Efficient Data Representation of Large Job Schedules

Dalibor Klusáček, Hana Rudová

xklusac@fi.muni.cz, hanka@fi.muni.cz

Faculty of Informatics,
Masaryk University,
Brno, Czech Republic

7th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science
14th - 16th October, 2011 | Lednice | Czech Republic

Introduction

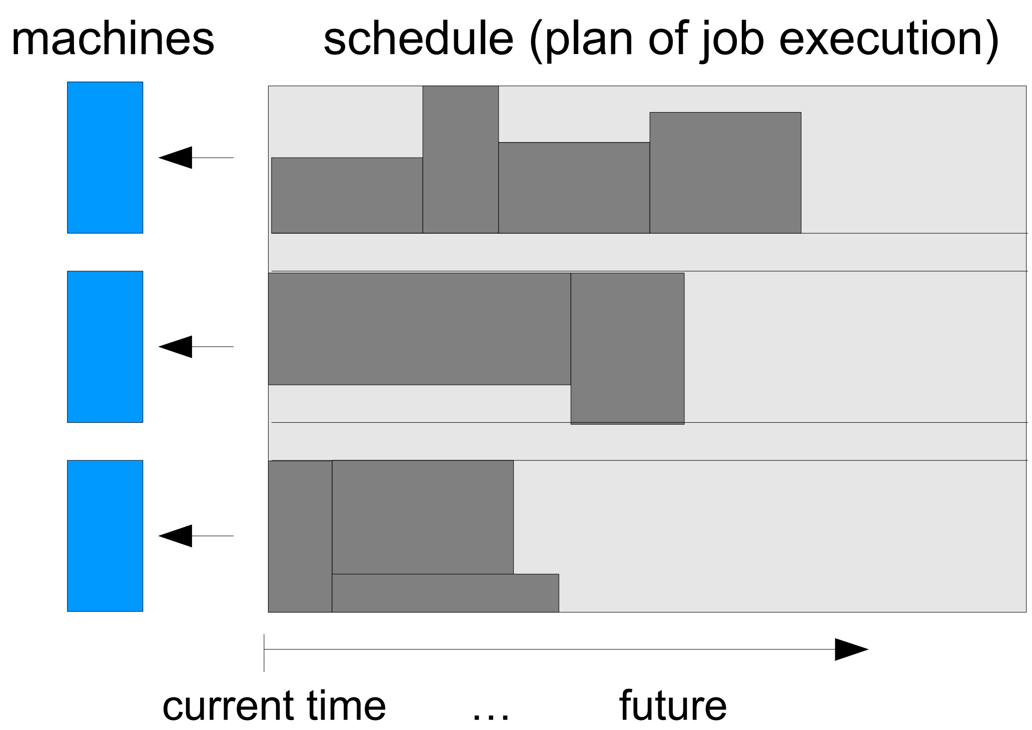
- Motivation
  - Practical problems we faced during our research in the area of Grid scheduling
  - Proposal of efficient scheduling algorithms
  - Implementation
- Even good algorithm may be very inefficient when implemented in a wrong fashion or when the scale of the problem increases
- This paper describes how to efficiently represent large job schedules
  - wrt. memory requirements
  - wrt. runtime requirements

Problem Description

- Grid
    - Large system of distributed (computational) resources
    - Executing users' applications
    - Highly dynamic, heterogeneous
- Grid scheduling
    - Job allocation on resources in time
        - Subject to (often complex) objective criteria
    - Must be fast ("on-line scheduling")
        - Difficult task due to dynamic behavior and uncertainty

Schedule-based Approach

- Instead of queue(s), schedule (plan of job execution) is built
  - Allows to plan when and where jobs will be executed
  - Preditability (useful for the user)
  - Evaluation (helps to identify problems, inefficiencies)
  - Optimization (helps to fix problems and inefficiencies)

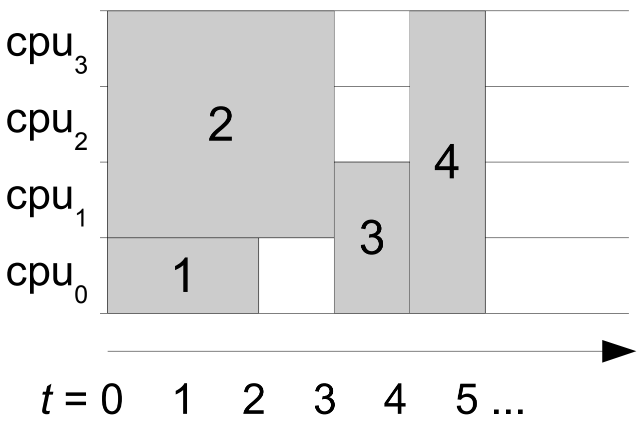


How To Efficiently Represent Schedule

- Unlike the queue, schedule is more complicated structure
- The Grid system is often huge and hundreds of jobs are planned at the same moment
- Data representation should be
  - Memory efficient (schedules are huge – many CPUs, many jobs)
  - Time efficient (wrt. common schedule-related operations)

Schedule Representation (1)

"Human readable" schedule



Matrix-like representation

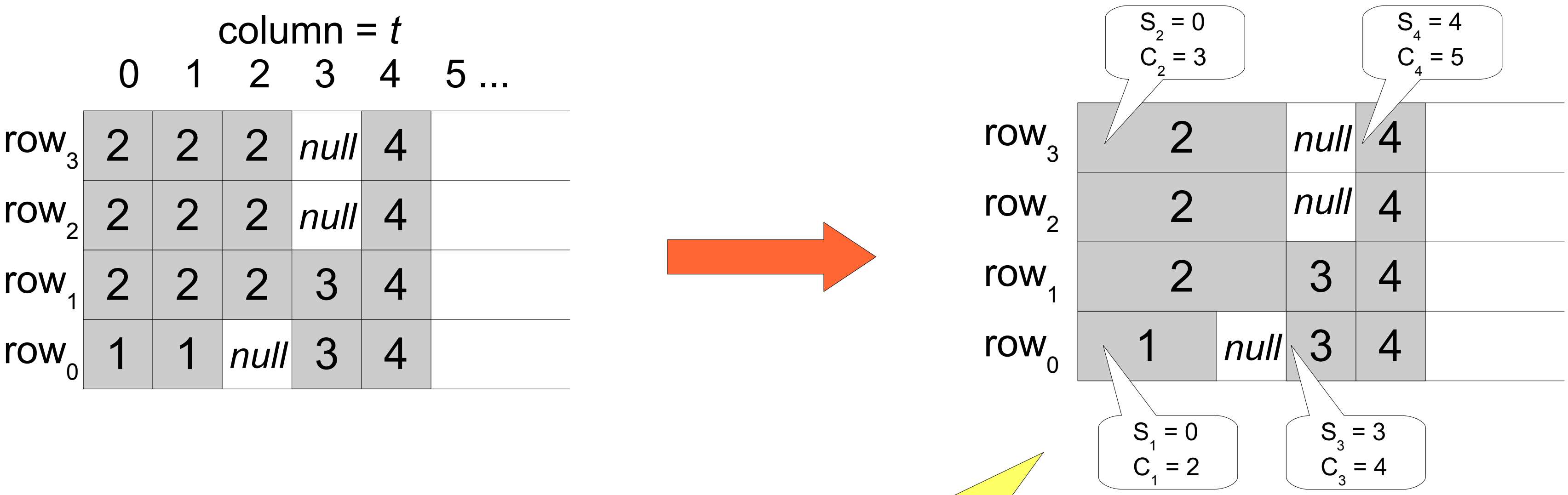
column = *t*

	0	1	2	3	4	5 ...
row_3	2	2	2	*null*	4	
row_2	2	2	2	*null*	4	
row_1	2	2	2	3	4	
row_0	1	1	*null*	3	4	

This representation does not scale well w.r.t. the length of the schedule.
One month for 1 CPU would require 2,6 milions cells in case that 1 cell = second.

The size of such structure is proportional to $\mathbf{m \cdot C_{max}}$

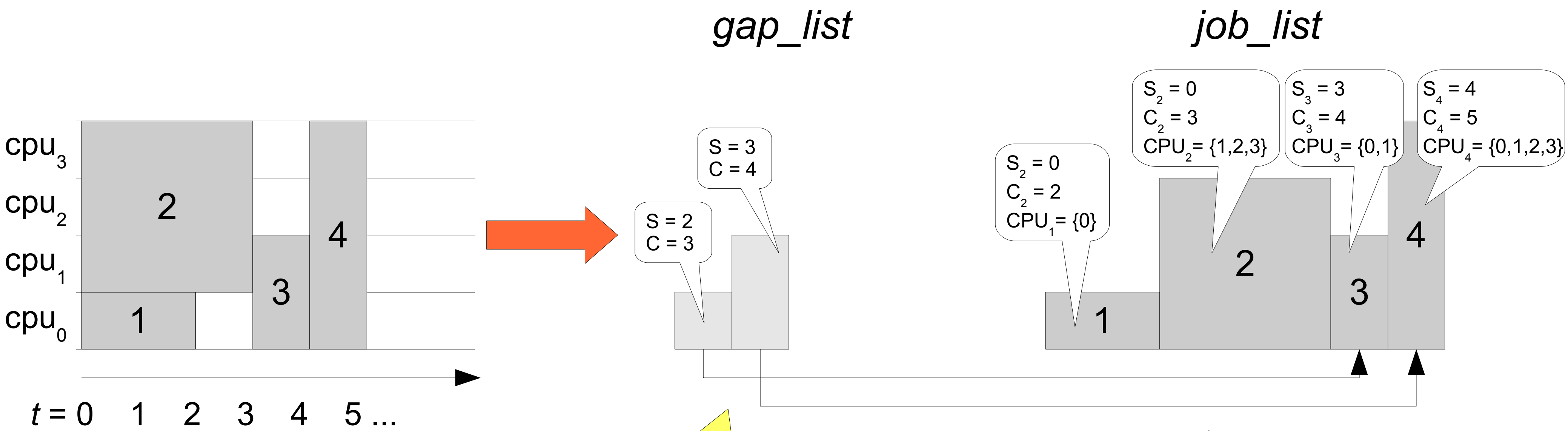
Schedule Representation (2)



Information about given job can be "scattered" in cells with **different mutual position**.
E.g., job 3 is stored in second cell (row_1) and third cell (row_0), respectively.

The size of such structure is proportional to **m·n**

Schedule Representation (3)



Gaps are stored in a separate list.
It is useful as they can be used to for new jobs.
Saves **computational time**, do not change **guaranteed start times** of previous jobs.

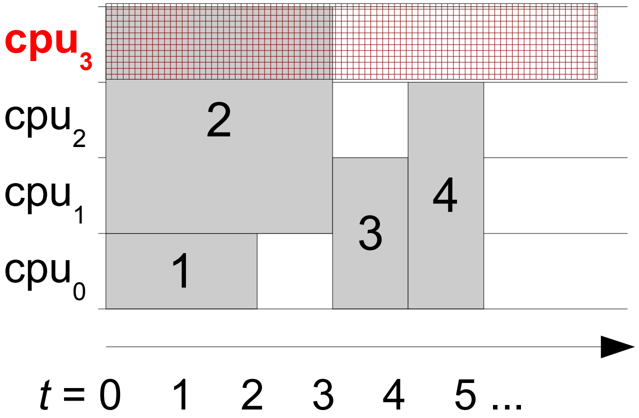
The size of such structure is proportional to **2·n**

Schedule Consistency

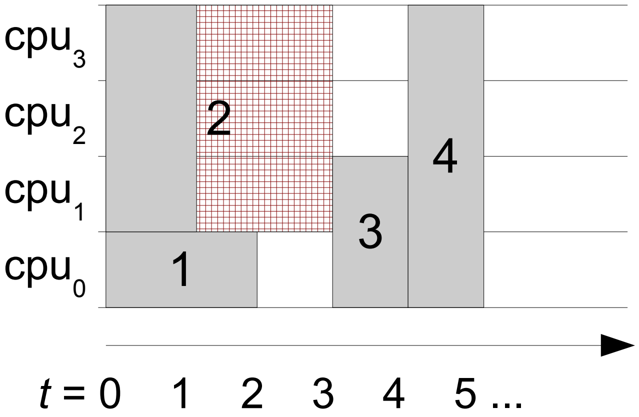
- **On-line scheduling** therefore
  - Schedule becomes inconsistent with new state of the system
  - Something happens
    - Machine fails
    - Job arrives
    - Job completes prematurely
    - Optimization (i.e., modifications of existing schedule)
    - etc.
- **Schedule must be updated**

When to Update the Schedule?

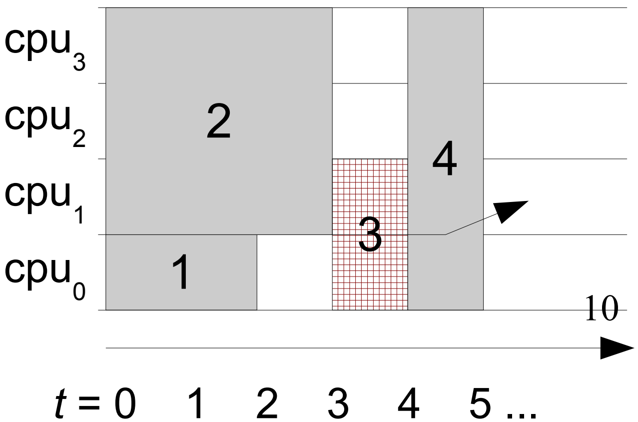
- Machine fails
  - Use only working CPUs



- Job finishes earlier
  - Shift later jobs to ealier time slot



- Job position has changed (e.g., by optimization algorithm)



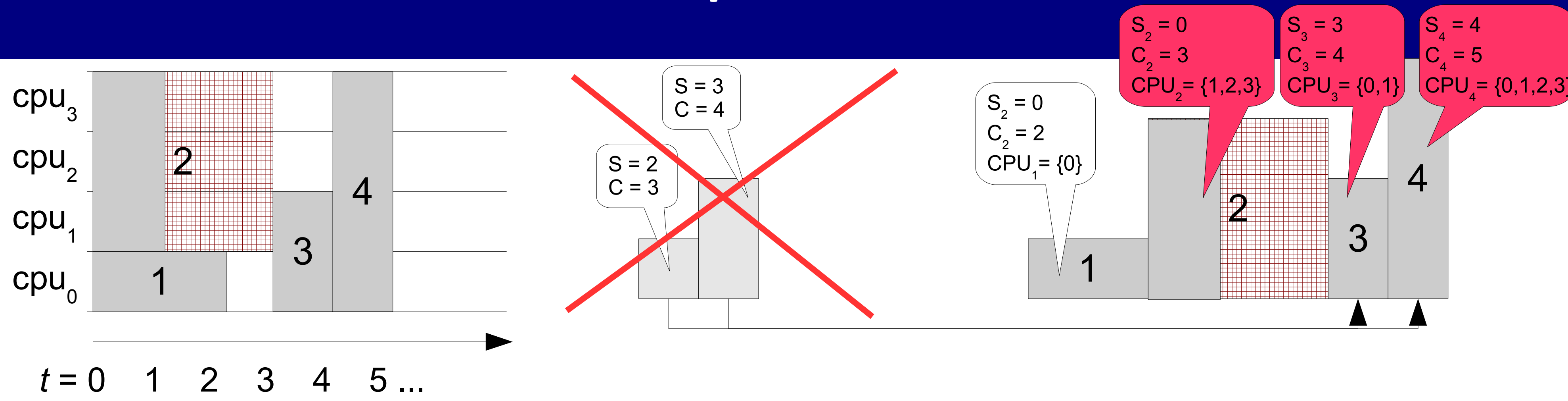
How to Update the Schedule?

- **Update procedure**
  - Recomputes job "coordinates" for each job
    - start time
    - completion time
    - set of assigned CPUs
  - The gap list is recreated

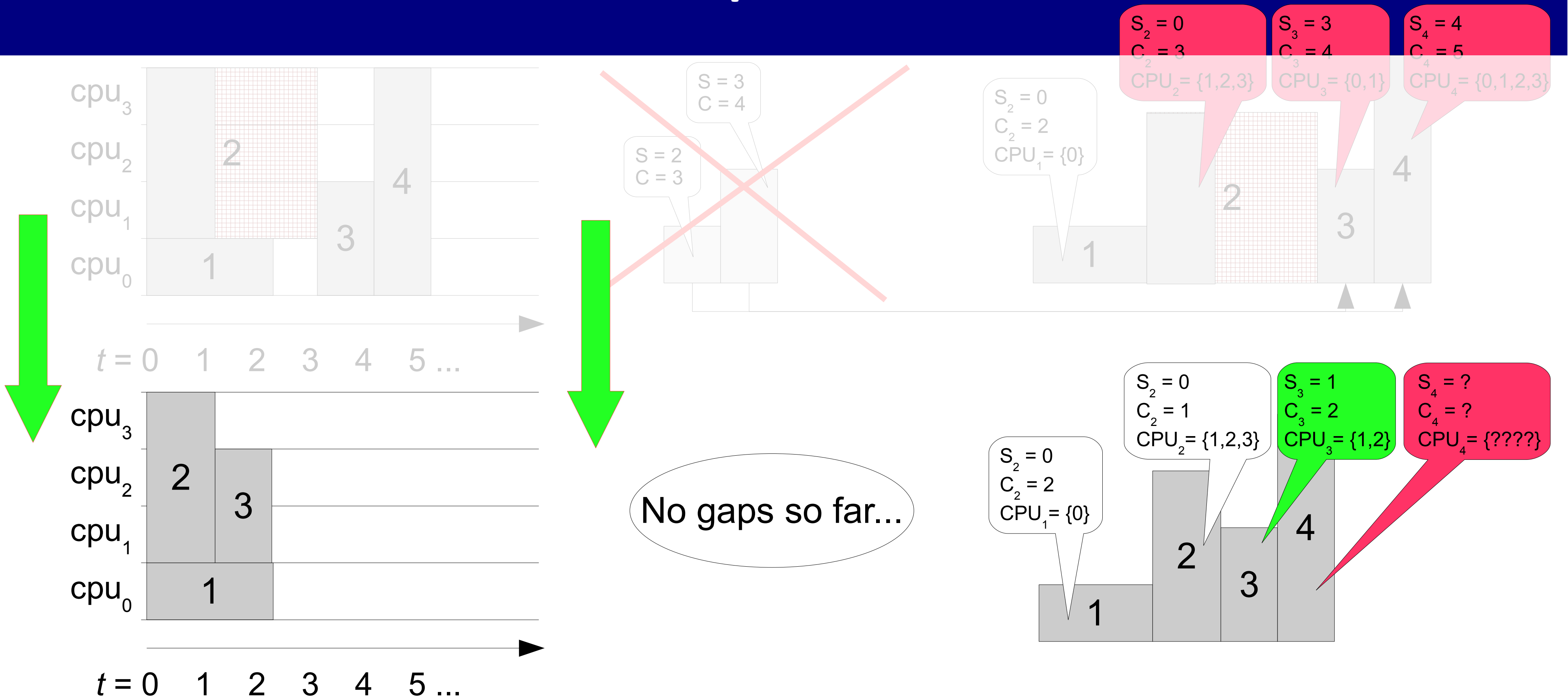
```
gap_list := null;

for i:=1 to n do
  job:= i-th job from job_list;
  find earliest start time of job;
  compute completion time of job;
  compute the set of CPUs assigned to job;
  extend gap_list with new gaps that could appear "in front" of job;
end for
```

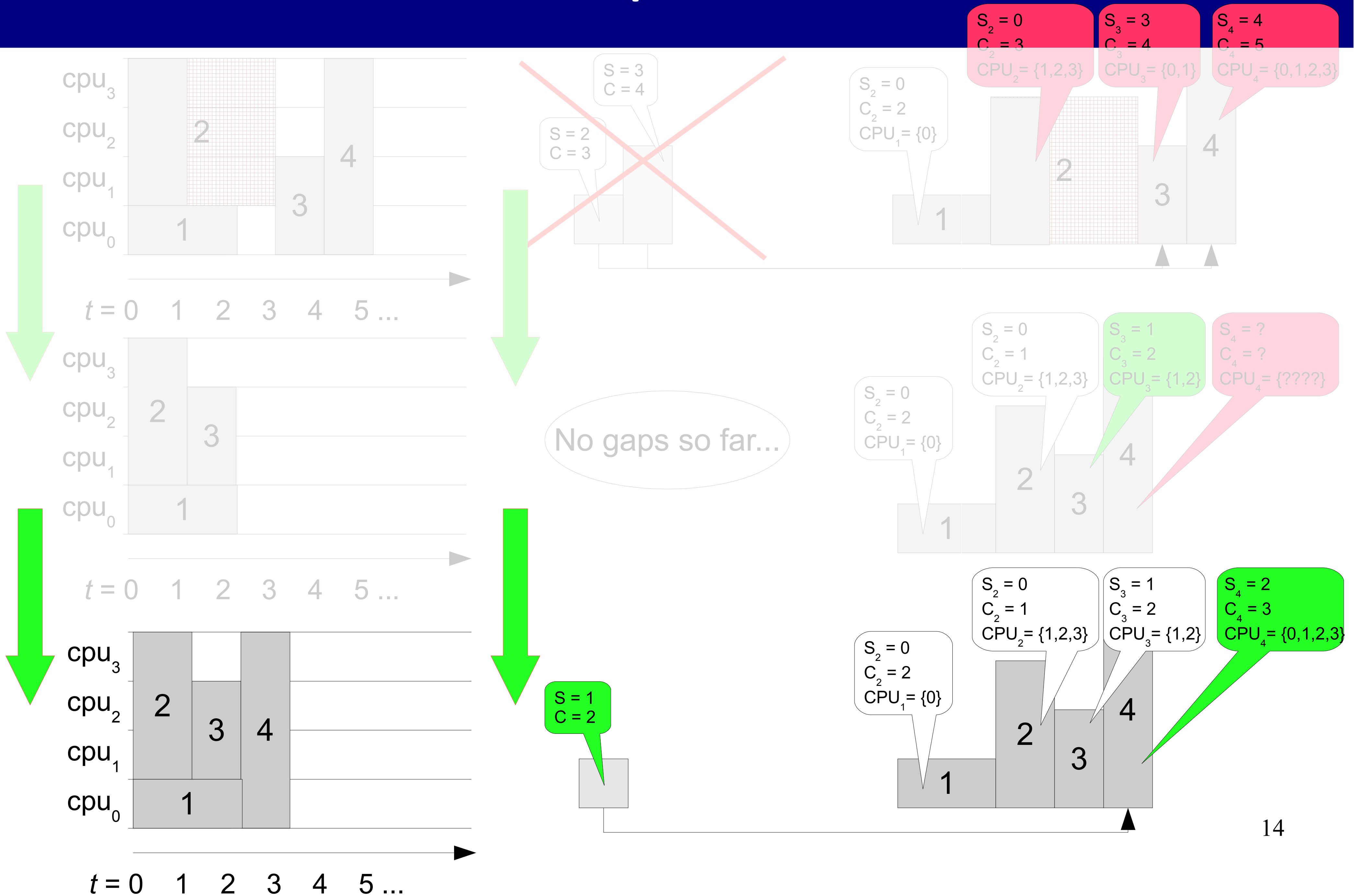
How the Update Goes...



How the Update Goes...



How the Update Goes...



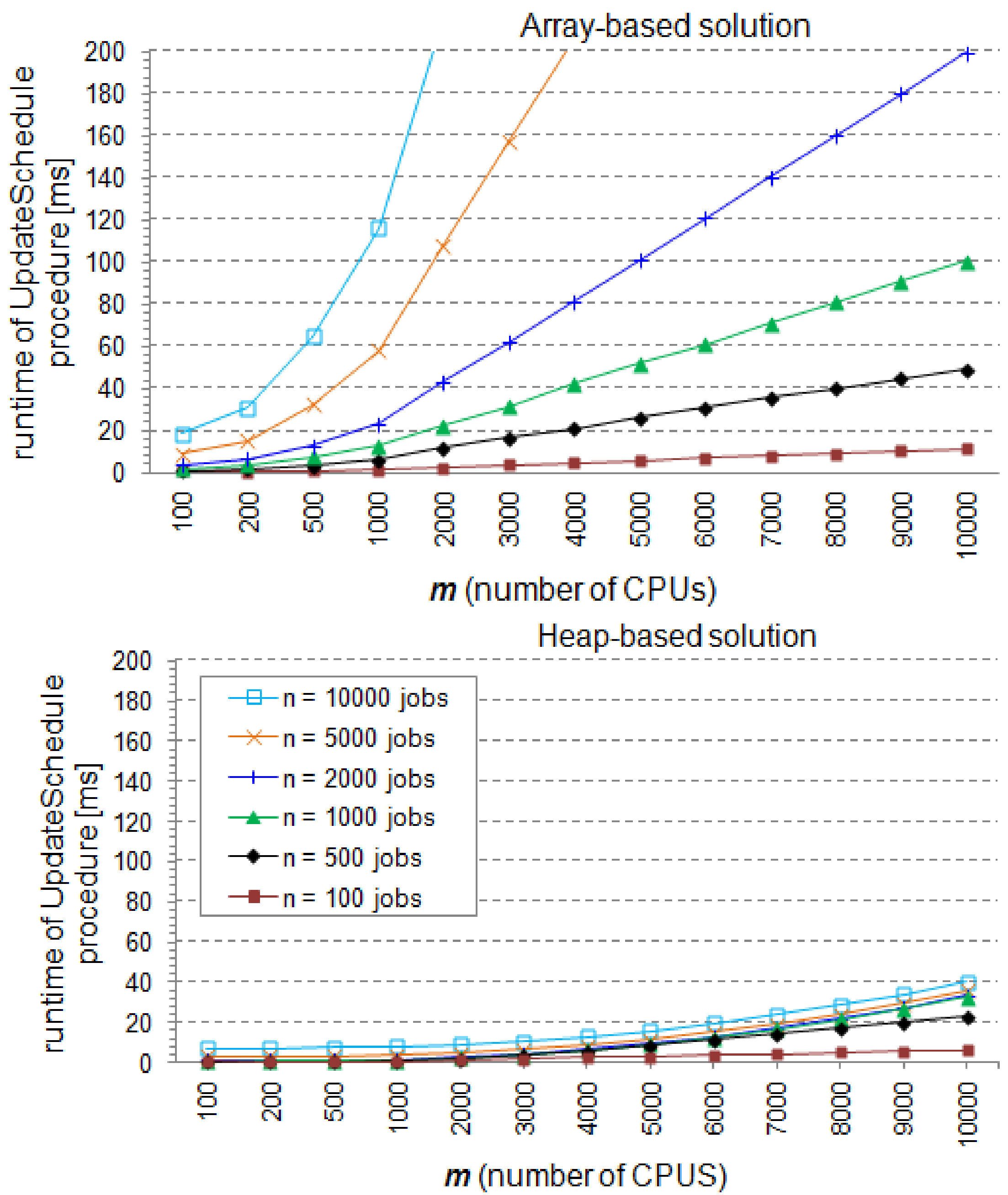
Time Complexity

- Key operation:
  - Finding earliest time slot + CPU selection
  - Let n be the number of jobs, m be the number of CPUs
  - Naive implementation using unordered array: $O(m^2 \cdot n)$
- Binary heap-based structure
  - Each node contains list of CPUs that are free at time = node key
  - Reduces time needed to find earliest time slot
    - best case: $O(1)$
    - worst case: $O(m \cdot \log m)$
  - Heap update: $O(m + \log m) = O(m)$
- The complexity of UpdateProcedure is in $O(m \cdot n)$

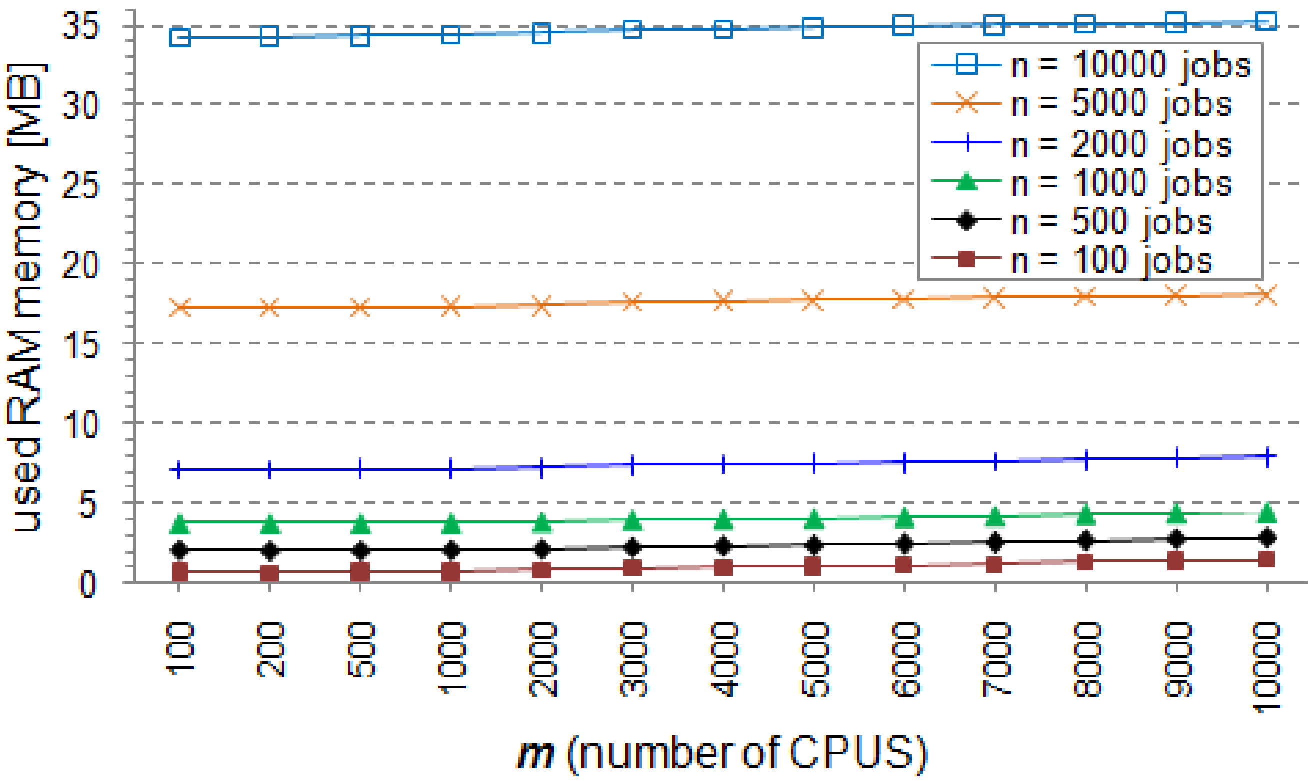
Experimental Evaluation

- Measures the scalability of the schedule structure
- When both ***m*** and ***n*** and is increasing
  - Runtime needed to update the schedule structure
  - RAM usage
- Experiment setup
  - ***n*** = {100, 500, 1000, 2000, 5000, 10,000} jobs
  - ***m*** = {100, 200, 500, 1000, 2000, 10,000} CPUs
  - Job paralelism = {1, 2, … , 128} CPUs per job
  - Each experiment repeated 20 times

Runtime: Array vs. Heap



RAM Usage

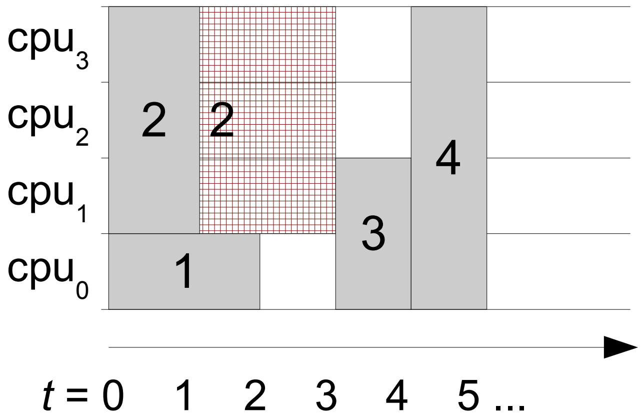


used RAM memory [MB]
35
30
25
20
15
10
5
0
n = 10000 jobs
n = 5000 jobs
n = 2000 jobs
n = 1000 jobs
n = 500 jobs
n = 100 jobs
100
200
500
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
m (number of CPUS)

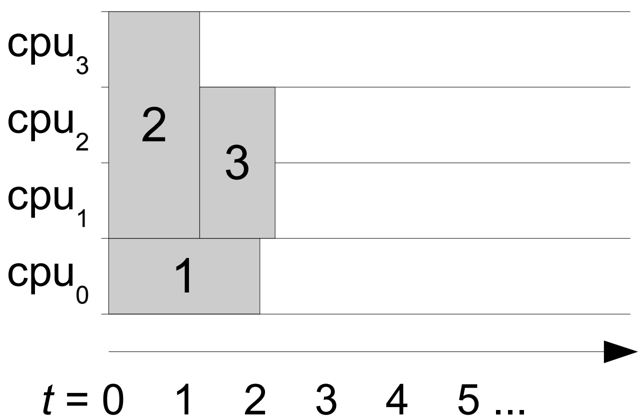
Conclusion

- Efficient schedule representation
  - Scales linearly wrt. number of jobs
  - Gaps are stored in a separate list (useful for scheduling)
- Efficient update procedure
  - Thanks to the use of binary heap
  - Even huge schedules are updated within few miliseconds
- **Current and future work**
  - Implementation of such a structure in production scheduler
  - Torque Resource Management System in MetaCentrum

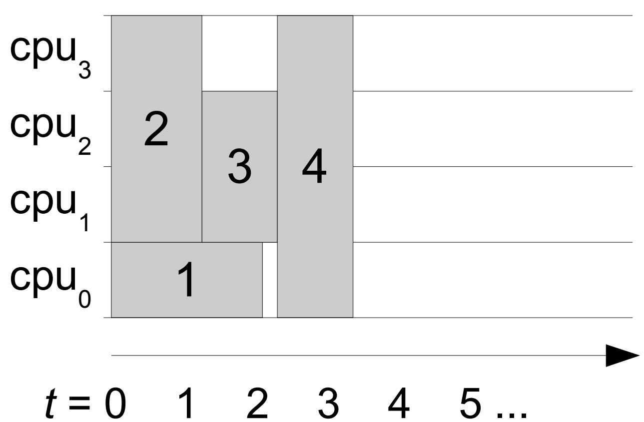
Algorithm Runtime



- Job 2 finished earlier
- Update is started



- Jobs 1 and 2 are inserted (as in previous case)
- Job 3 is inserted
- Earliest start time, completion time and a set of CPUs are found for job 3



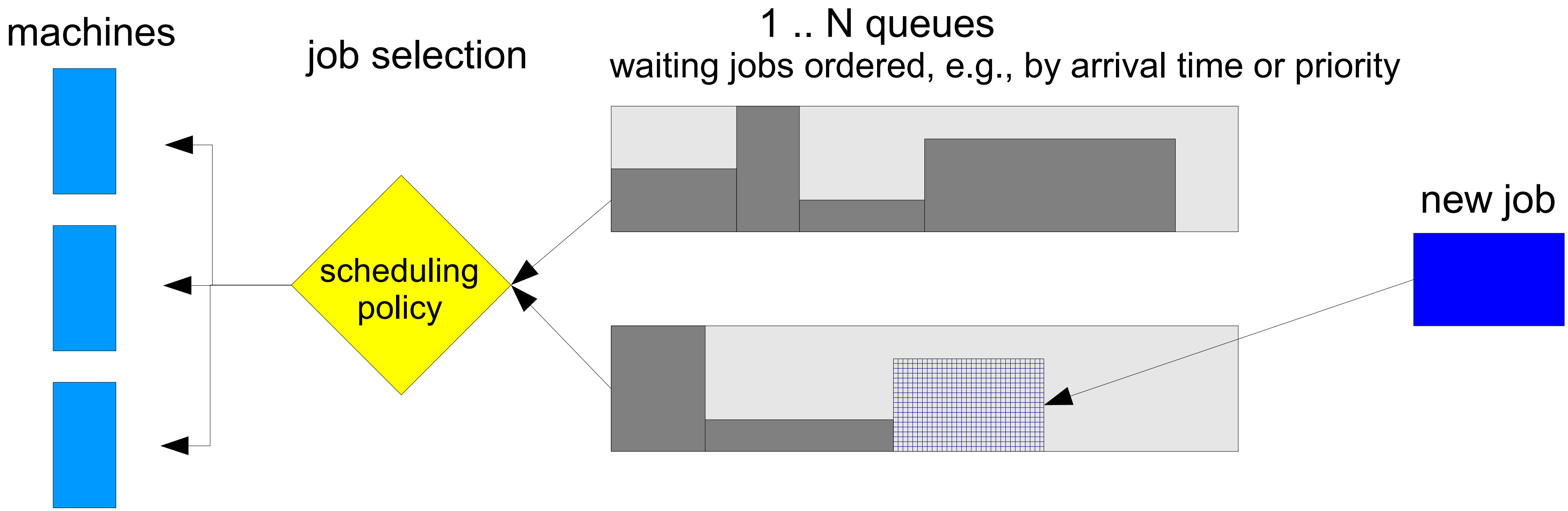
- Job 4 is inserted (2 gaps appear)

Time Complexity

- 1 job in $O(m \cdot \log m)$
- n jobs in $O(m \cdot n)$ – why?
- At the beginning, the heap contains 1 node
- Heap size is at most m
- Each job inserts at most 1 node => $O(m)$
- => all n jobs cannot extract more than n nodes
- => $O(n \cdot \log m)$
- Together $O(n \cdot m) + O(n \cdot \log m) = O(n \cdot (m + \log m)) =$
  $= O(n \cdot m)$.

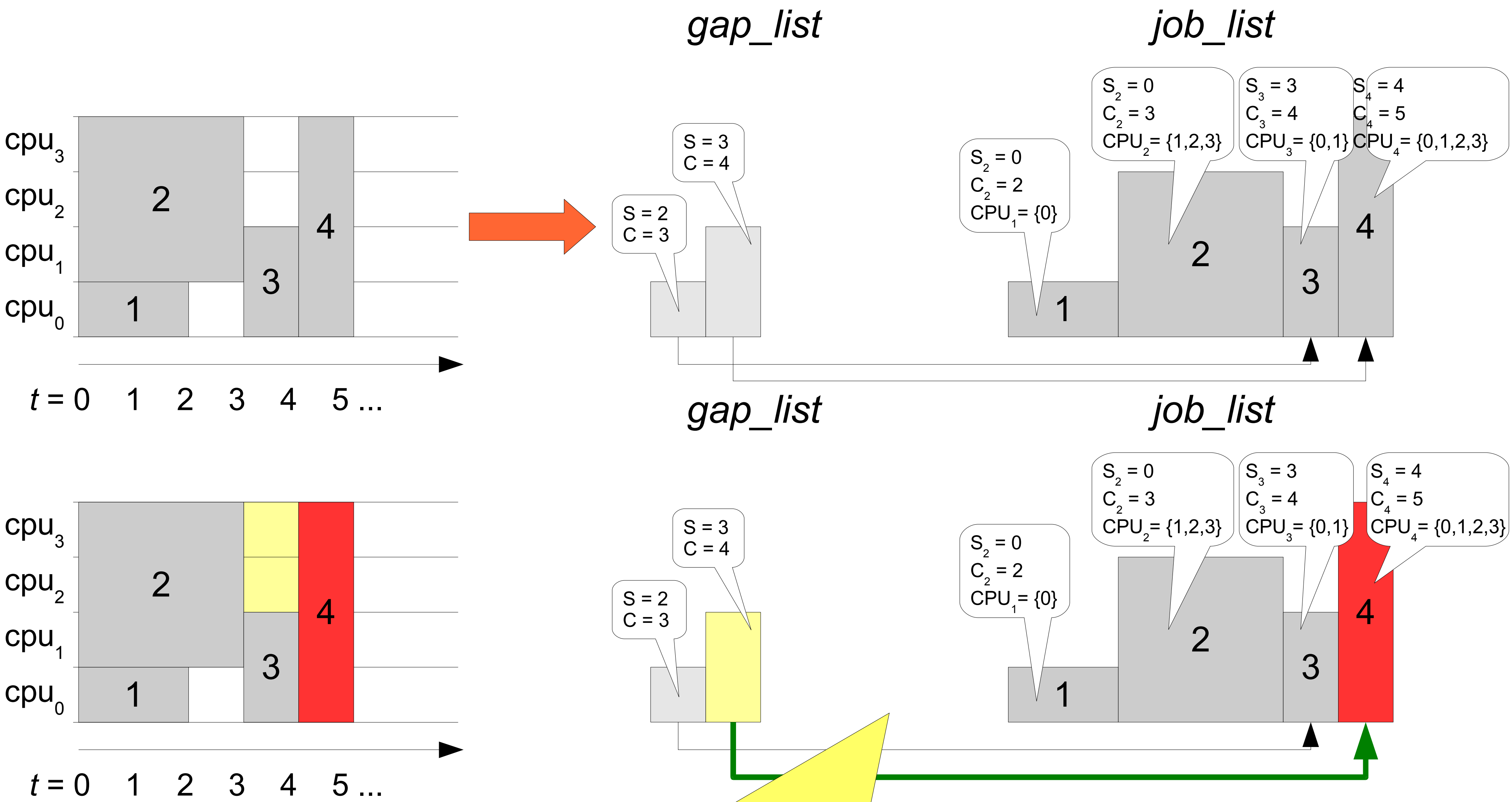
Queue-based Approach

- Standard solution in production systems (PBS, LSF, Torque,...)



- Limited "self control"
- Work in an "ad hoc" fashion
- Limited evaluation, limited prediction

Schedule Representation (3)



The size of such structure is proportional to **2·n**

Runtime of Update Procedure