling
- Many list scheduling algorithms are possible
  - Depending on how to select the ready task to schedule next

List scheduling example

3 Processors

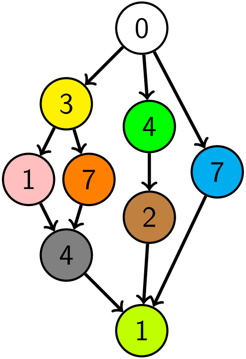


List scheduling example



3 Processors

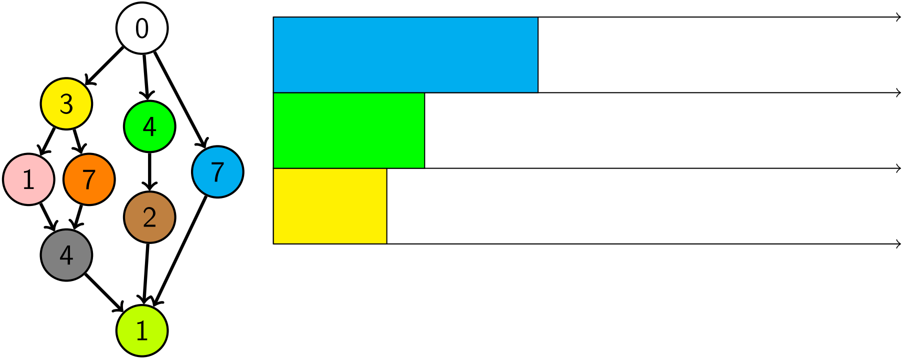
List scheduling example



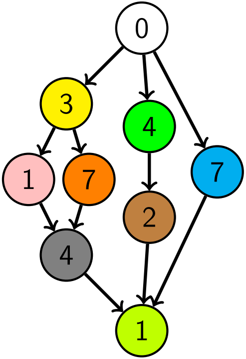
3 Processors

List scheduling example

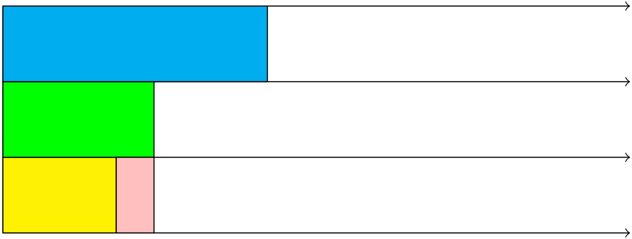
3 Processors



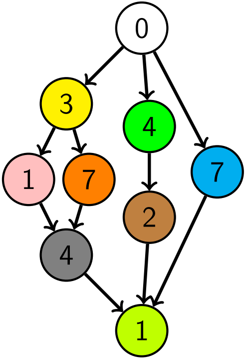
List scheduling example



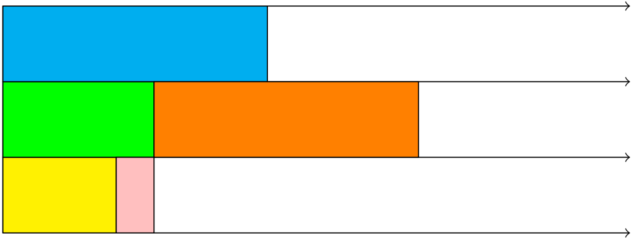
3 Processors



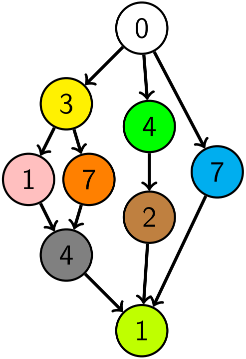
List scheduling example



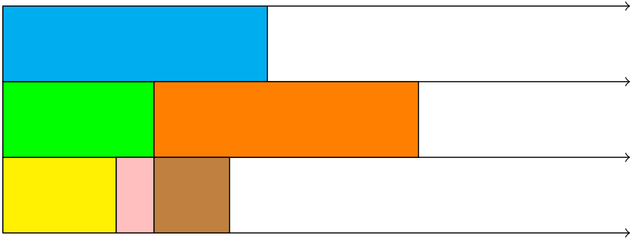
3 Processors



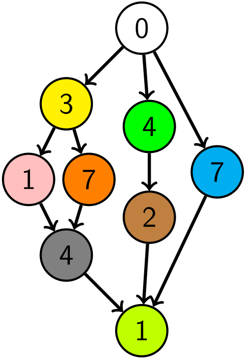
List scheduling example

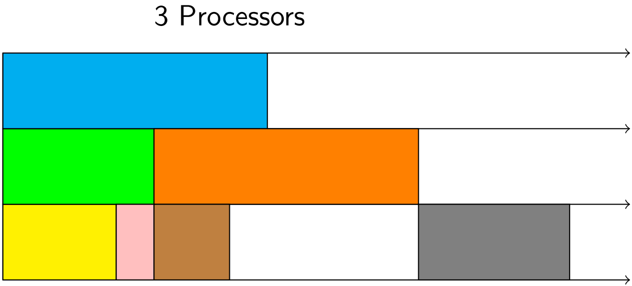


3 Processors

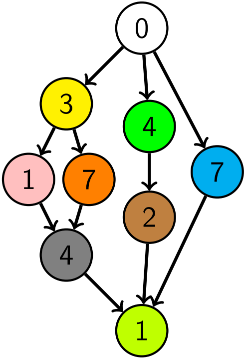


List scheduling example

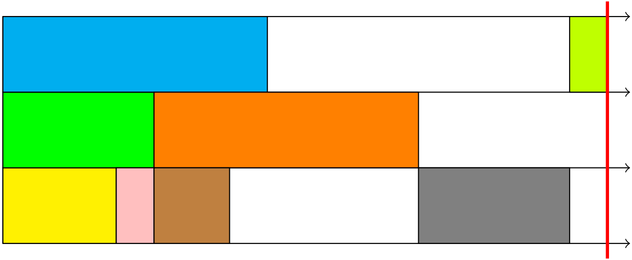




List scheduling example



3 Processors



Makespan = 16; CP Length = 15

Idle Time = 1+5+5+8 = 19

List scheduling

Theorem (fundamental)

List scheduling is a $(2 - \frac{1}{p})$-approximation

- Doesn't matter how the next ready task is selected
- Let's prove this theorem informally
  - Really simple proof if one doesn't use the typical notations for schedules
  - I never use these notations in public ☺

List scheduling

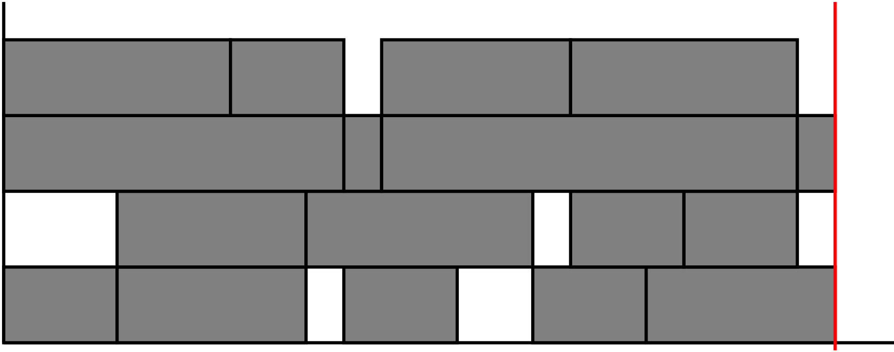
Theorem (fundamental)

List scheduling is a $(2 - \frac{1}{p})$*-approximation*

- Doesn't matter how the next ready task is selected
- Let's prove this theorem informally
  - Really simple proof if one doesn't use the typical notations for schedules
  - I never use these notations in public ☺

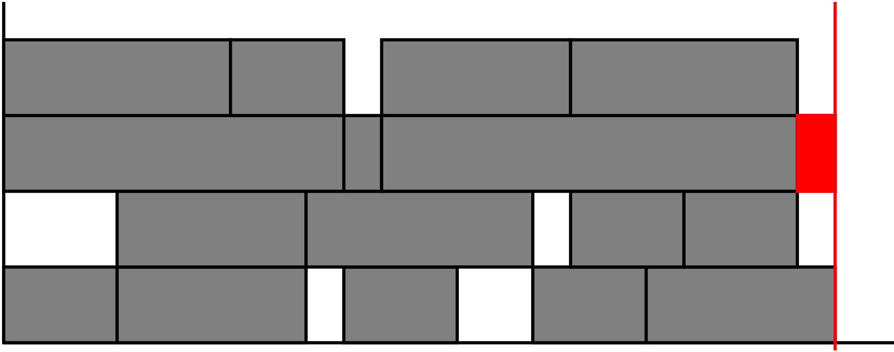
Approximation ratio

Let's consider a list-scheduling schedule



Approximation ratio

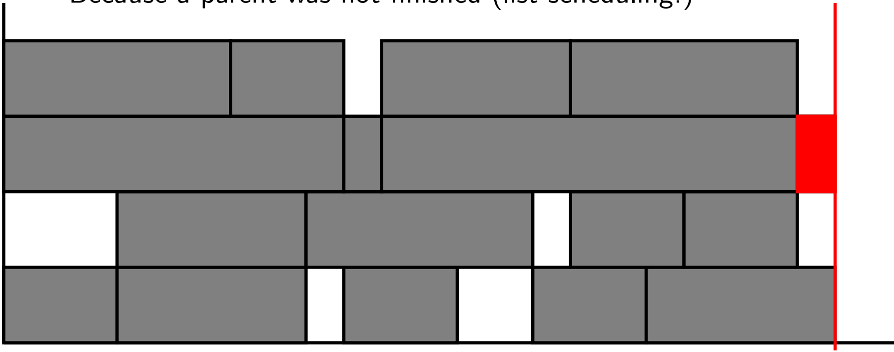
Let's consider one of the tasks that finishes last



Approximation ratio

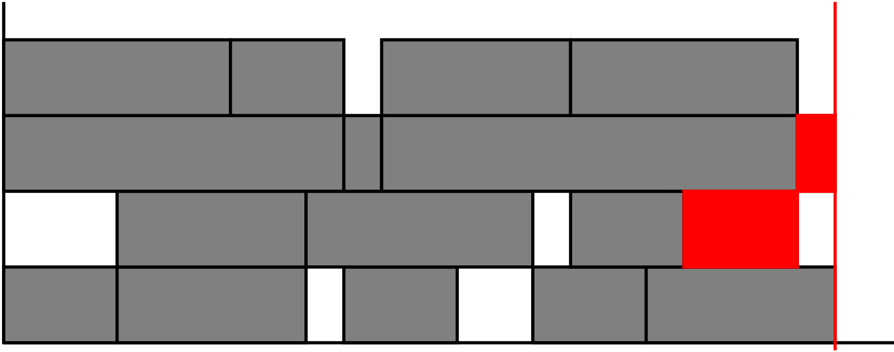
Why didn't this task run during an earlier idle period?

Because a parent was not finished (list scheduling!)



Approximation ratio

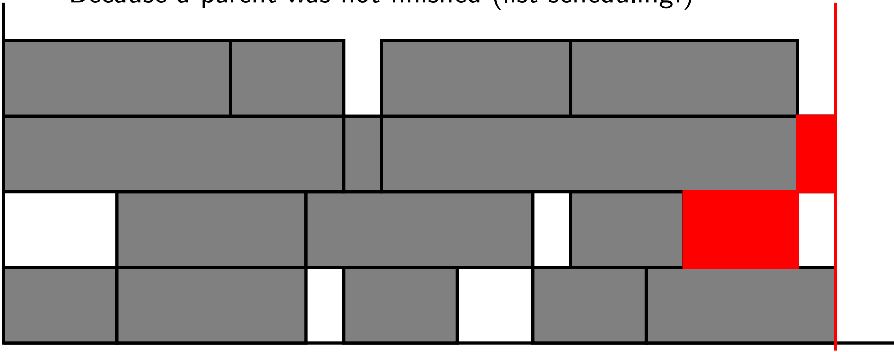
Let's look at a parent



Approximation ratio

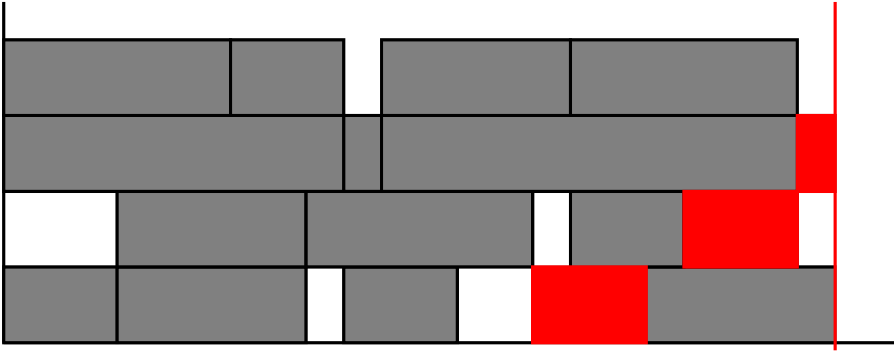
Why didn't this task run during an earlier idle period?

Because a parent was not finished (list scheduling!)



Approximation ratio

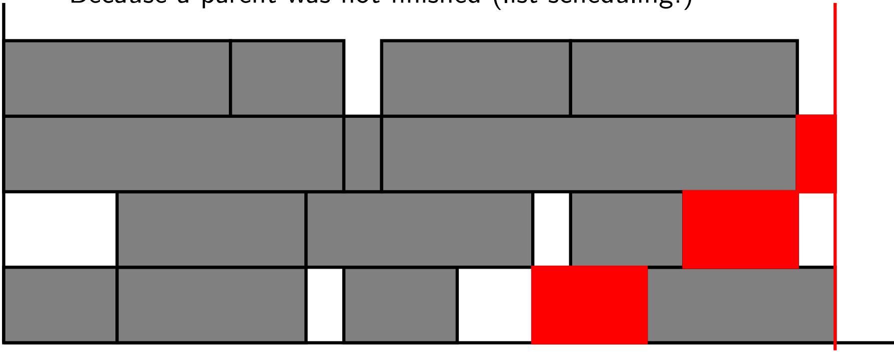
Let's look at a parent



Approximation ratio

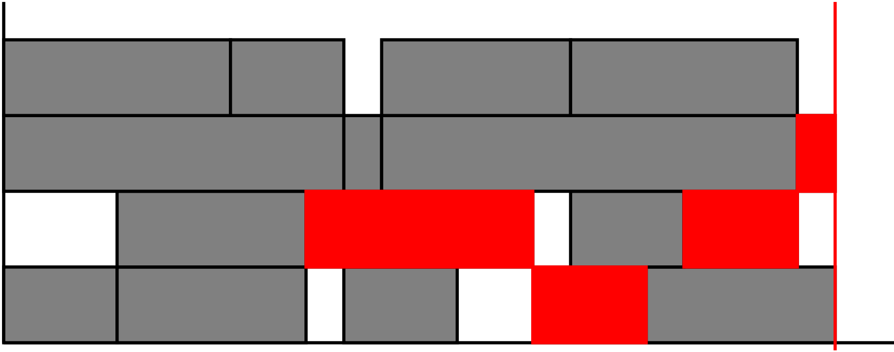
Why didn't this task run during an earlier idle period?

Because a parent was not finished (list scheduling!)



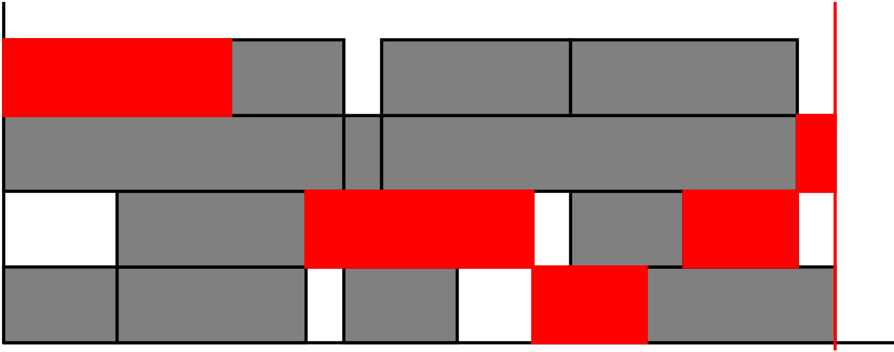
Approximation ratio

Let's look at a parent



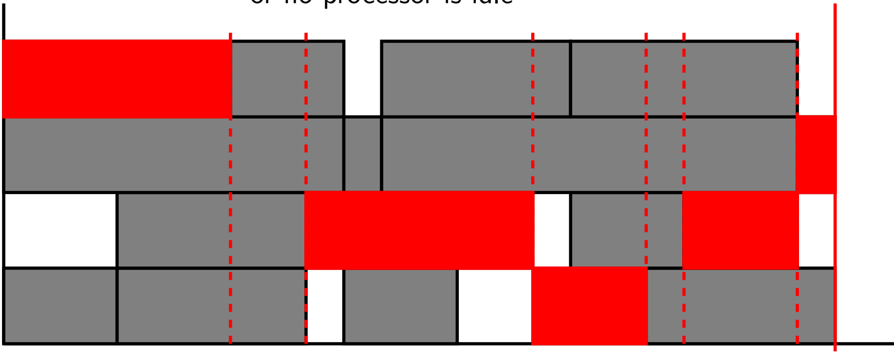
Approximation ratio

And so on...



Approximation ratio

At any point in time either a task on the red path is running or no processor is idle



Approximation ratio

- Let L be the length of the red path (in seconds), p the number of processors, I the total idle time, M the makespan, and S the sum of all task weights
- $I \leq (p-1)L$
  - processors can be idle only when a red task is running
- $L \leq M_{opt}$
  - The optimal makespan is longer than any path in the DAG
- $M_{opt} \geq S/p$
  - S/p is the makespan with zero idle time
- $p \times M = I + S$
  - rectangle's area = white boxes + non-white boxes

$$\Rightarrow \quad p \times M \leq (p-1)M_{opt} + pM_{opt} \quad \Rightarrow \quad M \leq (2 - \frac{1}{p})M_{opt} \qquad \square$$

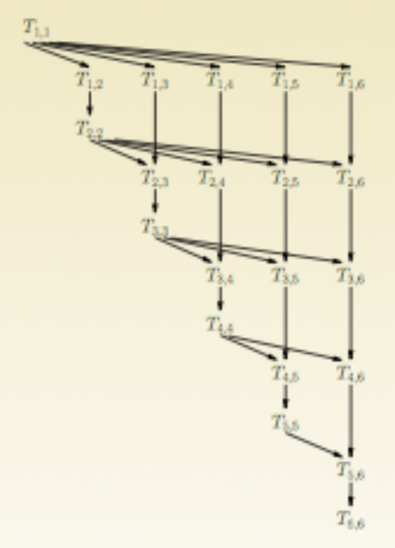
Good list scheduling?

- All list scheduling algorithms thus have the same approximation ratio
- But there are many options for list scheduling
  - Many ways of sorting the ready tasks...
- In practice, some may be better than others
- One well-known option, Critical path scheduling

Critical path scheduling

- When given a set of ready tasks, which one do we pick to schedule?
- Idea: pick a task on the CP
  - If we prioritize tasks on the CP, then the CP length is reduced
  - The CP length is a lower bound on the makespan
  - So intuitively it's good for it to be low
- For each (ready) task, compute its <u>bottom level</u>, the length of the path from the task to the sink
- Pick the task with the <u>largest</u> bottom level

Your turn to work EXERCISE



Outline

Graham's notation

- There are SO many variations on the scheduling problem that Graham has proposed a standard notation: $\alpha|\beta|\gamma$
  - *alpha*: processors
  - *beta*: tasks
  - *gamma*: objective function
- Let's see some examples for each

α: processors

- 1: one processor
- Pn: n identical processors (if n not fixed, not given)
- Qn: n uniform processors (if n not fixed, not given)
  - Each processor has a (different) compute speed
- Rn: n unrelated processors (if n not fixed, not given)
  - Each processor has a (different) compute speed for each (different) task (e.g., P_1 can be faster than P_2 for T_1, but slower for T_2)

β: tasks

- r_j: tasks have release dates
- d_j: tasks have deadlines
- $p_j = x$: all tasks have weight x
- *prec*: general precedence constraints (DAG)
- *tree*: tree precedence constraints
- *chains*: chains precedence constraints (multiple independent paths)
- *pmtn*: tasks can be preempted and restarted (on other processors)
  - Makes scheduling easier, and can often be done in practice
- ...

γ: objective function

- C_{max}: makespan
- $\sum C_i$: mean flow-time (completion time minus release date if any)
- $\sum w_i C_i$: average weighted flow-time
- L_{max}: maximum lateness ($\max(0, C_i - d_i)$)
- ...

Examples of scheduling problems

- The classification is not perfect and variations among authors are common
- Some examples:
  - $P2||C_{max}$, which we called $\mathrm{INDEP}(2)$
  - $P||C_{max}$, which we called $\mathrm{INDEP}(\mathrm{P})$
  - $P|prec|C_{max}$, which we called DAG scheduling
  - $R2|chains|\sum C_i$
    - Two unrelated processors, chains, minimize sum-flow
  - $P|r_j; p_j \in \{1, 2\}; d_j; pmtn|L_{max}$
    - Identical processors, tasks with release dates and deadlines, task weights either 1 or 2, preemption, minimize maximum lateness

Where to find known results

- Luckily, the body of knowledge is well-documented (and Graham's notation widely used)
- Several books on scheduling that list known results
  - Handbook of Scheduling, Leung and Anderson
  - Scheduling Algorithms, Brucker
  - Scheduling: Theory, Algorithms, and Systems, Pinedo
  - ...
- Many published survey articles

Example list of known results

- Excerpt from Scheduling Algorithm, P. Brucker

	$P2 \parallel C_{max}$	Lenstra et al. [155]
*	$P \parallel C_{max}$	Garey & Johnson [98]
*	$P \mid p_i = 1; intree; r_i \mid C_{max}$	Brucker et al. [35]
*	$P \mid p_i = 1; prec \mid C_{max}$	Ullman [203]
*	$P2 \mid chains \mid C_{max}$	Du et al. [86]
*	$Q \mid p_i = 1; chains \mid C_{max}$	Kubiak [129]
*	$P \mid p_i = 1; outtree \mid L_{max}$	Brucker et al. [35]
*	$P \mid p_i = 1; intree; r_i \mid \sum C_i$	Lenstra [150]
*	$P \mid p_i = 1; prec \mid \sum C_i$	Lenstra & Rinnooy Kan [152]
*	$P2 \mid chains \mid \sum C_i$	Du et al. [86]
*	$P2 \mid r_i \mid \sum C_i$	Single-machine problem
	$P2 \parallel \sum w_i C_i$	Bruno et al. [58]
*	$P \parallel \sum w_i C_i$	Lenstra [150]
*	$P2 \mid p_i = 1; chains \mid \sum w_i C_i$	Timkovsky [201]
*	$P2 \mid p_i = 1; chains \mid \sum U_i$	Single-machine problem
*	$P2 \mid p_i = 1; chains \mid \sum T_i$	Single-machine problem

Table 5.3: $\mathcal{NP}$-hard parallel machine problems without preemption.

Outline

Conclusion

- Scheduling problems are diverse and often difficult
- Relevant theoretical questions:
  - Is it in $\mathcal{P}$?
  - Is it $\mathcal{NP}$-complete?
    - Are there approximation algorithms?
    - Are there PTAS or FPTAS?
    - Are there are least decent non-guaranteed heuristics?
- Luckily, scheduling problems have been studied a lot
- Come up with the Graham notation for your problem and check what is known about it!

Outline

What about communications?

- If the processors are on a network (as opposed to in a shared memory machine), then we need to account for the cost of communication of data among tasks
- Each edge in the DAG now has a weight
  - e.g., data transfer time on a reference network
- Common Assumption: If two tasks are scheduled on the same processor the edge weight is ignored
  - Or at least it's made very small
- There is now a notion of network topology as well, which may be regular or irregular
- Accounting for communication costs makes things much more complicated

Model for communications

A very simple model (things are already complicated enough): the macro-data flow model. If there is some data-dependence between T and T', the communication cost is

$$c(T,T') = \begin{cases} 0 & \text{if alloc}(T) = \text{alloc}(T') \\ c(T,T') & \text{otherwise} \end{cases}$$

Definition.

A DAG with communication cost (say cDAG) is a directed acyclic graph $G = (V, E, w, c)$ where vertexes represent tasks and edges represent dependence constraints. $w : V \to \mathbb{N}^*$ is the computation time function and $c : E \to \mathbb{N}^*$ is the communication time function. Any valid schedule has to respect the dependence constraints.

$$\forall e = (v, v') \in E,$$

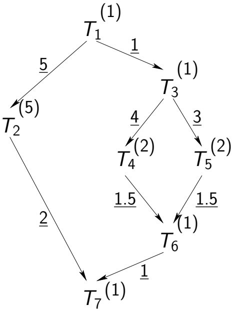
$$\begin{cases} \sigma(v) + w(v) \leqslant \sigma(v') & \text{if alloc}(v) = \text{alloc}(v') \\ \sigma(v) + w(v) + c(v; v') \leqslant \sigma(v') & \text{otherwise.} \end{cases}$$

Complexity with communications

Even $Pb(\infty)$ is NP-complete !!! (EXERCISE)

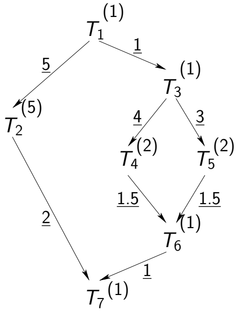
You constantly have to figure out whether you should use more processors (but then pay more for communications) or not. Finding the good trade-off is a real challenge.

Example



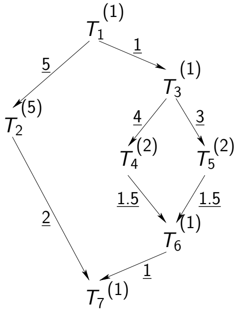
- All tasks on same processor: $MS_{opt}(1) =$?
- One task per processor: $MS_{opt}(\infty) =$?
- Optimal makespan = ?

Example



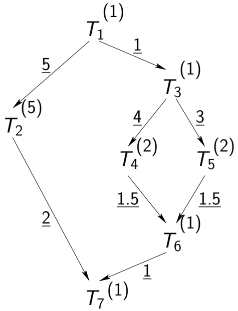
- All tasks on same processor: $MS_{opt}(1) =$ 13
- One task per processor: $MS_{opt}(\infty) =$
- Optimal makespan $=$

Example



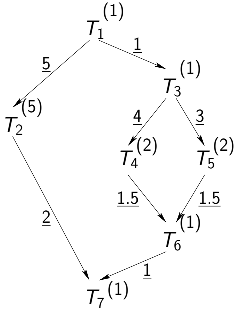
- All tasks on same processor: $MS_{opt}(1) =$ 13
- One task per processor: $MS_{opt}(\infty) =$ 14
- Optimal makespan =

Example

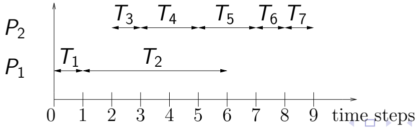


- All tasks on same processor: $MS_{opt}(1) =$ 13
- One task per processor: $MS_{opt}(\infty) =$ 14
- Optimal makespan = 9

Example



- All tasks on same processor: $MS_{opt}(1) =$ 13
- One task per processor: $MS_{opt}(\infty) =$ 14
- Optimal makespan = 9

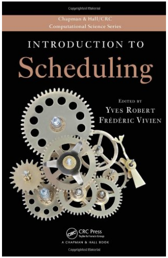


Outline

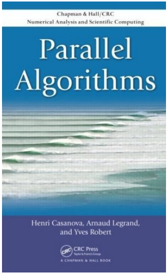
$R||C_{\max}$

- Linear programming formulation for the $R||C_{\max}$ problem, and integrality gap.
- It is possible to derive a 2-approximation algorithm from this linear programming formulation, see the paper "Approximation algorithms for scheduling unrelated parallel machines" by Lenstra, Shmoys and Tardos for all details (but it is difficult)

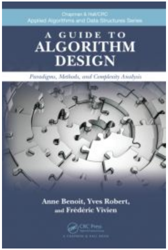
Sources and acknowledgments



Y. Robert
F. Vivien



H. Casanova
A. Legrand
Y. Robert



A. Benoit
Y. Robert
F. Vivien

Thanks to Loris Marchal and Yves Robert for some of these slides ☺