Practical Steady-State Scheduling for Tree-Shaped Task Graphs

Sékou DIAKITÉ[1], Loris MARCHAL[2], Jean-Marc NICOD[1], Laurent PHILIPPE[1] - 19/11/2009

1: Laboratoire d'Informatique de Franche-Comté
Université de France Comté, France

2: Laboratoire de l'Informatique du Parallélisme
CNRS - INRIA - Université de Lyon, France

INRIA GRAAL

Outline

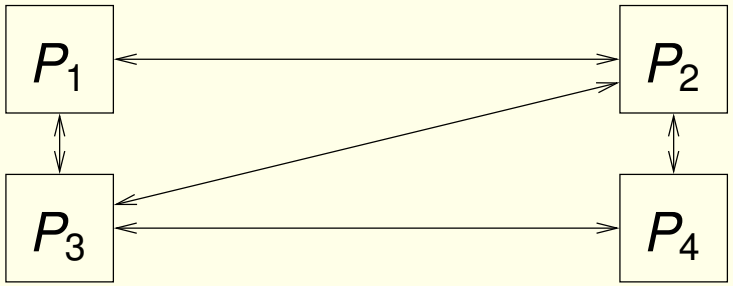
Scheduling problem

Definitions

Execution platform

- undirected graph, $G_p = (V_p, E_p)$
  - $V_p = \{P_1, ..., P_n\}$: n processors
  - E_p : communication links between the processors
    - bidirectional one-port model
    - $c_{i,j}$ is the time needed to send a unit of data from P_i to P_j

Example



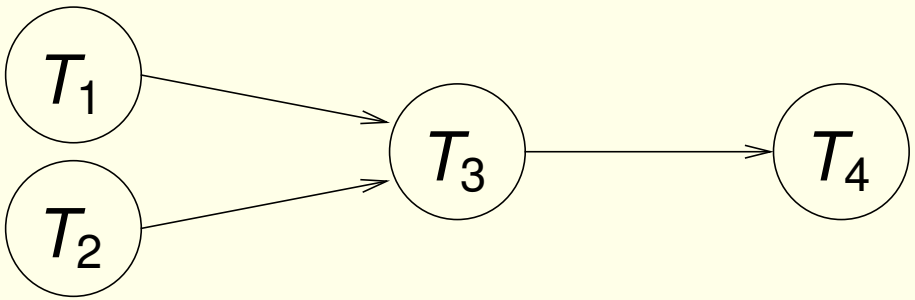
Scheduling problem

Definitions

Application

- DAG with no forks (in-trees), $G_a = (V_a, E_a)$
  - $V_a = \{T_1, ..., T_k\}$: k tasks
    - unrelated computation model, $w_{i,k}$: time needed by P_i to execute T_k
  - E_a dependencies between tasks
    - $F_{k,l}$ is the amount of data (File) produced by T_k and consumed by T_l

Example



10 to 1000 times

Scheduling problem

How to ?

Problem

Executing a batch of graphs (from 10 to 1000)

Objective

Minimizing the makespan C_{max}

Chosen method

Steady-state technique which is asymptotically optimal (throughput)

Outline

Principle of steady-state scheduling

Overview

This study is based on

O. Beaumont, A. Legrand, L. Marchal and Y. Robert. Steady-state scheduling on heterogeneous clusters. Int. J. of Foundations of Computer Science, 16(2) :163-194, 2005.

Principle of steady-state scheduling

Overview

Converting the scheduling problem to a linear program

- the steady-state is characterized by activities variables
  - the average number of T_k processed by P_i in one time unit
  - the average number of $F_{k,l}$ sent by P_i to P_j in one time unit
- these activities variables allow us to write constraints
  - on processor speeds and link bandwidths
  - "conservation laws" to state that $F_{k,l}$ has to be produced by T_k and consumed by T_l
- these constraints describe a valid steady-state schedule
- by adding the objective of maximizing the steady-state throughput, we obtain a linear program

Principle of steady-state scheduling

Overview

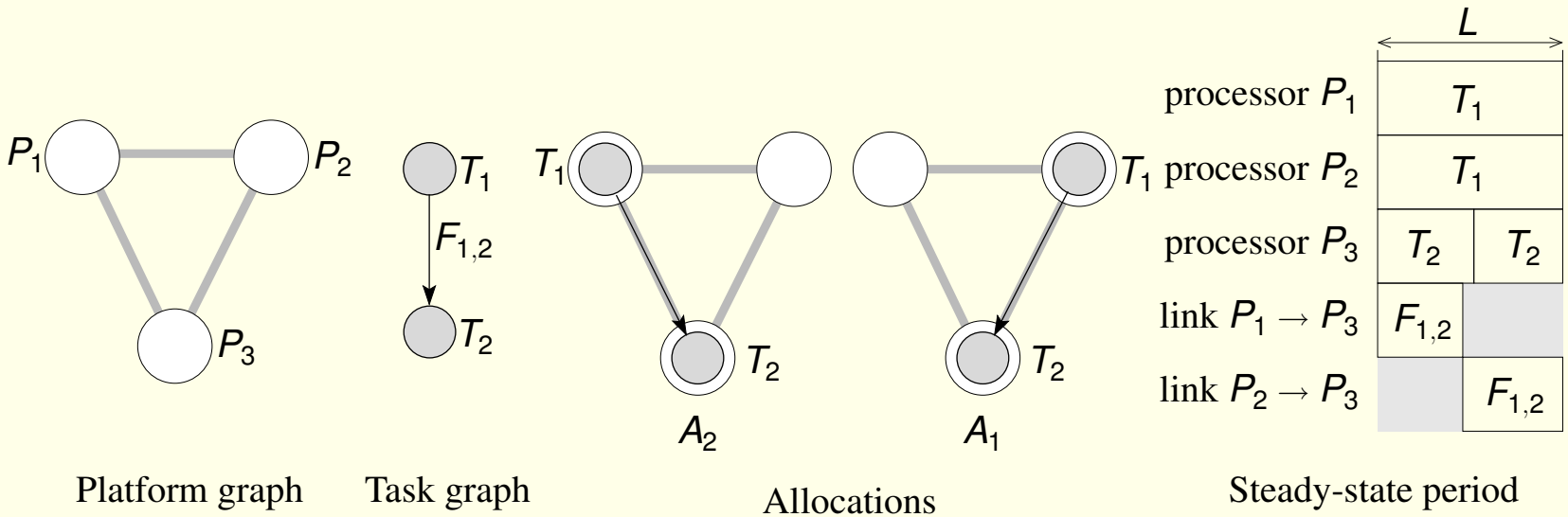
From the linear program to a periodic schedule (period)

- the optimal solution of the linear program gives rational activities
- we can not split tasks and files
  - $\rightarrow$ the period length L is equal to the LCM of activities denominators
  - $\rightarrow$ we multiply every activity by L, activities are now integers
- L is large but bounded
- the period allows us to schedule any number of graphs, the final schedule consists in 3 phases
  - initialization
  - steady-state : $n \times$ periods
  - clean-up

Principle of steady-state scheduling

Overview

Example



Platform graph Task graph Allocations Steady-state period

Outline

Principle of steady-state scheduling

Shortcomings

Long latency

- several periods are necessary to process an instance
  - → drawback for interactive applications
  - → lead to large buffers : at every time step, a large number of ongoing job has to be stored

Long initialization and clean-up phases

- the period contains a large number of ongoing job
  - → long initialization phase to enter steady-state
  - → long clean-up phase to leave steady-state
- initialization and clean-up are done with heuristic scheduling
  - → we lose the benefit of the optimal steady-state phase

Principle of steady-state scheduling

Shortcomings

Addressing the shortcomings

- the original steady-state algorithm reaches good C_{max} as soon as the number of instances is large enough
- in this study, we aim at reducing this threshold

Principle of steady-state scheduling

Addressing the shortcomings

Means of actions

- decrease the length of the period
  - hard to do when we want to keep an optimal period
- reduce the latency (inter/intra dependencies)
  - side benefit : less work to do in initialization and clean-up (gain on C_{max})
- reduce the period length by allowing a small reduction of the throughput
  - side benefit : reducing the latency

Outline

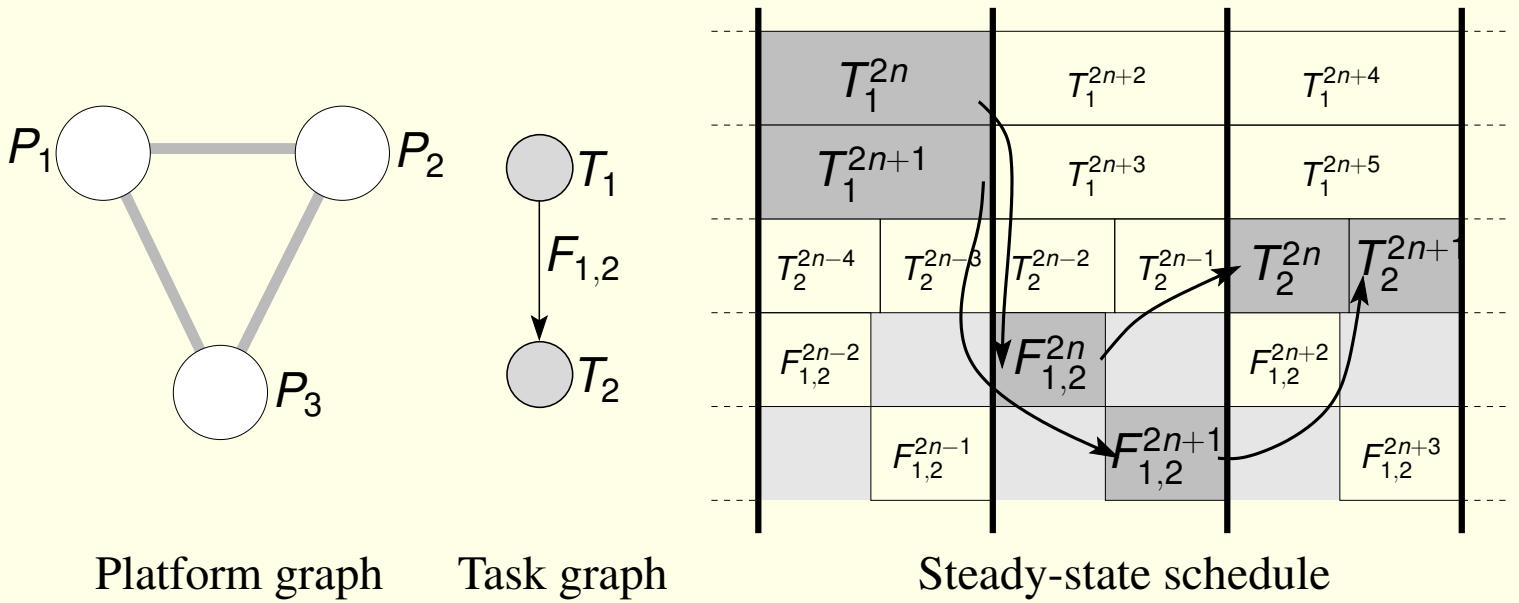
Reducing the latency

Dependencies

How the reduce the latency ?

Intra-period dependencies.

The original steady-state (only inter-period dependencies)

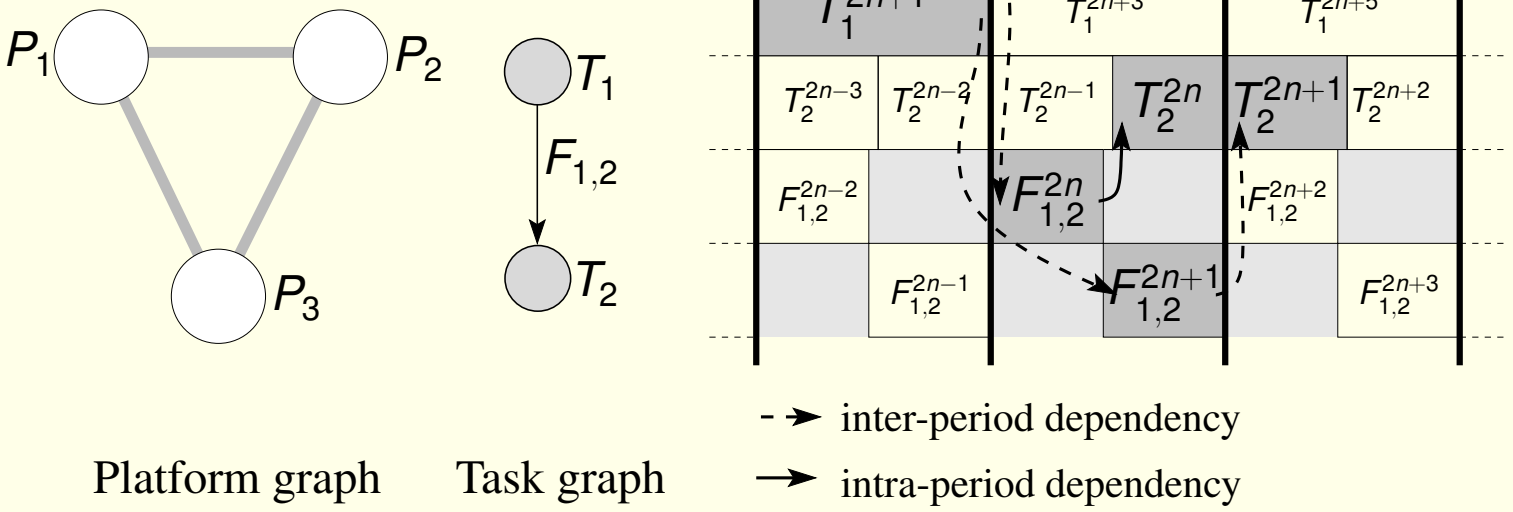


Platform graph Task graph Steady-state schedule

Reducing the latency

Dependencies

The steady-state with intra-period dependencies



Outline

Reducing the latency

Mixed Integer Program

Ordering

- Tasks (T_j, T_k) on the same processor P_i
  - binary variable $y_{j,k} = 1$ if and only if T_j is processed before T_k
  - t_j is the starting time of task T_j, L is the length of the period

$$t_j - t_k \geq -y_{j,k} \times L \tag{1}$$

$$y_{j,k} + y_{k,j} = 1 \tag{2}$$

$$t_k - (t_j + w_{i,j}) \geq (y_{j,k} - 1) \times L \tag{3}$$

$$t_j + w_{i,j} \leq L \tag{4}$$

Reducing the latency

Mixed Integer Program

Dependencies

- For each dependency $T_j \rightarrow T_k$
  - binary variable $e_{j,k} = 1$ intra-period dependency ($e_{j,k} = 0$ inter-period)

$$t_k - (t_j + w_{i,j}) \geq (e_{j,k} - 1) \times L \quad (5)$$

Objective

$$\begin{cases} \text{Maximize } \sum e_{j,k} \\ \text{under the constraints (1), (2), (3), (4) and (5)} \end{cases}$$

Outline

Reducing the latency

Heuristic approach

Limitations

The heuristic algorithm is not allowed to move tasks inside the period

Algorithme 1 : Heuristic algorithm

IntraDep $\leftarrow \{\}$;
Prod $\leftarrow$ set of all sources of the dependencies (sorted by completion time);
Cons $\leftarrow$ set of all destinations of the dependencies (sorted by starting time);
forall $T_{src} \in Prod$ **do**
  forall $T_{dst} \in Cons$ **do**
    if *There is a dependency* $T_{src} \rightarrow T_{dst}$ **then**
      if $end(T_{src}) \leq start(T_{dst})$ **then**
        remove T_{dst} from *Cons*;
        $IntraDep \leftarrow IntraDep \cup \{(T_{src} \rightarrow T_{dst})\}$;
        continue

Outline

Using non-conservative solutions

Motivation

How to reduce the period length ?

- one of the main drawbacks of the method
- solution of a linear program
  - hard to find an other optimal solution with a smaller period

→ modify the solution

A sub-optimal solution

- decrease the system throughput
- gain flexibility on the period length

→ our claim : much shorten the period at the cost of a slight reduction of the throughput

  - side benefits : shorter latencies and smaller buffers

Using non-conservative solutions

Principle

Steady state scheduling and allocations

- allocations : $A_1, \ldots, A_m$
- throughput of A_k $\rho_k = \alpha_k/\beta_k$ and total throughput $\rho = \sum_k \rho_k$
- period length $T = \text{lcm}_k \beta_k$

$\rightarrow$ in one period A_k processed $T \times \alpha_k/\beta_k \in \mathbb{N}$

Influence of large value of β_k

- contribution to a small amount the total throughput
- responsible of large period size

$\rightarrow$ suppress β_k from the computation of T (scale $\lfloor (\alpha_j \times T)/\beta_j \rfloor$)

- hope : loss in ρ compensated by a shorter value of T

Using non-conservative solutions

Algorithm

Algorithme 2 : Shorten the period of the steady state schedule

Data : N_{total} instances, m allocations A_i with the throughput α_i/β_i.
Parameters : $K = 0.25$ (ratio initialization/total) and $L = 0.85$ (maximum degradation).
Sort allocation by non-increasing β_k, so that $\beta_1 \geq \beta_2 \geq \cdots \geq \beta_m$
$N_{init} \leftarrow$ estimateInitTermJobCount$(\rho_1, \ldots, \rho_m)$
$i \leftarrow 1$; $\rho^{orig} = \sum_{k=1}^{m} \alpha_k/\beta_k$
while $i < m-1$ *and* $(N_{init}/N_{total} > K)$ *and* $(\rho > L \times \rho^{orig})$ **do**
  $T \leftarrow \text{lcm}\{\beta_i, \ldots, \beta_m\}$
  foreach *allocation* A_k *in* $\{A_1, \ldots A_m\}$ **do**
    $\rho_k^{\text{rollback}} \leftarrow \rho_k$ $\qquad$ $\rho_k \leftarrow \lfloor(\alpha_k \times T)/\beta_k\rfloor$
  $\rho \leftarrow \sum_{k=1}^{m} \rho_k$
  $N_{init} \leftarrow$ estimateInitTermJobCount$(\rho_1, \ldots, \rho_m)$
  $i \leftarrow i + 1$
if $(N_{init}/N_{total} \leq K)$ *or* $(\rho \leq L \times \rho^{orig})$ **then**
  foreach *allocation* A_k *in* $\{A_1, \ldots A_m\}$ **do** $\rho_k \leftarrow \rho_k^{\text{rollback}}$
return $(\rho_1, \ldots, \rho_m)$

Outline

Experimental results

Comparison of 6 algorithms

- the original steady-state implementation ;
- the steady-state with the reduction of inter-period using MIP (steady-state+MIP)
- the steady-state with the reduction of inter-period using the greedy heuristic (steady-state+greedy)
- the steady-state with the non-conservative period reduction (steady-state+suboptimal)
- the steady-state with both the greedy heuristic and the non-conservative period reduction (steady-state+heuristic+suboptimal)
- a classical list scheduling algorithm based on HEFT

Experimental results

Simulation settings

Simulator

- results are obtained with a simulator on top of SimGrid
- simulations of 200 random platform/application scenarios
- batches from 1 to 1000 task graphs
- MIP solving using CPLEX

Limitations

- the MIP solver was able to find a solution within 15 minutes for 142 SIMPLE scenarios
- in the GENERAL case we do not give the results for the MIP approach

Outline

Experimental results

Inter-period dependencies

Results

- SIMPLE case : the MIP solves 32% of the intra-period dependencies
- SIMPLE case : the heuristic solves 26% of the intra-period dependencies
- GENERAL case : the heuristic solves 25% of the intra-period dependencies

Notes

- both the MIP and the heuristic achieve good performances for the resolution of intra-period dependencies
- how does they perform on other metrics ?

Outline

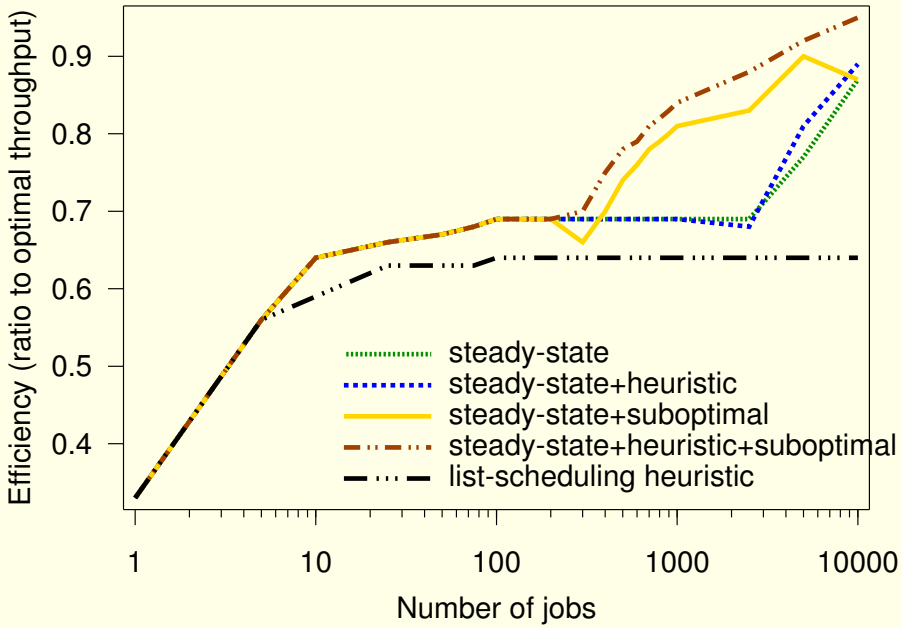
Experimental results

Scheduling efficiency

Efficiency

ratio to the optimal throughput obtained by an algorithm

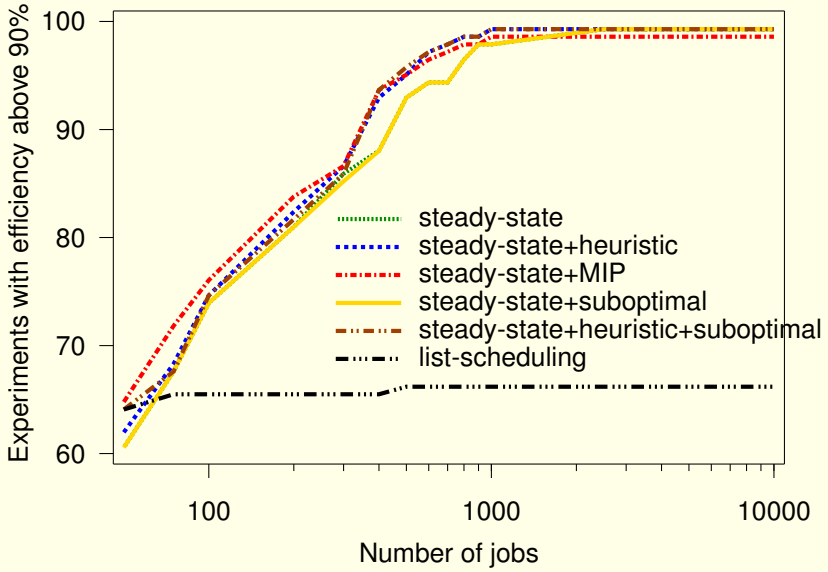
One complex example



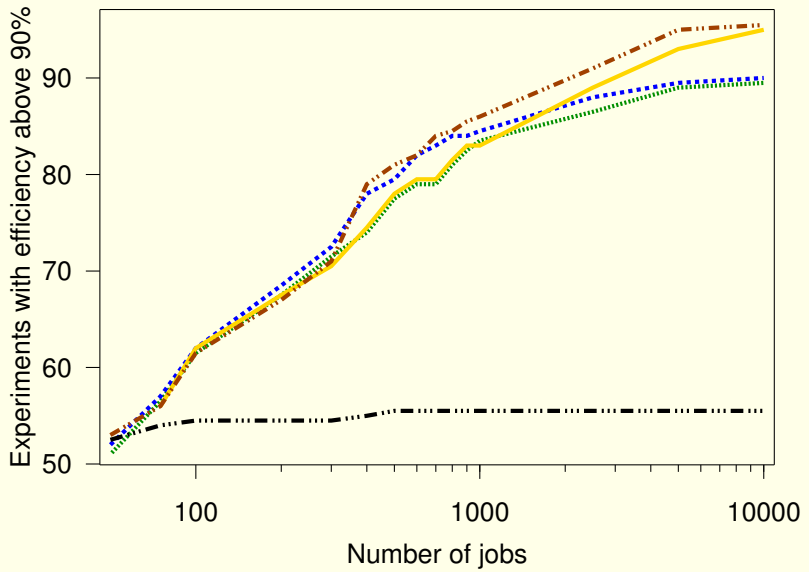
Experimental results

Scheduling efficiency

SIMPLE



GENERAL



Notes

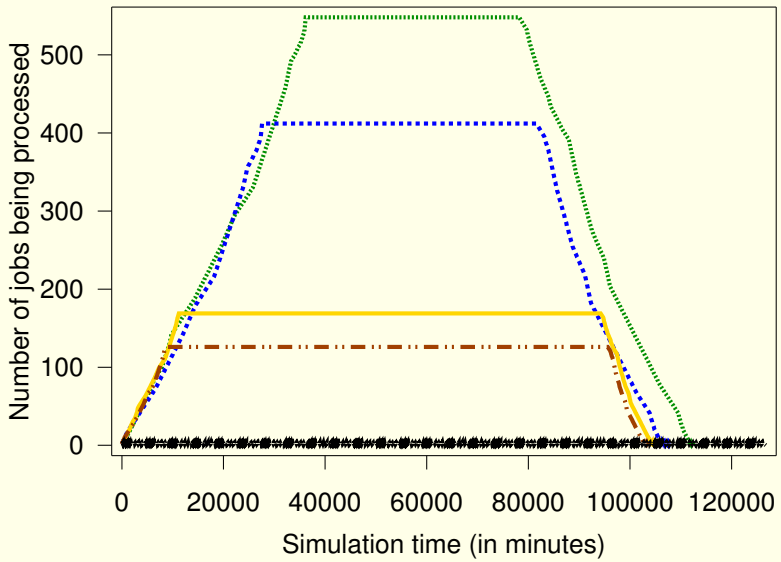
Proportion of scenarios where we reach 90% of the optimal throughput

Outline

Experimental results

Number of running instances

Example



Results

- SIMPLE case : MIP induces 30% less running instances
- SIMPLE case : heuristic induces 24% less running instances
- GENERAL case : heuristic + non-conservative induces 37% less running instances (548 down to 126)

→ shorten the buffer size

Experimental results

Latency and buffer size

SIMPLE and GENERAL

Algorithm	SIMPLE scenarios			GENERAL scenarios		
	average latency	maximum latency	max. num. of running jobs	average latency	maximum latency	max. num. of running jobs
MIP	94%	67%	70%	N/A	N/A	N/A
heuristic	95%	74%	76%	90%	90%	93%
suboptimal	100%	100%	100%	53%	93%	88%
heuristic+ suboptimal	95%	74%	75%	33%	67%	63%

TAB.: Performance of the algorithms in latency and buffer size relatively to original steady-state implementation. (Smaller latency and number of running jobs are better.)

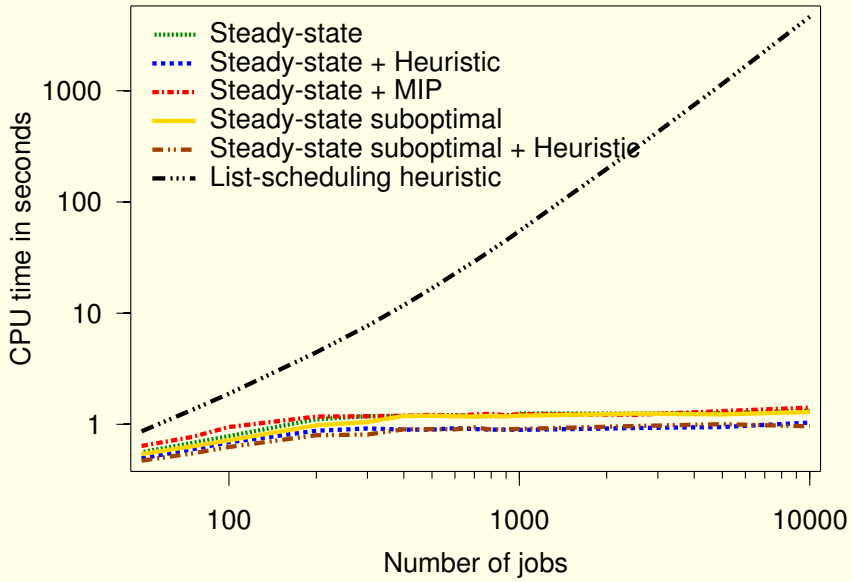
NB : GENERAL cases include SIMPLE cases too, so the decrease is important for complex cases

Outline

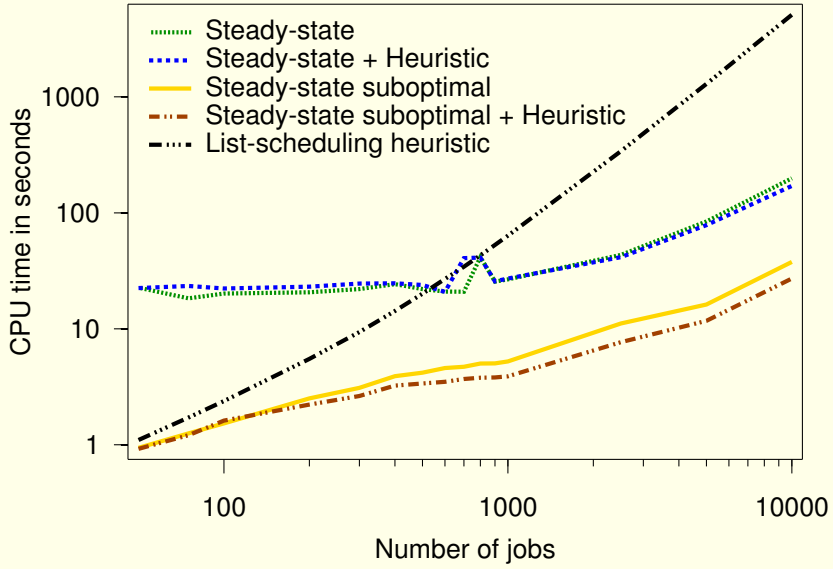
Experimental results

Running time of the algorithms

SIMPLE



GENERAL



Notes

Average CPU-time in seconds

Outline

Synthesis

Conclusion

Summary

- study of an adaptation of the steady state techniques in practical conditions :
  - medium size batches
  - performance metrics (throughput) and pratical interest (latency and buffer)
- two optimizations :
  - dependency reorganization (NP-Complet : MIP + heuristic)
  - shorten the period by decreasing the throughput ($< 15\%$)
- mesure of the impact of our optimizations (efficiency, buffer size, latency)

Conclusion

- steady-state scheduling is an efficient tool for dealing collections of task graphs

Synthesis

Future work

- steady state techniques and tolerance to the variation the platform capabilities
- evaluation onto a GRID context (cf MAO project)

Thank you for your attention

Questions ?