Scheduling

Master 2 Research Lecture: Parallel Systems

Vincent Danjean, MCF UJF, LIG/INRIA/Moais
Derick Kondo, CR INRIA, LIG/INRIA/Mescal
Arnaud Legrand, CR CNRS, LIG/INRIA/Mescal
Jean-François Méhaut, PR UJF, LIG/INRIA/Mescal
Bruno Raffin, CR INRIA, LIG/INRIA/Moais
Jean-Louis Roch, MCF ENSIMAG, LIG/INRIA/Moais
Alexandre Termier, MCF UJF, LIG/Hadas

LIG laboratory, arnaud.legrand@imag.fr

October 20, 2008

Outline

Outline

Analyzing a Simple Code

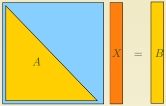
Solving $A.x = B$ where A is lower triangular matrix

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

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

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

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



Analyzing a Simple Code

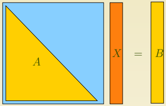
Solving $A.x = B$ where A is lower triangular matrix

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

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

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

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



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.

Analyzing a Simple Code

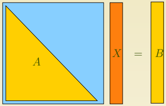
Solving $A.x = B$ where A is lower triangular matrix

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

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

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

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



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.

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.

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

Bernstein Conditions

Definition.

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

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

Those conditions are known as Bernstein's conditions [5].

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 [5].

We can check that:

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

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 [5].

We can check that:

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

```
for i = 1 to n do
    Task T_{i,i}: x(i) ← b(i)/a(i,i)
    for j = i+1 to n do
        Task T_{i,j}: b(j) ← b(j) − a(j,i) × 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 [5].

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

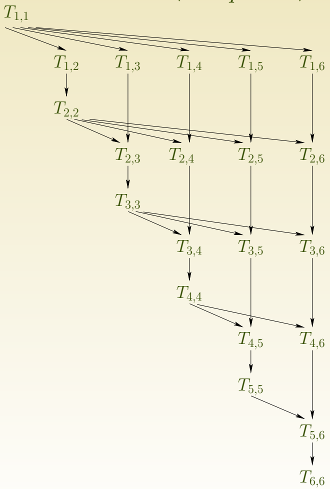
Precedence

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 $(<_{seq} \cap \perp)$.

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

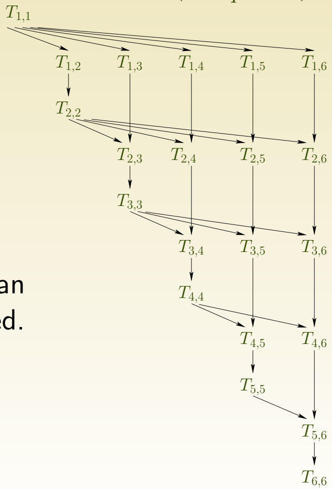
A dependence graph G is used.



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



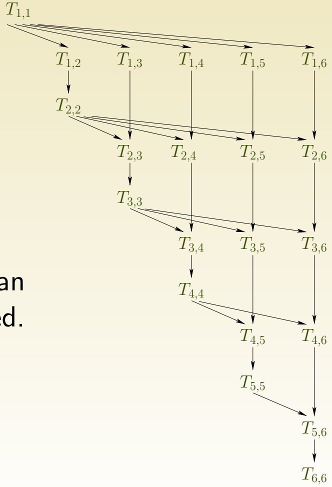
A dependence graph G is used.

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

Precedence

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

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.

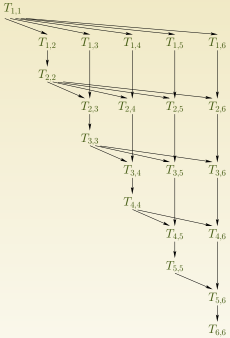
The previous task graph comes from a low-level analysis of the code.

It probably makes little sense to do a parallel implementation with MPI with such a low task granularity.

Can totally make sense with OpenMP.

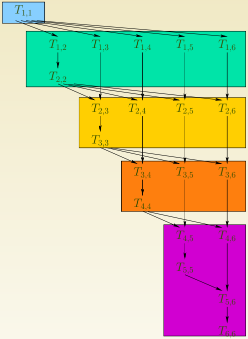
Such task graphs can also be used by compilers to do code optimization by exploiting multiple functional units, pipelines functional units, etc.

With blocking these tasks could become MPI (parallel) tasks.



From Coarse-grain Task Graphs...

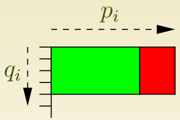
The previous task graph comes from a low-level analysis of the code.
It probably makes little sense to do a parallel implementation with MPI with such a low task granularity.
Can totally make sense with OpenMP.
Such task graphs can also be used by compilers to do code optimization by exploiting multiple functional units, pipelines functional units, etc.
With blocking these tasks could become MPI (parallel) tasks.



Hide applications' complexity

3 versions:

- Rigid Tasks

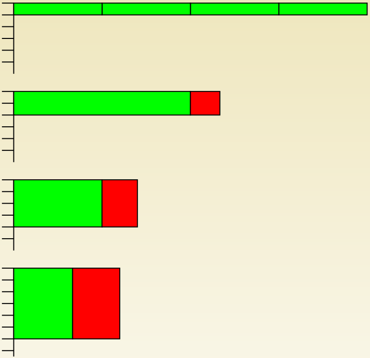


The execution time *generally* decreases with the number of processors but the penalty incurred by communications and synchronizations increases.

Hide applications' complexity

3 versions:

- Rigid Tasks
- Moldable Tasks

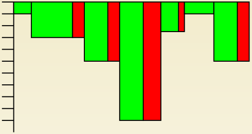


The execution time *generally* decreases with the number of processors but the penalty incurred by communications and synchronizations increases.

Hide applications' complexity

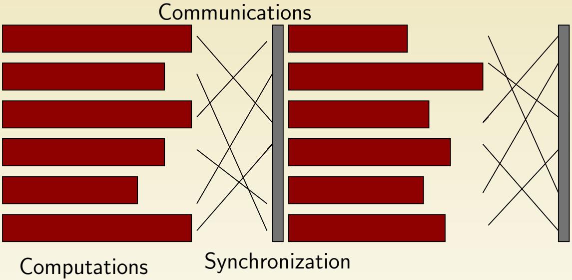
3 versions:

- Rigid Tasks
- Moldable Tasks
- Malleable Tasks



The execution time *generally* decreases with the number of processors but the penalty incurred by communications and synchronizations increases.

Bulk Synchronous Parallel is a programming paradigm whose principle is a series of independent steps of computations and communication/synchronization.



The cost of a superstep is determined as the sum of three terms:

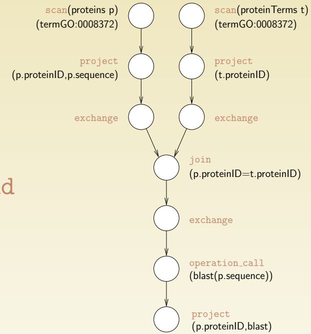
$$T = \max_i w(i) + \max h(i)g + l$$

Scheduling under BSP is about finding a tradeoff between load-balancing and number of communication/synchronizations.

Workflow

Task-graph do not necessarily come from instruction-level analysis.

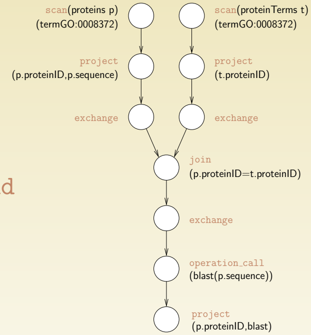
```
select p.proteinID,
       blast(p.sequence)
from proteins p, proteinTerms t
where p.proteinID = t.proteinID and
t.term = GO:0008372
```



Workflow

Task-graph do not necessarily come from instruction-level analysis.

```
select p.proteinID,
       blast(p.sequence)
from proteins p, proteinTerms t
where p.proteinID = t.proteinID and
t.term = GO:0008372
```

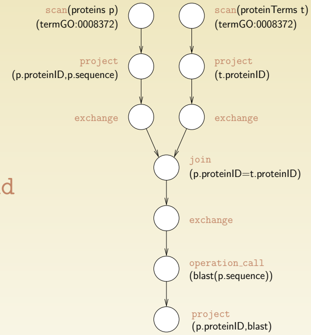


- Each task may be parallel, preemptable, divisible, ...

Workflow

Task-graph do not necessarily come from instruction-level analysis.

```
select p.proteinID,
       blast(p.sequence)
from proteins p, proteinTerms t
where p.proteinID = t.proteinID and
t.term = GO:0008372
```

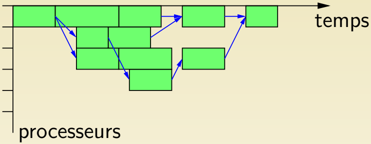


- Each task may be parallel, preemptable, divisible, . . .
- Each edge depicts a dependency i.e. most of the times some data to transfer.

Outline

Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)

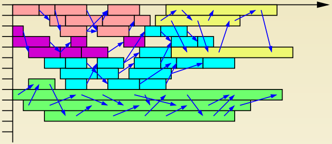


Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)

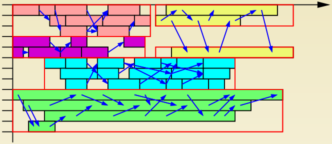
Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)



- When one purchases a cluster, typically many users want to use it.
  - One cannot let them step on each other's toes
  - Every user wants to be on a dedicated machine
  - Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.

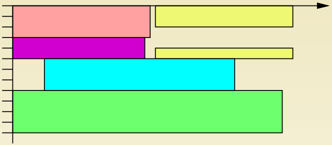
- Parallel Tasks from Scientific Computations (simulation, medical)



- When one purchases a cluster, typically many users want to use it.
  - One cannot let them step on each other's toes
  - Every user wants to be on a dedicated machine
  - Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.

Need for Batch Scheduling

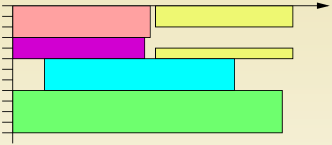
- Parallel Tasks from Scientific Computations (simulation, medical)



- When one purchases a cluster, typically many users want to use it.
  - One cannot let them step on each other's toes
  - Every user wants to be on a dedicated machine
  - Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.

Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)



- When one purchases a cluster, typically many users want to use it.
  - One cannot let them step on each other's toes
  - Every user wants to be on a dedicated machine
  - Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.

The Job Scheduler is the entity that prevents them from stepping on each other's toes

The Job Scheduler gives out nodes to applications

Batch Scheduling

Each job is defined as a Number of nodes (q_i) and a Time (p_i):

I want 6 nodes for 1h

Typically users are "charged" against an "allocation": e.g. "*You only get 100 CPU hours per week*".

A batch scheduler is a central middleware to manage resources (e.g. processors) of parallel machines:

- accept jobs (computing tasks) submitted by users
- decide when and where jobs are executed
- start jobs execution

They take into account:

- unavailability of some nodes
- users jobs mutual exclusion
- specific needs for jobs (memory, network, ...)

While trying to :

- maximize resources usage
- be fair among users

Typical wanted features:

- Interactive mode
- Batch mode
- Parallel jobs support
- Multi-queues with priorities
- Admission policies (limit on usage, notions of user groups, power users)
- Resources matching
- File staging
- Jobs dependences
- Backfilling
- Reservations
- Best effort jobs
- Environment reconfiguration

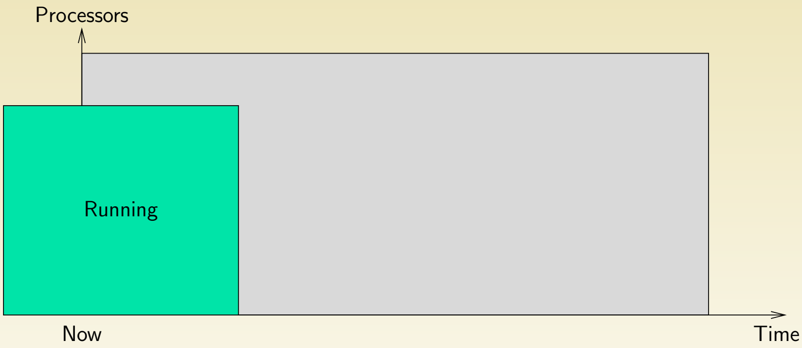
There are many existing batch schedulers : LSF, PBS/Torque, Maui scheduler, Sun Grid Engine, EASY, OAR, ...

These are complex systems with many config options !

Main Batch Schedulers Features

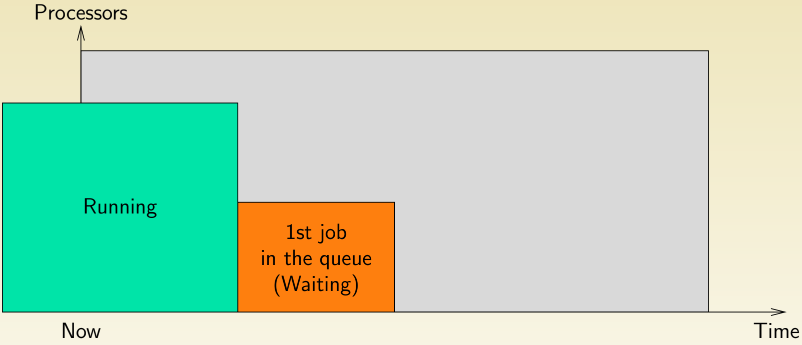
	OpenPBS	SGE	Maui Scheduler (+ OpenPBS)	OAR
Interactive mode	×	×	×	×
Batch mode	×	×	×	×
Parallel jobs support	×	×	×	×
Multi-queues with priorities	×	×	×	×
Resources matching	×	×	×	×
Admission policies	×	×	×	×
File staging	×	×	×	
Jobs dependences	×	×	×	
Backfilling			×	×
Reservations			×	×
Best effort jobs				×
Environment reconfiguration				×
Fair sharing			×	×

General Principle



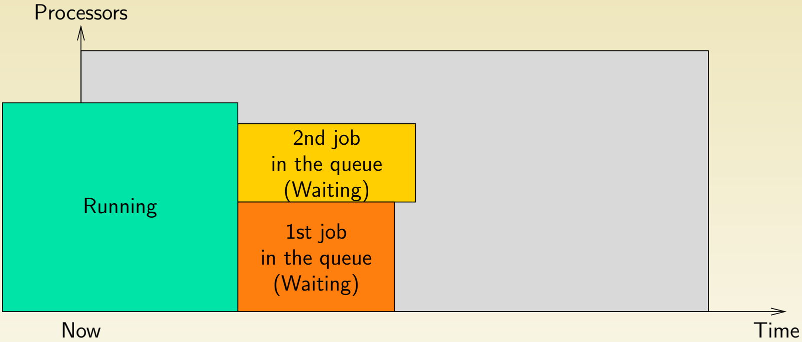
- Jobs arrive one after the other and are scheduled at arrival.

General Principle



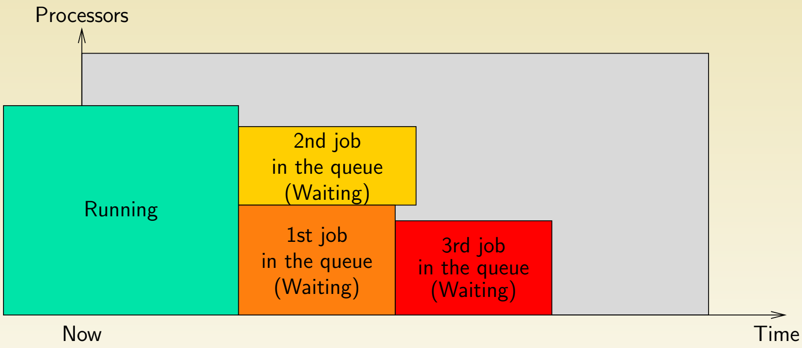
- Jobs arrive one after the other and are scheduled at arrival.

General Principle



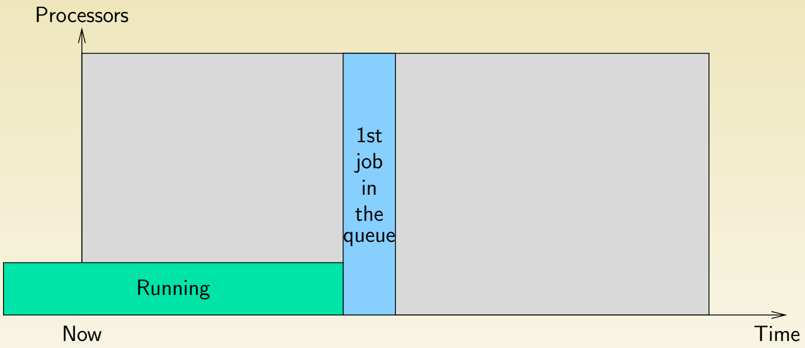
- Jobs arrive one after the other and are scheduled at arrival.

General Principle



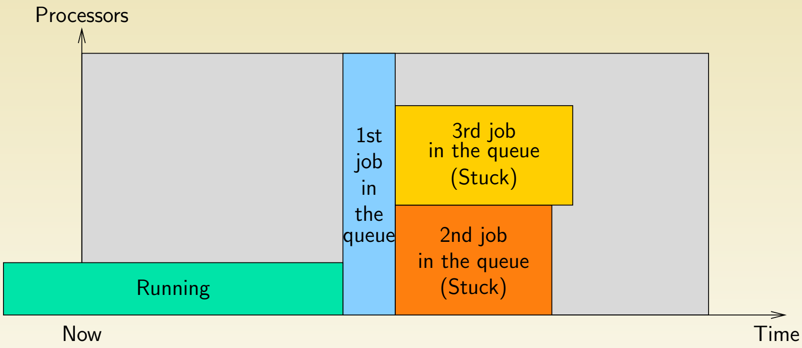
- Jobs arrive one after the other and are scheduled at arrival.

First Come First Served



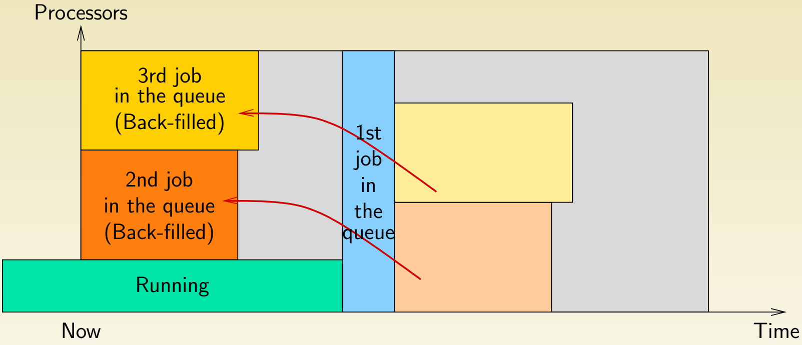
- FCFS = simplest scheduling option
- Fragmentation $\leadsto$ need for backfilling

First Come First Served



- FCFS = simplest scheduling option
- Fragmentation $\rightsquigarrow$ need for backfilling

First Come First Served



- FCFS = simplest scheduling option
- Fragmentation $\rightsquigarrow$ need for backfilling

Backfilling: Question

- Which job(s) should be picked for promotion through the queue?
- Many heuristics are possible
- Two have been studied in detail
  - EASY
  - Conservative Back Filling (CBF)
- In practice EASY (or variants of it) is used, while CBF is not.
- Although, OAR, a recently proposed batch scheduler implements CBF.

EASY Backfilling

Extensible Argonne Scheduling System

Maintain only one *reservation*, for the first job in the queue.

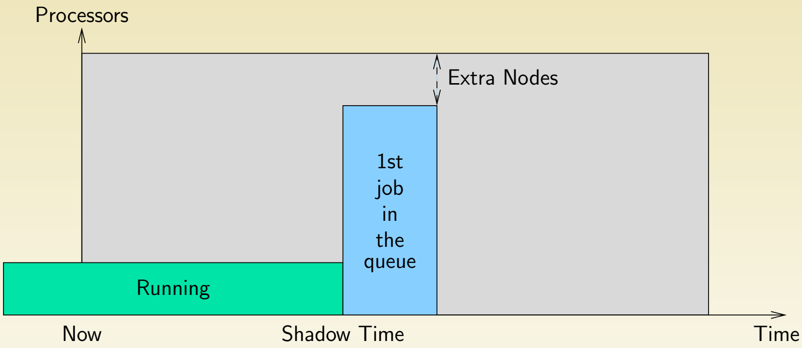
Definitions:

Shadow time
: time at which the first job in the queue starts execution

Extra nodes
: number of nodes idle when the first job in the queue starts execution

1. Go through the queue in order starting with the 2nd job.
2. Backfill a job if it will terminate by the shadow time, or it needs less than the extra nodes.

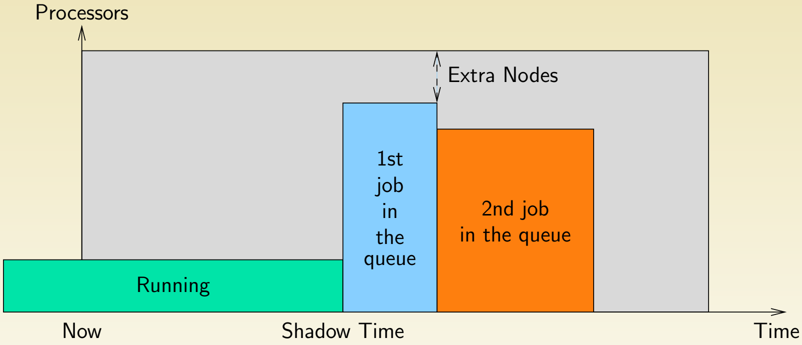
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs

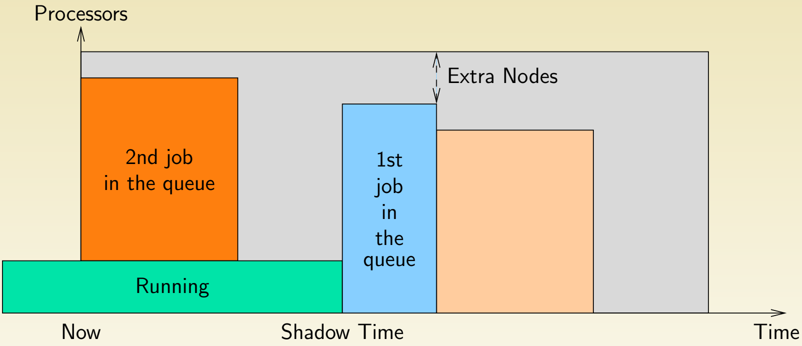
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs

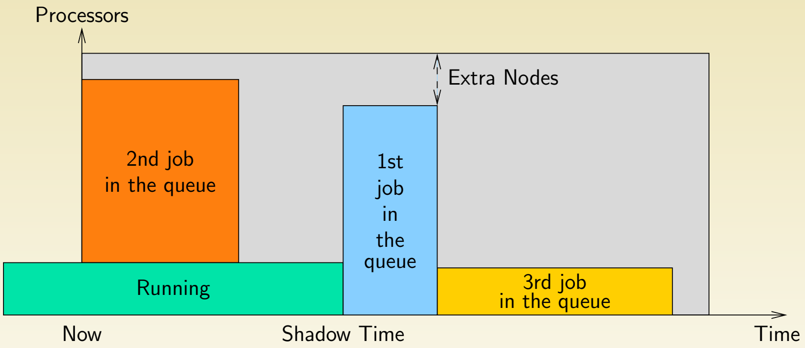
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs

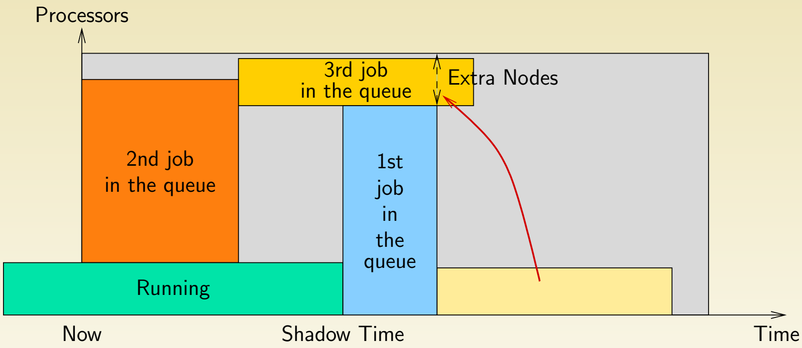
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs

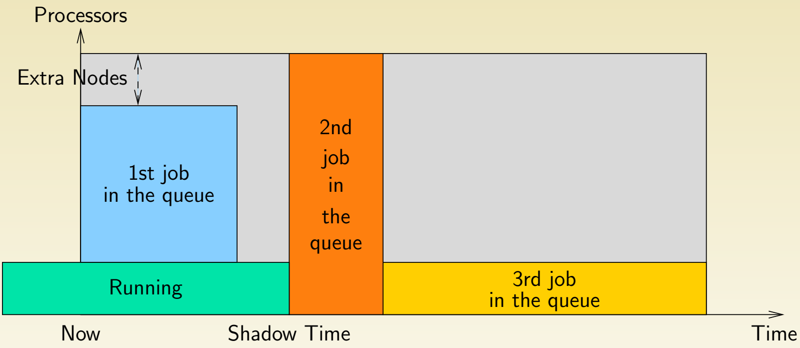
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs

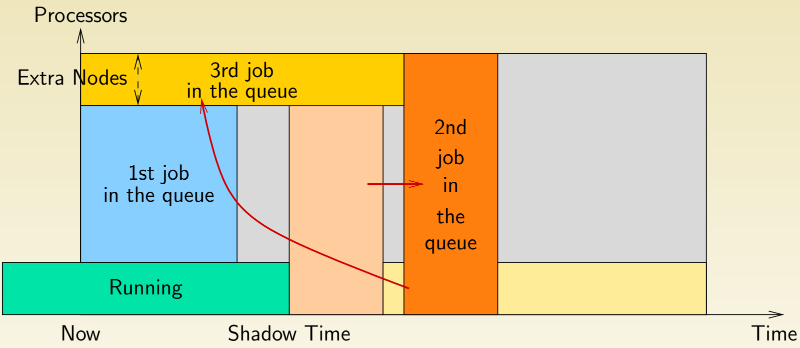
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!

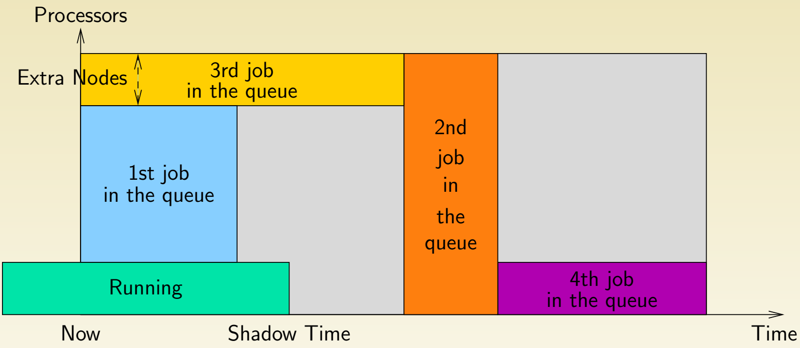
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!

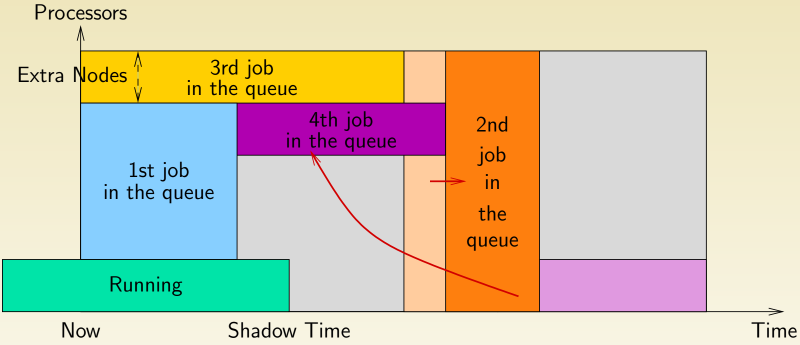
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!

EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!

EASY Properties

Unbounded Delay.
- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!

No Starvation.
- Delay of first job is bounded by runtime of current jobs
- When the first job finishes, the second job becomes the first job in the queue
- Once it is the first job, it cannot be delayed further

Other approach.
- Conservative Backfilling. *EVERY* job has a *reservation*. A job may be backfilled only if it does not delay any other job ahead of it in the queue.
- Fixes the unbounded delay problem that EASY has. More complicated to implement (The algorithm must find holes in the schedule) though.
- EASY favors small long jobs and harms large short jobs.

When Does Backfilling Happen?

Possibly when

- A new job arrives

When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts

When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts
- When a job finishes early

When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts
- When a job finishes early

Users provide job runtime estimates (Jobs are killed if they go over).

When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts
- When a job finishes early

Users provide job runtime estimates (Jobs are killed if they go over).
Trade-off:

- provide a conservative estimate: you goes through the queue faster (may be backfilled)
- provide a loose estimate: your job will not be killed

When Does Backfilling Happen?

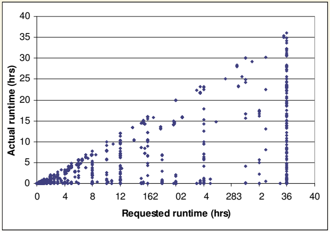
Possibly when

- A new job arrives
- The first job in the queue starts
- When a job finishes early

Users provide job runtime estimates (Jobs are killed if they go over).
Trade-off:

- provide a conservative estimate: you goes through the queue faster (may be backfilled)
- provide a loose estimate: your job will not be killed

Are estimates accurate?



How Good is the Schedule ?

All of this is great, but how do we know what a "good" schedule is? FCFS, EASY, CFB, Random?

What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs

How Good is the Schedule ?

All of this is great, but how do we know what a "good" schedule is? FCFS, EASY, CFB, Random?

What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs

1. Turn-around time or flow (Wait time + Run time).
   Job 1 needs 1h of compute time and waits 1s
   Job 2 needs 1s of compute time and waits 1h
   Clearly Job 1 is really happy, and Job 2 is not happy at all

How Good is the Schedule ?

All of this is great, but how do we know what a "good" schedule is? FCFS, EASY, CFB, Random?

What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs

1. Turn-around time or flow (Wait time + Run time).
   Job 1 needs 1h of compute time and waits 1s
   Job 2 needs 1s of compute time and waits 1h
   Clearly Job 1 is really happy, and Job 2 is not happy at all
2. Wait time (equivalent to "user happiness")
   Job 1 asks for 1 nodes and waits 1 h
   Job 2 asks for 512 nodes and waits 1h
   Again, Job 1 is unhappy while Job 2 is probably sort of happy.
   We need a metric that represents happiness for small, large, short, long jobs.

How Good is the Schedule ?

All of this is great, but how do we know what a "good" schedule is? FCFS, EASY, CFB, Random?

What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs

1. Turn-around time or flow (Wait time + Run time).
   Job 1 needs 1h of compute time and waits 1s
   Job 2 needs 1s of compute time and waits 1h
   Clearly Job 1 is really happy, and Job 2 is not happy at all
2. Wait time (equivalent to "user happiness")
   Job 1 asks for 1 nodes and waits 1 h
   Job 2 asks for 512 nodes and waits 1h
   Again, Job 1 is unhappy while Job 2 is probably sort of happy.
   We need a metric that represents happiness for small, large, short, long jobs.
3. Slowdown or Stretch (turn-around time divided by turn- around time if alone in the system)
   Doesn't really take care of the small/large problem. Could think of some scaling, but unclear !

Now What ?

Now we have a few metrics we can consider

We can run simulations of the scheduling algorithms, and see how they fare.

We need to test these algorithms in representative scenarios

Supercomputer/cluster traces. Collect the following for long periods of time:

- Time of submission
- How many nodes asked
- How much time asked
- How much time was actually used
- How much time spent in the queue

Uses of the traces:

1. Drive simulations
2. Come up with models of user behaviors

Sample Results

A type of experiments that people have done: replace user estimate by f times the actual run time

Possible to improve performance by multiplying user estimates by 2!

	EASY	CBF
Mean Slowdown		
KTH	-4.8%	-23.0%
CTC	-7.9%	-18.0%
SDSC	+4.6%	-14.2%
Mean Response time		
KTH	-3.3%	-7.0%
CTC	-0.9%	-1.6%
SDSC	-1.6%	-10.9%

Message

- These are all heuristics.
- They are not specifically designed to optimize the metrics we have designed.
- It is difficult to truly understand the reasons for the results.
- But one can derive some empirical wisdom.
- One of the reasons why one is stuck with possibly obscure heuristics is that we're dealing with an *on-line* problem: We don't know what happens next.
- We cannot wait for all jobs to be submitted to make a decision. But we can wait for a while, accumulate jobs, and schedule them together.

Summary

Batch Schedulers are what we're stuck with at the moment.
They are often hated by users.

- I submit to the queue asking for 10 nodes for 1 hour.
- I wait for two days.
- My code finally starts, but doesn't finish within 1 hour and gets killed!!

Summary

Batch Schedulers are what we're stuck with at the moment.
They are often hated by users.

- I submit to the queue asking for 10 nodes for 1 hour.
- I wait for two days.
- My code finally starts, but doesn't finish within 1 hour and gets killed!!

A lot of research, a few things happening "in the field".
When you go to a company that has clusters (like most of them), they typically have a job scheduler, so it's good to have some idea of what it is.

Summary

Batch Schedulers are what we're stuck with at the moment.
They are often hated by users.

- I submit to the queue asking for 10 nodes for 1 hour.
- I wait for two days.
- My code finally starts, but doesn't finish within 1 hour and gets killed!!

A lot of research, a few things happening "in the field".
When you go to a company that has clusters (like most of them), they typically have a job scheduler, so it's good to have some idea of what it is.
A completely different approach is gang scheduling, which we discuss next.

Outline

Gang Scheduling: Basis

- All processes belonging to a job run at the same time (the term gang denotes all processors within a job).
- Each process runs alone on each processor.
- BUT: there is rapid coordinated context switching.
- It is possible to suspend/preempt jobs arbitrarily

- All processes belonging to a job run at the same time (the term gang denotes all processors within a job).
- Each process runs alone on each processor.
- BUT: there is rapid coordinated context switching.
- It is possible to suspend/preempt jobs arbitrarily $\rightsquigarrow$ May allow more flexibility to optimize some metrics.

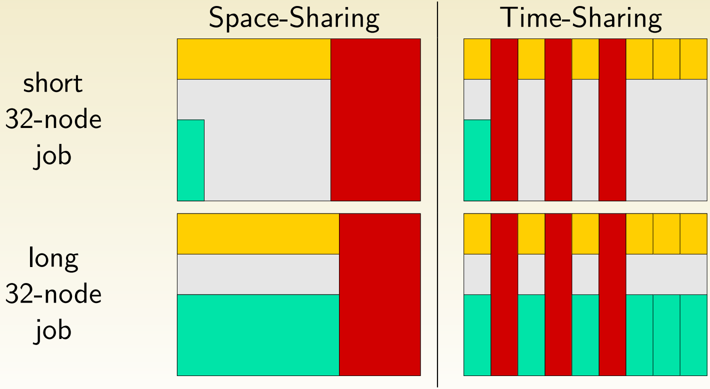
Gang Scheduling: Basis

- All processes belonging to a job run at the same time (the term gang denotes all processors within a job).
- Each process runs alone on each processor.
- BUT: there is rapid coordinated context switching.
- It is possible to suspend/preempt jobs arbitrarily $\rightsquigarrow$ May allow more flexibility to optimize some metrics.
- If processing times are not known in advance (or grossly erroneous), preemption can help short jobs that would be "stuck" behind a long job.
- Should improve machine utilization.

Gang Scheduling: an Example

- A 128 node cluster.
- A running 64-node job.
- A 32-node job and a 128-node job are queued.

Should the 32-node job be started ?



More uniform slowdown, better resource usage.

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency(Frequent cache flushing).

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency(Frequent cache flushing).
- RAM Pressure (more jobs must fit in memory, swapping to disk causes unacceptable overhead).

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency(Frequent cache flushing).
- RAM Pressure (more jobs must fit in memory, swapping to disk causes unacceptable overhead).
- Typically not used in production HPC systems (batch scheduling is preferred).

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency(Frequent cache flushing).
- RAM Pressure (more jobs must fit in memory, swapping to disk causes unacceptable overhead).
- Typically not used in production HPC systems (batch scheduling is preferred).
- Some implementations (MOSIX, Kerighed).

Batch Scheduling it is then

So it seems we're stuck with batch scheduling.
Why don't we like Batch Scheduling?

Batch Scheduling it is then

So it seems we're stuck with batch scheduling.
Why don't we like Batch Scheduling? Because queue waiting times are difficult to predict.

- depends on the status of the queue
- depends on the scheduling algorithm used
- depends on all sorts of configuration parameters set by system administrator
- depends on future job completions!
- etc.

So I submit my job and then it's in limbo somewhere, which is eminently annoying to most users.

Batch Scheduling it is then

So it seems we're stuck with batch scheduling.
Why don't we like Batch Scheduling? Because queue waiting times are difficult to predict.

- depends on the status of the queue
- depends on the scheduling algorithm used
- depends on all sorts of configuration parameters set by system administrator
- depends on future job completions!
- etc.

So I submit my job and then it's in limbo somewhere, which is eminently annoying to most users.
That is why there is more and more demand for reservation support.
Users build (badly?) the schedule by themselves.

Batch Scheduling and Grids

Grids result from the collaboration of many Universities/Computing Centers.

Everyone runs its own Batch Scheduler that cannot be bypassed. How to decide where we should submit our jobs?

Batch Scheduling and Grids

Grids result from the collaboration of many Universities/Computing Centers.

Everyone runs its own Batch Scheduler that cannot be bypassed. How to decide where we should submit our jobs?

When in doubt, a brute-force approach is to:

- Do multiple submissions for different numbers of nodes
- Cancel all submissions but the first one that comes back
- Or possibly make some ad-hoc call regarding whether to keep a potentially poor request in the hope of getting a better one through shortly after.

Batch Scheduling and Grids

Grids result from the collaboration of many Universities/Computing Centers.

Everyone runs its own Batch Scheduler that cannot be bypassed. How to decide where we should submit our jobs?

When in doubt, a brute-force approach is to:

- Do multiple submissions for different numbers of nodes
- Cancel all submissions but the first one that comes back
- Or possibly make some ad-hoc call regarding whether to keep a potentially poor request in the hope of getting a better one through shortly after.

What happens if everybody does this?

Batch Scheduling and Grids

Grids result from the collaboration of many Universities/Computing Centers.

Everyone runs its own Batch Scheduler that cannot be bypassed. How to decide where we should submit our jobs?

When in doubt, a brute-force approach is to:

- Do multiple submissions for different numbers of nodes
- Cancel all submissions but the first one that comes back
- Or possibly make some ad-hoc call regarding whether to keep a potentially poor request in the hope of getting a better one through shortly after.

What happens if everybody does this?

Other issues:

- File Staging ?
- Load Balancing between sites ?

Sequential Job Scheduling for Grids

A set unrelated processors $P_1, \ldots, \mathcal{P}_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).

Let's try a few natural scheduling strategies. We denote by a_i the time at which P_i is available (at the beginning $a_i = 0$ for all P_i):

Sequential Job Scheduling for Grids

A set unrelated processors $P_1, \ldots, \mathcal{P}_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).

Let's try a few natural scheduling strategies. We denote by a_i the time at which P_i is available (at the beginning $a_i = 0$ for all P_i):

Min-Min Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each J_j and choose the one with the smallest C_j. Update the corresponding a_i (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

Sequential Job Scheduling for Grids

A set unrelated processors $P_1, \dots, \mathcal{P}_n$ and a set of sequential jobs $J_1, \dots, J_n$ (processing time $p_{i,j}$).

Let's try a few natural scheduling strategies. We denote by a_i the time at which P_i is available (at the beginning $a_i = 0$ for all P_i):

Min-Min Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each J_j and choose the one with the smallest C_j. Update the corresponding a_i (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

Max-Min Choose J_j with the largest C_j and update the corresponding a_i (its best host) accordingly.

Sequential Job Scheduling for Grids

A set unrelated processors $P_1, \ldots, \mathcal{P}_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).

Let's try a few natural scheduling strategies. We denote by a_i the time at which P_i is available (at the beginning $a_i = 0$ for all P_i):

Min-Min
: Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each J_j and choose the one with the smallest C_j. Update the corresponding a_i (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

Max-Min
: Choose J_j with the largest C_j and update the corresponding a_i (its best host) accordingly.

Sufferage
: S_j is the difference between the best completion time of J_j and its second best completion time. Choose the job with the largest sufferage and schedule it on its best processor.

Sequential Job Scheduling for Grids

A set unrelated processors $P_1, \ldots, \mathcal{P}_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).

Let's try a few natural scheduling strategies. We denote by a_i the time at which P_i is available (at the beginning $a_i = 0$ for all P_i):

Min-Min Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each J_j and choose the one with the smallest C_j. Update the corresponding a_i (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

Max-Min Choose J_j with the largest C_j and update the corresponding a_i (its best host) accordingly.

Sufferage S_j is the difference between the best completion time of J_j and its second best completion time. Choose the job with the largest sufferage and schedule it on its best processor.

Problem: How do you get an estimate of $p_{i,j}$?

So Where are we ?

- Batch schedulers are complex pieces of software that are used in practice.
- A lot of experience on how they work and how to use them.
- But ultimately everybody knows they are an imperfect solution.
- Many view the lack of theoretical foundations as a big problem.
- Let's look at what theoreticians think of job scheduling.
- The first step is to define the scheduling problem (On-line vs. Off-line, Preemption vs. No preemption).

Outline

The Job Scheduling Problem

- When do jobs "arrive"?

  On-line We know when they arrive (periodic, aperiodic, etc.)

  We don't: batch scheduling, gang scheduling.

  We only get upper bounds on the real processing times (kind of non-clairvoyant).

  Off-line more related to application scheduling but should be studied before everything else.

- Control of the resources
  - With preemption: Gang Scheduling
  - Without preemption: Batch Scheduling
- The practical implementations (batch and gang) are only heuristics and do not consider the problem at a theoretical level.
  In fact, they don't optimize any metric each individual user cares about.

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.

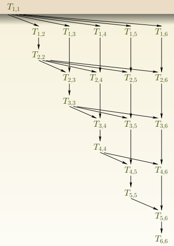
Task system

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 $\sigma : V \to \mathbb{N}^*$ such that:

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

Schedule and Allocation

Definition: Schedule.

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

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

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

Definition: Allocation.

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

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

Schedule and Allocation

Definition: Schedule.

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

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

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

Definition: Allocation.

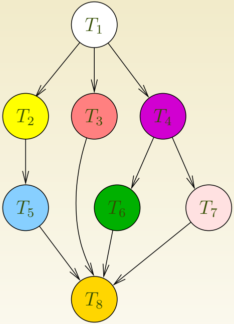
An allocation of a task system $G = (V, E, w)$ is a function $\pi : V \to \mathcal{P}$ such that:

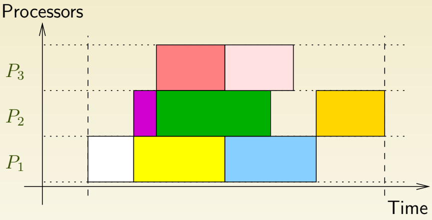
$$\pi(T) = \pi(T') \Leftrightarrow \begin{cases} \sigma(T) + w(T) \leqslant \sigma(T') \text{ or} \\ \sigma(T') + w(T') \leqslant \sigma(T) \end{cases}$$

Depending on the application and platform model, much more complex definitions can be proposed.

Gantt-chart

Manipulating functions is generally not very convenient. That is why Gantt-chart are used to depict schedules and allocations.





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.

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. □

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 $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$. 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. □

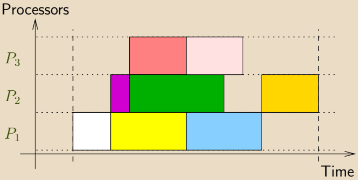
Therefore all task systems we will be considering in the following are Directed Acyclic Graphs.

Makespan

Definition: **Makespan**.

The makespan of a schedule is the total execution time :

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



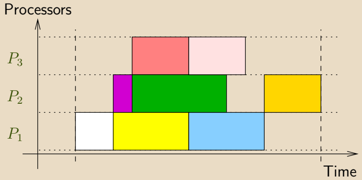
The makespan is also often referred as $C_{\max}$ in the literature.

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

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.

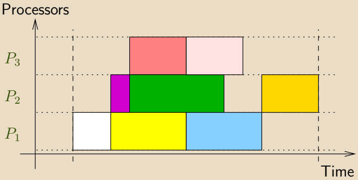
$$C_{\max} = \max_{v \in V} C_v$$

- $Pb(p)$: find a schedule with the smallest possible makespan, using at most p processors. $MS_{opt}(p)$ denotes the optimal makespan using only p processors.

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.

Critical path

Let $\Phi = (T_1, T_2, \ldots, T_n)$ be a path in G. w can be extended to paths in the following way :

$$w(\Phi) = \sum_{i=1}^{n} w(T_i)$$

Critical path

Let $\Phi = (T_1, T_2, \ldots, T_n)$ be a path in G. w can be extended to paths in the following way :

$$w(\Phi) = \sum_{i=1}^{n} w(T_i)$$

Lemma 1.

Let $G = (V, E, w)$ be a DAG and σ_p a schedule of G using p processors. For any path Φ in G, we have $MS(\sigma_p) \geqslant w(\Phi)$.

Critical path

Let $\Phi = (T_1, T_2, \ldots, T_n)$ be a path in G. w can be extended to paths in the following way :

$$w(\Phi) = \sum_{i=1}^{n} w(T_i)$$

Lemma 1.

Let $G = (V, E, w)$ be a DAG and σ_p a schedule of G using p processors. For any path Φ in G, we have $MS(\sigma_p) \geqslant w(\Phi)$.

Proof.

Let $\Phi = (T_1, T_2, \ldots, T_n)$ be a path in G: $(T_i, T_{i+1}) \in E$ for $1 \leqslant i < n$. Therefore we have $\sigma_p(T_i) + w(T_i) \leqslant \sigma_p(T_{i+1})$ for $1 \leqslant i < n$, hence

$$MS(\sigma_p) \geqslant w(T_n) + \sigma_p(T_n) - \sigma_p(T_1) \geqslant \sum_{i=1}^{n} w(T_i) = w(\Phi)\ . \qquad \square$$

Work, Cost, Speed-up and Efficiency

Definition.

Let $G = (V, E, w)$ be a DAG and σ_p a schedule of G using only p processors:

- **Work**: $W(\sigma_p) = \sum_{v \in V} w(v)$.

  On such DAGs, the work does not change with the schedule and communications are not taken into account. However, when the tasks are parallel (rigid, moldable, malleable), their work depends on the number of processors they are alloted!

  Indeed, parallel algorithms generally do not do the same operations as the sequential ones. They often have to do more.

Work, Cost, Speed-up and Efficiency

Definition.

Let $G = (V, E, w)$ be a DAG and σ_p a schedule of G using only p processors:

- Work: $W(\sigma_p) = \sum_{v \in V} w(v)$.

  On such DAGs, the work does not change with the schedule and communications are not taken into account. However, when the tasks are parallel (rigid, moldable, malleable), their work depends on the number of processors they are alloted!

  Indeed, parallel algorithms generally do not do the same operations as the sequential ones. They often have to do more.

- Cost: $C(\sigma_p) = p.MS(\sigma_p)$.

  The cost accounts for the idle time of the processing units.

Work, Cost, Speed-up and Efficiency

Definition.

Let $G = (V, E, w)$ be a DAG and σ_p a schedule of G using only p processors:

- Work: $W(\sigma_p) = \sum_{v \in V} w(v)$.

  On such DAGs, the work does not change with the schedule and communications are not taken into account. However, when the tasks are parallel (rigid, moldable, malleable), their work depends on the number of processors they are alloted!

  Indeed, parallel algorithms generally do not do the same operations as the sequential ones. They often have to do more.
- Cost: $C(\sigma_p) = p.MS(\sigma_p)$.

  The cost accounts for the idle time of the processing units.
- Speed-up: $s(\sigma_p) = \frac{Seq}{MS(\sigma_p)}$, where $Seq = MS_{opt}(1) = \sum_{v \in V} w(v)$.

Work, Cost, Speed-up and Efficiency

Definition.

Let $G = (V, E, w)$ be a DAG and σ_p a schedule of G using only p processors:

- **Work**: $W(\sigma_p) = \sum_{v \in V} w(v)$.

  On such DAGs, the work does not change with the schedule and communications are not taken into account. However, when the tasks are parallel (rigid, moldable, malleable), their work depends on the number of processors they are alloted!
  Indeed, parallel algorithms generally do not do the same operations as the sequential ones. They often have to do more.

- **Cost**: $C(\sigma_p) = p.MS(\sigma_p)$.

  The cost accounts for the idle time of the processing units.

- **Speed-up**: $s(\sigma_p) = \dfrac{Seq}{MS(\sigma_p)}$, where $Seq = MS_{opt}(1) = \sum_{v \in V} w(v)$.

- **Efficiency**: $e(\sigma_p) = \dfrac{s(\sigma_p)}{p} = \dfrac{Seq}{p \times MS(\sigma_p)}$.

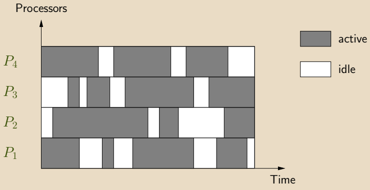
Speed-up and Efficiency (Cont'd)

Theorem 2.

Let $G = (V, E, w)$ be a DAG. For any schedule σ_p using p processors:

$$0 \leqslant e(\sigma_p) \leqslant 1 .$$

Proof.



Let $Idle$ denote the total idle time. $Seq + Idle$ is then equal to the total surface of the rectangle, i.e. $p \times MS(\sigma_p)$.

Therefore $e(\sigma_p) = \dfrac{Seq}{p \times MS(\sigma_p)} \leqslant 1$. □

The speed-up is thus bounded by the number of processors. No supra-linear speed-up in our model!

A Trivial Result

Theorem 3.

Let $G = (V, E, w)$ be a DAG. We have

$$Seq = MS_{opt}(1) \geqslant \ldots \geqslant MS_{opt}(p) \geqslant MS_{opt}(p+1) \geqslant \ldots \geqslant MS_{opt}(\infty) .$$

A Trivial Result

Theorem 3.

Let $G = (V, E, w)$ be a DAG. We have

$$Seq = MS_{opt}(1) \geqslant \ldots \geqslant MS_{opt}(p) \geqslant MS_{opt}(p+1) \geqslant \ldots \geqslant MS_{opt}(\infty) \; .$$

Allowing to use more processors cannot hurt.

A Trivial Result

Theorem 3.

Let $G = (V, E, w)$ be a DAG. We have

$$Seq = MS_{opt}(1) \geqslant \ldots \geqslant MS_{opt}(p) \geqslant MS_{opt}(p+1) \geqslant \ldots \geqslant MS_{opt}(\infty)\ .$$

Allowing to use more processors cannot hurt.

However, using more processors may hurt, especially in a model where communications are taken into account.

If we define $MS'(p)$ as the smallest makespan of schedules using exactly p processors, we may have $MS'(p) > MS'(p')$ with $p < p'$.

Graham Notation

Many parameter can change in a scheduling problem. Graham has then proposed the following classification : $\langle \alpha | \beta | \gamma \rangle$ [6]

Graham Notation

Many parameter can change in a scheduling problem. Graham has then proposed the following classification : $\langle\alpha|\beta|\gamma\rangle$ [6]

- α is the processor environment (a few examples):
  - $\emptyset$: single processor;
  - P: identical processors;
  - Q: uniform processors;
  - R: unrelated processors;

Graham Notation

Many parameter can change in a scheduling problem. Graham has then proposed the following classification : $\langle\alpha|\beta|\gamma\rangle$ [6]

- α is the processor environment (a few examples):
  - $\emptyset$: single processor;
  - P: identical processors;
  - Q: uniform processors;
  - R: unrelated processors;
- β describe task and resource characteristics (a few examples):
  - $pmtn$: preemption;
  - $prec$, $tree$ or $chains$: general precedence constraints, tree constraints, set of chain constraints and independent tasks otherwise;
  - r_j: tasks have release dates;
  - $p_j = p$ or $\underline{p} \leqslant p_j \leqslant \overline{p}$: all task have processing time equal to p, or comprised between $\underline{p}$ and $\overline{p}$, or have arbitrary processing times otherwise;
  - $\tilde{d}$: deadlines;

Graham Notation

Many parameter can change in a scheduling problem. Graham has then proposed the following classification : $\langle\alpha|\beta|\gamma\rangle$ [6]

- α is the processor environment (a few examples):
  - $\emptyset$: single processor;
  - P: identical processors;
  - Q: uniform processors;
  - R: unrelated processors;
- β describe task and resource characteristics (a few examples):
  - $pmtn$: preemption;
  - $prec$, $tree$ or $chains$: general precedence constraints, tree constraints, set of chain constraints and independent tasks otherwise;
  - r_j: tasks have release dates;
  - $p_j = p$ or $\underline{p} \leqslant p_j \leqslant \overline{p}$: all task have processing time equal to p, or comprised between $\underline{p}$ and $\overline{p}$, or have arbitrary processing times otherwise;
  - $\tilde{d}$: deadlines;
- γ denotes the optimization criterion (a few examples):
  - $C_{\max}$: makespan;
  - $\sum C_i$: average completion time;
  - $\sum w_i C_i$: weighted A.C.T;
  - $L_{\max}$: maximum lateness $(\max C_i - d_i)$;
  - ...

Complexity Results

If we have an infinite number of processors, the "as-soon-as-possible" schedule is optimal. $MS_{opt}(\infty) = \max_{\Phi \text{ path in } G} w(\Phi)$.

Complexity Results

If we have an infinite number of processors, the "as-soon-as-possible" schedule is optimal. $MS_{opt}(\infty) = \max_{\Phi \text{ path in } G} w(\Phi)$.

- $\langle P, 2||C_{\max}\rangle$ is weakly NP-complete (2-Partition);

Complexity Results

If we have an infinite number of processors, the "as-soon-as-possible" schedule is optimal. $MS_{opt}(\infty) = \max_{\Phi \text{ path in } G} w(\Phi)$.

- $\langle P, 2||C_{\max}\rangle$ is weakly NP-complete (2-Partition);

Proof.

By reduction to 2-Partition: can $\mathcal{A} = \{a_1, \ldots, a_n\}$ be partitioned into two sets $\mathcal{A}_1$, $\mathcal{A}_2$ such $\sum_{a \in \mathcal{A}_1} a = \sum_{a \in \mathcal{A}_2} a$?

$\square$

Complexity Results

If we have an infinite number of processors, the "as-soon-as-possible" schedule is optimal. $MS_{opt}(\infty) = \max_{\Phi \text{ path in } G} w(\Phi)$.

- $\langle P, 2||C_{\max}\rangle$ is weakly NP-complete (2-Partition);

Proof.

By reduction to 2-Partition: can $\mathcal{A} = \{a_1, \ldots, a_n\}$ be partitioned into two sets $\mathcal{A}_1$, $\mathcal{A}_2$ such $\sum_{a \in \mathcal{A}_1} a = \sum_{a \in \mathcal{A}_2} a$?
$p = 2$, $G = (V, E, w)$ with $V = \{v_1, \ldots, v_n\}$, $E = \emptyset$ and $w(v_i) = a_i, \forall 1 \leqslant i \leqslant n$.
Finding a schedule of makespan smaller or equal to $\frac{1}{2}\sum_i a_i$ is equivalent to solve the instance of 2-Partition. $\square$

Complexity Results

If we have an infinite number of processors, the "as-soon-as-possible" schedule is optimal. $MS_{opt}(\infty) = \max_{\Phi \text{ path in } G} w(\Phi)$.

- $\langle P, 2||C_{\max}\rangle$ is weakly NP-complete (2-Partition);

Proof.

By reduction to 2-Partition: can $\mathcal{A} = \{a_1, \ldots, a_n\}$ be partitioned into two sets $\mathcal{A}_1$, $\mathcal{A}_2$ such $\sum_{a \in \mathcal{A}_1} a = \sum_{a \in \mathcal{A}_2} a$?
$p = 2$, $G = (V, E, w)$ with $V = \{v_1, \ldots, v_n\}$, $E = \emptyset$ and $w(v_i) = a_i, \forall 1 \leqslant i \leqslant n$.
Finding a schedule of makespan smaller or equal to $\frac{1}{2}\sum_i a_i$ is equivalent to solve the instance of 2-Partition. $\square$

- $\langle P, 3|prec|C_{\max}\rangle$ is strongly NP-complete (3DM);

Complexity Results

If we have an infinite number of processors, the "as-soon-as-possible" schedule is optimal. $MS_{opt}(\infty) = \max_{\Phi \text{ path in } G} w(\Phi)$.

- $\langle P, 2||C_{\max}\rangle$ is weakly NP-complete (2-Partition);

Proof.

By reduction to 2-Partition: can $\mathcal{A} = \{a_1, \dots, a_n\}$ be partitioned into two sets $\mathcal{A}_1$, $\mathcal{A}_2$ such $\sum_{a \in \mathcal{A}_1} a = \sum_{a \in \mathcal{A}_2} a$?
$p = 2$, $G = (V, E, w)$ with $V = \{v_1, \dots, v_n\}$, $E = \emptyset$ and $w(v_i) = a_i, \forall 1 \leqslant i \leqslant n$.
Finding a schedule of makespan smaller or equal to $\frac{1}{2}\sum_i a_i$ is equivalent to solve the instance of 2-Partition. $\square$

- $\langle P, 3|prec|C_{\max}\rangle$ is strongly NP-complete (3DM);
- $\langle P|prec, p_j = 1|C_{\max}\rangle$ is strongly NP-complete (max-clique);

Complexity Results

If we have an infinite number of processors, the "as-soon-as-possible" schedule is optimal. $MS_{opt}(\infty) = \max_{\Phi \text{ path in } G} w(\Phi)$.

- $\langle P, 2||C_{\max}\rangle$ is weakly NP-complete (2-Partition);

Proof.

By reduction to 2-Partition: can $\mathcal{A} = \{a_1, \dots, a_n\}$ be partitioned into two sets $\mathcal{A}_1$, $\mathcal{A}_2$ such $\sum_{a \in \mathcal{A}_1} a = \sum_{a \in \mathcal{A}_2} a$?
$p = 2$, $G = (V, E, w)$ with $V = \{v_1, \dots, v_n\}$, $E = \emptyset$ and $w(v_i) = a_i, \forall 1 \leqslant i \leqslant n$.
Finding a schedule of makespan smaller or equal to $\frac{1}{2}\sum_i a_i$ is equivalent to solve the instance of 2-Partition. □

- $\langle P, 3|prec|C_{\max}\rangle$ is strongly NP-complete (3DM);
- $\langle P|prec, p_j = 1|C_{\max}\rangle$ is strongly NP-complete (max-clique);
- $\langle P, p \geqslant 3|prec, p_j = 1|C_{\max}\rangle$ is open;

If we have an infinite number of processors, the "as-soon-as-possible" schedule is optimal. $MS_{opt}(\infty) = \max_{\Phi \text{ path in } G} w(\Phi)$.

- $\langle P, 2||C_{\max}\rangle$ is weakly NP-complete (2-Partition);

Proof.

By reduction to 2-Partition: can $\mathcal{A} = \{a_1, \ldots, a_n\}$ be partitioned into two sets $\mathcal{A}_1$, $\mathcal{A}_2$ such $\sum_{a \in \mathcal{A}_1} a = \sum_{a \in \mathcal{A}_2} a$?
$p = 2$, $G = (V, E, w)$ with $V = \{v_1, \ldots, v_n\}$, $E = \emptyset$ and $w(v_i) = a_i, \forall 1 \leqslant i \leqslant n$.
Finding a schedule of makespan smaller or equal to $\frac{1}{2}\sum_i a_i$ is equivalent to solve the instance of 2-Partition. □

- $\langle P, 3|prec|C_{\max}\rangle$ is strongly NP-complete (3DM);
- $\langle P|prec, p_j = 1|C_{\max}\rangle$ is strongly NP-complete (max-clique);
- $\langle P, p \geqslant 3|prec, p_j = 1|C_{\max}\rangle$ is open;
- $\langle P, 2|prec, 1 \leqslant p_j \leqslant 2|C_{\max}\rangle$ is strongly NP-complete;

List Scheduling

When simple problems are hard, we should try to find good approximation heuristics. A ρ-approximation is an algorithm whose output is never more than a factor ρ times the optimum solution.

Natural idea: using greedy strategy like trying to allocate the most possible task at a given time-step. However at some point we may face a choice (when there is more ready tasks than available processors).

List Scheduling

When simple problems are hard, we should try to find good approximation heuristics. A ρ-approximation is an algorithm whose output is never more than a factor ρ times the optimum solution.

Natural idea: using greedy strategy like trying to allocate the most possible task at a given time-step. However at some point we may face a choice (when there is more ready tasks than available processors).

Any strategy that does not let on purpose a processor idle is efficient [7]. Such a schedule is called list-schedule.

Theorem 4: Coffman.

Let $G = (V, E, w)$ be a DAG, p the number of processors, and σ_p a list-schedule of G.

$$MS(\sigma_p) \leqslant \left(2 - \frac{1}{p}\right) MS_{opt}(p) \ .$$

List Scheduling

When simple problems are hard, we should try to find good approximation heuristics. A ρ-approximation is an algorithm whose output is never more than a factor ρ times the optimum solution.

Natural idea: using greedy strategy like trying to allocate the most possible task at a given time-step. However at some point we may face a choice (when there is more ready tasks than available processors).

Any strategy that does not let on purpose a processor idle is efficient [7]. Such a schedule is called list-schedule.

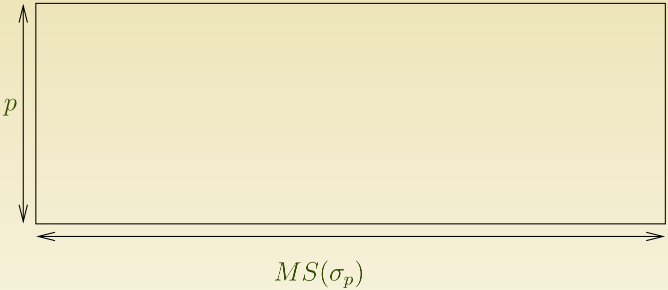
Theorem 4: Coffman.

Let $G = (V, E, w)$ be a DAG, p the number of processors, and σ_p a list-schedule of G.

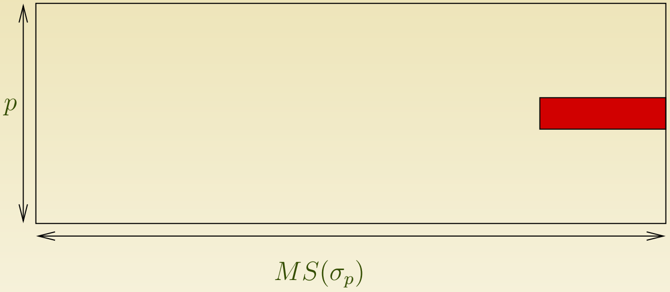
$$MS(\sigma_p) \leqslant \left(2 - \frac{1}{p}\right) MS_{opt}(p) \ .$$

Most of the time, list-heuristics are based on the critical path.

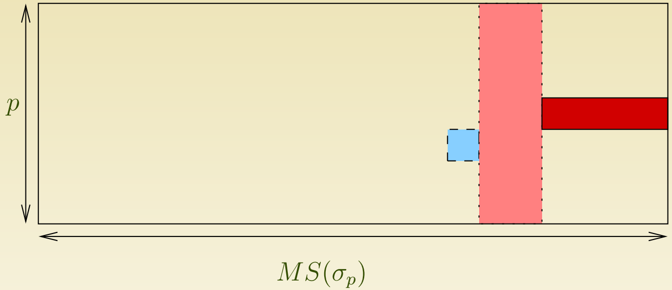
List Scheduling: proving the Coffman result



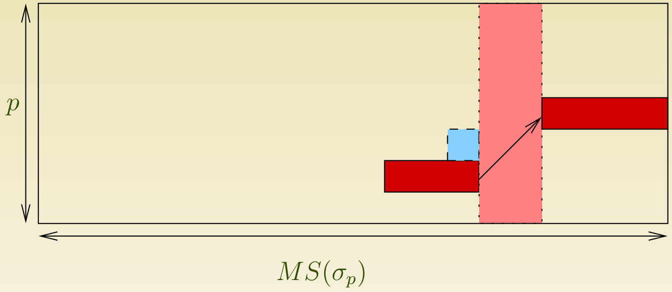
List Scheduling: proving the Coffman result



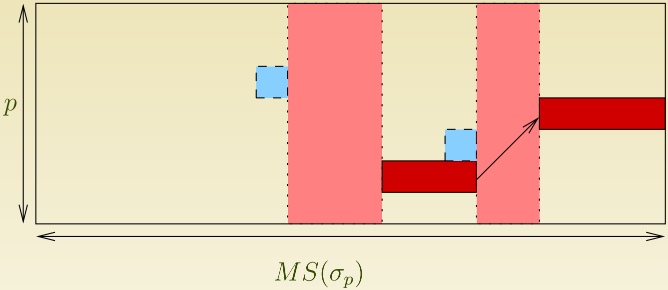
List Scheduling: proving the Coffman result



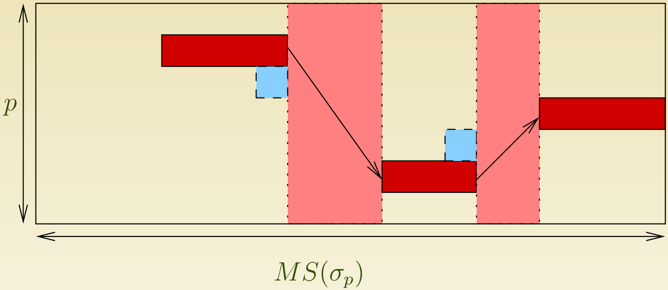
List Scheduling: proving the Coffman result



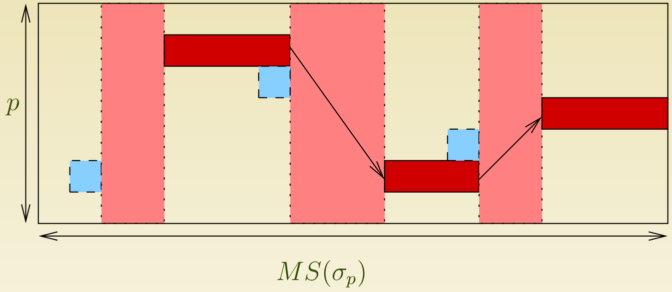
List Scheduling: proving the Coffman result



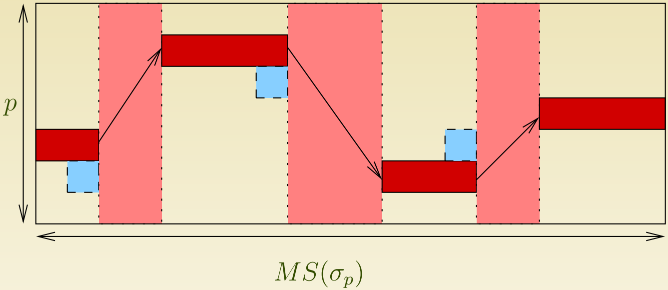
List Scheduling: proving the Coffman result



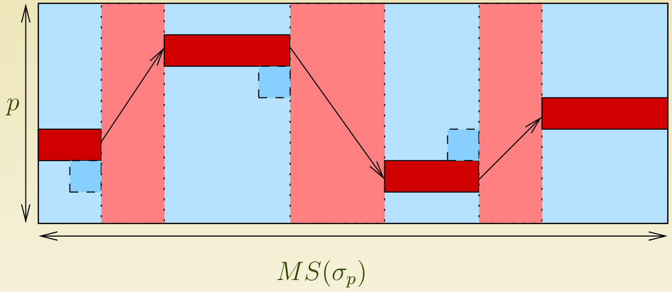
List Scheduling: proving the Coffman result



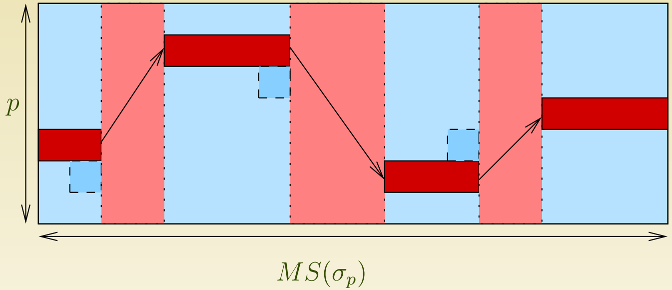
List Scheduling: proving the Coffman result



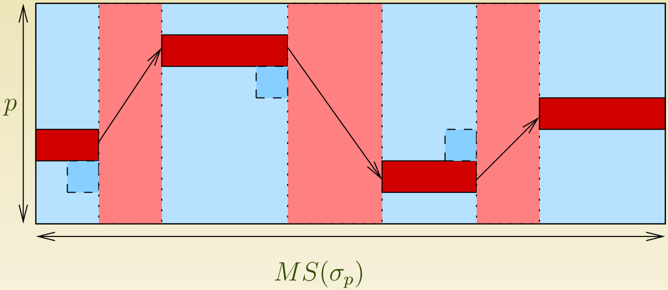
List Scheduling: proving the Coffman result



List Scheduling: proving the Coffman result

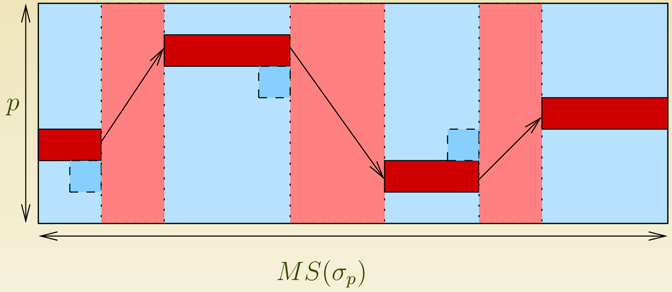


Therefore, $Idle \leqslant (p-1).w(\Phi)$ for some Φ



Therefore, $Idle \leqslant (p-1).w(\Phi)$ for some Φ
Hence,

$$\begin{aligned} p.MS(\sigma_p) &= Idle + Seq \leqslant (p-1)w(\Phi) + Seq \\ &\leqslant (p-1)MS_{opt}(p) + p.MS_{opt}(p) = (2p-1)MS_{opt}(p) \end{aligned}$$

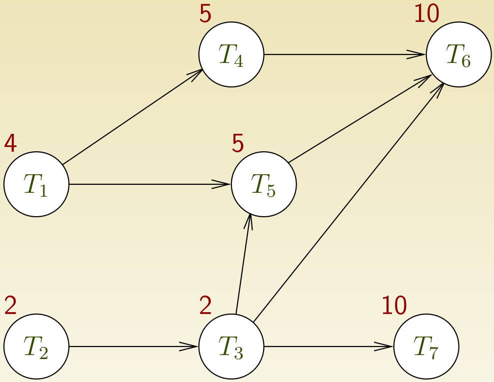


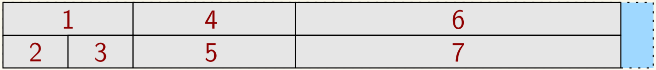
Therefore, $Idle \leqslant (p-1).w(\Phi)$ for some Φ
Hence,

$$\begin{aligned} p.MS(\sigma_p) &= Idle + Seq \leqslant (p-1)w(\Phi) + Seq \\ &\leqslant (p-1)MS_{opt}(p) + p.MS_{opt}(p) = (2p-1)MS_{opt}(p) \end{aligned}$$

One can actually prove that this bound cannot be improved.

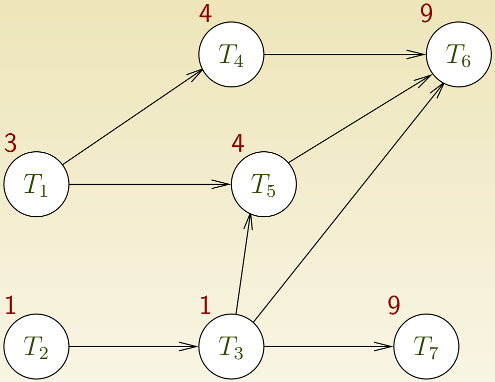
List scheduling Anomalies

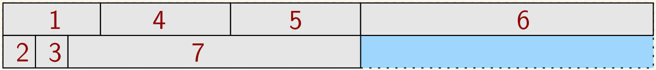




$$MS = 19$$

List scheduling Anomalies





$$MS = 20$$

List Scheduling for Parallel Rigid Tasks

Let us assume we have n independent rigid jobs $J_1 = (p_1, q_1), \ldots, J_n = (p_n, q_n)$ and m machines.
Let us denote by T^* the optimal makespan for this instance.

List Scheduling for Parallel Rigid Tasks

Let us assume we have n independent rigid jobs $J_1 = (p_1, q_1), \dots, J_n = (p_n, q_n)$ and m machines.

Let us denote by T^* the optimal makespan for this instance.

Let us consider a list schedule of makespan T. Let us denote by $q(t)$ the number of active processors at time t.

We have $\forall t_1, t_2 \in [0, T] : t_1 \leqslant t_2 - T^* \Rightarrow q(t_1) + q(t_2) > m$ (otherwise, the tasks running at time t_2 could have been run at time t_1).

Let us assume we have n independent rigid jobs $J_1 = (p_1, q_1), \ldots, J_n = (p_n, q_n)$ and m machines.

Let us denote by T^* the optimal makespan for this instance.

Let us consider a list schedule of makespan T. Let us denote by $q(t)$ the number of active processors at time t.

We have $\forall t_1, t_2 \in [0, T] : t_1 \leqslant t_2 - T^* \Rightarrow q(t_1) + q(t_2) > m$ (otherwise, the tasks running at time t_2 could have been run at time t_1).

Let us assume that $T > 2T^*$. Then we have:

$$mT^* \geqslant \sum_i q_i p_i = \int_0^T q(t) = \int_0^{2T^*} q(t) + \int_{2T^*}^T q(t)$$

$$\geqslant \underbrace{\int_0^{T^*} q(t) + q(t + T^*)}_{>mT^*} + \underbrace{\int_{2T^*}^T q(t)}_{\geqslant 0}, \text{which is absurd.}$$

Let us assume we have n independent rigid jobs $J_1 = (p_1, q_1), \ldots, J_n = (p_n, q_n)$ and m machines.

Let us denote by T^* the optimal makespan for this instance.

Let us consider a list schedule of makespan T. Let us denote by $q(t)$ the number of active processors at time t.

We have $\forall t_1, t_2 \in [0, T] : t_1 \leqslant t_2 - T^* \Rightarrow q(t_1) + q(t_2) > m$ (otherwise, the tasks running at time t_2 could have been run at time t_1).

Let us assume that $T > 2T^*$. Then we have:

$$mT^* \geqslant \sum_i q_i p_i = \int_0^T q(t) = \int_0^{2T^*} q(t) + \int_{2T^*}^T q(t)$$

Theorem 5.

List-scheduling has a approximation factor of 2 for minimizing the Cmax of Parallel Rigid Tasks.

Taking Communications into Account

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

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

Taking Communications into Account

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.

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

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

Taking Communications into Account (cont'd)

Even Pb(∞) is NP-complete !!!

You constantly have to figure out whether you should use more processor (but then pay more fore communications) or not. Finding the good trade-off is a real challenge.
$4/3$-approximation if all communication times are smaller than computation times.
Finding guaranteed approximations for other settings is really hard, but really useful (file staging).

Results More Related to Job Scheduling

	$model = \emptyset$	$model = pmtn$
$\langle 1 \vert r_j; model \vert \max w_j F_j \rangle$	NP([3])	$\downarrow$
$\langle P \vert r_j; model \vert \max w_j F_j \rangle$	$\uparrow$	$\downarrow$
$\langle Q \vert r_j; model \vert \max w_j F_j \rangle$	$\uparrow$	$\downarrow$
$\langle R \vert r_j; model \vert \max w_j F_j \rangle$	$\uparrow$	P(Lin. Prog)
$\langle 1 \vert r_j; model \vert \sum F_j \rangle$	NP([9])	P(SRPT [1])
$\langle P \vert r_j; model \vert \sum F_j \rangle$	$\uparrow$	NP(Numerical-3DM [2])
$\langle Q \vert r_j; model \vert \sum F_j \rangle$	$\uparrow$	$\uparrow$
$\langle R \vert r_j; model \vert \sum F_j \rangle$	$\uparrow$	$\uparrow$
$\langle 1 \vert r_j; model \vert \sum S_j \rangle$	NP	?
$\langle P \vert r_j; model \vert \sum S_j \rangle$	$\uparrow$	?
$\langle Q \vert r_j; model \vert \sum S_j \rangle$	$\uparrow$	?
$\langle R \vert r_j; model \vert \sum S_j \rangle$	$\uparrow$	?
$\langle 1 \vert r_j; model \vert \sum w_j F_j \rangle$	NP([9])	NP(Numerical-3DM [8])
$\langle P \vert r_j; model \vert \sum w_j F_j \rangle$	$\uparrow$	$\uparrow$
$\langle Q \vert r_j; model \vert \sum w_j F_j \rangle$	$\uparrow$	$\uparrow$
$\langle R \vert r_j; model \vert \sum w_j F_j \rangle$	$\uparrow$	$\uparrow$

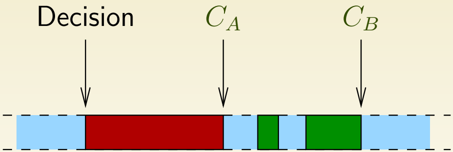
Significance of These Results

- In the previous table we saw that with preemption many problems become "easier".
  This is probably a good indication that the only hope to optimize a "user centric" performance metric is to allow preemption.
  Gang scheduling does preemption! Perhaps one can do just a little bit of preemption and be ok?
- Also, all the previous results are for off-line situations, when we know EVERYTHING about the stream of tasks/jobs.
  What about the on-line case?
  Competitive ratio: How close does an on-line scheduling algorithm come to the optimal offline algorithm in the worst case.

Flow Minimization (Sum Flow)

$\langle 1|r_j; pmtn|\sum F_j\rangle$ One processor, preemption is allowed, release dates, minimize average flow-time.

Shortest Remaining Processing Time is optimal: Upon job arrival/departure, ensure that the job with the shortest remaining processing time has the processor ($\leadsto$ use preemption).



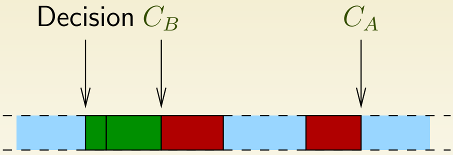
NP-complete for multiple processors or with no preemption.

Approximation Algorithm with logarithmic competitive ratio on multiple processors exists.

Flow Minimization (Sum Flow)

$\langle 1|r_j;pmtn|\sum F_j \rangle$ One processor, preemption is allowed, release dates, minimize average flow-time.

Shortest Remaining Processing Time is optimal: Upon job arrival/ departure, ensure that the job with the shortest remaining processing time has the processor ($\rightsquigarrow$ use preemption).



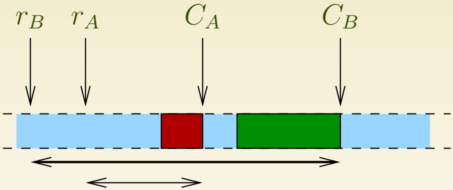
NP-complete for multiple processors or with no preemption.

Approximation Algorithm with logarithmic competitive ratio on multiple processors exists.

Flow Minimization (Max Flow)

$\langle 1|r_j; pmtn|F_{\max}\rangle$ One processor, preemption is allowed, release dates, minimize maximum flow-time.

First Come First Served is optimal ($\rightsquigarrow$ preemption is not needed).

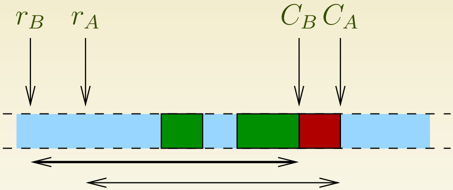


NP-complete for multiple processors when preemption is not allowed.

Flow Minimization (Max Flow)

$\langle 1|r_j; \mathit{pmtn}|F_{\max}\rangle$ One processor, preemption is allowed, release dates, minimize maximum flow-time.

First Come First Served is optimal ($\leadsto$ preemption is not needed).



NP-complete for multiple processors when preemption is not allowed.

Stretch Minimization (Max Stretch)

$\langle 1|r_j; pmtn|S_{\max}\rangle$ One processor, preemption is allowed, release dates, minimize maximum slowdown.

Offline algorithm based on linear programming and/or deadlines (preemption is needed).

Online algorithm There is no $\frac{1}{2}\Delta^{\sqrt{2}-1}$-competitive algorithms for max-stretch (where Δ is the ratio between largest processing time and the smallest processing time).
There are deadline-based online algorithms that are $O(\sqrt{\Delta})$-competitive for max-stretch [3, 4].

FCFS is Δ competitive for $S_{\max}$

Two job-sizes then the best known competitive ratio is $\frac{1+\sqrt{5}}{2}$ and $\sqrt{2}$ is an upper bound on the competitive ratio.

Stretch Minimization (Sum Stretch)

$\langle 1|r_j; \textit{pmtn}|S_{\max}\rangle$ One processor, preemption is allowed, release dates, minimize average slowdown.

Complexity is open (offline)

SRPT is 2-competitive.

FCFS is Δ^2-competitive.

NP-complete when preemption is not allowed.

On a single processor minimizing sum-flow is easier than minimizing sum-stretch.

On multiple processors SRPT is 14-competitive.

And so on...

A large literature with results here and there. Max-stretch/Max-flow is kind of about "fairness", Sum- stretch/Sum-flow is kind of about "performance" $\rightsquigarrow$ It would be nice to sort of optimize both.

And so on...

A large literature with results here and there. Max-stretch/Max-flow is kind of about "fairness", Sum- stretch/Sum-flow is kind of about "performance" $\rightsquigarrow$ It would be nice to sort of optimize both.

Depressing result:

Theorem 6.

Any $\rho(\Delta)$-competitive algorithm for AF such that $\rho(\Delta) < \Delta$ (i.e. more clever than FCFS) leads to starvation.

Theorem 7.

Any $\rho(\Delta)$-competitive algorithm for AS such that $\rho(\Delta) < \Delta^2$ (i.e. more clever than FCFS) leads to starvation.

And so on...

A large literature with results here and there. Max-stretch/Max-flow is kind of about "fairness", Sum- stretch/Sum-flow is kind of about "performance" $\rightsquigarrow$ It would be nice to sort of optimize both.
Depressing result:

Theorem 6.

Any $\rho(\Delta)$-competitive algorithm for AF such that $\rho(\Delta) < \Delta$ (i.e. more clever than FCFS) leads to starvation.

Theorem 7.

Any $\rho(\Delta)$-competitive algorithm for AS such that $\rho(\Delta) < \Delta^2$ (i.e. more clever than FCFS) leads to starvation.

In Practice

Being good for a sum-based metric is easy (smaller or weighted smaller first).
Relaxed deadline-based approaches are good for max-based metrics.

Outline

Conclusion

Theory Most of the time, the only thing we can do is to compare heuristics. There are three ways of doing that:

- Theory: being able to guarantee your heuristic;
- Experiment: Generating random graphs and/or typical application graphs along with platform graphs to compare your heuristics.
- Smart: proving that your heuristic is optimal for a particular class of graphs (fork, join, fork-join, bounded degree, . . .).

However, remember that the first thing to do is to look whether your problem is NP-complete or not. Who knows? You may be lucky...

Practice We do batch scheduling, which completely disregards all this. But theory says that preemption is key.

As usual there is a major disconnect. Only a few authors have read both types of work.

Great opportunity for research is there anything from the theory that should guide the practice?

K. Baker.
Introduction to Sequencing and Scheduling.
Wiley, New York, 1974.

P. Baptiste, P. Brucker, M. Chrobak, C. Durr, S. A. Kravchenko, and F. Sourd.
The complexity of mean flow time scheduling problems with release times, 2006.
Available at http://arxiv.org/abs/cs/0605078.

M. A. Bender, S. Chakrabarti, and S. Muthukrishnan.
Flow and stretch metrics for scheduling continuous job streams.
In *Proceedings of the 9th Annual ACM-SIAM Symposium On Discrete Algorithms (SODA'98)*, pages 270–279. Society for Industrial and Applied Mathematics, 1998.
Available at http://citeseer.nj.nec.com/bender98flow.html.

M. A. Bender, S. Muthukrishnan, and R. Rajaraman.
Improved algorithms for stretch scheduling.

In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 762–771, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

A. Bernstein.
Analysis of programs for parallel processing.
IEEE Transactions on Electronic Computers, 15:757–762, Oct. 1966.

P. Brucker.
Scheduling Algorithms.
Springer, Heidelberg, 2 edition, 1998.

E. G. Coffman.
Computer and job-shop scheduling theory.
John Wiley & Sons, 1976.

J. Labetoulle, E. L. Lawler, J. Lenstra, and A. Rinnooy Kan.
Preemptive scheduling of uniform machines subject to release dates.

In W. R. Pulleyblank, editor, *Progress in Combinatorial Optimization*, pages 245–261. Academic Press, 1984.

J. Lenstra, A. Rinnooy Kan, and P. Brucker.
Complexity of machine scheduling problems.
Annals of Discrete Mathematics, 1:343–362, 1977.