Multi-organization scheduling

Denis TRYSTRAM

June, 2010

Outline

Outline

Context: emergence of new HPC platforms

The evolution of high-performance execution platforms leads to physical or logical distributed entities (organizations) which have their own local rules. Each organization is composed of multiple users who compete for the resources, and they aim at optimizing their own objectives. Such systems are often hierarchical (many-core).

Proposal:

To create a general framework for studying the resource allocation problem for most situations corresponding to actual parallel platforms and to propose efficient solutions for some of these problems.
Oriented towards theoretical analysis.

Multi-organization scheduling

The target multi-organization scheduling problem is generic and corresponds to many possible situations.
Informally, a set of users have some applications to execute on distributed resources. These resources belong potentially to multiple organizations that may have their local control and rules.
The objectives of the users are not necessarily the same, but they are related to a metric on the completion times (i.e. the finishing times) of the jobs.

Synthetic view of the problem

More formally

The problem is to allocate the jobs to the available resources according of a certain objective. Then, the jobs are scheduled locally.

Both problems correspond to determine two functions π (allocation) and σ (schedule).

The set of jobs is available at time 0, the execution is performed by a series of batches or by successive time frames.

Notations

- m machines
- K clusters, i-th cluster owns m_i processors. Sometimes, the clusters correspond to organizations.
- n jobs with weights p_j for $1 \leq j \leq n$ (and resource requirements q_j in case of parallel jobs)
- N users owing each n_i jobs
- $[m]$ is the set of machines, $[n]$ is the set of jobs

Classification of problems

Key parameters

- **Users:** single or multiple, uniform or heterogeneous
- **Type of applications (jobs):** sequential, parallel (rigid or malleable), divisible loads
- **Resources:** single, identical, hierarchical, heterogeneous
- **Control:** centralized or distributed
- **Objectives:** related to metrics involving the completion times, Cmax, ΣC_i, stretch

Methodology

- Hypothesis (and examples if needed)
- Formal definition of the problem
- Complexity analysis (including inapproximability)
- Algorithm(s)
- Analysis (worst case bounds or in average by the way of simulations/experiments)
- Synthesis (related works, practical issues, remaining open variants)

Outline

1. Introduction and Motivation
2. Classical Scheduling
3. Strip Packing
4. Distributed list Scheduling
5. Basic multi-organization Scheduling
6. Multiple strip Packing
7. Multiple organizations with parallel jobs

Starting smoothly with well-known results

A preliminary basic problem

- **Users:** **single** or multiple, uniform or heterogeneous
- **Jobs:** **sequential**, parallel (rigid or malleable), divisible loads
- **Resources:** single, **identical** , hierarchical, heterogeneous
- **Control:** **centralized** or distributed
- **Objectives:** **Cmax**, ΣC_i, stretch

Classical P-Cmax problem

Informally, this corresponds to the situation of a single (homogeneous) cluster.
Scheduling *n* independent jobs on *m* arbitrary parallel identical processors aiming at minimizing Cmax.

Complexity:

The problem is weakly NP-hard [Ullman 75].

PTAS for Pm,,Cmax [Tutorial Woeginger] (here, *m* is fixed).
Dynamic programming scheme leads to a PTAS [Hochbaum and Shmoys].

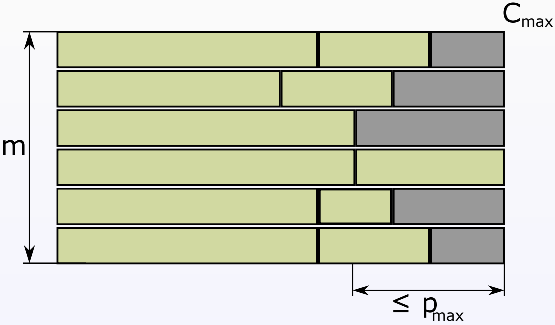
List scheduling

Algorithm framework:
List-scheduling [Graham 69] (greedy) whose principle is to build a list of ready jobs, and to execute any of these jobs as soon there are available processors. This algorithm has a guarantee in the worst case.

Remarks:
(asymptotically) optimal algorithm for a large number of jobs. It works also purely on-line algorithms.

Analysis 1

The idea is based on a geometrical proof on the Gantt chart:



$m.Cmax = W + S_{idle}$ (where $W = \Sigma_j p_j$)

Analysis 2

Proposition.

List scheduling is a 2-approximation.

$m.Cmax = W + S_{idle}$

Lower bounds:

$Cmax^* \geq \frac{W}{m}$ and $Cmax^* \geq p_{max}$

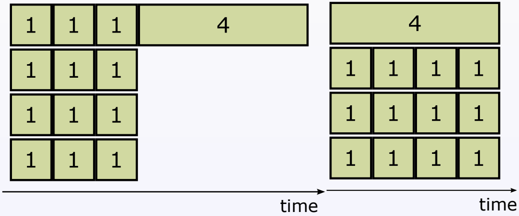
$S_{idle} \leq (m-1).p_{max} \leq (m-1).Cmax^*$

$Cmax = \frac{W}{m} + \frac{S_{idle}}{m} \leq (1 + \frac{m-1}{m}).Cmax^*$

Tightness for general list scheduling

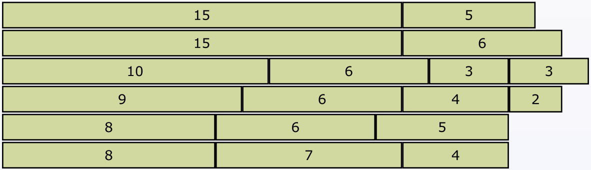
Proposition.

The worst case bound of $2 - \frac{1}{m}$ for list scheduling is tight.



LPT rule

Based on the tightness of the 2-approximation ratio, we can improve the bound by considering the specific LPT policy (largest first):

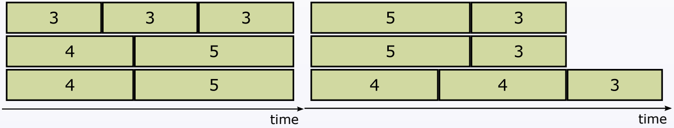


Approximation bound: $\frac{4}{3}$ (for $m \geq 2$)

Tightness of LPT

Proposition.

The worst case bound of $\frac{4}{3} - \frac{1}{3m}$ for LPT is tight.



$\frac{4}{3} - \frac{1}{3m} = \frac{11}{9}$ for $m = 3$.

Synthesis

List is a very nice framework which realizes a good trade-off between simplicity and efficiency.
It can be extended to many cases, sometimes it is possible to analyze theoretically.

- other objectives (ΣC_i with the "reverse" SPT policy wich is optimal for n independent jobs on m machines)
- taking into account communication costs
- Parallel rigid jobs

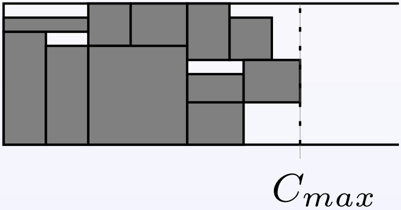
Outline

Problem

- **Users:** **single** or multiple, uniform or heterogeneous
- **Type of applications:** sequential, **parallel (rigid** or malleable), divisible loads
- **Resources:** single, **identical**, hierarchical, heterogeneous
- **Control:** **centralized** or distributed
- **Objectives:** **Cmax**, $\sum C_i$, stretch

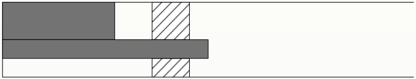
Problem statement

- **Problem:** Given *n* independent rigid tasks and 1 cluster with *m* machines, schedule all the jobs minimizing the makespan *Cmax*.



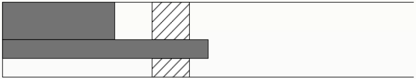
Continous Vs non-continuous, that is the question

- Rigid job scheduling

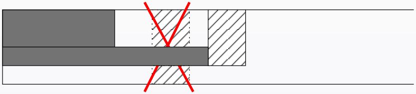


Continous Vs non-continuous, that is the question

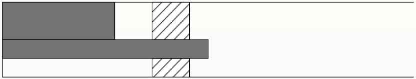
- Rigid job scheduling



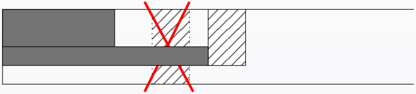
- Rectangle packing = rigid job continuous scheduling



Continous Vs non-continuous, that is the question

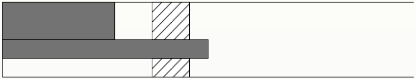


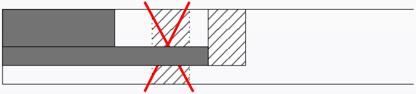
- Rigid job scheduling



- Rectangle packing = rigid job continuous scheduling
  - Algorithms for non continuous case generally donot apply to continuous case but ..

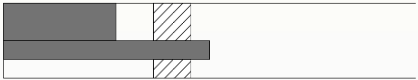
Continous Vs non-continuous, that is the question



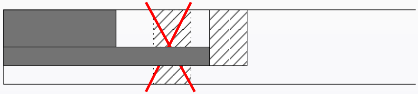


- Rigid job scheduling
- Rectangle packing = rigid job continuous scheduling
  - Algorithms for non continuous case generally donot apply to continuous case but ..
  - Proofs for continuous case may not apply to non contiguous case, as the non contiguous optimal could be smaller than the continous one

Continous Vs non-continuous, that is the question



- Rigid job scheduling



- Rectangle packing = rigid job continuous scheduling
  - Algorithms for non continuous case generally donot apply to continuous case but ..
  - Proofs for continuous case may not apply to non contiguous case, as the non contiguous optimal could be smaller than the continous one
  - However, continuous case is generally harder, and approximation algorithm/proofs for continous case based on surfaces arguments apply also to non continuous case

Continous Vs non-continuous, that is the question

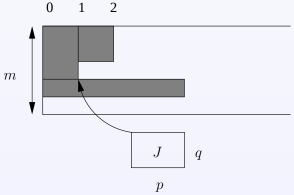
Two important remarks:

- Why does classical analysis of List Scheduling fails for continuous scheduling?

Continous Vs non-continuous, that is the question

Two important remarks:

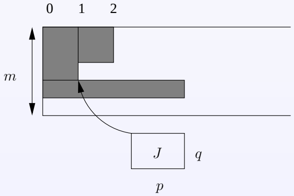
- Why does classical analysis of List Scheduling fails for continuous scheduling?
- Main argument for LS: J cannot be scheduled at time 1 $\Longrightarrow$ more than $m - q$ processors are busy at time 1



Continous Vs non-continuous, that is the question

Two important remarks:

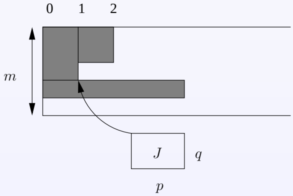
- Why does classical analysis of List Scheduling fails for continuous scheduling?
- Main argument for LS: J cannot be scheduled at time 1 $\Longrightarrow$ more than $m - q$ processors are busy at time 1
- No longer hold for continuous scheduling!

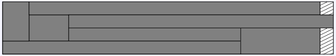


Continous Vs non-continuous, that is the question

Two important remarks:

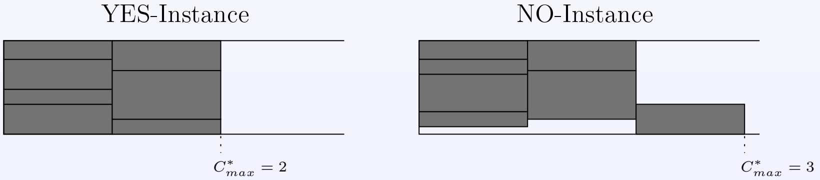
- Why does classical analysis of List Scheduling fails for continuous scheduling?
- Main argument for LS: J cannot be scheduled at time 1 $\Longrightarrow$ more than $m - q$ processors are busy at time 1
- No longer hold for continuous scheduling!
- **What is the gap between continuous and non continuous optimal?**





Overview of complexity results for one strip

- both versions (continuous or not) are $\frac{3}{2}$-inapproximable unless $P = NP$
- \+ LS is still a $(2 - \frac{1}{m})$-approx.. for non continuous case only!
- \+ Steinberg/Schiermeyer: fast 2-approx available for both versions
- \+ Jansen: very costly $(\frac{3}{2} + \epsilon)$-approx available for both versions
- \+ Kenyon-Remila: *AsymptoticFPTAS* available for both versions



Including extra rules

HF (Higest first) is a natural extension (similar to LPT).

Bad news: no better approximation as for general list (open question)
Good news: nce dominance rule (there exists a time moment for which the allocation before this moment is heavy loaded and the allocation after is weakly loaded).

Outline

1. Introduction and Motivation
2. Classical Scheduling
3. Strip Packing
4. **Distributed list Scheduling**
5. Basic multi-organization Scheduling
6. Multiple strip Packing
7. Multiple organizations with parallel jobs

Distributed control

coming back to sequential jobs

- **Users:** **single** or multiple, uniform or heterogeneous
- **Type of applications:** **sequential**, parallel (rigid or malleable), divisible loads
- **Resources:** single, **identical**, hierarchical, heterogeneous
- **Control:** centralized or **distributed**
- **Objectives:** **Cmax**, $\sum C_i$, stretch

Motivations

List scheduling is a nice and helpful technique. However, it is inefficient in case of fine grain computations for a large number of processors (because of the centralized nature of the list). Scheduling a big workload (bag of tasks) or n independent tasks on m arbitrary parallel identical processors aiming at minimizing $Cmax$.

Joint work with Marc Tchiboukdjian (paper available, just ask me)

Weakness of a central control

Most existing algorithms are based on the management of a global list of jobs.
Jobs generated by a running task are inserted into the list. When a processor is idle, it retrieves a job from the list.

Problem:
The list is accessed concurrently by several processors It should be protected by a lock (or we use a lock-free list).

There is a big overhead for managing the list.
This technique does not scale well in practice!

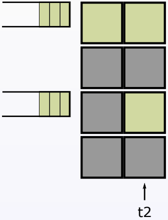
Description of the execution

Platform with m synchronized identical processors.

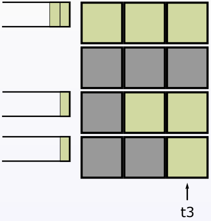
- Workload composed of n independent jobs with processing time p_j. Each processor i owns a list of jobs, put in the local queue Q_i.
- An active processor (non-empty list) executes one unit of work.
- An idle processor (sometimes called *thief*) randomly chooses another processor (victim). If the victims list is not empty, the thief steals "half" of the tasks and resumes execution at the next time slot. Otherwise, it tries again at the next time slot.
- Contention on lists: if several thieves target the same victim a random succeed, others fail.

Example

Example



Example



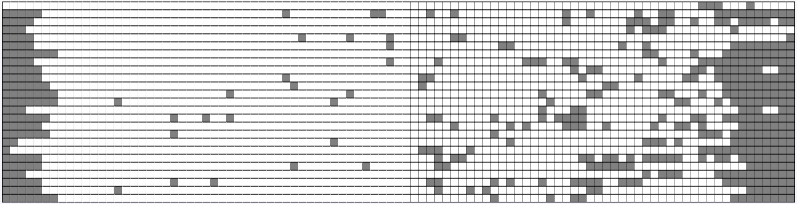
Notations

At time t

- $w_i(t)$ is the amount of work in queue Q_i, i.e. $w_i(t) = \sum_{j \in Q_i(t)} p_j$, and
- $w(t) = \sum_{i=1}^{m} w_i(t)$ is the total work in all the queues. The initial workload W is equal to $w(0)$ (n or Σp_i).
- $\alpha(t)$ is the number of active processors, i.e. the number of processors with a non-empty queue.

Principle of the analysis

Again, the analysis is based on the Gantt chart: the idle surface is bounded, here there is another factor. Below is a typical execution of $W = 2000$ unit independent tasks on $m = 25$ processors.



Potential function

First, we define a potential function $\Phi(t)$ that is, to a multiplicative factor, the variance of the queue sizes.
At time t, the potential function Φ is defined as:

$$\Phi(t) = \sum_{i=1}^{m} \left(w_i(t) - \frac{w(t)}{m} \right)^2$$

Moreover, let $\Delta\Phi(t) = \Phi(t) - \Phi(t+1)$.

Properties of Φ

$\Phi(t)$ is proportional to the variance of the queue sizes.

- When $\Phi(t) = 0$, we have $\forall i \; w_i(t) = \frac{w(t)}{m}$. No more work requests occurs until the end of the schedule as each processor has the same amount of work.
- The potential function is maximal when all the work is in a single queue.

$$\Phi(t) \leq \left(1 - \frac{1}{m}\right) \cdot w(t)^2 \leq \left(1 - \frac{1}{m}\right) \cdot W^2$$

- The potential function can also be written:

$$\Phi(t) = \sum_{i=1}^{m} w_i^2(t) - \frac{w^2(t)}{m}$$

Main results

- First, the expected *Cmax* applied on a bag of tasks W is equal to the absolute lower bound $\frac{W}{m}$ plus an additive term in $\frac{4e}{e-1} \log_2 W \leq 6.33 \log_2 W$.
  The analysis is tight up to a constant factor. It holds also when the jobs are not fully divisible which makes the analysis difficult when the number of jobs per queue is small.
- Second, we extend the previous analysis to weighted independent jobs with unknown processing times.
  The additive term becomes $O(\frac{p_{\max}}{p_{\min}} \cdot \log W)$ where $p_{\min}$ and $p_{\max}$ are the extremal values of the processing times.

Sketch of the proof for a bag of tasks

Decrease of the potential function for an active processor

Let $\delta_i(t)$ be the decrease of the potential function due to job execution and work requests on processor i at time t.

If processor i does not receive any work request at time t, we have $\delta_i(t) = 2w_i(t) - 1$.

If processor i receives at least one work request, we have $\delta_i(t) \geq w_i^2(t)/2 + w_i(t) - 1$.

Sketch of the proof for a bag of tasks

Decrease of the potential function in one step

The execution of one slot of the schedule decreases the potential function by:

$$\Delta\Phi(t) \geq \frac{1}{2} \cdot p_r(\alpha(t)) \cdot \Phi(t)$$

where $p_r(\alpha)$ is the probability that a processor receives a work request if $m - \alpha$ processors are idle:

$$p_r(\alpha) = 1 - \left(1 - \frac{1}{m-1}\right)^{m-\alpha}$$

Sketch of the proof for a bag of tasks

Finally, we can bound the expected number of work requests.

Proposition.

The expected makespan for W unit independent jobs scheduled by Distributed List Scheduling is bounded by

$$Cmax \leq \frac{W}{m} + \frac{2e}{e-1} \cdot \log_2 W + 1$$

This is optimal up to a constant factor in $\log_2 W$.

Extension for weighted jobs

Let us consider now the number of work requests for weighted independent tasks.
Each job j has a processing time p_j which is unknown. Let $p_{\min}$ and $p_{\max}$ be the minimum and maximum processing times.
During a work request, half of the tasks are transfered from the active processor to the idle processor.

Proposition.

The expected makespan for weighted independent jobs of total processing time W scheduled by DLS is bounded by

$$Cmax \leq \frac{W}{m} + O\left(\frac{p_{\max}}{p_{\min}} \cdot \log W + p_{\max} \cdot \log m\right)$$

Synthesis

As studied in this elementary case, the impact of distribution is high.
How to extend to hierarchical platforms?

Outline

1. Introduction and Motivation
2. Classical Scheduling
3. Strip Packing
4. Distributed list Scheduling
5. **Basic multi-organization Scheduling**
6. Multiple strip Packing
7. Multiple organizations with parallel jobs

Add local constraints

- **Users:** **single** or multiple, uniform or heterogeneous
- **Type of applications:** **sequential**, parallel (rigid or malleable), divisible loads
- **Resources:** single, **identical**, hierarchical, heterogeneous
- **Control:** **centralized** or distributed
- **Objectives:** **Cmax**, **ΣC_i**, stretch

Joint work with Johanne Cohen, Daniel Cordeiro and Frédéric Wagner (paper to appear in Euro-Par 2010)

Motivations

In the concept of grid computing, different *organizations* share processors and exchange jobs in order to maximize the profits of the whole community.

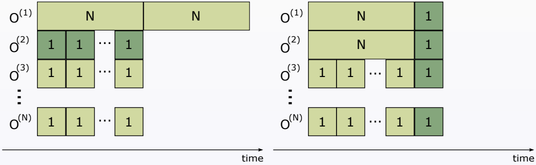
Locally, an organization can act *selfishly* and refuse to cooperate if in the final schedule one of its (migrated) jobs could be executed earlier in one of its own processors.

The focus here is to study the impact on the global performance ($Cmax$) of cooperation between selfish organizations. The local objectives are to minimize $Cmax$ or ΣC_i.

Notation:

MOSP($Cmax$) or MOSP(ΣC_i).

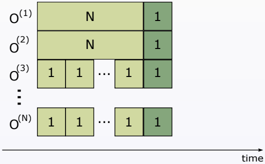
MOSP constraints



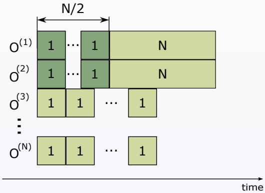
(a) Initial instance

(b) Global optimum without constraints

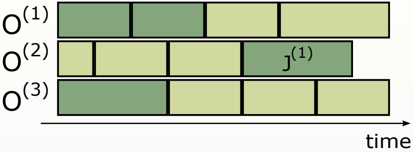
MOSP constraints



(c) Global optimum without constraints

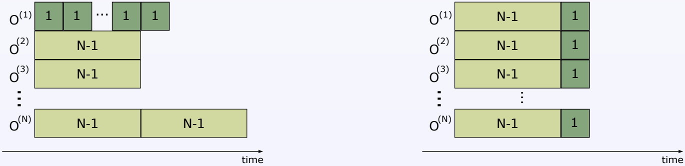


(d) Optimum with local constraints



Inapproximation.

Ratio between approximation algorithms with and without selfishness restrictions: $\geq 2 - \frac{2}{N}$



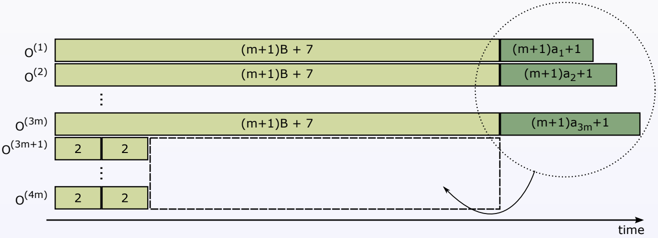
(e) Optimal with selfishness restrictions

(f) Optimal with local constraints

Complexity 1

MOSP(*Cmax*) is strongly NP-complete.

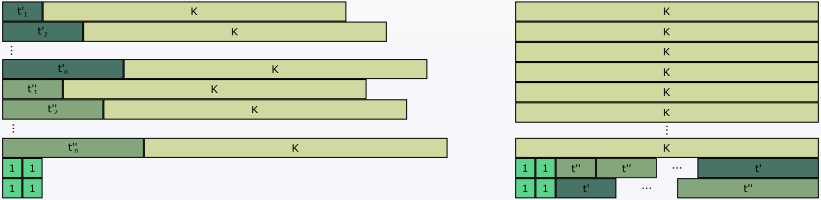
Proof. Reduction from 3-PARTITION.



Complexity 2

MOSP(ΣC_i) is NP-complete.

Proof. Reduction from PARTITION.



(g) Initial instance (h) Optimum

Approximation algorithms

The idea is to mix LPT and SPT for solving MOSP with selfish restrictions.

Phase 1: If solving MOSP($Cmax$), each organization applies LPT locally for its own jobs – or SPT if solving MOSP(ΣC_i).

Phase 2: Global LPT: each time an organization becomes idle, the longest job that does not have started yet is migrated and executed by the idle organization.

Analysis

Proposition:

MOSP is a 2-approximation for both objectives.

Proof.

- Phase 2 works as a list scheduling, so Graham's classical approximation ratio $2 - \frac{1}{N}$ holds for all of them.
- It is feasible since the migrated jobs are always executed earlier than the original schedule; this guarantees that the *selfishness restriction is always respected* and that both $Cmax$ and ΣC_i of the original organization is not increased;

Motivation for considering a Game

An alternative to classical combinatorial optimization is to study this problem by a non-cooperative game.
Game theory has some useful tools for considering cooperation and selfishness (each organization put emphasis on its own local objective).

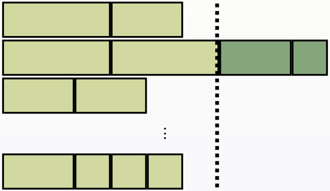
The problem here is to minimize the global *Cmax* of *N* organizations composed of one processor.

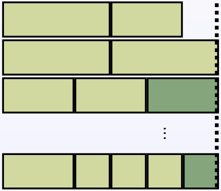
Game model

Each **player** k is an organization responsible for an “application” (a set of $n^{(k)}$ jobs) and wants to minimize its $cost^{(k)}$ (here $Cmax$ or ΣC_i of its jobs);

- Each organization applies locally a scheduling algorithm (LPT, SPT, etc.) with "my jobs first" policy (MJF);
- A strategy $S^{(k)}$ for player k is a vector of $n^{(k)}$ elements such that $S^{(k)}(i)$ (for $1 \leq i \leq n^{(k)}$) corresponds to the organization chosen by player k for job $J_i^{(k)}$;
- A configuration (profile) M is the vector $(S^{(1)}, S^{(2)}, \dots, S^{(N)})$ such that $S^{(k)}$ is a strategy of player k.

Game description





Nash equilibrium

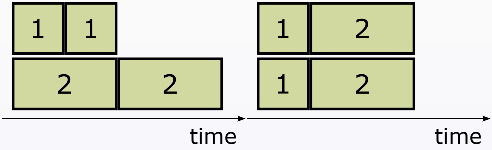
Definition:

A configuration $M = (S^{(1)}, S^{(2)}, \ldots, S^{(N)})$ is a Nash equilibrium if all the players (organizations indexed by k) satisfy the following property.

$\forall s \in \mathcal{S}^{(k)}, cost^{(k)}(M) \leq cost^{(k)}(s, M_{-k})$, where M_{-k} is a vector $(S^{(1)}, S^{(2)}, S^{(k-1)}, S^{(k+1)} \ldots, S^{(N)})$

Do there always exist Nash Equilibria for MOSP($Cmax$) or MOSP(ΣC_i)?
Preliminary results show that for this game there are instances that do not have pure Nash Equilibria.

Example of Nash equilibrium for $N = 2$



There exist several Nash equilibria. The first one does not correspond to a good (efficient) solution, the second one is optimal.

Price of Anarchy

As seen in the previous example, uncoordinated, selfish behaviour can lead to sub-optimal global makespan.
We are interested in studying the price of anarchy (ratio between the social cost of a worst-case Nash equilibrium and the social cost of an optimal assignment - $Cmax$).

Definition:
For any instance G of the MOSP game with N machines. Let Nash(G) denote the set of all strategy profiles that are Nash equilibria for G, and let opt(G) denote the minimum social cost over all the assignments. Then:

$$\mathrm{PoA(N)} = \max_G \max_{P \in \mathrm{Nash(G)}} \frac{cost(P)}{opt(G)}$$

Brief synthesis

Game theory with centralized coordination does not help too much (from the theoretical point of view).

It is hard to guaranty a bound on the convergence towards Nash equilibria (when they exist).

Outline

1. Introduction and Motivation
2. Classical Scheduling
3. Strip Packing
4. Distributed list Scheduling
5. Basic multi-organization Scheduling
6. **Multiple strip Packing**
7. Multiple organizations with parallel jobs

Problem

- **Users:** **single** or multiple, uniform or heterogeneous
- **Type of applications:** sequential, **parallel** (rigid or malleable), divisible loads
- **Resources:** single, identical, **hierarchical**, heterogeneous
- **Control:** **centralized** or distributed
- **Objectives:** related to metrics involving the completion times, **Cmax**, ΣC_i, stretch

Joint work with Marin Bougeret, Pierre-Francois Dutot, Klaus Jansen and Christina Otte

Scheduling rigid parallel jobs

Informally, this corresponds to the situation of a computational grid composed of several (homogeneous) clusters.

Outline

General multi-organization problem

- **Users: single** or multiple, uniform or heterogeneous
- **Type of applications:** sequential, **parallel rigid tasks**, divisible loads
- **Resources:** single, **identical**, **hierarchical**, heterogeneous
- **Control: centralized with local constraints** or distributed
- **Objectives: Cmax**, ΣC_i, stretch

Motivations

Let us consider now a computing platform comosed of several separate organizations. Parallel applications are submitted locally, but can be moved to other organizations if it helps to improve the local schedules.

Scheduling n parallel tasks on k clusters of identical m processors aiming at minimizing Cmax with distributed control.

Joint work with Pierre-Francois Dutot, Fanny Pascual and krzysztof Rzadca (preliminary version published in EuroPar 2007, extended version to appear in IEEE TPDS).