Scheduling algorithms for CPU and GPU altogether: algorithmic tips and tricks

Grégory Mounié

Univ. of Grenoble-Alpes
Grenoble-INP/Ensimag
Laboratoire d'Informatique de Grenoble (LIG - UMR 5217)
team-Inria project Datamove

2018-12-06
UNIFESP, São José dos Campos

Outline

Univ. Grenoble Alpes, Grégory Mounié, Ensimag

Univ. Grenoble Alpes

- 60 000 students, 0.7 M people in the city
- Oldest (and largest) French CS academic research center (1951-1952, 1500 people)

Grégory Mounié, Associate Professor

Teaching Operating Systems; UNIX; Concurrent programming; Distributed Systems; Write an OS from Scratch.

Research Scheduling for High Performance Computing (HPC)

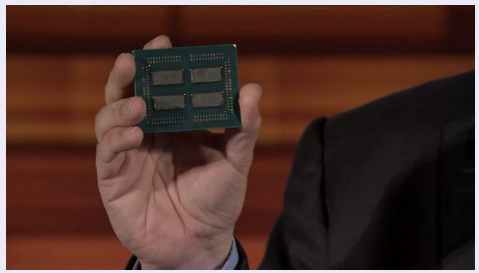
Hardware is becoming more and more complex

The end of frequency scaling induce complex techniques to improve performances

Out-of-order execution; Multi-core; Numa; Accelerators; Static routing in high speed network; Burst buffer; non-volatile RAM (NVRAM), etc.

Processors are not "single chip" anymore (again)

The 32-core AMD EPYC is build by gluing 4 8-core modules



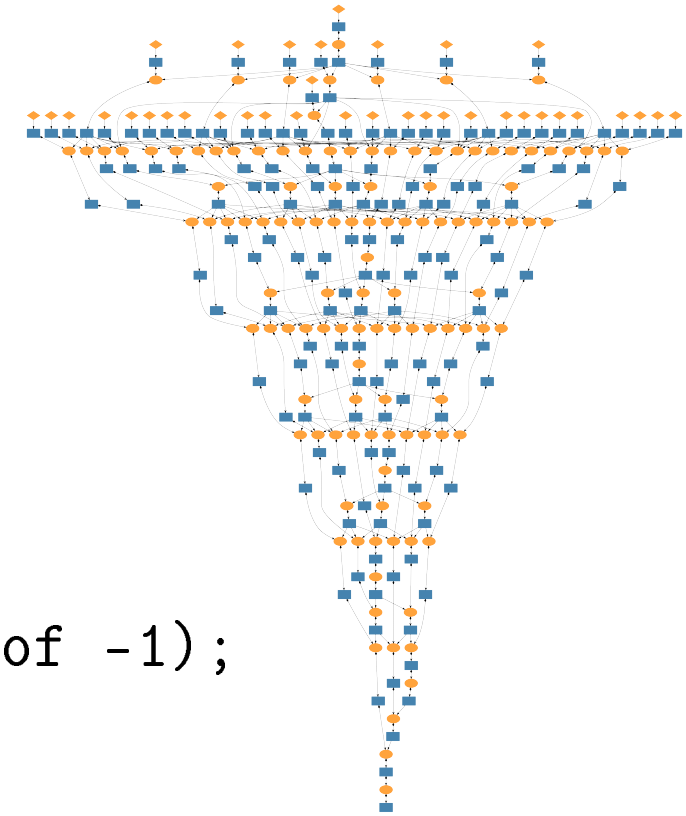
The Scheduling Problem of the Task Graph

A modern parallel application is a task graph (*OpenMP*, Go, Rust, Julia, etc.).

```
long long int fibo(int n) {
    if (n <= 2) return 1;

    long long int r1;
#pragma omp task shared(r1)
    r1 = fibo(n-1, prof -1);

    long long int r2 = fibo(n-2, prof -1);
#pragma omp taskwait
    return r1 + r2;
}
```



The scheduling problem

Definition (The scheduling problem)

The scheduling problem is to choose, for each of the n tasks, the processor to execute it and the date of the execution

This problem is *Strongly NP-Hard* save in very trivial cases (eg. 1 mono-core processor).
Thus, the optimal solution is very very very very very long to compute

Scheduling algorithm computes good solutions ($\neq$ optimal)

The goal is not to compute the best solution, but to compute, quickly, a good solution minimizing the completion time of the execution of the task graph, and often simultaneously others criteria

Criteria: Energy consumption (key for Exascale supercomputer)

In order to reduce the execution time and the energy consumption, the graphic card became accelerators (GPU).

GPU are everywhere

Hardware Laptop, mobile phone, supercomputer, etc.

Software Crypto-Currency, Deep Learning

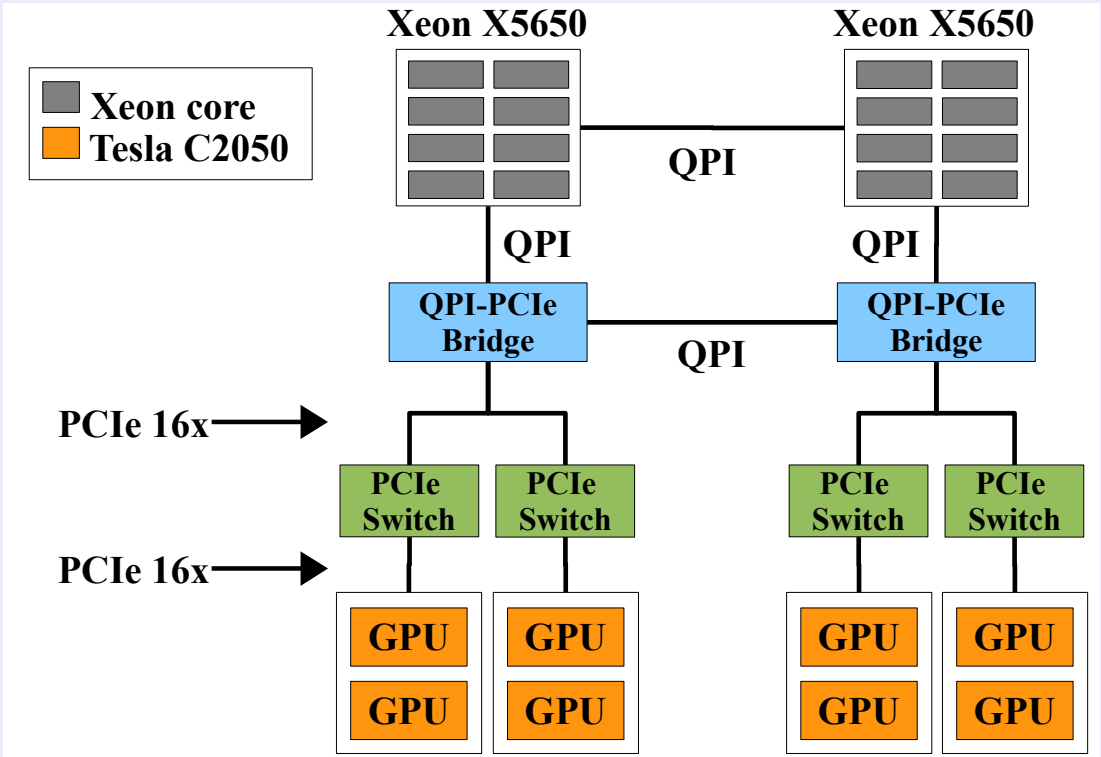
GPU are one key to get top performance on supercomputer

But, not the only way (Top 500: SunWay TaihuLight, 3rd, SuperMUC-NG, 8th), and definitely not yet the simplest.

Hybrid computing and the scheduling problem

How to use efficiently m cores (CPU) and k accelerators (GPU) ?

Two choices: The programmer implements ad'hoc scheduling decision or the program take efficient automatic scheduling decision



Content of the rest of this talk

1. Basics of scheduling theory and their application in designing heuristics
2. Two advanced algorithmic schemes useful in designing the heuristics for CPU+GPU, and how to combine them.

Limitations of this talk

No discussion about some important aspects of CPU+GPU scheduling such as:

- latency of the data exchange
- sharing of communication channels (shared PciE link)
- performance evaluation
- duplication
- checkpoint/restart
- etc.

Focus on low complexity algorithms

Our heuristics of interest have a low complexity ($\simeq O(n \log n)$) and a performance guarantee.

Performance guarantee (1/2)

Definition (Solution value)

The scheduling algorithm $\mathcal{A}$ applied to the instance I produce a solution σ_I with a value $v(\sigma_I)$ for the considered criterion

Definition (Solution quality)

For the instance I, the solution with the best value is denoted σ_I^*.
The quality of the solution is the performance ratio

$$\frac{v(\sigma_I)}{v(\sigma_I^*)}$$

Performance guarantee (2/2)

Definition (Performance guarantee of an algorithm)

The performance guarantee of algorithm $\mathcal{A}$ is the worst case ratio

$$\rho^* = \max_{\forall I} \frac{V(\sigma_I)}{V(\sigma_I^*)}$$

Definition (Lower bound of the optimal value)

The optimal solution is not computable in reasonnable time, thus in fact a lower bound $LB(\sigma_I^*) \leq V(\sigma_I^*)$ is used instead.

$$\rho^* \leq \rho = \max_{\forall I} \frac{V(\sigma_I)}{LB(\sigma_I^*)}$$

Worst case versus average case

The performance bounded by a worst case analysis (the performance guarantee)

Worst case analysis advantages

- no hypothesis on the instances (all the instances are better than the worst case)
- no frequency analysis of the instances required for any average case analysis
- still relevant if ρ is small
- the worst case structures highlight some deficiencies of the algorithm

But the worst case performance may be quite far from the average case performance (like for *quick sort* versus *heap sort*)

Notations (1/2)

- n tasks $t_i (1 \leq i \leq n)$: execution time of p_i on CPU and g_i on GPU.
- The acceleration factor of a task t_i is the ratio $a_i = \frac{p_i}{g_i}$
- m CPU and k GPU
- the completion time of a task t_i in a scheduling σ is denoted C_i

Definition (The makespan: the basic criterion)

The optimized criterion is the completion time of the last task in a scheduling σ (the makespan) denoted C_{max}.

$$C_{max} = \max_{1 \leq i \leq n} C_i$$

The optimal makespan (unknow) is denoted C^*_{max}.

Notations (2/2)

Warning !!

In this talk, the execution time of a task is function of its mapping (CPU or GPU)

Definition (Total work and critical path)

For any scheduling σ, two values summarize it well:

- the work $W_\sigma = \sum_{\sigma(t_i) \in CPU} p_i + \sum_{\sigma(t_i) \in GPU} g_i$
- the critical path CP_σ (the longest path in the task graph).

Lower bounds

$W_\sigma/(m+k)$ and the critical path CP_σ are lower bounds of C_{max}

List scheduling [Graham 1966, 1969, 1973, 1975,...]

Two main ideas:

1. Keep the List of the tasks to execute, sorted by priority
2. Apply the following rule: start as soon as possible one of the task of the list on a free processor. The task is the ready task starting the earliest (first) with the highest priority (second).

One basic idea: fill the processors !

Keep the processors busy if it is possible, even by running a low priority task first

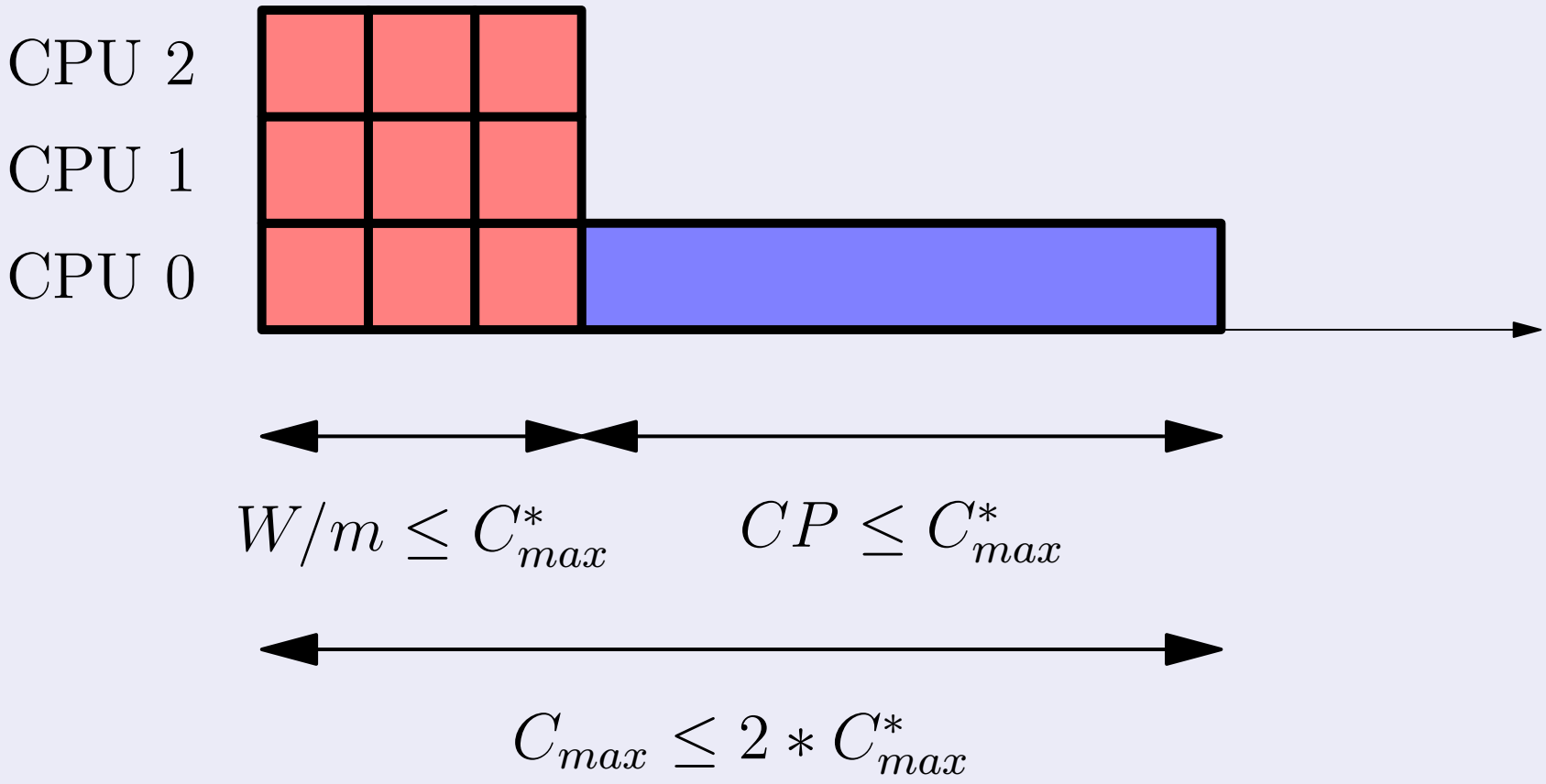
Execution order of the tasks is not guaranteed

The order of the list is not the order of execution. E.g. precedence constraints or resources constraints change the order.

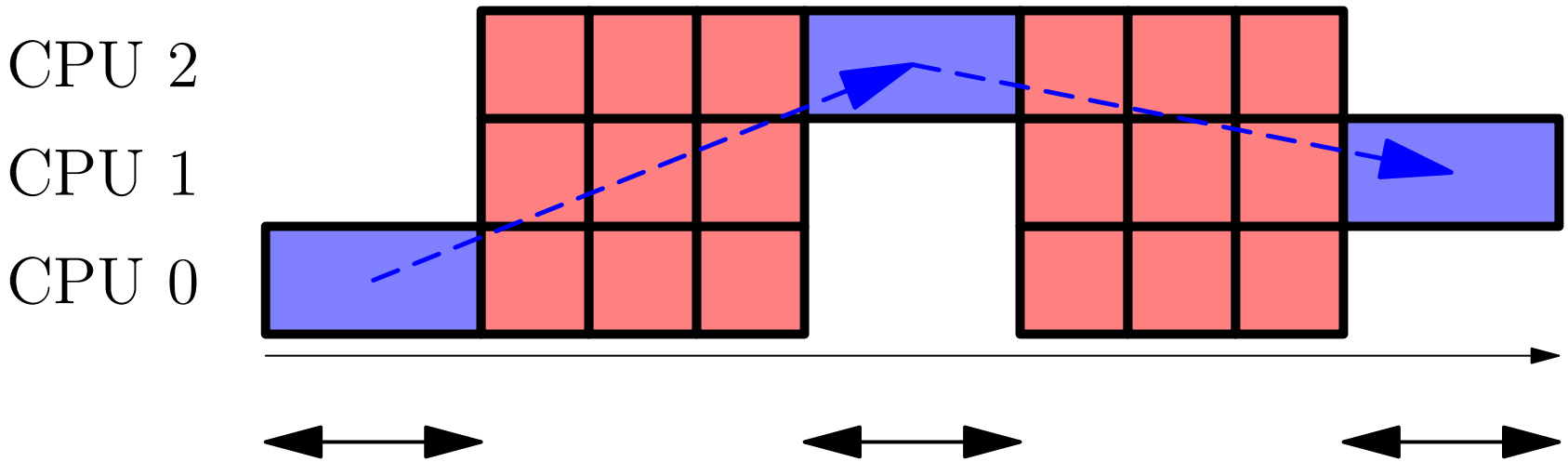
Graham with independent tasks

Task list $= p_i = (1, 1, 1, 1, 1, 1, 1, 1, 1, 6)$, $m = 3$ CPU, $k = 0$ GPU

Scheduling



Graham with precedence constraints

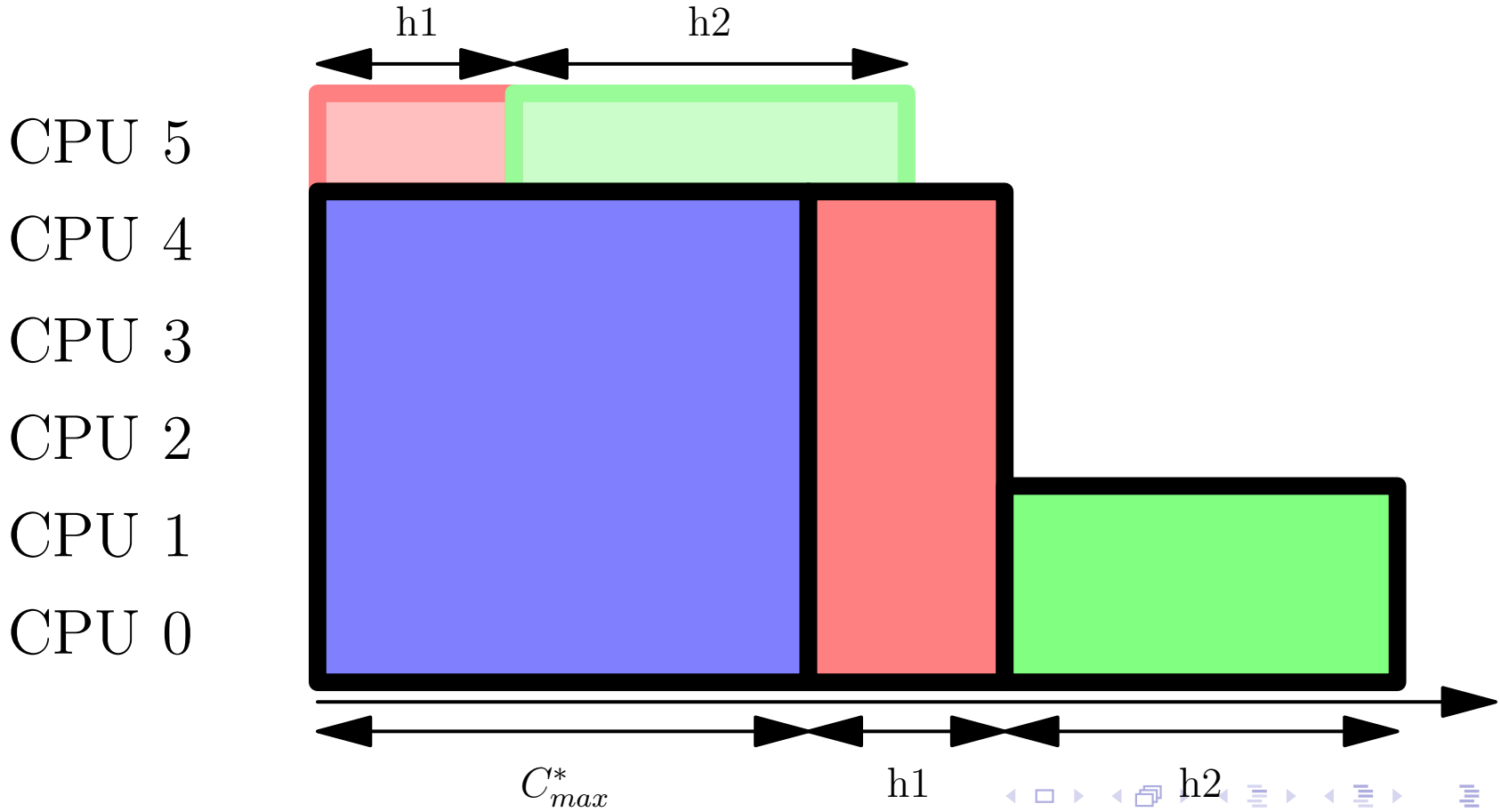


$$CP \leq C^*_{max}$$

$$W/m \leq C^*_{max}$$

Graham with resource constraints (simplified: rigid multiprocessor tasks)

Contradiction: a makespan larger than $2C^*_{max}$ would required more than mC^*_{max} work



1+1 = 2

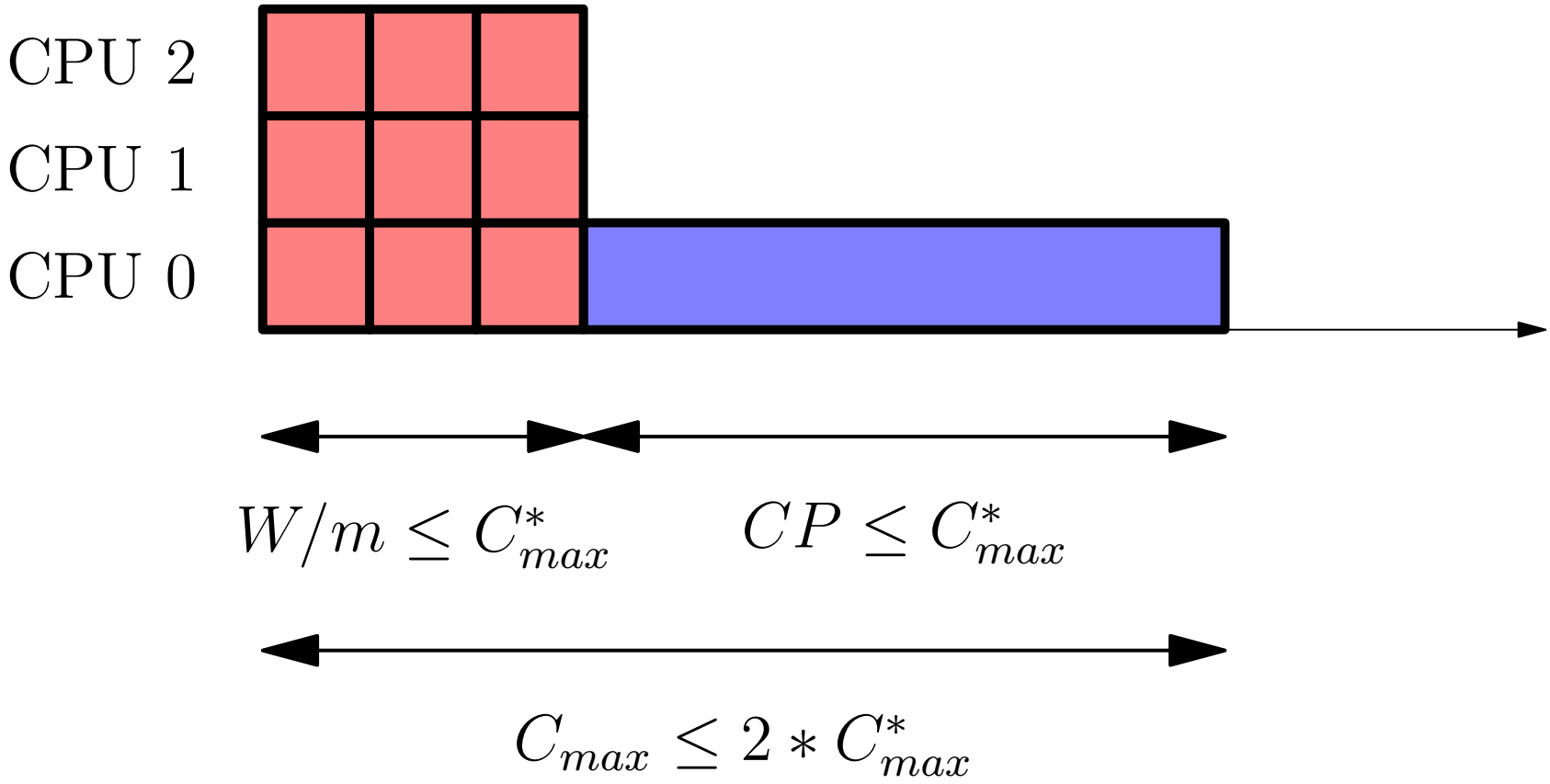
All these results use the same concepts:

- 1 + 1 The makespan is lower than $\frac{W}{m} + CP$
- 2 $\frac{W}{m}$ and CP are two lower bounds of C^*_{max}. Then keeping busy the processors as much as possible is sufficient to be close to the optimal makespan (at a factor of 2).

Non-clairvoyant algorithms use everywhere in practice

The Graham's rules does not need any information on the ready tasks to get the factor 2 ! It is applied (indirectly) by most schedulers of most systems: process and threads, disk io, network paquet io, etc.

Graham's independent task (again)



Partially sorting of the priority list allows better results:

m largest first, in decreasing order $\rho = 3/2$

$2m$ largest first, in decreasing order $\rho = 4/3$

Work stealing: the default parallel dynamic scheduler of process and threads for multi-core (Linux, Windows, OpenMP, ...)

Work stealing algorithm

- One list of tasks per processor
- a free processor execute the first task of its list
- unless the list of the processor is empty, then the processor select randomly a victim and steal a task of the victim's list.

It is basically the same idea as Graham's list (keeping busy the processors !) but with a distributed implementation.

Work stealing: the default parallel dynamic scheduler

Three main ideas help the implementations:

1. Most of the *overhead* of the distributed list management is done by the *free processors*
2. One list per computing resource: *low contention* using lock-free dequeue
3. Additional improvement: *Lazy task creation* at steal time (task creation is not free)

Non-uniform work stealing

variant of work-stealing is used also in distributed and parallel computing. Some ideas could also be used for (large) NUMA:

adaptive probability of victim selection the distance, the hierarchy of the computer, the number of failed steals

close and far simultaneous steals simultaneously doing a far away steal and a close steal

multiple simultaneous steals select two random victims and steal the "richest"

The precedence constraints are a lie (1/2)

FALSE: efficient scheduling decision are based on the precedence constraint

Dynamic parallel programming environment manipulate dynamic task graph with precedence constraints. Thus any useful scheduling algorithm need to manage task graph with precedence constraints.

TRUE: the runtimes schedule ready tasks

Ready task are independent, per definition. In addition, the future is not known:

- The readiness is not know in advance. All the tasks must be done as efficiently as possible (Work minimization).
- it would be better to do the tasks on the critical path first, but we do not know the critical path.

Precedence constraint management (2/2)

Task graph management in parallel runtime

Two main families of operations :

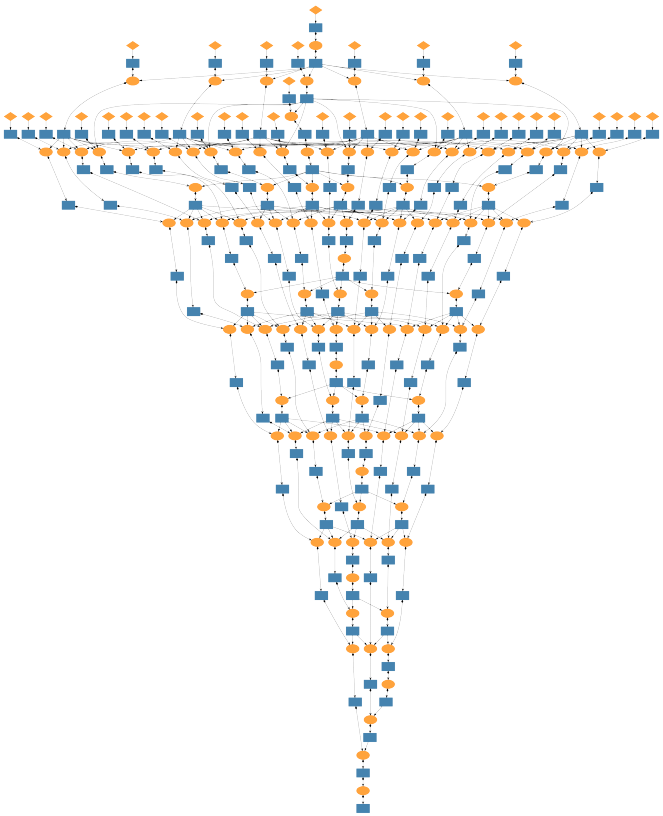
precedence constraint management event driven: creation and end of execution of the tasks allow to compute the readiness of each task in the list

scheduling of the ready tasks all ready tasks are independent.

The scheduled tasks are ready and thus independent.

It is a major point ! The scheduling decision, in practice, manage only independent tasks.

Dataflow graph



Any parallel runtime needs to know where are the data, especially in CPU+GPU. Thus the managed graph is not a task graph but a bipartite graph composed of two kind of nodes:

- computation node
- data node (every input and output of the computation nodes)

Summary for homogeneous computer

1. Most scheduling heuristics on identical machines have good performance guarantee (2) almost for free: keep the processors busy !
1. Most of the scheduling decision are done on independent tasks.

And with heterogeneous computer (CPU+GPU) ?

How to achieve similar performances on heterogeneous systems ?

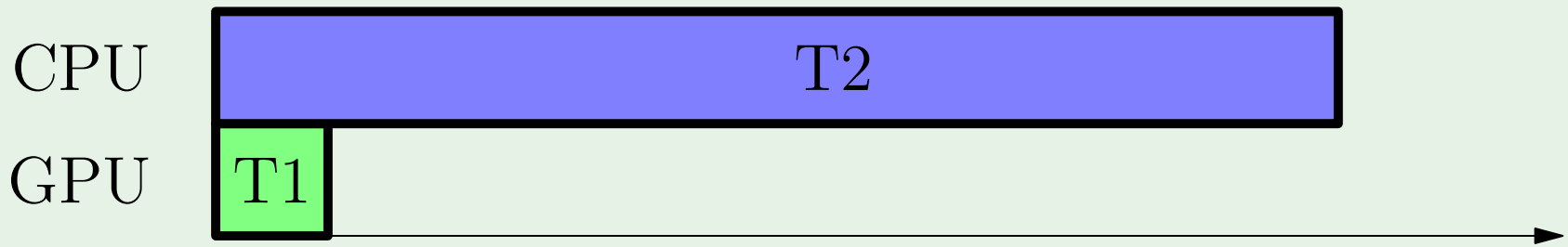
Heterogeneity and list algorithm

Graham's list scheduling does not have any performance guarantee on heterogeneous machines.

Counter-example:

	T1	T2
p_i (CPU)	100	100
g_i (GPU)	1	1

Any list algorithm schedules this list of 2 tasks with a makespan of $C_{max} = 100$. The optimal schedule has a makespan of $C^*_{max} = 2$



List algorithm on heterogeneous computer

Contrary to the identical machine case, keeping the computing resource busy is not always a good idea. Some computing resources must stay free.

Work stealing adaptation

A steal may failed even if some tasks are ready.

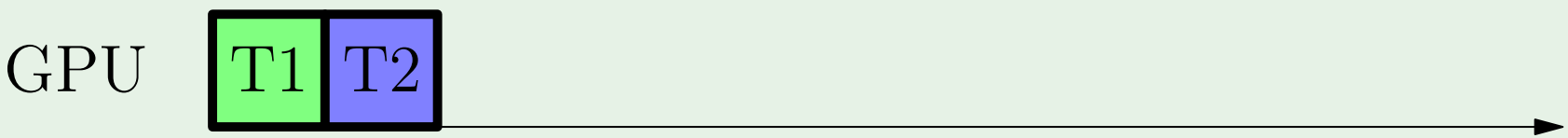
The simple, classical and standard solution: the programmer will do it

Static mapping

The programmer choose if a task will be executed on CPU or GPU.

Example

	T1	T2
CPU	$+\infty$	$+\infty$
GPU	1	1



How to build generic efficient heuristics ?

Two ways

system way implement interruption or round-robin on GPU

algorithmic way use clairvoyant heuristics

Algorithmic way

Some information on the tasks are required !

Typically useful data

- the execution time on CPU and GPU (or approximate values)
- the acceleration factor of every task

Classical heuristic: HEFT [Topcuoglu] (1/5)

Definition (Heterogeneous Earliest Finish Time First)

- A task list with the following priority :

$$Rank_i = Exec_i + \max_{j \in succ(i)} (Comm_{ij} + Rank_j)$$

. with:

  - $Exec_i = (mp_i + kg_i)/(m + k)$, the average execution time
  - $Comm_ij$, the average communication cost

- A mapping rule: Put the first task of the list on the resource where the task completes the earliest

HEFT (2/5)

Property:

- Low complexity
- May include a very accurate communication and execution model
- take into account the precedence constraints and the critical path
- Need the full graph to compute the correct rank

HEFT (3/5)

HEFT priority rule selects the ready task on top of the average critical path. Thus HEFT sort independent tasks by average execution time, decreasing.

HEFT counter-example: independent tasks (1/3)

Let use m CPU, 1 GPU and 3 kinds of tasks (filling tasks, $\mathcal{A}$ and $\mathcal{B}$):

Large in average filling tasks:

Initialize the processors by filling the CPU

	T1	...	Tm
CPU	ϵ	...	ϵ
GPU	$m+2$	...	$m+2$

HEFT (3/5) bis

HEFT counter-example: independent tasks (2/3)

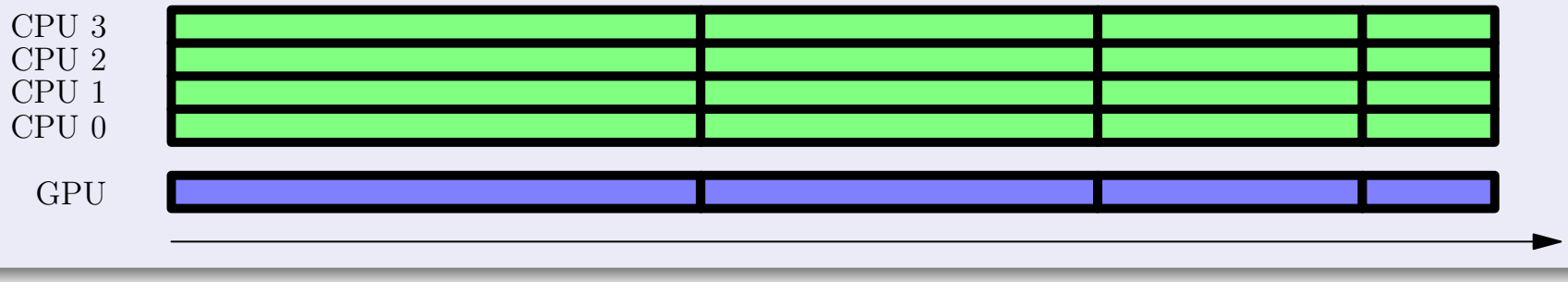
... and the following tasks
for $i = 0..m-1$

	$\mathcal{A}_{i1}$	$\mathcal{B}_{i1}$	...	$\mathcal{B}_{im-1}$
CPU	$1-i/m$	$1-i/m$	...	$1-i/m$
GPU	$1-i/m$	$1/m^2$	...	$1/m^2$

Layer by Layer, put 1 task $\mathcal{A}$ first, wrongly on GPU, then put m tasks $\mathcal{B}$, wrongly on CPU

HEFT (4/5)

HEFT counter-example schedule (3/3)



Optimal schedule

HEFT (5/5)

How HEFT could be so wrong ?

HEFT execute the top priority task with respect to the critical path.
Some resources will be used less efficiently and thus the total work will increase.
To keep the total work low, sorting by the acceleration factor would be better choice (not for the critical path)

Open question

Does HEFT with sorting on acceleration factor has a performance guarantee ?

Heterogeneity

HEFT does not take into account one major point of our problem

There are only two types of computing resources

There are only two types of computing resources: CPU and GPU.
If a task is not executed by a GPU, it has to be executed by a CPU.
This is true also for the optimal solution !

Knapsack

Minimizing the total work is similar to the knapsack problem: select the best set of tasks executed by the GPU (and thus the complementary set of tasks executed by the CPU).

But how to choose the knapsack size ?

use Dual Approximation ! (Next slide)

Dual Approximation (1/2) [Shmoys,Hochbaum]

Definition (Dual aproximation)

Choose an arbitrary value λ and use it as *guess* of the optimal solution value.

The algorithm has the following binary output:

ACCEPT it provides a schedule with $C_{max} \leq \rho\lambda$

REJECT $C^*_{max} \geq \lambda$

λ value will guide the structure of the solution

All tasks should execute in a time less than λ and the work should be less than $(m + k)\lambda$

Dual Approximation (1/2)

How to choose λ ?

Using binary search (with ϵ accuracy) on the value of λ, we get:

- a solution with $C_{max} = \rho\lambda + \epsilon$
- and $C^*_{max} \geq \lambda$

Thus, the performance guarantee of the solution is $\rho + \epsilon$.

Optimal solution structure

If λ is the optimal value:

1. All execution time are lower than λ (Critical path)
2. Every computing resource compute at most λ, thus the total work is lower than $(m + k)\lambda$ (Work)

CPU+GPU scheduling and dual approximation

Application with $\rho = 2$ [Monna, Bleuse et al.]

sort	sort the tasks by acceleration factor
preloading	list schedule the tasks constrained to a single type of resources ($p_i > \lambda$ or $g_i > \lambda$)
knapsack	list schedule the GPU with the tasks by decreasing acceleration factor until the sum of work on GPU is larger than $k\lambda$. The GPU are filled less than 2λ.
CPU filling	list schedule all remaining tasks The CPU are filled less than 2λ

CPU+GPU scheduling and dual approximation

REJECT

- $\exists i, p_i > \lambda and g_i > \lambda$
- Total work larger than $(m+k)\lambda$
- One CPU is filled more than 2λ

Knapsack work is lower than the work of the optimal solution

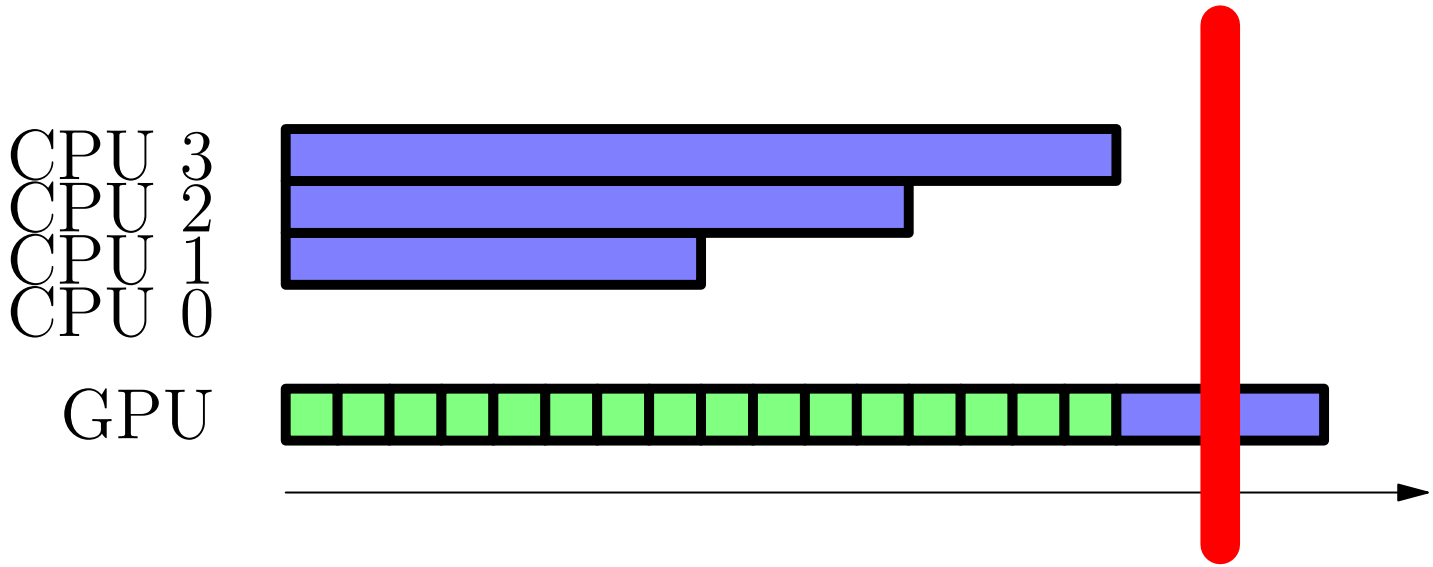
$If\, \lambda \geq C^*_{max}, W_\sigma \leq W_{\sigma^*}$

HEFT counter-example

$for\, i = 0..m-1$

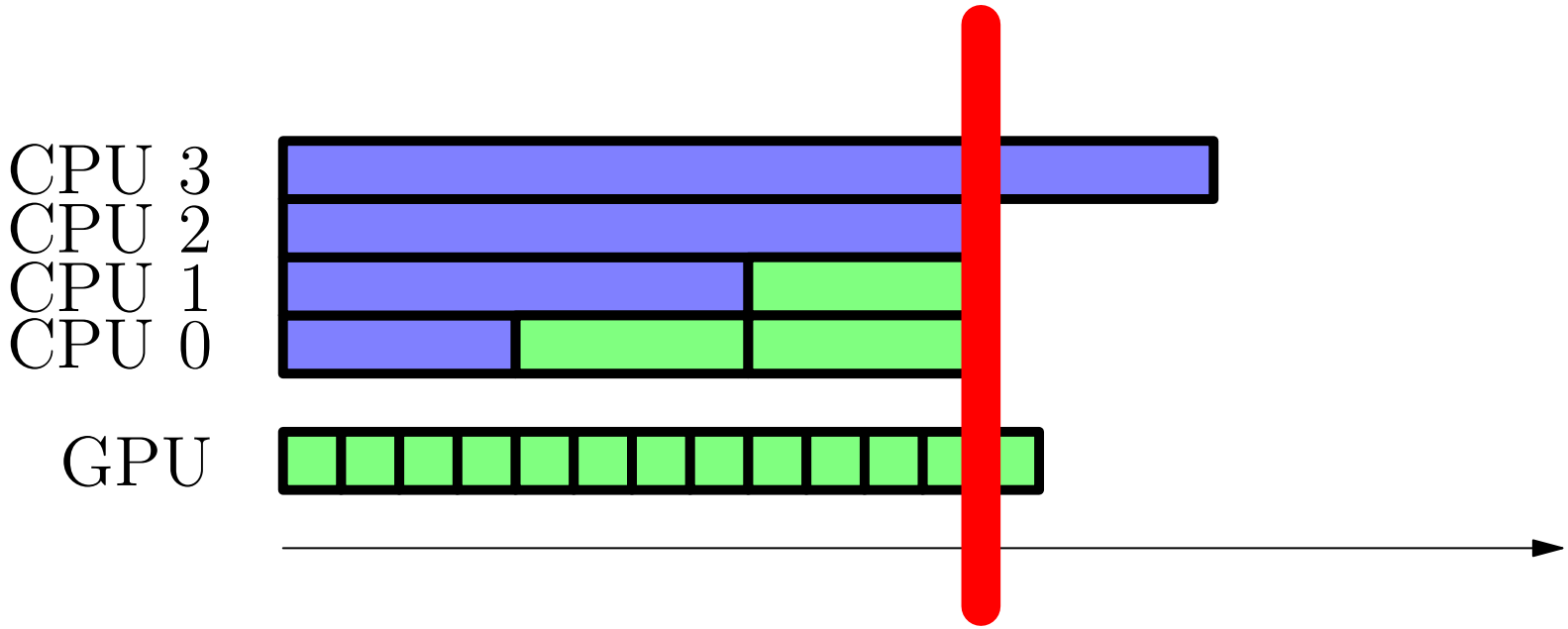
	$\mathcal{A}_{i1}$	$\mathcal{B}_{i1}$	$\ldots$	$\mathcal{B}_{im-1}$
CPU	$1 - i/m$	$1 - i/m$	$\ldots$	$1 - i/m$
GPU	$1 - i/m$	$1/m^2$	$\ldots$	$1/m^2$

If λ larger than C^*_{max}



ACCEPT: λ is a bit too large

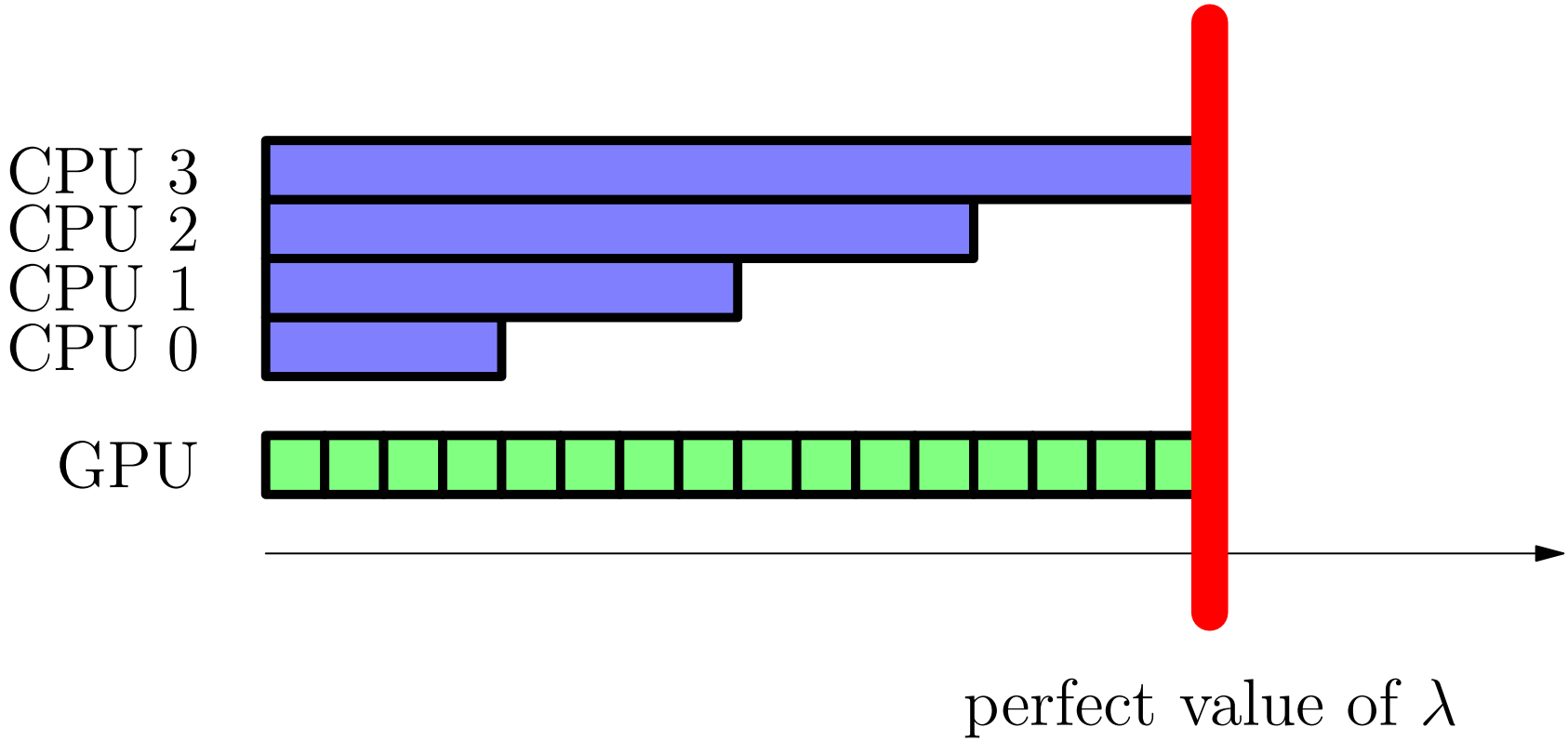
If λ smaller than C^*_{max}



REJECT: λ a bit too small (too much work)

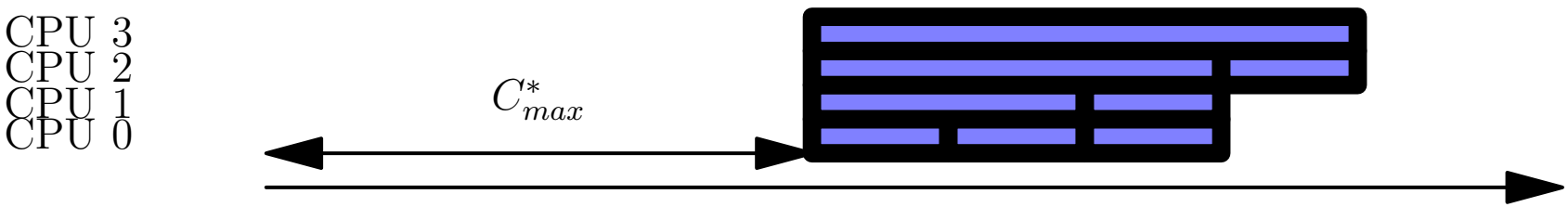
λ is at the right value

The algorithm build the optimal solution for this instance (not always the case)



Wein and Stein scheme (1/3)

If you add C^*_{max} to all tasks of a schedule produced by an algorithm of performance guarantee ρ, you got a solution with a guarantee of $\rho + 1$.
We lost time (makespan) but we gain space (idle resources) to optimize an additional criteria.



Wein and Stein scheme (2/3)

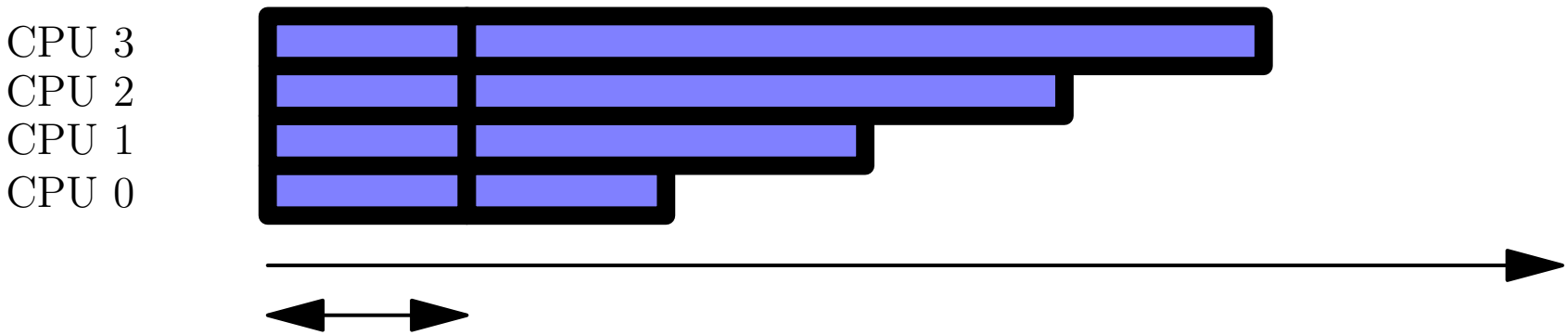
To maximize the throughput ($\sum C_i$) of tasks, the smallest tasks must be done first

SPTF versus LPTF

- SPTF (Smallest Processing Time First) is optimal for $\sum C_i$ and has a guarantee of 2 for the makespan (it is a list scheduling algorithm !)
- LPTF (Largest Processing Time First) has a guarantee of 4/3 for the makespan and no guarantee for $\sum Ci$

Wein and Stein scheme (3/3)

SPTF and LPTF merge using Wein and Stein scheme



$< 1/3 C^*_{max}$

SPTF and LPTF merge using Wein and Stein scheme

- SPTF for 1/3 of the LPTF 4/3 solution
- then schedule the remaining tasks identically to the first LPTF (or redo LPTF if better)

The bicriteria algorithm has then two guarantees $(\frac{5}{3}, 5)$:

- 5/3 on makespan
- 5 on $\sum C_i$

Other criteria [from Saule et al on reliability]

The Wein and Stein scheme is usable also with a criterion unrelated to C_i

Rational

Data movement are costly. If possible, it is better to apply "Owner Compute Rule" like mapping.

Definition (Binary affinity)

All ready task score the resources with an affinity score function Φ. A single resource is non-zero

The affinity of a scheduling is the sum of affinity of the tasks mapped on their scored resources.

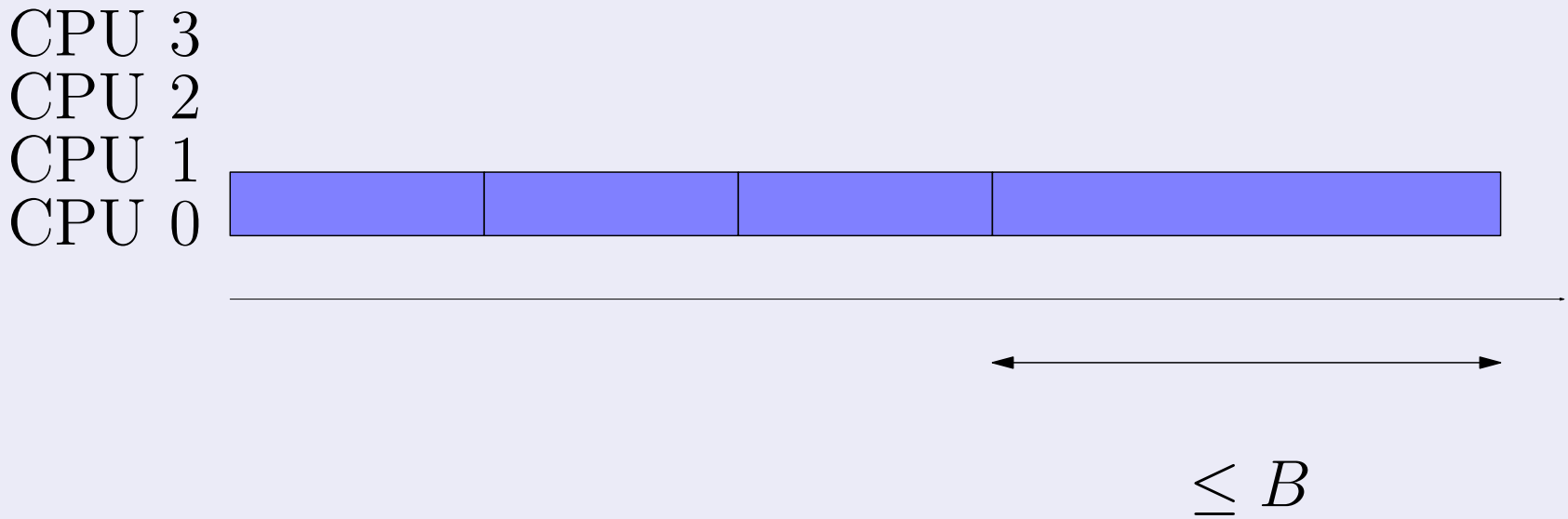
Affinity

How to maximize affinity of a scheduling ?

Execute every tasks on their scored resource.

But the makespan can be much larger than the optimal

It is worse with 2-Heterogeneity if all tasks want the same slow resource (e.g. at the beginning of the computation)



Affinity and makespan

Inapproximability of the bicriteria affinity/makespan

An bicriteria algorithm (affinity/makespan) with constant performance guarantee on both criteria does not exist (even for identical machines) !

Affinity with a maximum makespan

We will compare affinity of a schedule with the best affinity possible with a schedule of same makespan.

Definition (Affinity with bounded makespan)

$\Phi^*(B)$ is the best affinity for any schedule with a makespan lower than B

Affinity with a maximum makespan

Definition (Algorithm to maximize affinity with constraint B)

- sort the tasks by decreasing value of the ratio $\frac{\Phi_i}{p_i}$
- fill the resources, in order of the task, unless a resource is filled more than B
- list schedule the remaining tasks

Affinity quality of the schedule

$\Phi_\sigma \geq \Phi^*(B)$

Partial affinity with bounded makespan

Instead of filling resources more than B, we fill them more than αB. ($\alpha \geq 0$)

Affinity quality of the schedule

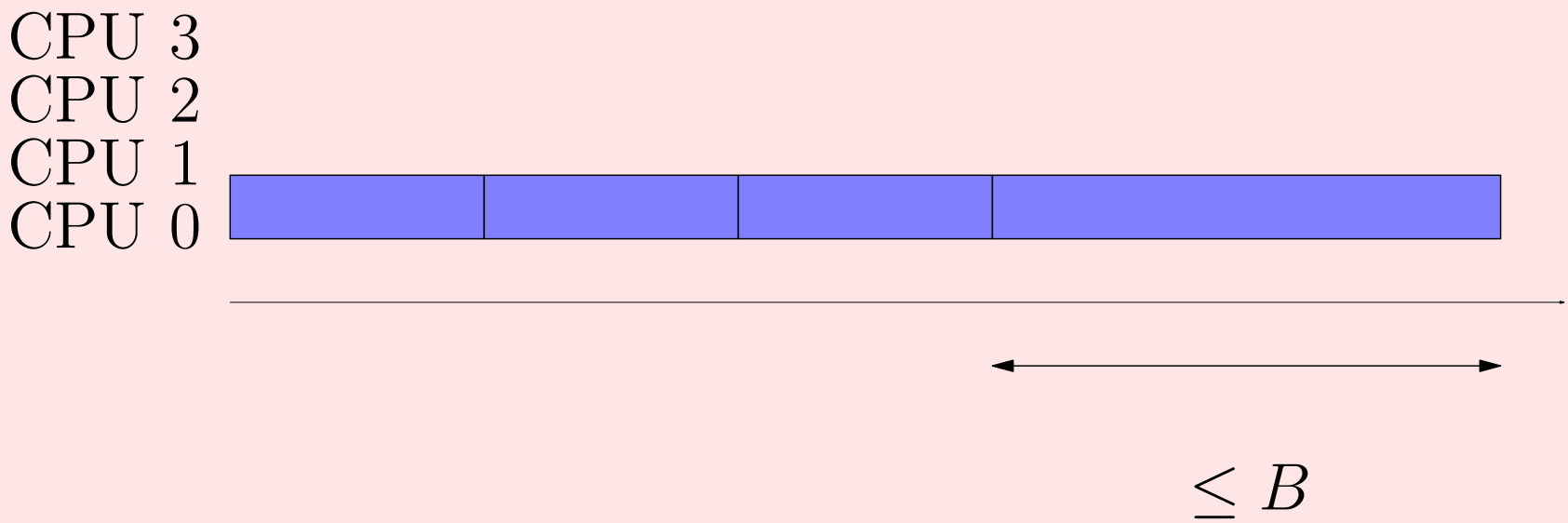
$\Phi_\sigma \geq \alpha \Phi^*(B)$ if $\alpha \leq 1$

2-heterogeneity constraint

All tasks of the best affinity schedule have a execution time lower than B

Makespan

The makespan of the tasks mapped by affinity is lower than $(1+\alpha)B$



Combining Affinity with dual approximation

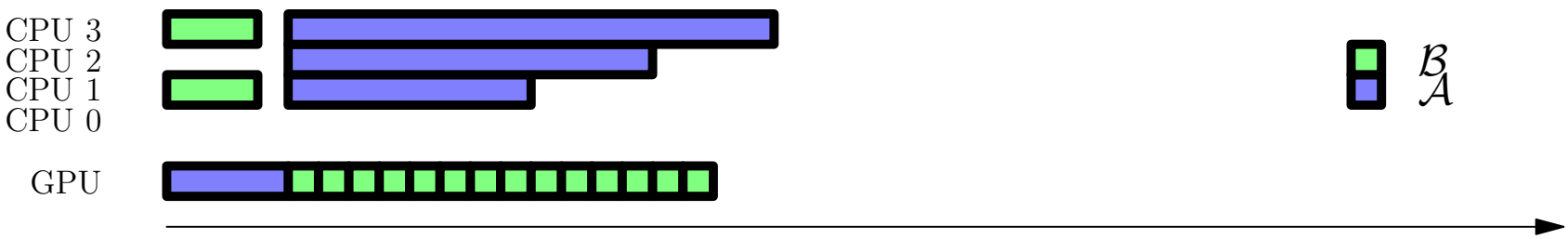
To keep affinity guarantee we need $B \geq \lambda$
Let $B = \lambda$

- $\$C_{max} \leq (1+\alpha + 2)\ \lambda + \epsilon\ \$$
- $\Phi_\sigma \geq \alpha\Phi^*(\lambda)$

Bicriteria guarantee

$(3+\alpha, \underline{\alpha})$

Combining Affinity with dual approximation



"Easy" extensions

- Better scheduling (LPTF) required to solve a knapsack with additional constraints: dynamic programming is ok for a small set of tasks.
- Affinity (not only binary) with lexicographic quality: maximize first choice, then max second choice, etc.
- Moldable tasks: select the number of resources use by a parallel task (CPU) at launch time
- etc.

Conclusion

- Optimizing only independent task scheduling algorithm is relevant in dynamic parallel programming environment
- Dual Approximation is a flexible way to build a scheduling heuristic with good load balancing.
- DA and Wein and Stein scheme offer secondary optimization option (affinity, reliability, throughput, etc.) at low cost.