Introduction to Scheduling

ICS632: Principles of High-Performance Computing

Henri Casanova (henric@hawaii.edu)

Fall 2015

Foreword

- This set of lecture notes will be a bit theoretical-ish
- We’ll refer to simple computational complexity concepts
- We’ll have a few hand-wavy proofs

- We could have a whole semester on scheduling, this will only scratch the surface

Outline

What is scheduling?

- 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 computation to processors and communications to network links so as to minimize application execution time**

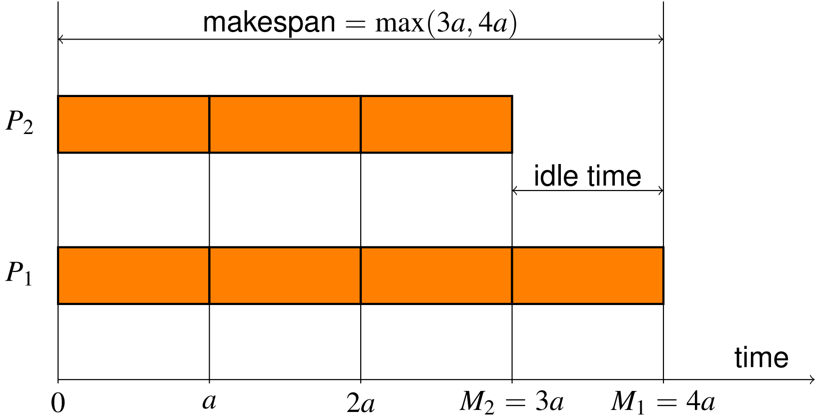
A simple scheduling problem

- 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. n compute tasks
     - Each task can run on either processor in a seconds
     - Tasks are *independent*: can be computed in any order
  3. Objective: minimize $\max(M_1, M_2)$ (makespan)
     - 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
- We have a trivial linear-time algorithm
  - For each task pick one of the two processors by comparing the index of the task with $n/2$
- In fact was have already seen an optimal algorithms for a more complex situation in which we have p *heterogeneous* processors

Gantt chart for INDEP(2) with 7 identical tasks



Non-identical tasks

- Task T_i, $i = 1, \ldots, n$ takes time $a_i \geq 0$
- There is no p-time algorithm to solve INDEP(2) (unless $\mathcal{P} = \mathcal{NP}$)
- 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 makespan bound to answer “Yes”
- Consider an instance of 2-PARTITION ($\mathcal{NP}$-complete):
  - Given n integers x_i, is there a subset I of $\{1, \ldots, n\}$ 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
- INDEP(2) is 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 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's 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 is $1/\epsilon$)
- *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's 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 is $1/\epsilon$)
- *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's 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 is $1/\epsilon$)
- *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's 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 is $1/\epsilon$)
- *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

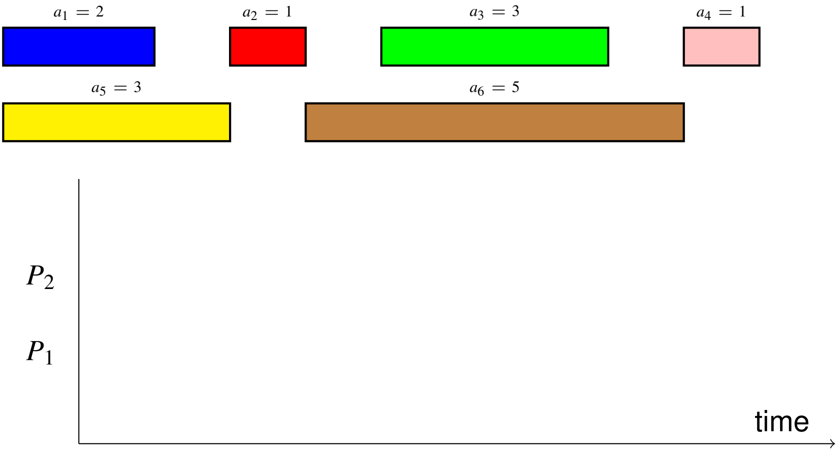
Greedy algorithms

- 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're 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
    - As if 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

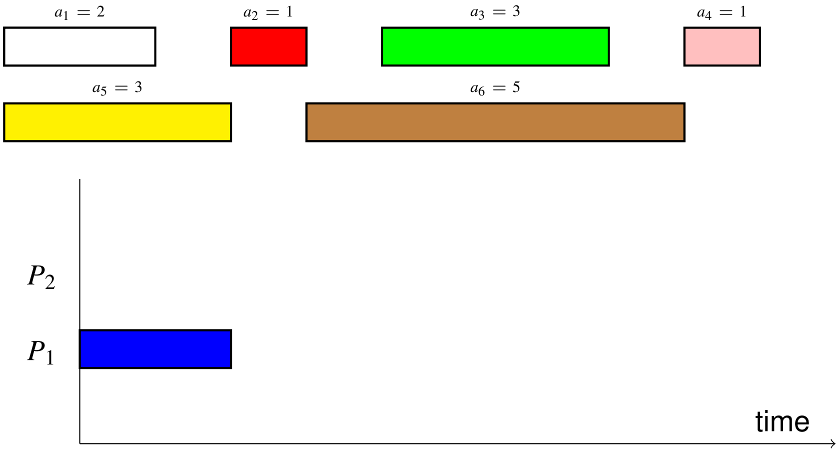
Greedy algorithms

- 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're 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
    - As if 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

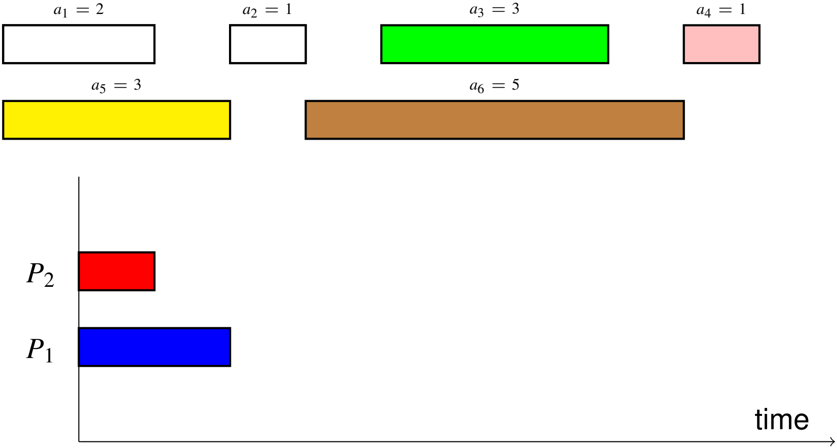
Example with 6 tasks: Online



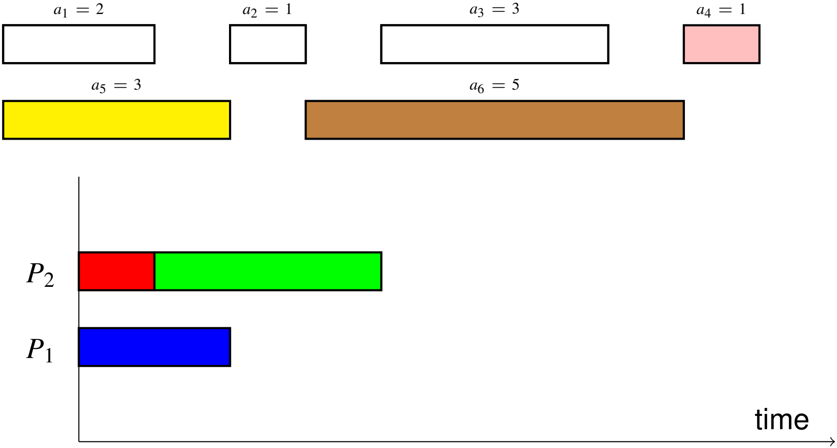
Example with 6 tasks: Online



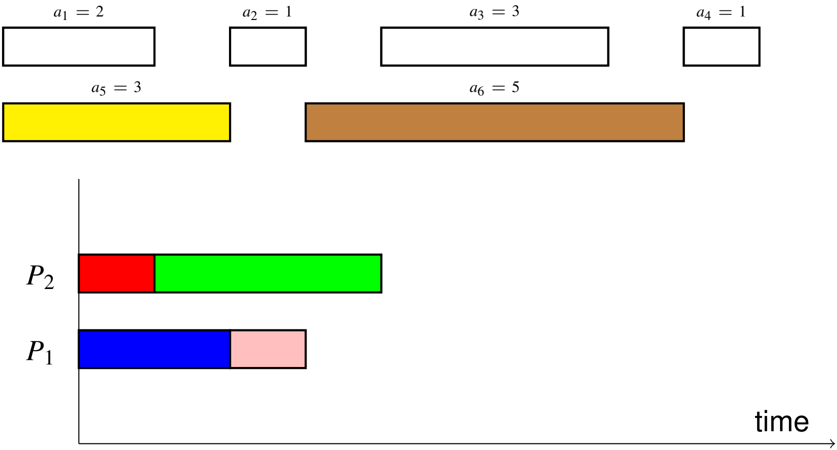
Example with 6 tasks: Online



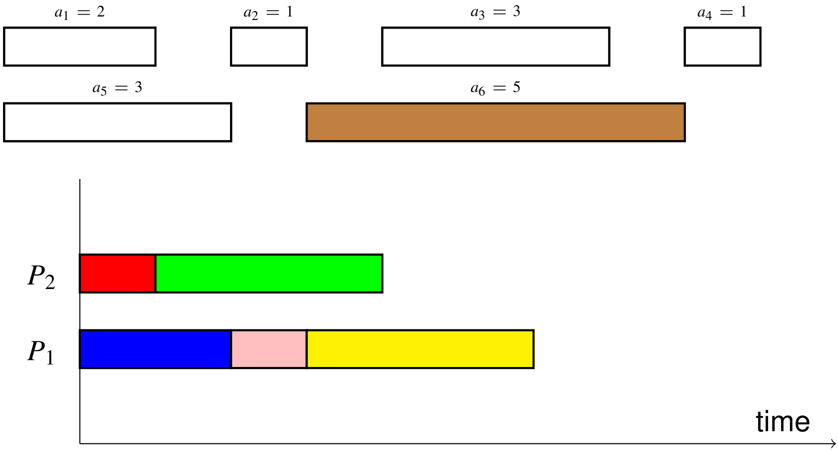
Example with 6 tasks: Online



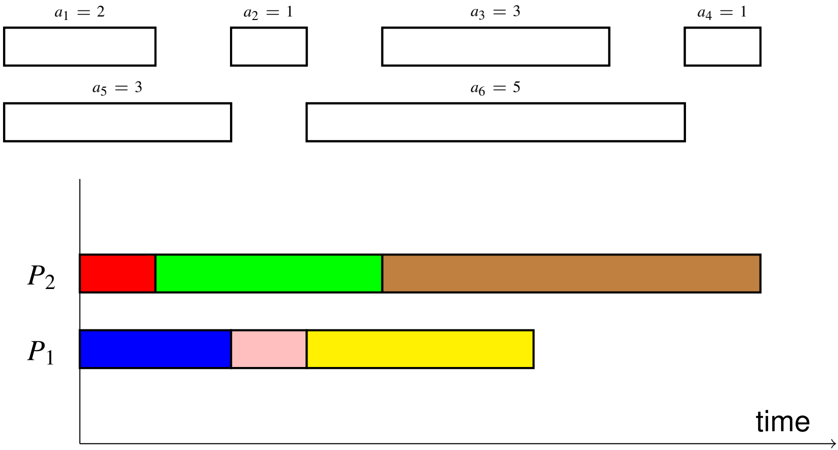
Example with 6 tasks: Online



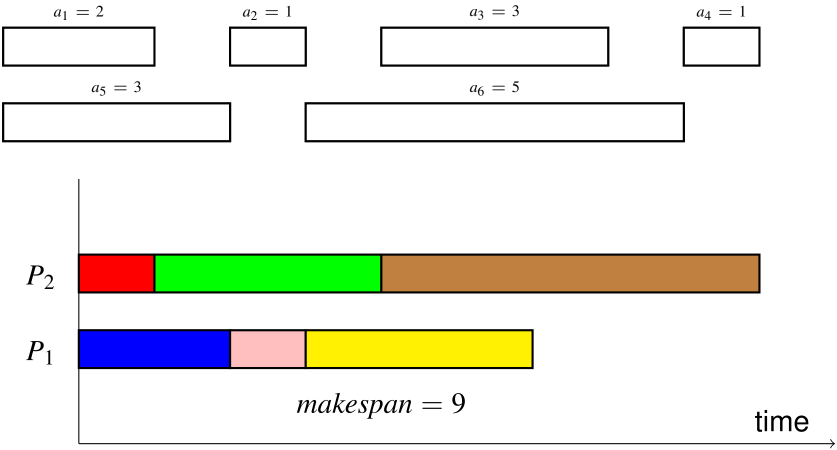
Example with 6 tasks: Online



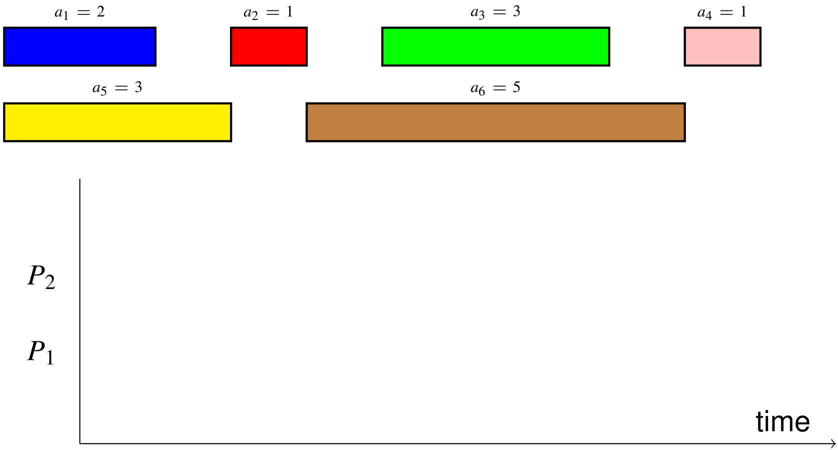
Example with 6 tasks: Online



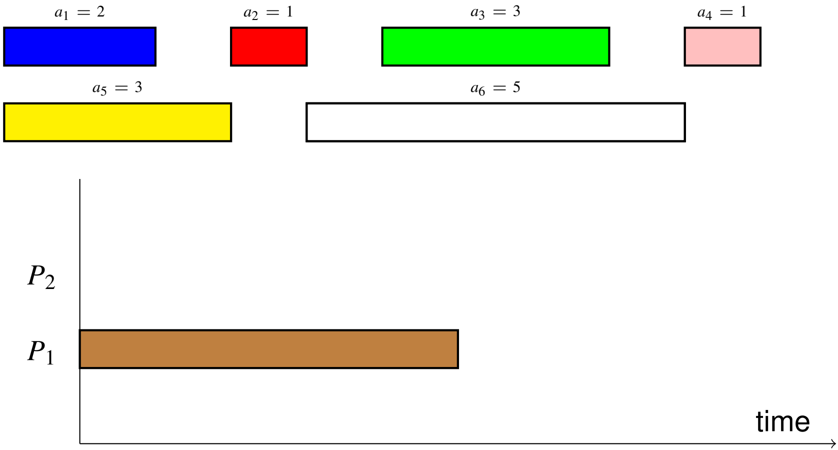
Example with 6 tasks: Online



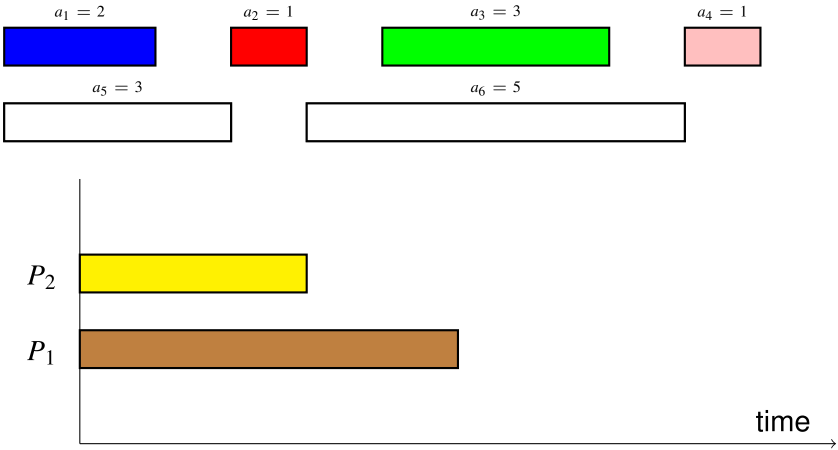
Example with 6 tasks: Offline



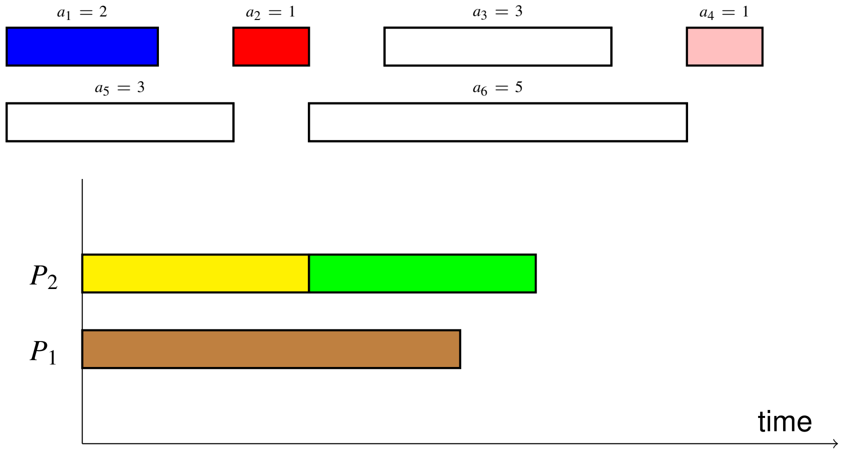
Example with 6 tasks: Offline



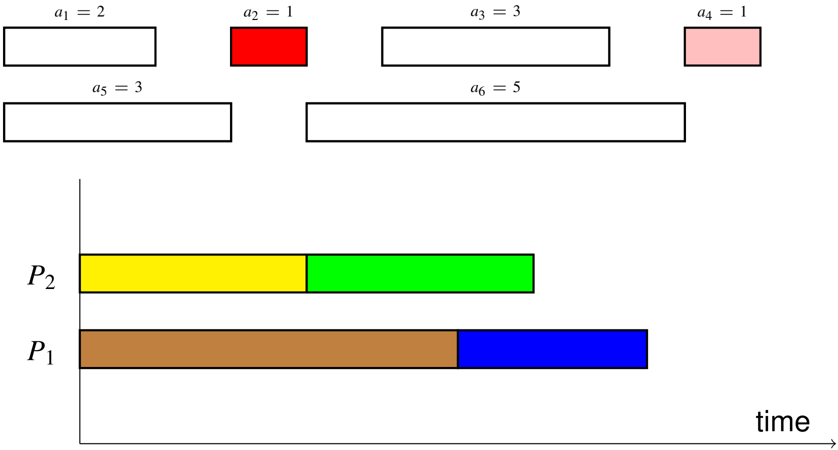
Example with 6 tasks: Offline



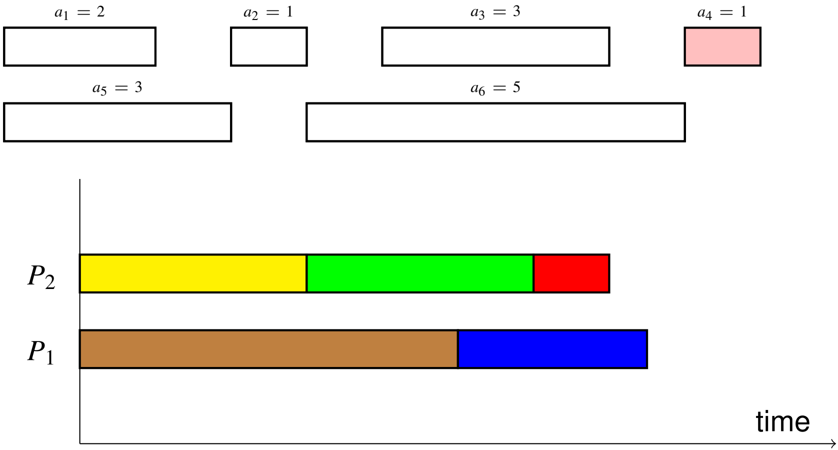
Example with 6 tasks: Offline



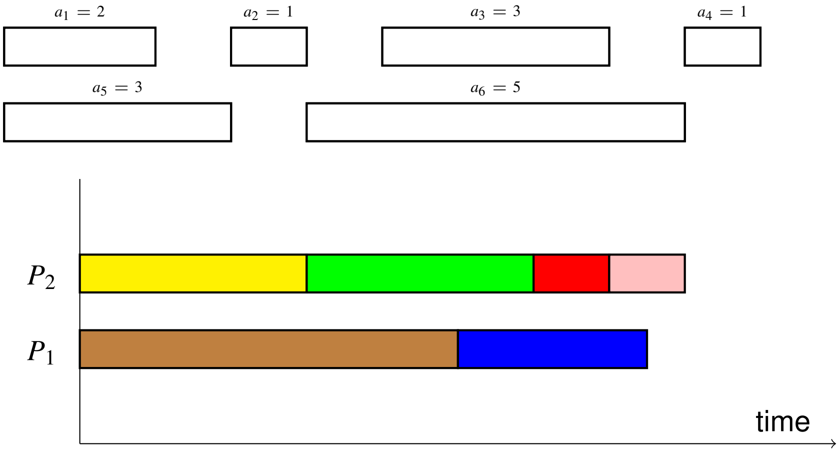
Example with 6 tasks: Offline



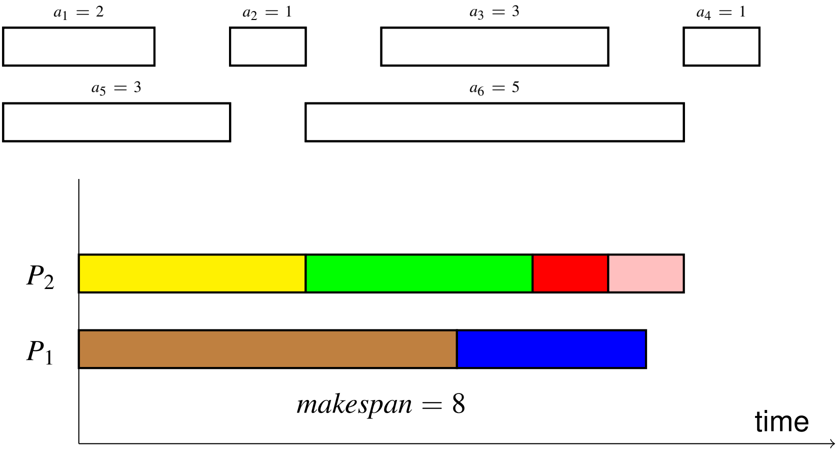
Example with 6 tasks: Offline



Example with 6 tasks: Offline



Example with 6 tasks: Offline



Greedy-online for INDEP(2)

Theorem

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

- Proof:
  - P_i finishes computing at time M_i (M stands for makespan)
  - Let us assume $M_1 \geq M_2$ ($M_{greedy} = M_1$)
  - Let T_j the last task to execute on P_1
  - Since the greedy algorithm put T_j on P_1, then $M_1 - a_j \leq M_2$
  - We have $M_1 + M_2 = \sum_i a_i = S$
  - $M_{greedy} = M_1 = \frac{1}{2}(M_1 + (M_1 - a_j) + a_j) \leq \frac{1}{2}(M_1 + M_2 + a_j) = \frac{1}{2}(S + a_j)$
  - but $M_{opt} \geq S/2$ (ideal lower bound on optimal)
  - and $M_{opt} \geq a_j$ (at least one task is executed)
  - Therefore: $M_{greedy} \leq \frac{1}{2}(2M_{opt} + M_{opt}) = \frac{3}{2}M_{opt}$ □

Greedy-online for INDEP(2)

Theorem

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

- Proof:
  - P_i finishes computing at time M_i (M stands for makespan)
  - Let us assume $M_1 \geq M_2$ ($M_{greedy} = M_1$)
  - Let T_j the last task to execute on P_1
  - Since the greedy algorithm put T_j on P_1, then $M_1 - a_j \leq M_2$
  - We have $M_1 + M_2 = \sum_i a_i = S$
  - $M_{greedy} = M_1 = \frac{1}{2}(M_1 + (M_1 - a_j) + a_j) \leq \frac{1}{2}(M_1 + M_2 + a_j) = \frac{1}{2}(S + a_j)$
  - but $M_{opt} \geq S/2$ (ideal lower bound on optimal)
  - and $M_{opt} \geq a_j$ (at least one task is executed)
  - Therefore: $M_{greedy} \leq \frac{1}{2}(2M_{opt} + M_{opt}) = \frac{3}{2}M_{opt}$ $\square$

Greedy-online for INDEP(2)

Theorem

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

- Proof:
  - P_i finishes computing at time M_i (M stands for makespan)
  - Let us assume $M_1 \geq M_2$ ($M_{greedy} = M_1$)
  - Let T_j the last task to execute on P_1
  - Since the greedy algorithm put T_j on P_1, then $M_1 - a_j \leq M_2$
  - We have $M_1 + M_2 = \sum_i a_i = S$
  - $M_{greedy} = M_1 = \frac{1}{2}(M_1 + (M_1 - a_j) + a_j) \leq \frac{1}{2}(M_1 + M_2 + a_j) = \frac{1}{2}(S + a_j)$
  - but $M_{opt} \geq S/2$ (ideal lower bound on optimal)
  - and $M_{opt} \geq a_j$ (at least one task is executed)
  - Therefore: $M_{greedy} \leq \frac{1}{2}(2M_{opt} + M_{opt}) = \frac{3}{2}M_{opt}$ $\square$

Greedy-offline for INDEP(2)

Theorem

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

- Proof:
  - If $a_j \leq \frac{1}{3}M_{opt}$, the previous proof can be used
    - $M_{greedy} \leq \frac{1}{2}(2M_{opt} + \frac{1}{3}M_{opt}) = \frac{7}{6}M_{opt}$
  - If $a_j > \frac{1}{3}M_{opt}$, then $j \leq 4$
    - if T_j was the 5th task, then, due to the task ordering, there would be 5 tasks with $a_i > \frac{1}{3}M_{opt}$
    - There would be at least 3 tasks on the same processor in the optimal schedule
    - Therefore $M_{opt} > 3 \times \frac{1}{3}M_{opt}$, a contradiction
  - One can check all possible scenarios for 4 tasks and show optimality □

Greedy-offline for INDEP(2)

Theorem

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

- Proof:
  - If $a_j \leq \frac{1}{3}M_{opt}$, the previous proof can be used
    - $M_{greedy} \leq \frac{1}{2}(2M_{opt} + \frac{1}{3}M_{opt}) = \frac{7}{6}M_{opt}$
  - If $a_j > \frac{1}{3}M_{opt}$, then $j \leq 4$
    - if T_j was the 5th task, then, due to the task ordering, there would be 5 tasks with $a_i > \frac{1}{3}M_{opt}$
    - There would be at least 3 tasks on the same processor in the optimal schedule
    - Therefore $M_{opt} > 3 \times \frac{1}{3}M_{opt}$, a contradiction
  - One can check all possible scenarios for 4 tasks and show optimality $\square$

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 a 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...

INDEP(P) is much harder

- INDEP(P) is $\mathcal{NP}$-complete by trivial reduction to 3-PARTITION:
  - Give $3n$ integers $a_1, \ldots, a_{3n}$ 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 and 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)
- This ratio is tight

- There is a PTAS for INDEP(P), a $(1+\epsilon)$-approximation (massively exponential in $1/\epsilon$)
- There is no known FPTAS, unlike for INDEP(2)

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)
- This ratio is tight

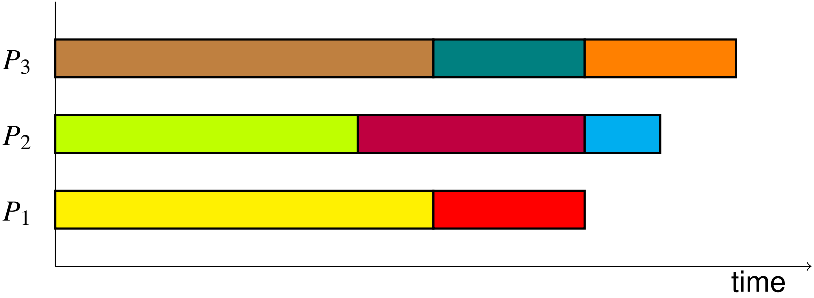
- There is a PTAS for INDEP(P), a $(1 + \epsilon)$-approximation (massively exponential in $1/\epsilon$)
- There is no known FPTAS, unlike for INDEP(2)

Outline

Why are many scheduling problems hard?

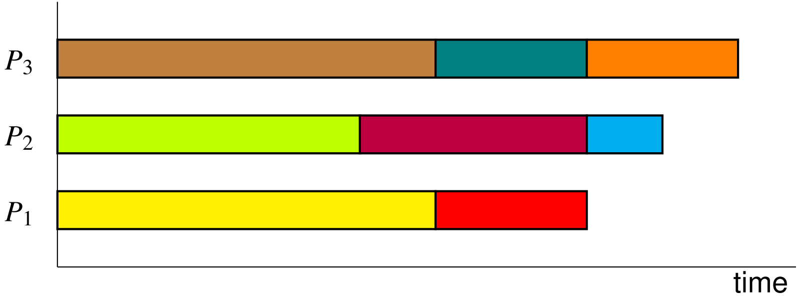
- Many scheduling problems are $\mathcal{NP}$-complete
- One contributing reason is that they involve integer constraints
  - The same reason why bin packing is difficult: you can't cut boxes into pieces!
- Let's see this on an example...

INDEP(P) example schedule (offline)



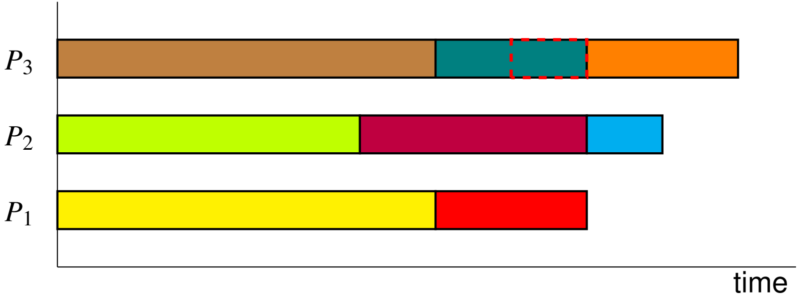
$\sum a_i = 21$; makespan = 8

INDEP(P) example schedule (offline)

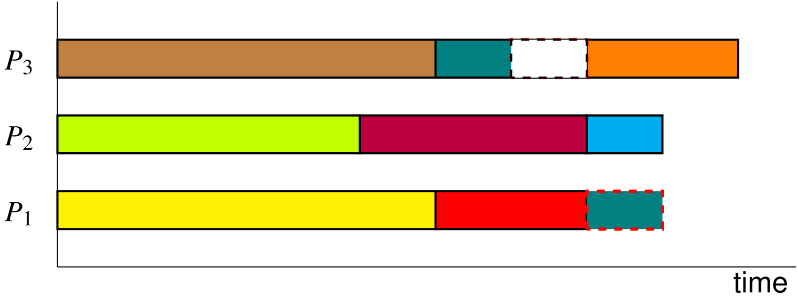


Let's modify the schedule using preemption/migration

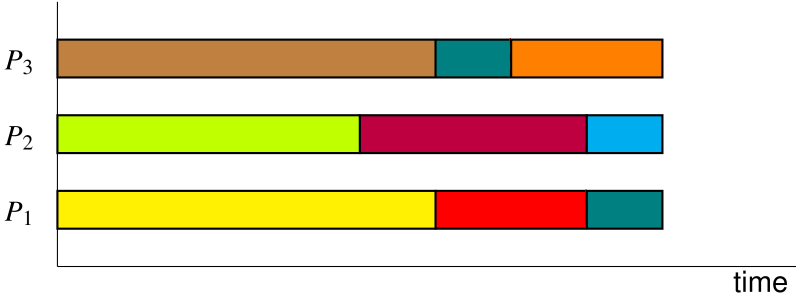
INDEP(P) example schedule (offline)



INDEP(P) example schedule (offline)



INDEP(P) example schedule (offline)



$\sum a_i = 21$; makespan = 7 (optimal: no idle time)

Cutting tasks

- By “cutting” a task in two, we’re able to have all processors finish at the same time
  - Zero idle time means the schedule is optimal
- If we were able to cut all tasks into tiny bits, then we would always be able to achieve zero idle time
  - Again, if you have a knife, binpacking is easy
  - Of course, there’d be “cutting overhead”...
- Question: Can this be done for real-world applications?

Divisible Load applications

- It turns out that many useful applications consist of very large numbers of small, independent, and identical tasks
  - task execution time $<<$ application execution time
  - tasks can be completed in any order
  - tasks all do the same thing, but on different data
- Example applications:
  - Ray tracing (1 task = 1 photon)
  - MPEG encoding of a movie (1 task = 1 frame)
  - Seismic event processing (1 task = 1 event)
  - High-Energy Physics (1 task = 1 particle)
- These applications are termed *Divisible Loads* (DLs)
  - So fine-grain that a *continuous load* assumption is valid
- This should make scheduling trivial (INDEP(P) with same-size tasks)

OpenMP Loops

- OpenMP is used primarily to parallelize loops in which all iterations are independent
- If the number of iterations is large, a loop is a divisible load!
- Simple divisible load assumption: If n iterations on p cores, then each core performs $\sim n/p$ iterations
- Easy, right?
- But:
  1. Not all iterations are always equal
     - So we want to create a lot of chunks to avoid idle time!
  2. Creating a chunk of iterations incurs overhead
     - So we want to create few chunks to avoid overhead!

OpenMP: chunk size $\sim n/p$

OpenMP

```
#pragma omp parallel for schedule(static)
for (i=0; i < N; i++) {
  // compute something
}
```

- Each thread performs $\sim n/p$ iterations
- Low overhead: “assign” work to each thread once
- High potential idle time if iterations are non-identical

OpenMP: chunk size = constant

OpenMP

```
#pragma omp parallel for schedule(dynamic, chunksize)
for (i=0; i < N; i++) {
  // compute something
}
```

- Each thread performs *chunksize* iterations (default = 1)
- High overhead (if $chunksize << N$): "assign" work to each thread many times
  - Implemented via a critical section to increment an index
- Low idle time (if $chunksize << N$): if iterations are wildly different then using $chunksize = 1$ corresponds to the on-line optimal algorithm for solving INDEP(P), but with overhead added to each task

OpenMP: chunk size = variable

OpenMP

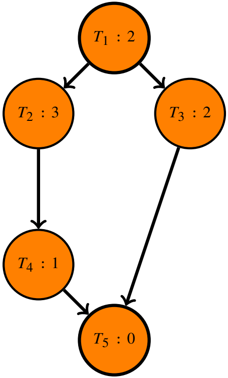
```
#pragma omp parallel for schedule(guided, min_chunksize)
for (i=0; i < N; i++) {
  // compute something
}
```

- Chunk sizes are created as follows and executed in an greedy fashion in this order
  - $N/2$ iterations partitioned into p chunks
  - $N/4$ iterations partitioned into p chunks
  - ...
  - until a minimal chunksize is reached (default=1)
- Goal:
  - Low overhead at the beginning, no idle time anyway
  - High overhead at the end but low idle time

Outline

Task dependencies

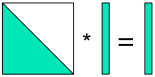
- In practice tasks often have *dependencies*
- A general model of computation is the Acyclic Directed Graph (DAG), $G = (V, E)$
- Each task has a *weight* (i.e., execution time in seconds), a *parent*, and *children*
- The first task is the *source*, the last task the *sink*
- Topological (partial)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



Where do DAGs come from?

- Consider a (lower) triangular linear system solve
  - What you would need to do after an LU factorization

$Ax = b$



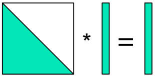
- Simple Algorithm

```
for (i = 0; i < n; i++) {
    x[i] = b[i] / a[i,i];
    for (j=i+1; i<n; i++) {
      b[j] = b[i] - a[j,i] * x[i];
    }
}
```

Where do DAGs come from?

- Consider a (lower) triangular linear system solve
  - What you would need to do after an LU factorization

$Ax = b$



- Simple Algorithm

```
for (i = 0; i < n; i++) {
    T_{i,i}: x[i] = b[i] / a[i,i];
    for (j=i+1; i<n; i++) {
      T_{i,j}: b[j] = b[i] - a[j,i] * x[i];
    }
}
```

Tasks, Dependencies, etc.

```
for (i = 0; i < n; i++) {
      Ti,i: x[i] = b[i] / a[i,i];
      for (j=i+1; i<n; i++) {
        Ti,j: b[j] = b[i] - a[j,i] * x[i];
      }
```

- All tasks $T_{i,*}$ are executed at iteration i of the outer loop
- There is a simple sequential order of the tasks

$T_{0,0} < T_{0,1} < \ldots < T_{0,n-1} < T_{1,0} < T_{1,1} < \ldots < T_{1,n-1} < \ldots$

- Of course, when considering a parallel execution, one tries to find independent tasks
- To see if tasks are independent one must examine their input (In) and their output (Out)

Tasks, Dependencies, etc.

```
for (i = 0; i < n; i++) {
      T_i,i: x[i] = b[i] / a[i,i];
      for (j=i+1; i<n; i++) {
        T_i,j: b[j] = b[i] - a[j,i] * x[i];
      }
```

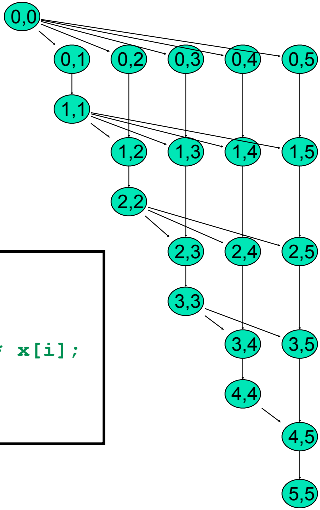
- Input and Output
  - $In(T_{i,i})$ = {b[i], a[i,i]}
  - $Out(T_{i,i})$ = {x[i]}
  - $In(T_{i,j})$ = {b[i], a[j,i], x[i]} for j > i
  - $Out(T_{i,j})$ = {b[j]} for j > i
- Bernstein Conditions
  - T and T' are independent if all 3 conditions are met
    - $In(T) \cap Out(T') = \emptyset$
    - $Out(T) \cap In(T') = \emptyset$
    - $Out(T) \cap Out(T') = \emptyset$

Task Graph

```
for (i = 0; i < n; i++) {
      T_{i,i}: x[i] = b[i] / a[i,i];
      for (j=i+1; i<n; i++) {
        T_{i,j}: b[j] = b[i] - a[j,i] * x[i];
      }
    }
```

- It is easy to see that
  - for all i, all $T_{i,j}$ are independent of each other for j > i
  - for all i, all $T_{i,j}$ depend on $T_{i,i}$, for j > i
  - for all i, all Ti,j depend on Ti-1,j for j >= i and i > 0
- Hence the task graph

Task Graph



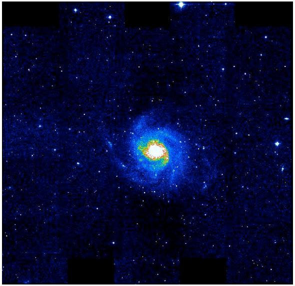

```
for (i = 0; i < n; i++) {
  T_i,i: x[i] = b[i] / a[i,i];
  for (j=i+1; i<n; i++) {
    T_i,j: b[j] = b[i] - a[j,i] * x[i];
  }
}
```

More taskgraphs

- The previous taskgraph comes from a low-level analysis of the code
  - It probably makes little sense to do a parallel implementation with MPI with such a low task granularity
  - Can totally make sense with OpenMP
  - Such task graphs can also be used by compilers to do code optimization by exploiting multiple functional units, pipelines functional units, etc.
  - With “blocking” these tasks could become MPI tasks
- Other taskgraphs are really how the application was build

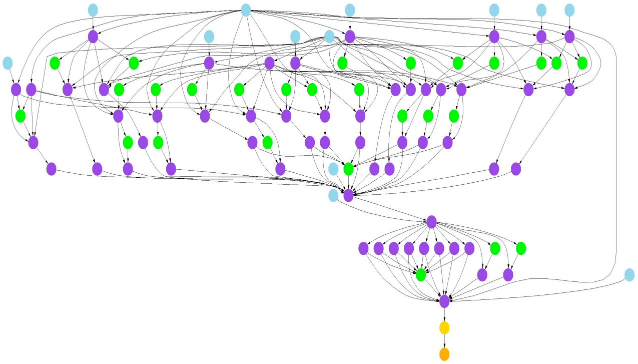
Scientific Workflows

- A popular way in which many scientific applications are constructed is as workflows
  - A scientists conceptually drags and drops computational kernels and connects their input-output
  - The result is a DAG (actually more general than a DAG) that does something useful
- Example Application: Montage
  - Produce Mosaic of the Sky
  - Based on multiple data sources
  - Given angle, coordinates, size, etc.
  - 10s of thousands of tasks
  - Example: M101 galaxy images



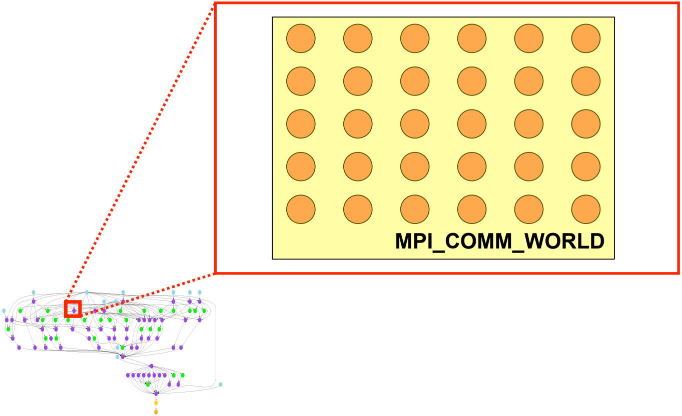
Many levels of parallelisms

- Montage Workflow



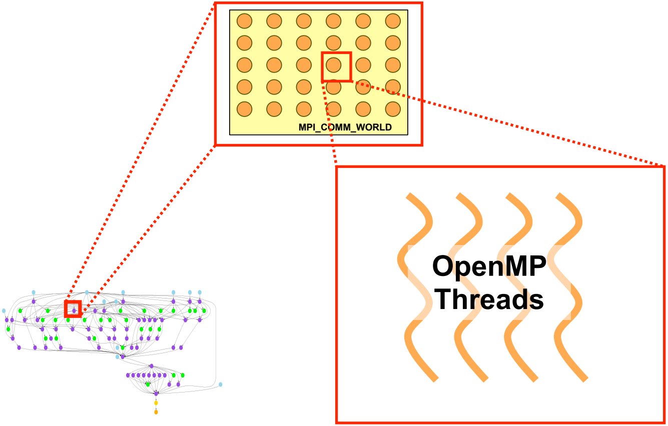
Many levels of parallelisms

- Montage Workflow



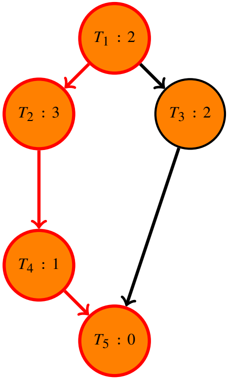
Many levels of parallelisms

- Montage workflow



Critical path

- Assume that the DAG executes on p processors
- The longest path (in seconds) 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)



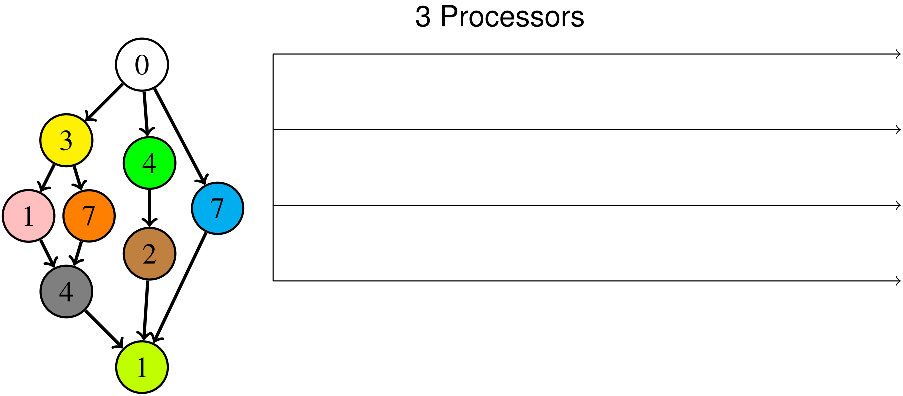
Complexity

- Unsurprisingly, DAG scheduling 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

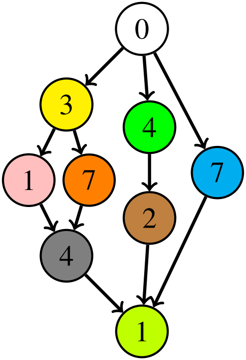
Complexity

- Unsurprisingly, DAG scheduling 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

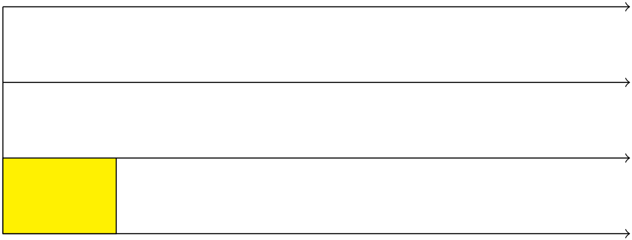
List scheduling example



List scheduling example

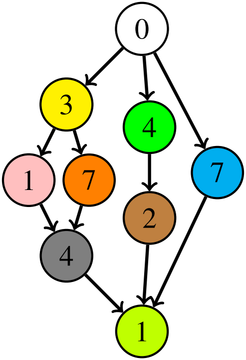


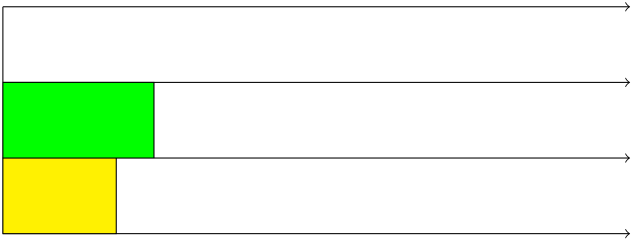
3 Processors



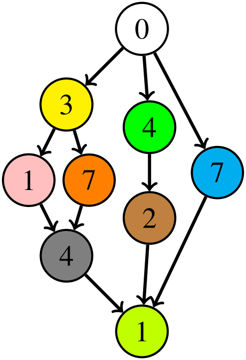
List scheduling example

3 Processors



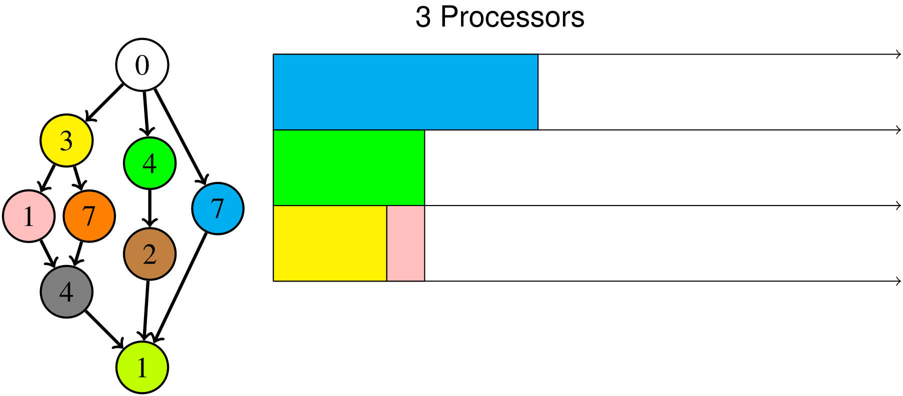


List scheduling example

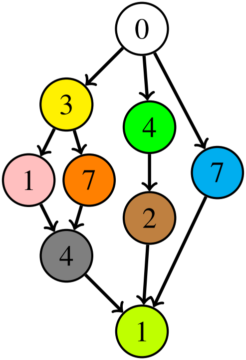


3 Processors

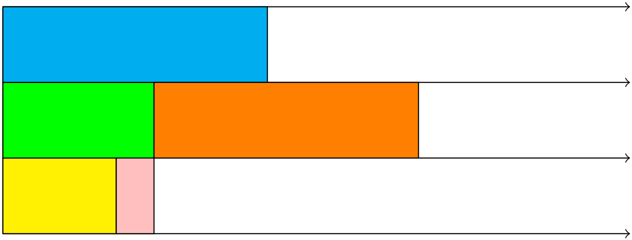
List scheduling example



List scheduling example

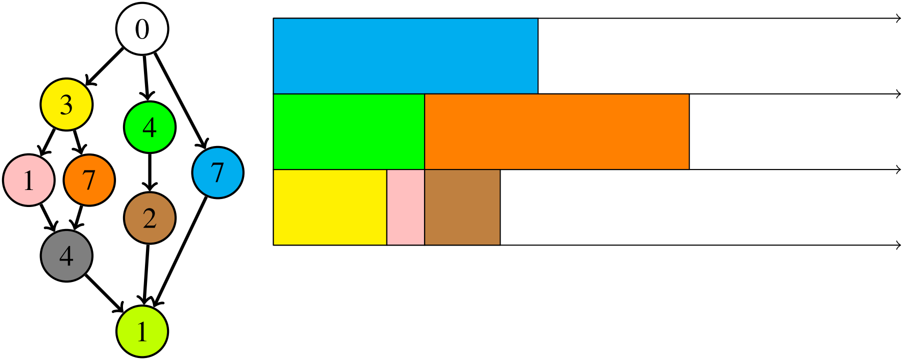


3 Processors

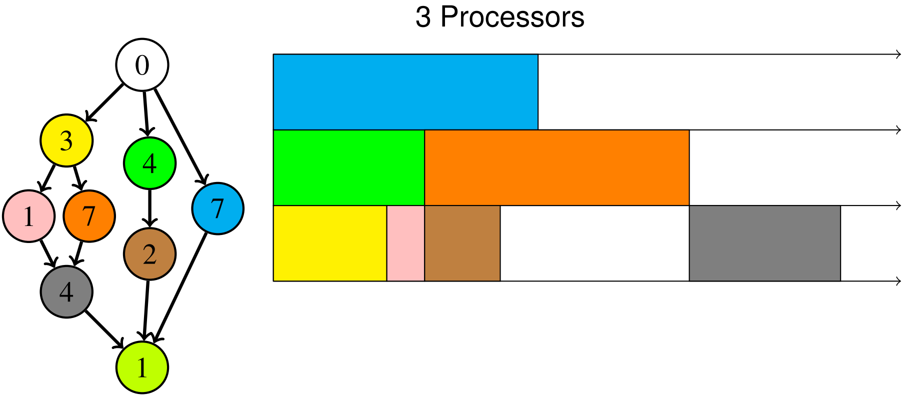


List scheduling example

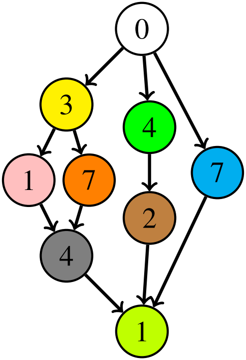
3 Processors



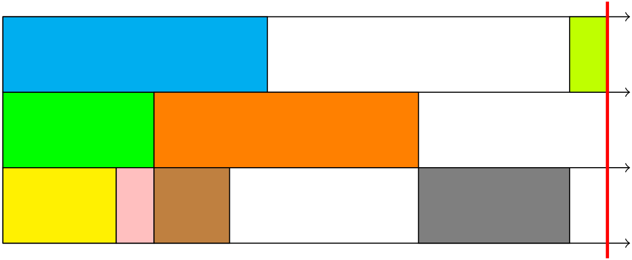
List scheduling example



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

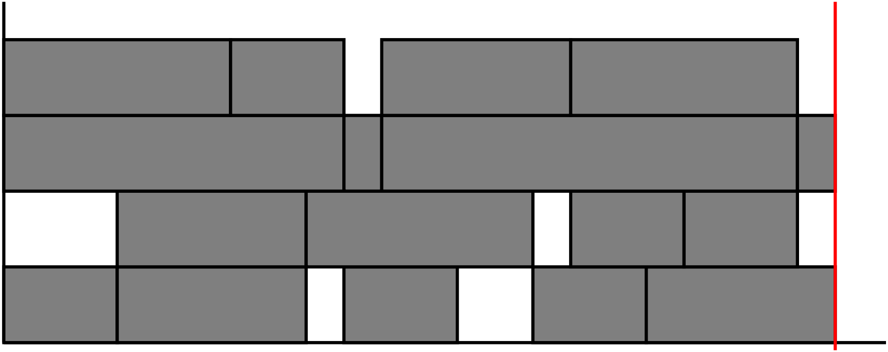
Theorem (fundamental)

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 ☺

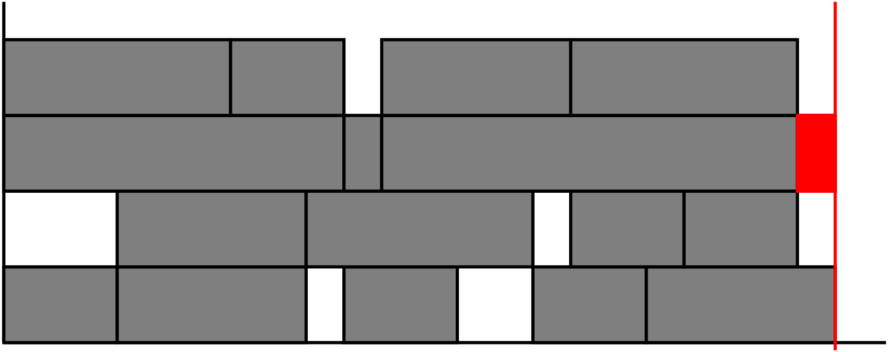
Approximation ratio

Let's consider a list-scheduling schedule



Approximation ratio

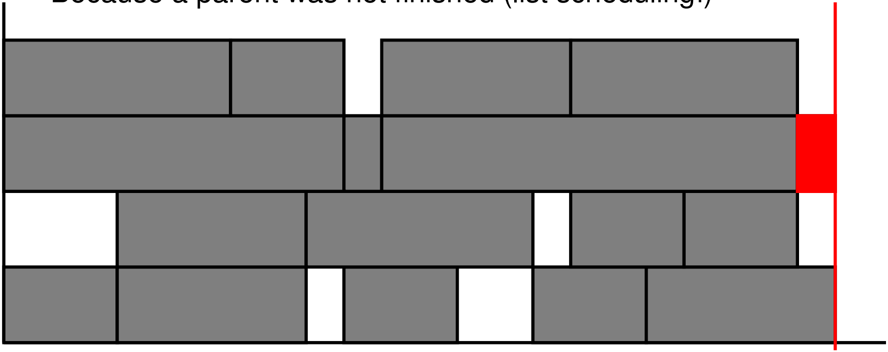
Let's consider one of the tasks that finishes last



Approximation ratio

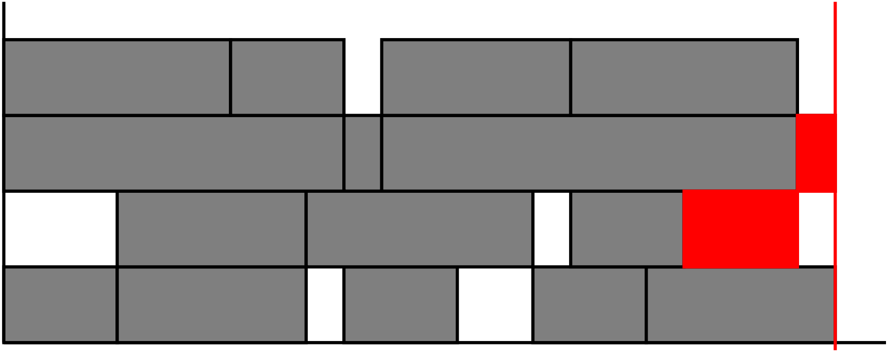
Why didn't this task run during an earlier idle period?

Because a parent was not finished (list scheduling!)



Approximation ratio

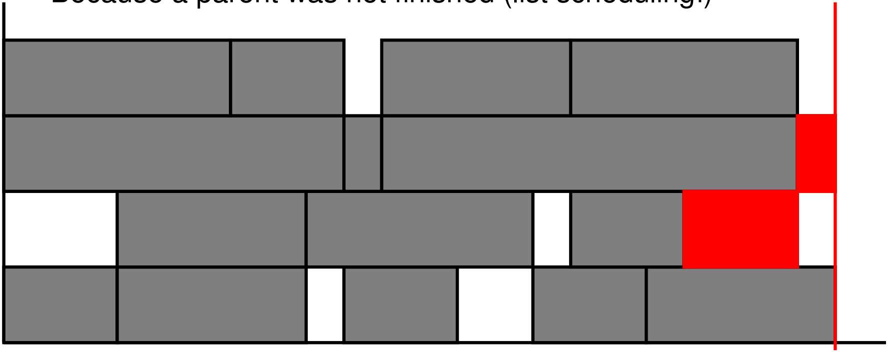
Let's look at a parent



Approximation ratio

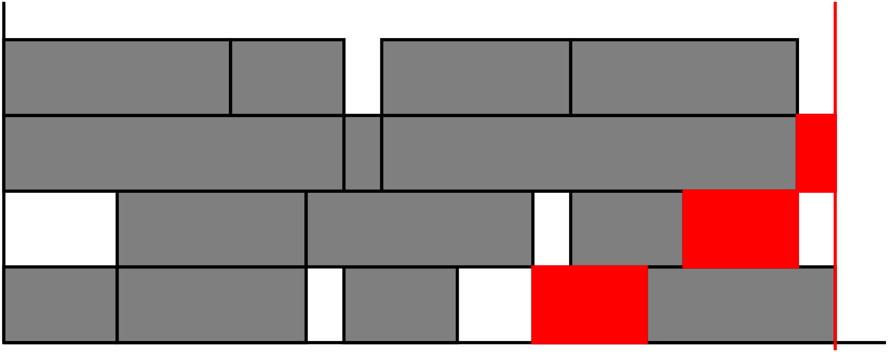
Why didn't this task run during an earlier idle period?

Because a parent was not finished (list scheduling!)



Approximation ratio

Let's look at a parent



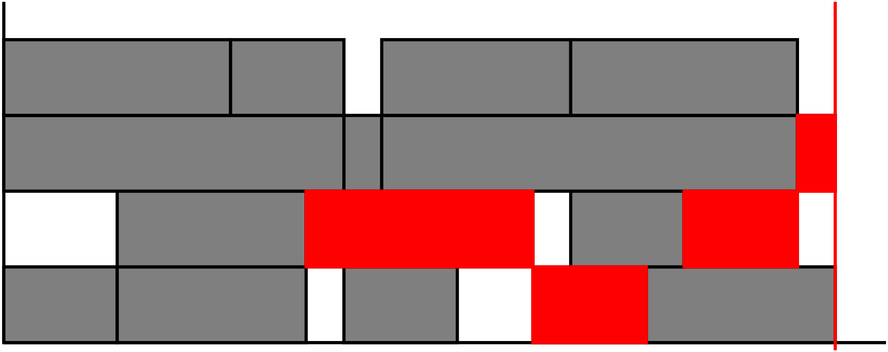
Approximation ratio

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

Because a parent was not finished (list scheduling!)

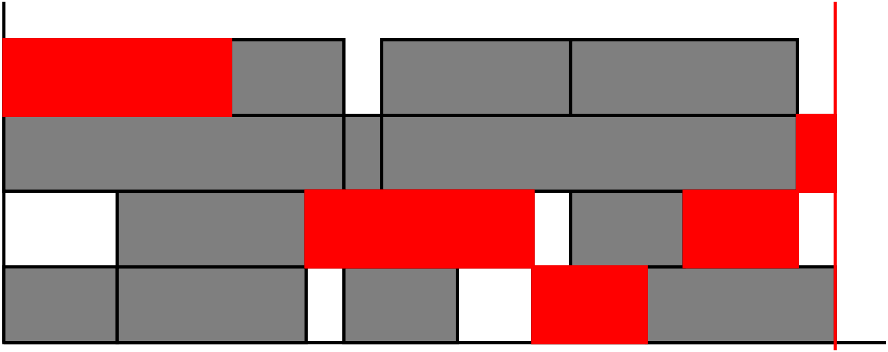
Approximation ratio

Let's look at a parent



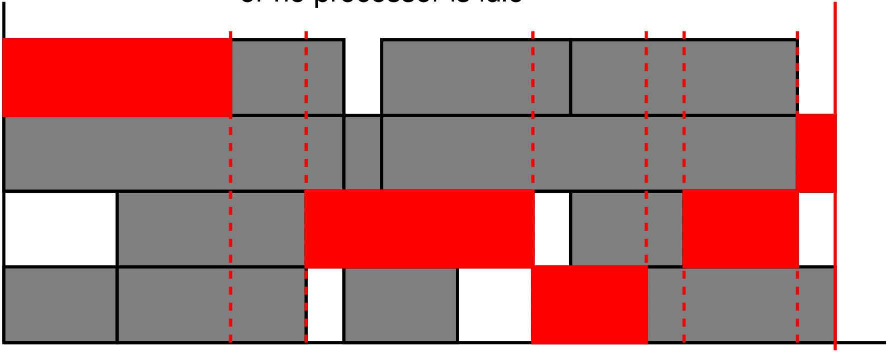
Approximation ratio

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

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

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 *bottom level*, the length of the path from the task to the sink
- Pick the task with the *largest* bottom level

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 ($\max(0, C_i - d_i)$)
- ...

Example scheduling problems

- The classification is not perfect and variations among authors are common
- Some examples:
  - $P2||C_{max}$, which we called INDEP(2)
  - $P||C_{max}$, which we called INDEP(P)
  - $P|prec|C_{max}$, which we called DAG scheduling
  - $R2|chains|\sum C_i$
    - Two related 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

Does any of this help?

- So far we've looked at very simple models
- What if we throw everything in?
  - Non-identical tasks
  - Task dependencies
  - Heterogeneous compute nodes
  - Complex network topologies with heterogeneous "links"
  - Task computation times not known ahead of time
  - Tasks not known ahead of time
  - Tasks are themselves parallel
  - Processor/network speeds change
  - etc.
- But how do we even reason about something this complicated and non-deterministic?
- One good practical option: dynamic execution

Master-Worker Dynamic execution

- Master process:
  - Keeps a list of “ready” tasks to compute with where input data can be found
- Worker processes
  - Request work from the master when idle
  - Create new tasks to compute and tell the master about them
- Each work request causes overhead
- Cleverness should be used to avoid long data transfers
  - i.e., leave data distributed and have the master try to assign a task to a process that already has the data

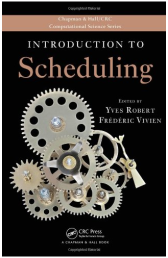
Work Stealing

- No Master
- Each process keeps a queue of tasks to perform and extracts a task from its local queue whenever idle
- If the queue is empty, then a process *steals* a task from another “victim” process
- Many options:
  - How many candidate victims should be considered?
  - Which victim do I pick?
  - How many tasks to I steal (overhead)?
- Under some assumptions, there are theoretical results:
  - Bounds on numbers of steals, with high probability
  - Bounds on overall makespan, with high probability
- Many efficient implementations in shared-memory (e.g., Cilk) and distributed-memory (e.g., Kaapi)

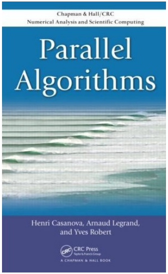
Conclusion

- 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 FTPAS?
    - 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!
- In the wild, dynamic scheduling may work well

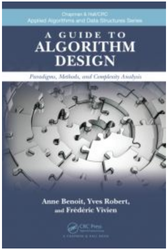
Sources



Y. Robert
F. Vivien



H. Casanova
A. Legrand
Y. Robert



A. Benoit
Y. Robert
F. Vivien