Trace-Based Analysis of Task Dependency Effects on Performance

15.07.2013 |

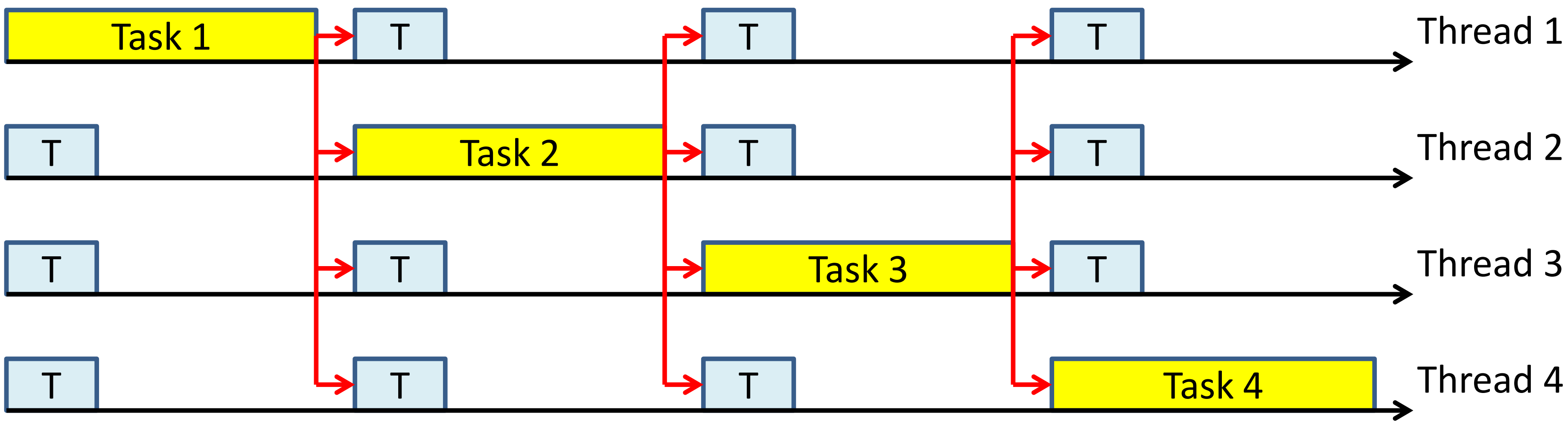
Daniel Lorenz
Jülich Supercomputing Centre

CScADS Tools Workshop
Madison, Wisconsin

Introduction

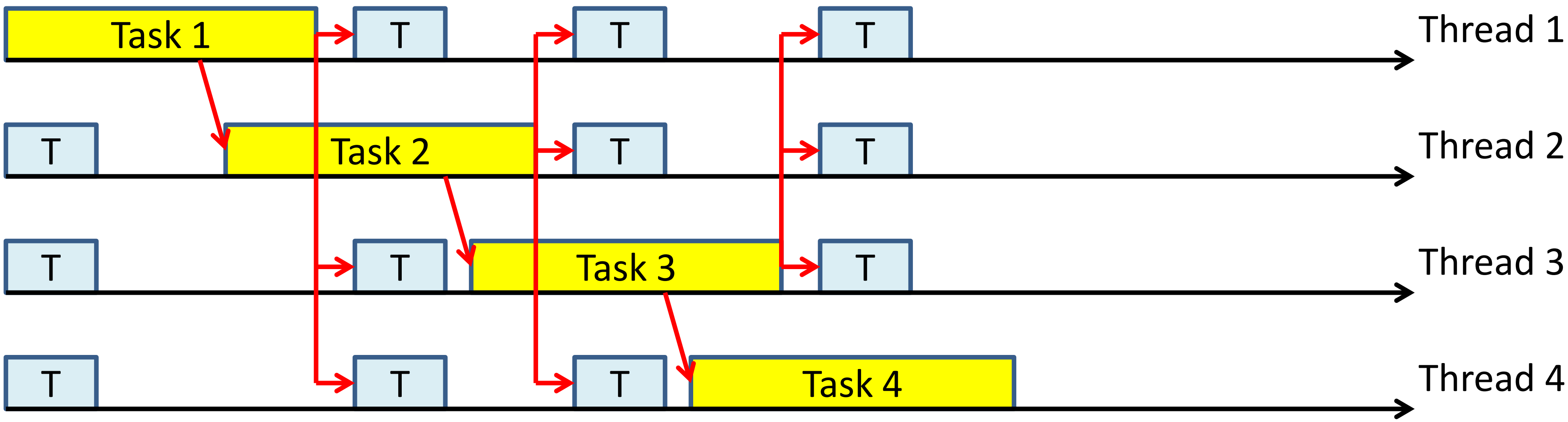
- Tasks provide automatic scheduling and load-balancing
- Profiling can be used to identify tasks of inappropriate size
  - Too small tasks create large management overhead
- Task dependencies can have effects that
  - Create performance loss
  - Execution time profile might provide too little information to understand the reasons

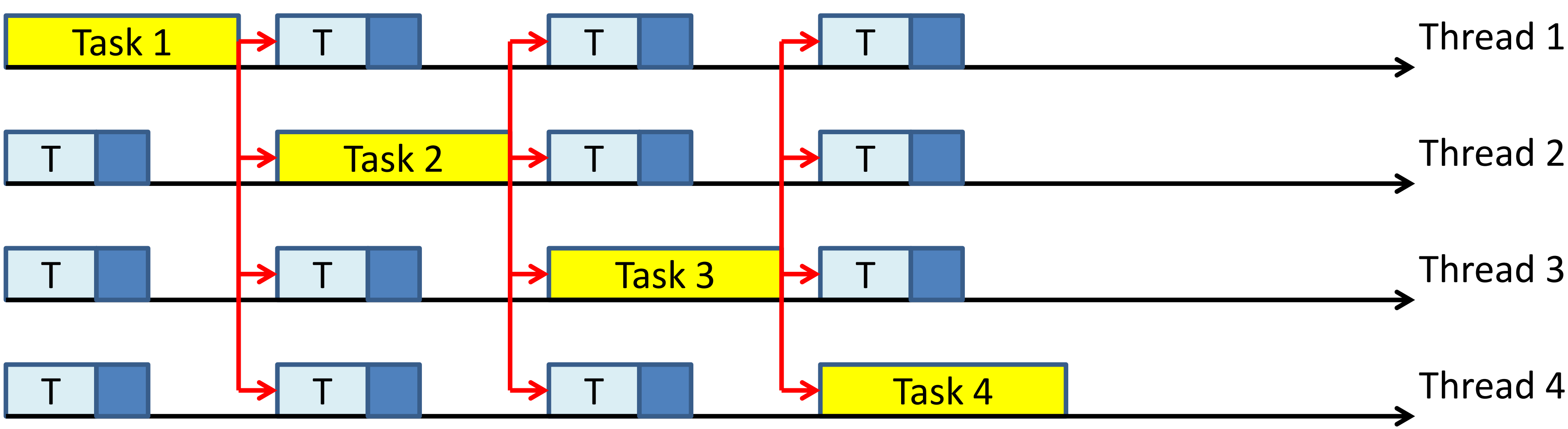
Task dependency case 1



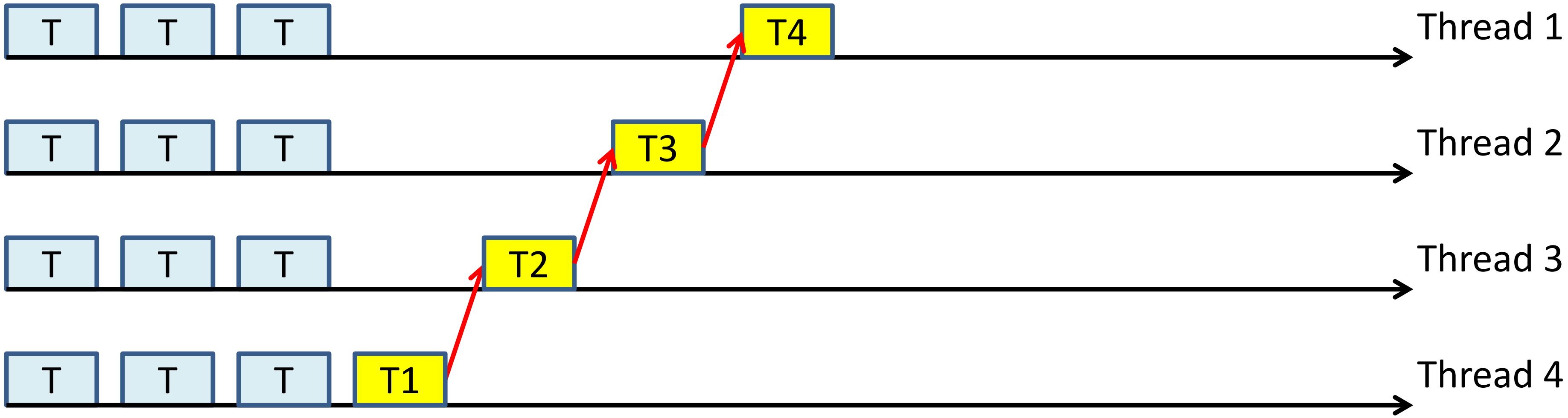
- Critical path determines runtime
  - Execution time of the critical path much larger than average execution time of all threads.
  - Execution time profile is perfectly balanced

Task dependency case 1 (possible improvements)



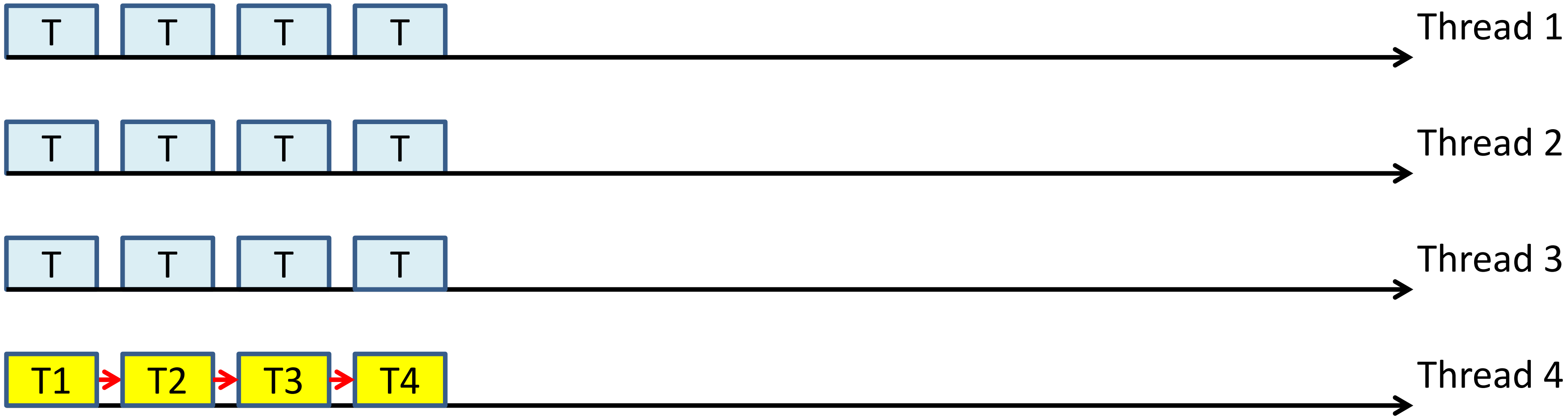


Task dependency case 2



- Suspension / Late start of critical path
  - The critical path is suspended for a significant amount of time
  - Execution time profile is perfectly balanced

Task dependency case 2 (optimal schedule)



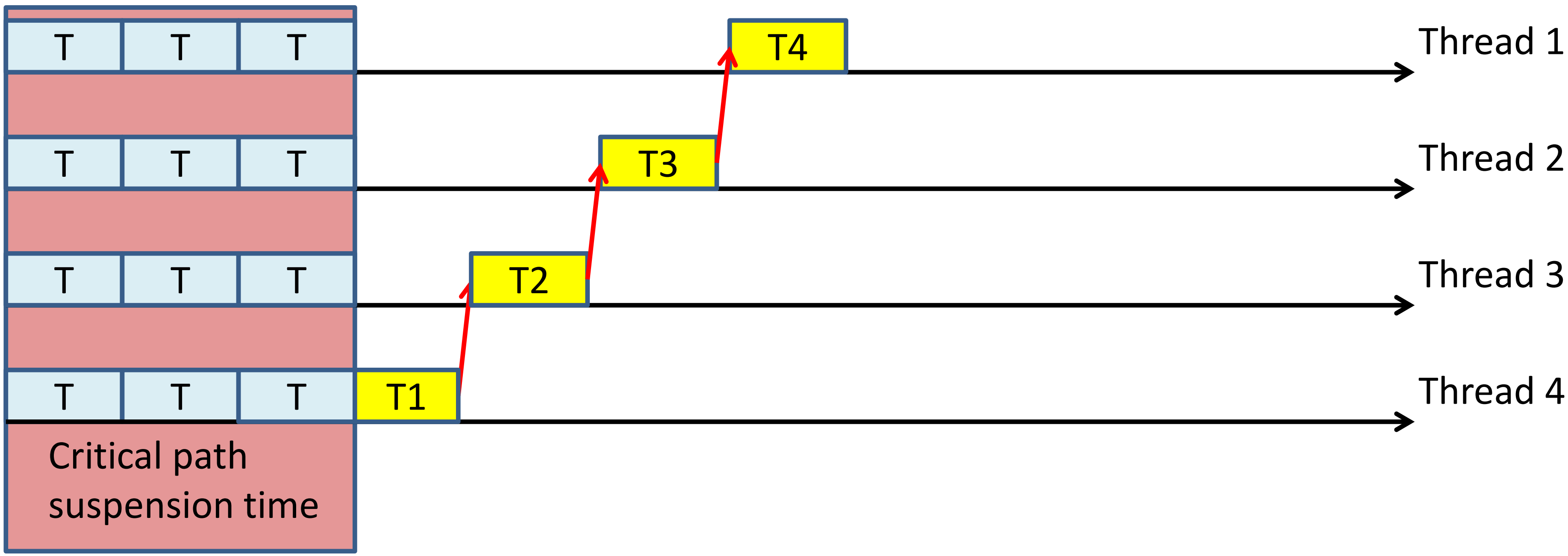
Goal

- Identify task dependency induced performance loss
  - Analysis of the task dependency graph
- Point to causes
- Automatic search over full program run
- Present the analysis result in a small high-level report
  - Manual scan through large number of tasks (e.g., in a time-line view) is tedious
  - The effects may be obscured by other task chains

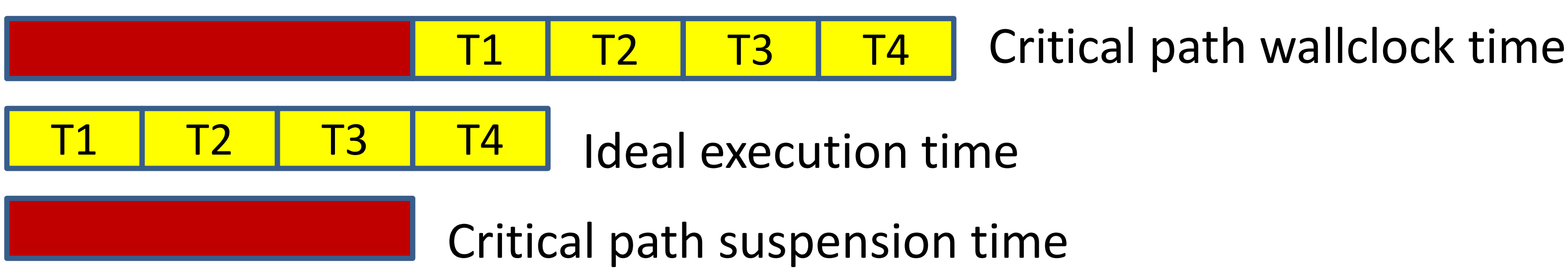
Performance issue detection

- Determine critical path
  - Chain with the longest wall-clock time
- Calculate ideal execution time
  - Sum of execution time of all tasks divided by the number of threads
- If critical path is not significantly longer than ideal execution time
  - No problem
- Else
  - Suspension time on the critical path is always a problem
  - Compare execution time of critical path with ideal execution time and determine imbalance

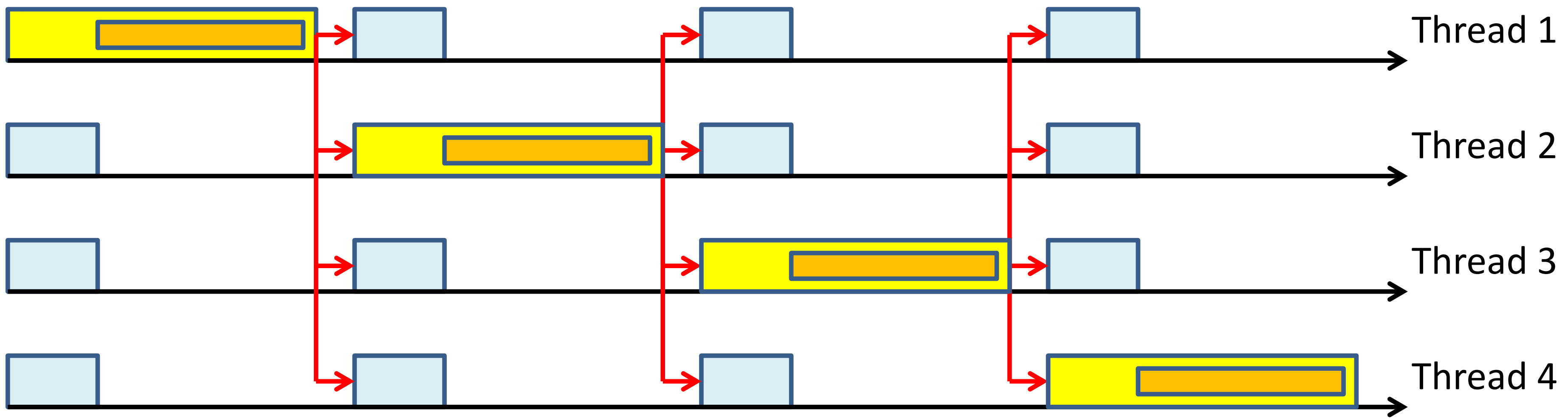
Bad scheduling



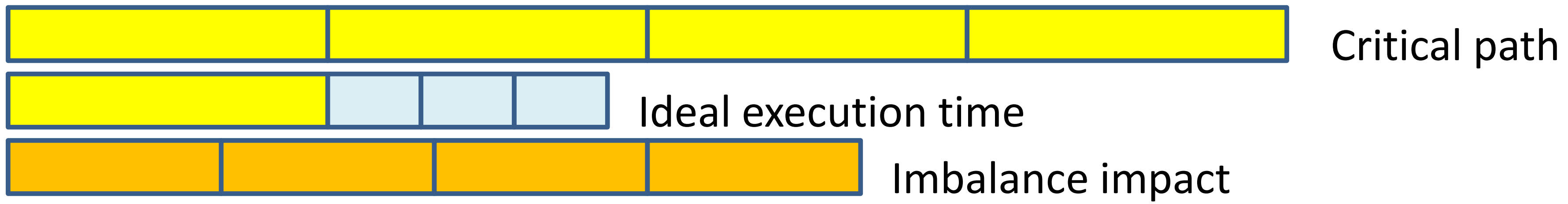
Critical path profile



Critical path imbalance



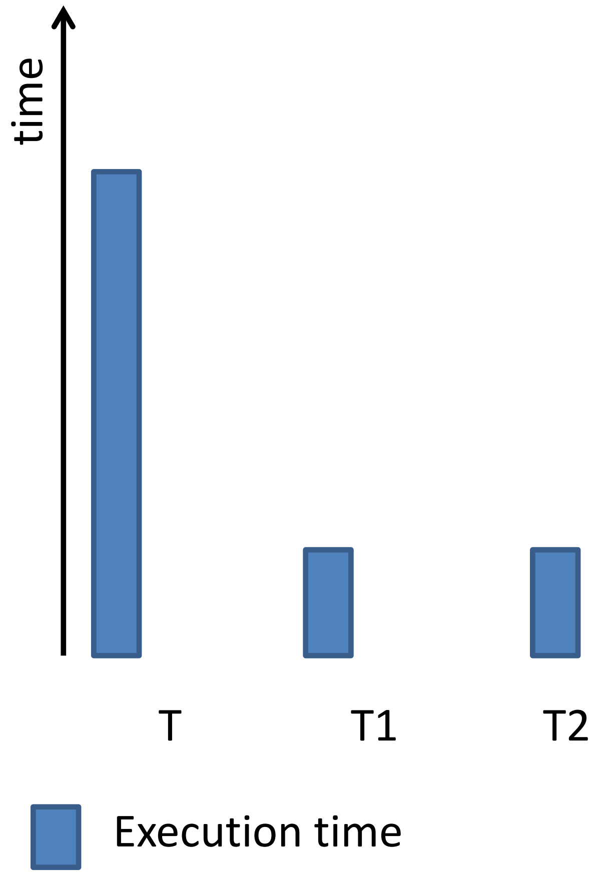
Critical path profile



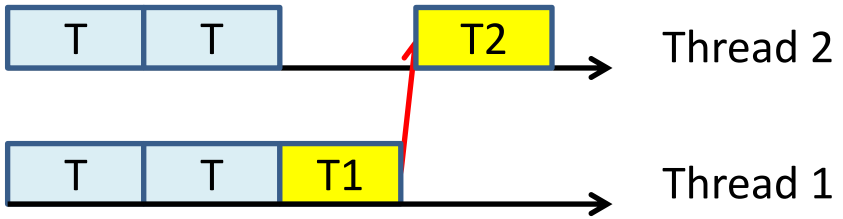
Presentation ideas

- Output on task instance level is too detailed.
  - Output information at an abstract level
  - Map to source code
- Goal: Add information to a profile

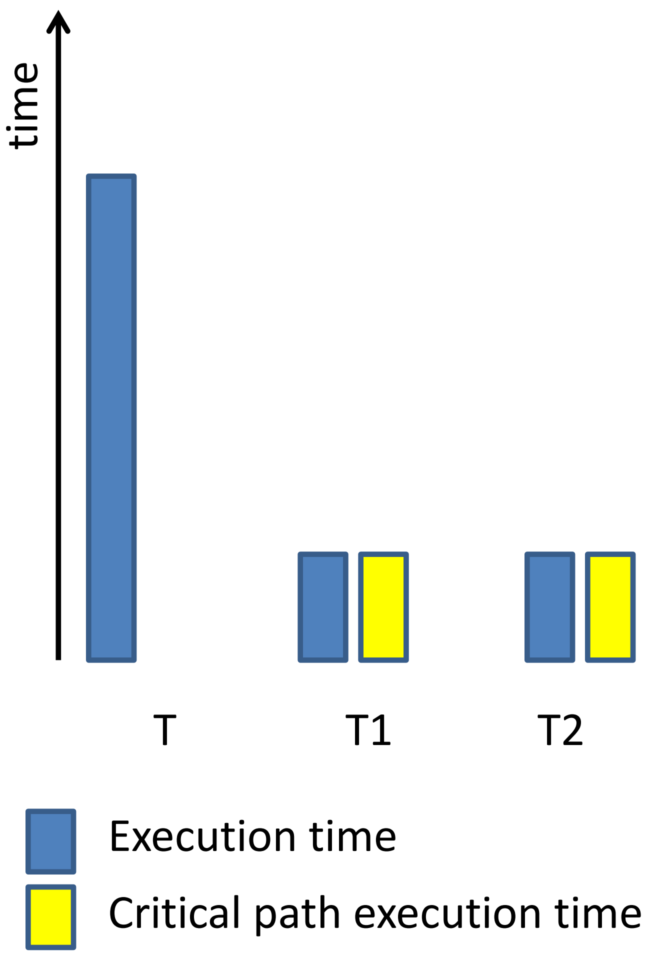
Execution time profile



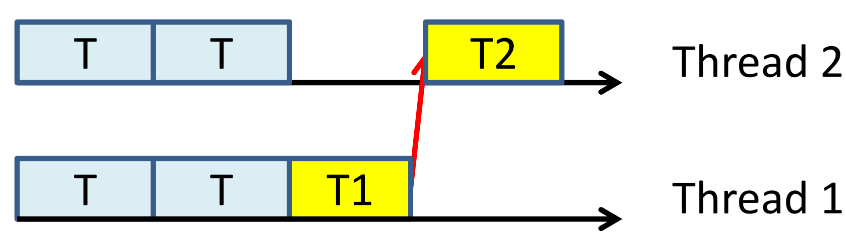
- Aggregates statistics for all visits of a code region / task region
- Does not show that T1 and T2 are the important tasks to improve



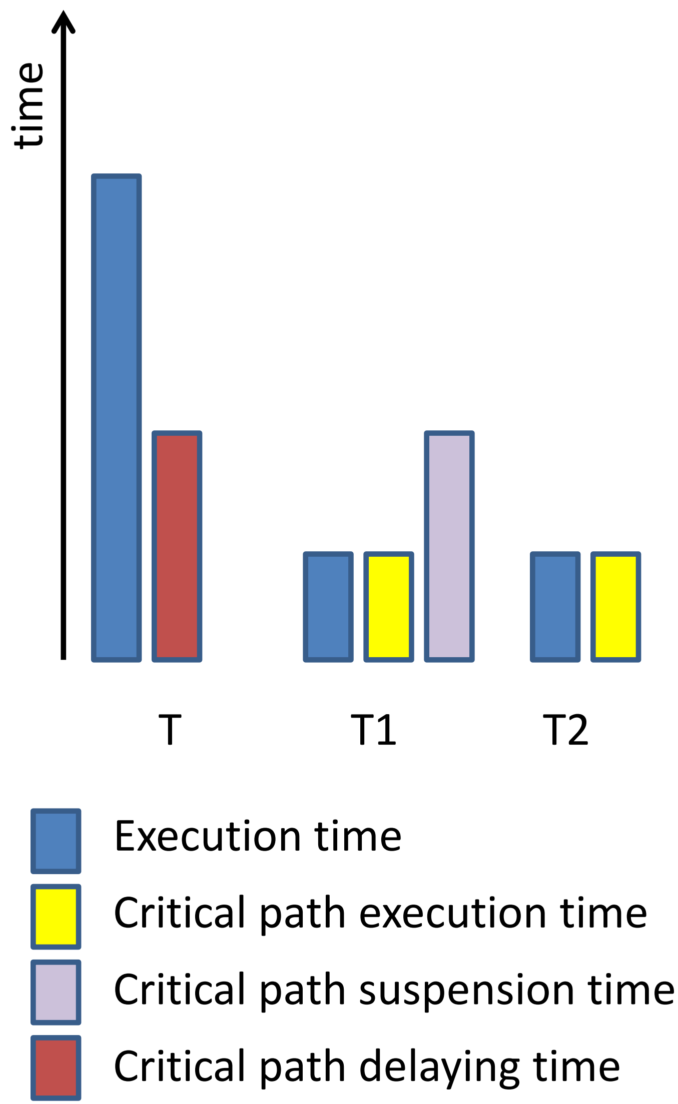
Profile with critical path execution time



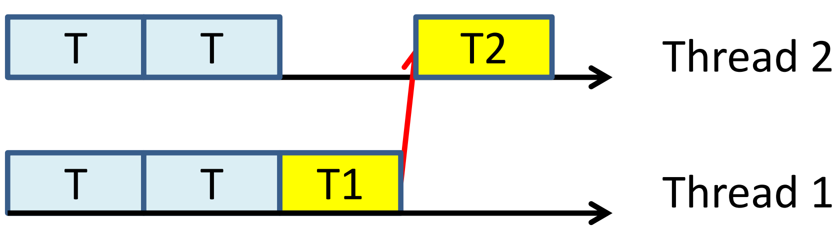
- Add the time spent on the critical path
- Shows that T1 and T2 are the limiting tasks
- But where does the wall clock time come from if the critical path is so short?



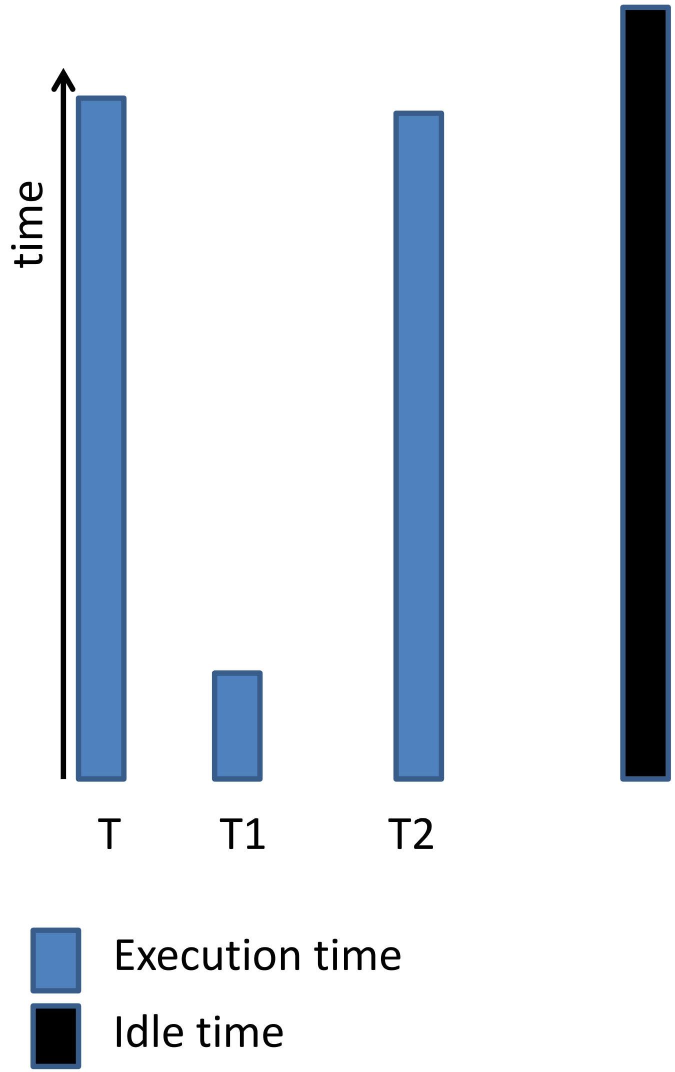
Profile with critical path suspension time



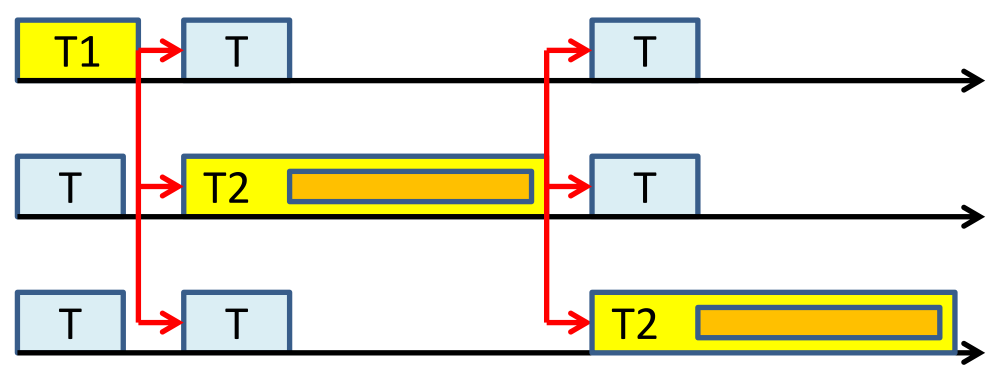
- Add the critical path suspension time to T1
- Blame T for the time it delays the execution of the critical path



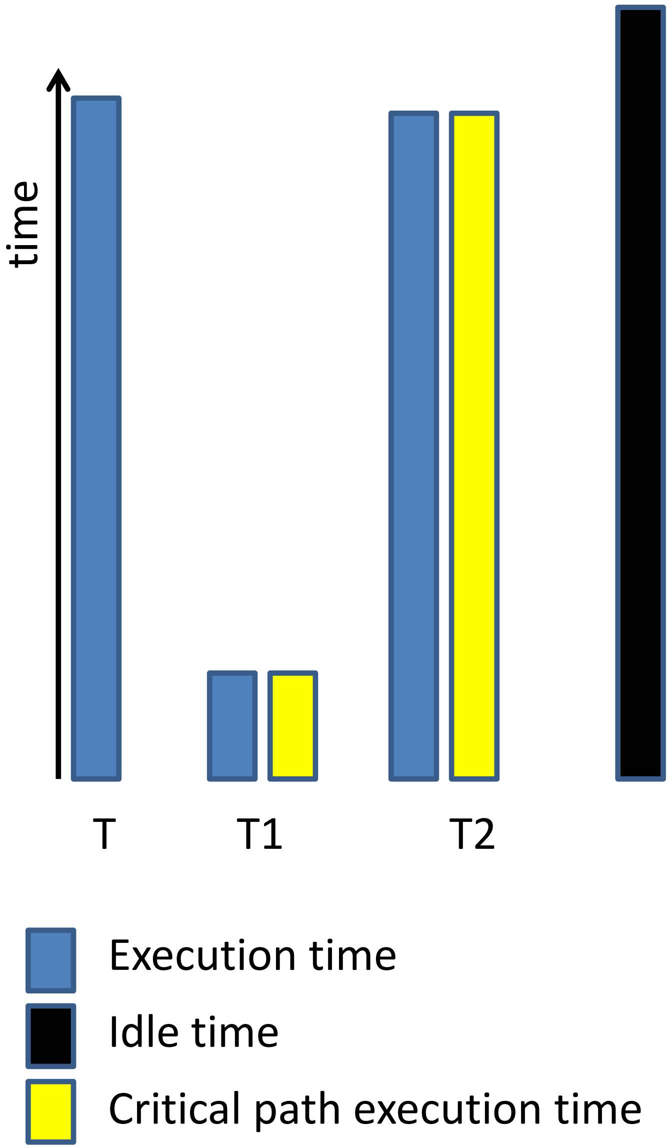
Execution time profile



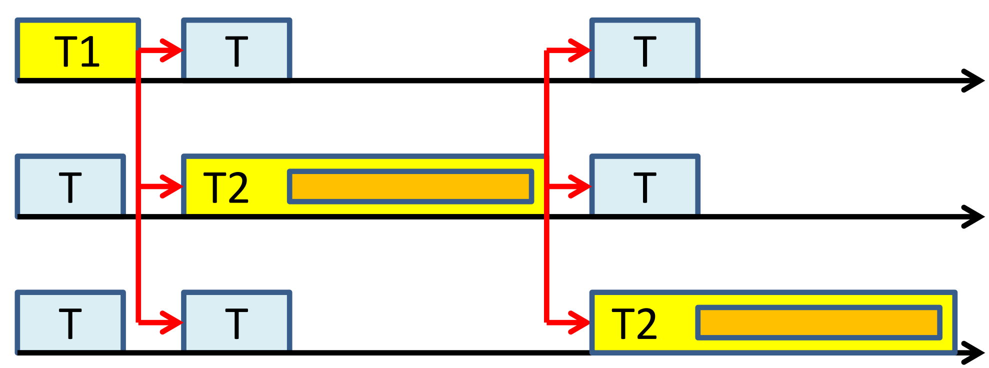
- The execution time does not provide the right hint
- What causes the idle time?



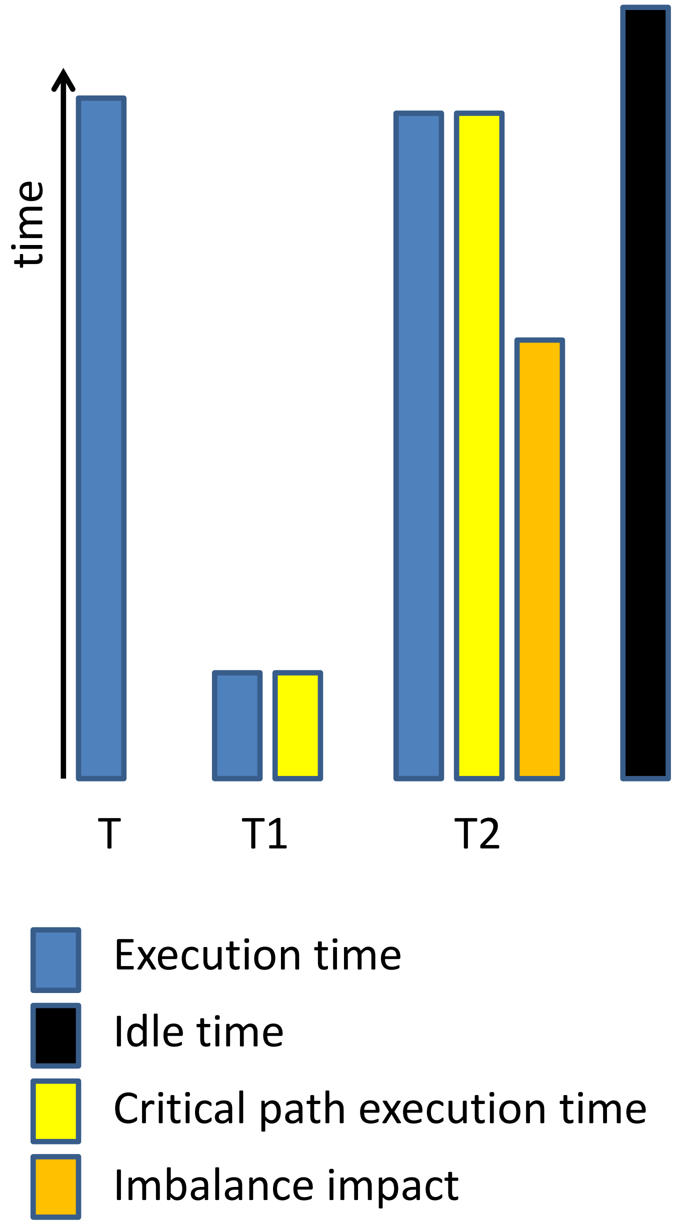
Profile with critical path execution time



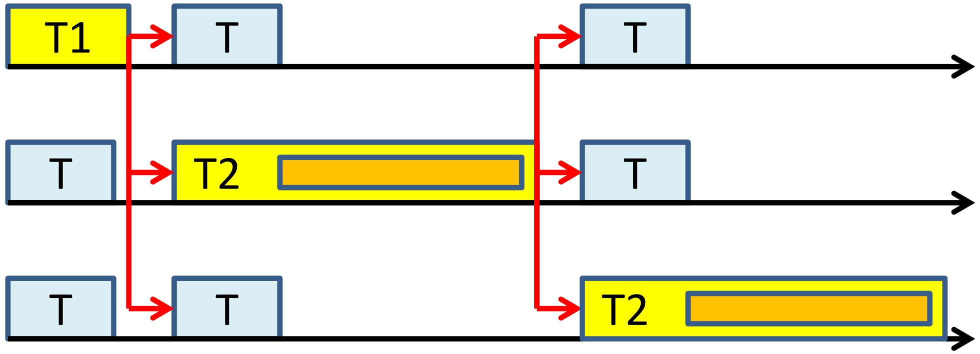
- Adding the critical path metric shows which tasks determine the execution time much better.



Profile with imbalance impact

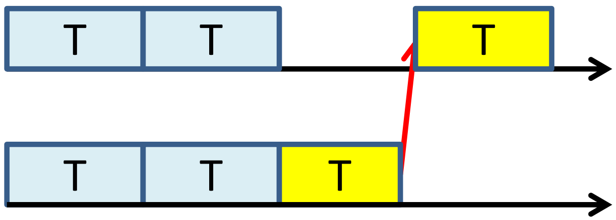


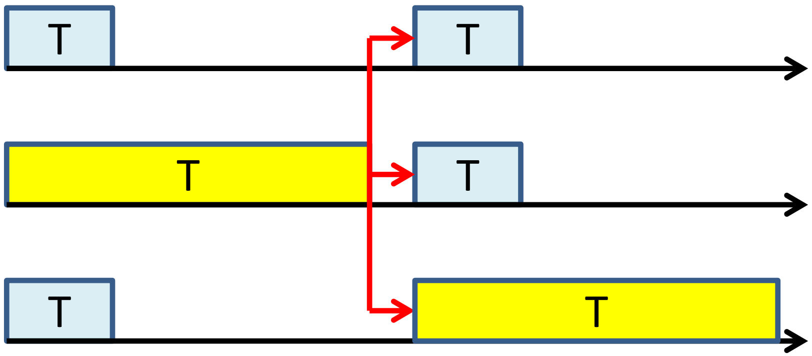
- The imbalance impact pin points where to optimize
- Show optimization potential



How to tell task instances apart?

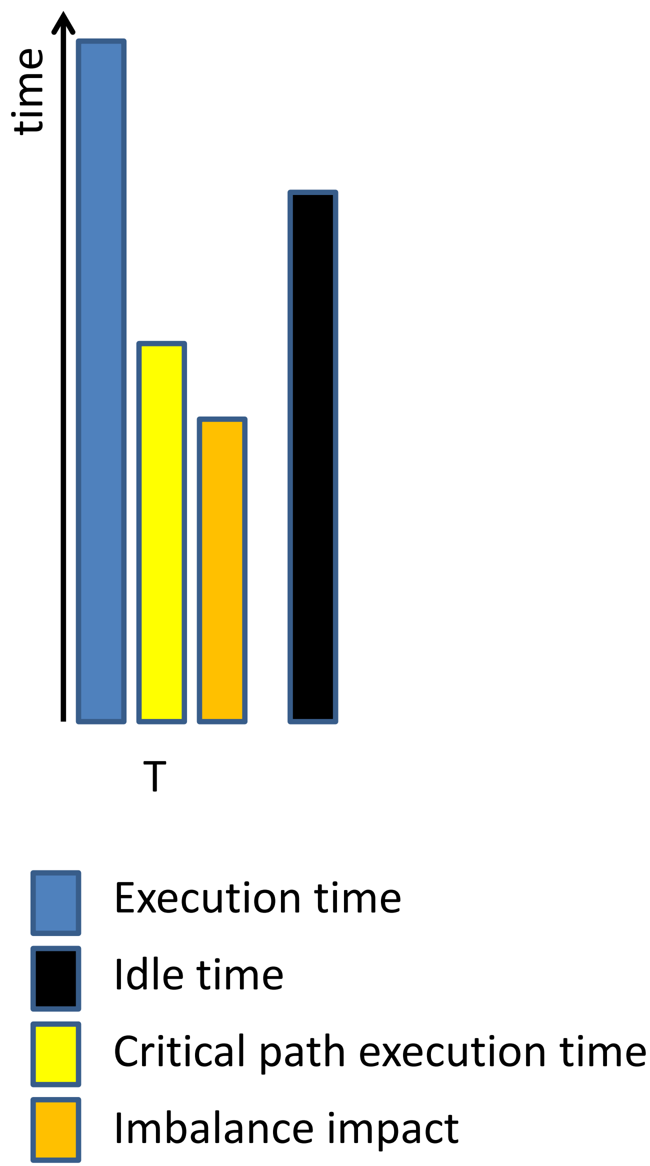
- The profile aggregates all tasks of the same source code region.





- Sometimes all tasks stem from the same region, but behave different
  - Caused by different internal execution path
  - Can be distinguished in a profile by providing
    - *Task-internal call-path data*
    - *Parameter data*

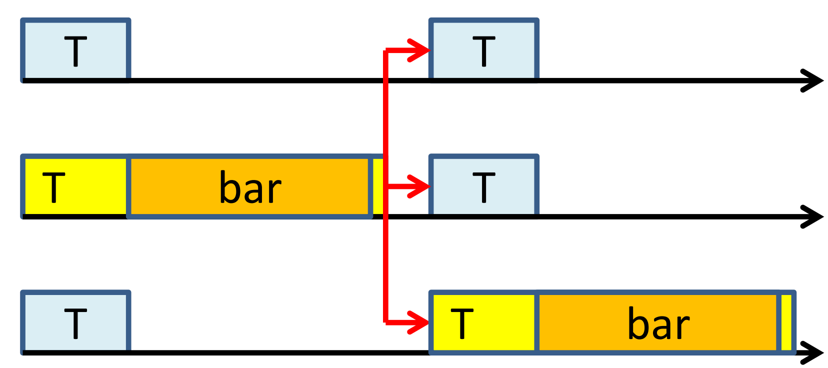
Tell critical path imbalance impact apart



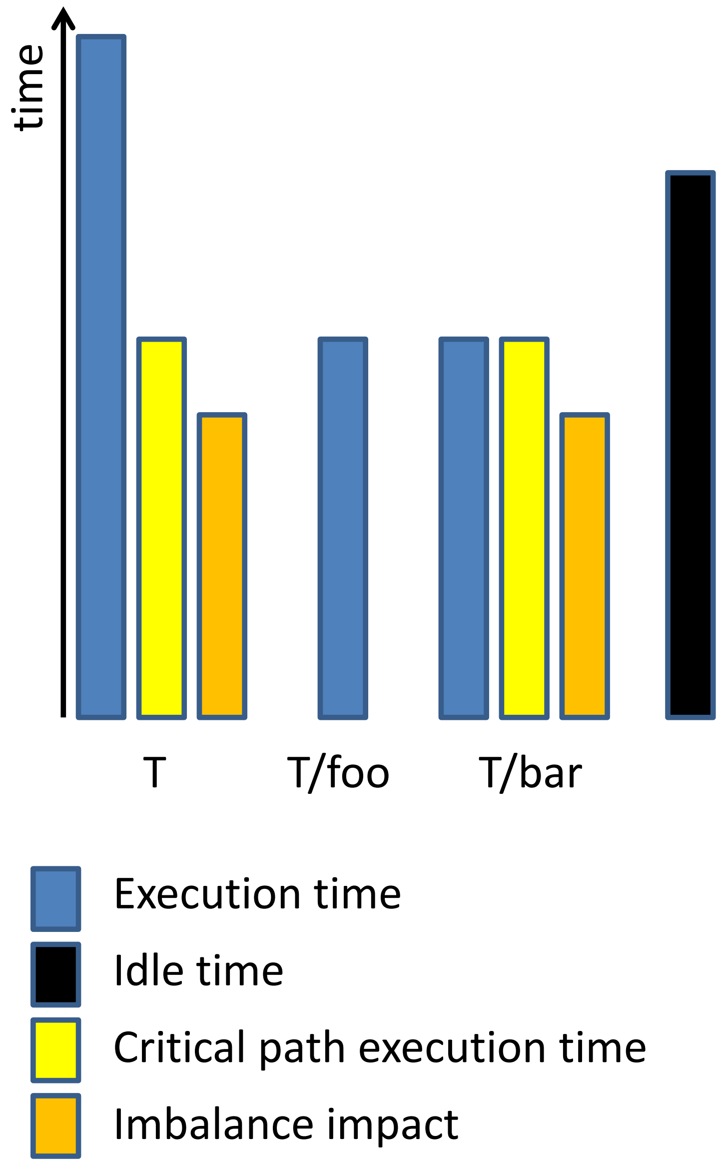
- Example task with conditional exection of foo or bar

```
#pragma omp task // T
{
    if (condition) foo();
    else bar();
}
```

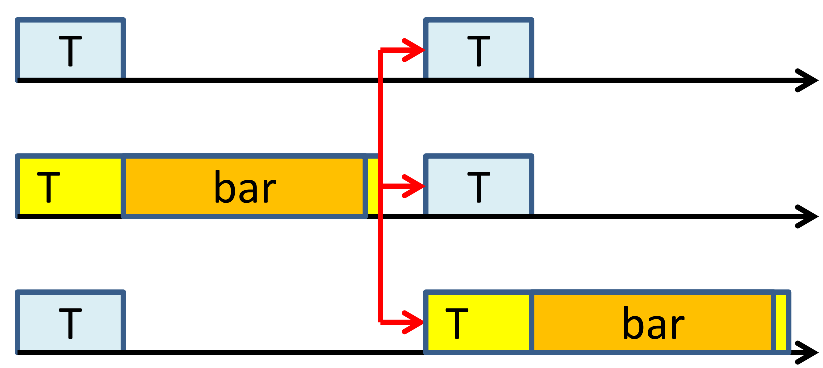
- The total sum tells us that there is some imbalance caused by T, but how?



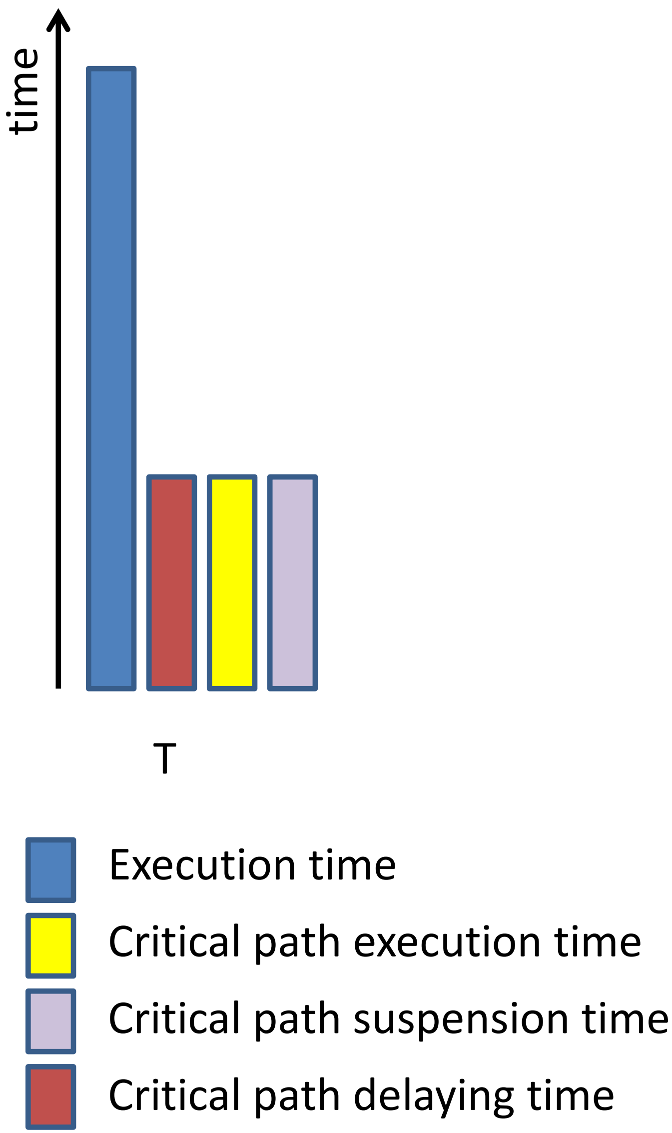
Tell critical path imbalance impact apart



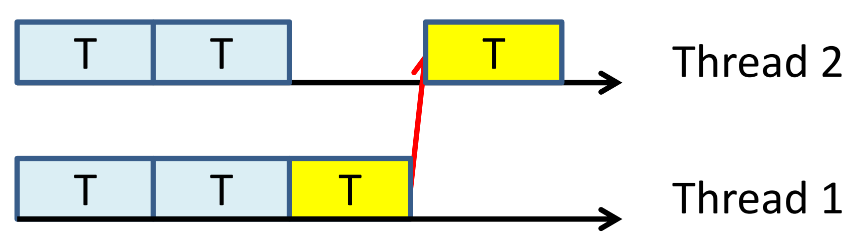
- T/foo does not contribute to the critical path
- The execution path through T/bar creates the long tasks and the imbalance



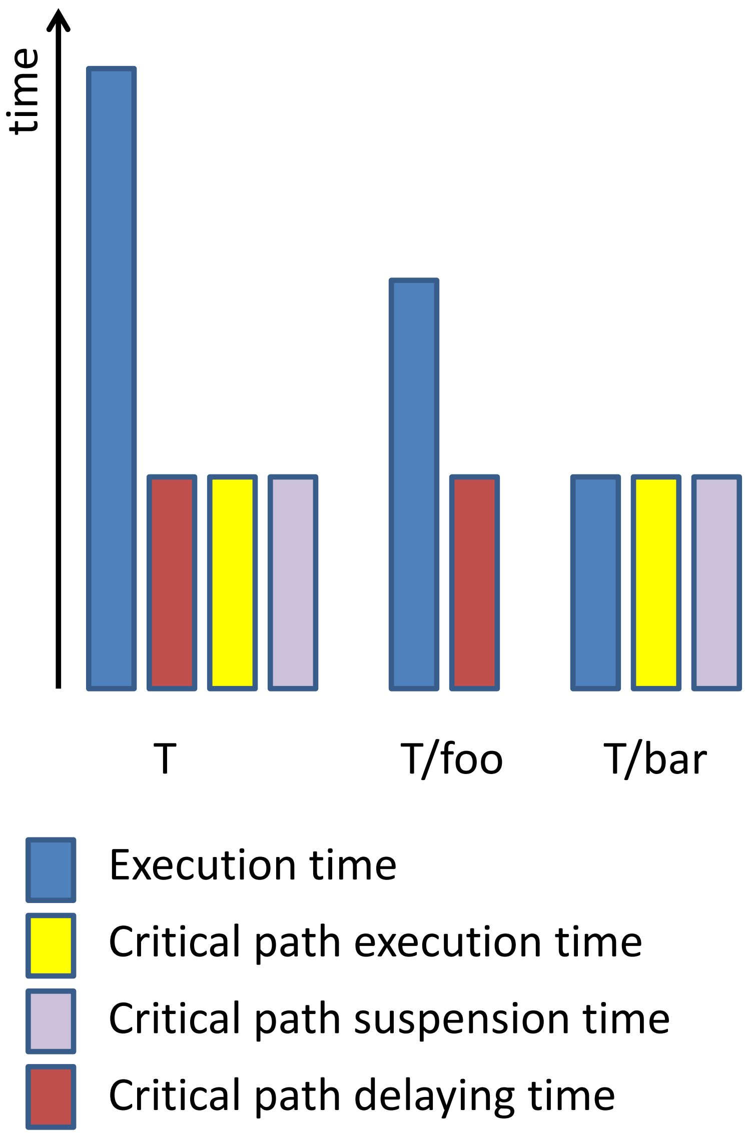
Tell critical path suspension apart



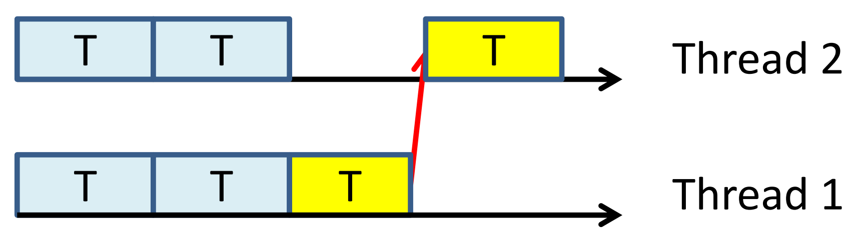
- The sum of all tasks T tells that
  - *Some instances are part of the critical path*
  - *The critical path is delayed by instances of T*
  - *The delayed tasks are instances of T*



Tell critical path suspension apart



- The sum of all tasks T tells that
  - *Some instances are part of the critical path*
  - *The critical path is delayed by instances of T*
  - *The waiting delayed tasks are instances of T*
- Separating the execution path reveals which tasks.



Current status

- Prototypical implementation by Youssef Hatem
  - Graph generation and analysis
  - Outputs dependency graph as intermediate result in CUBE format
  - Implementation produce more vertices and edges
    - *Task creation divides the creator task in two nodes*
- Complexity tests with BOTS benchmark suite
  - Preliminary results
- Missing
  - Mapping to call-path profile

Preliminary: Event, vertices and edge count

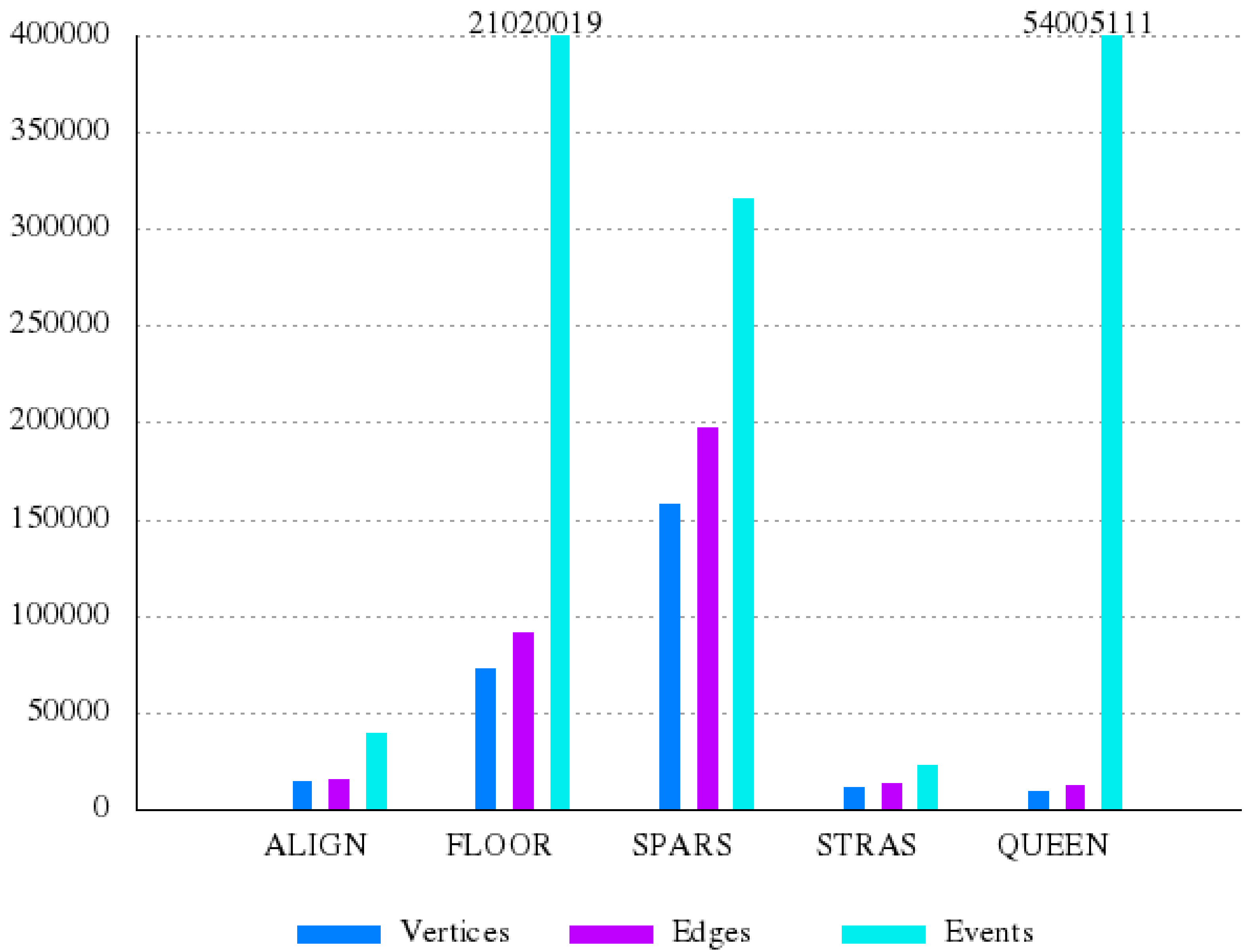


Figure from Youssef Hatem: “Critical pat Analysis of Parallel Applications Using OpenMP Tasks”

Preliminary: Analysis time

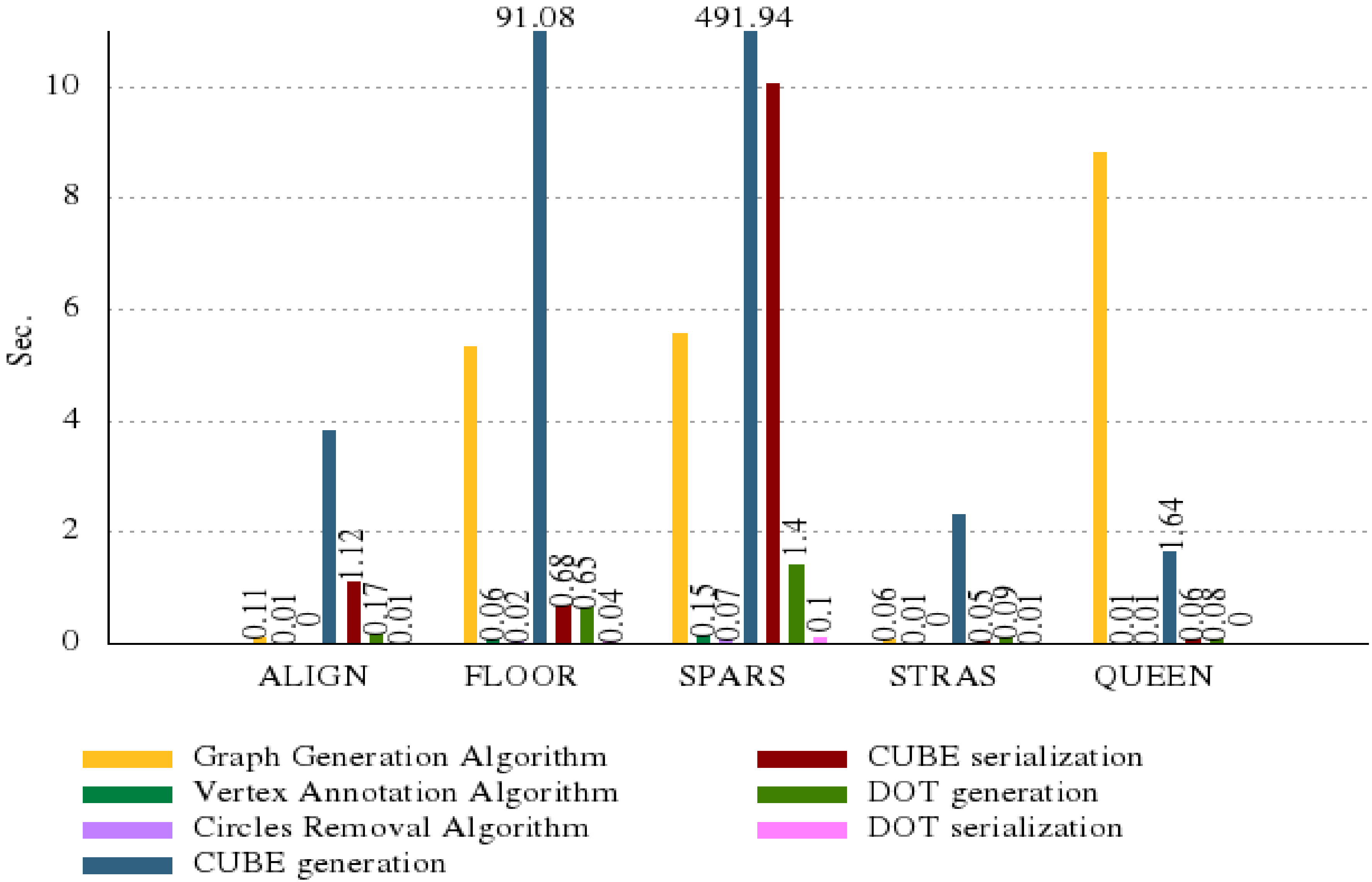


Figure from Youssef Hatem: “Critical pat Analysis of Parallel Applications Using OpenMP Tasks”

Preliminary: Number of events

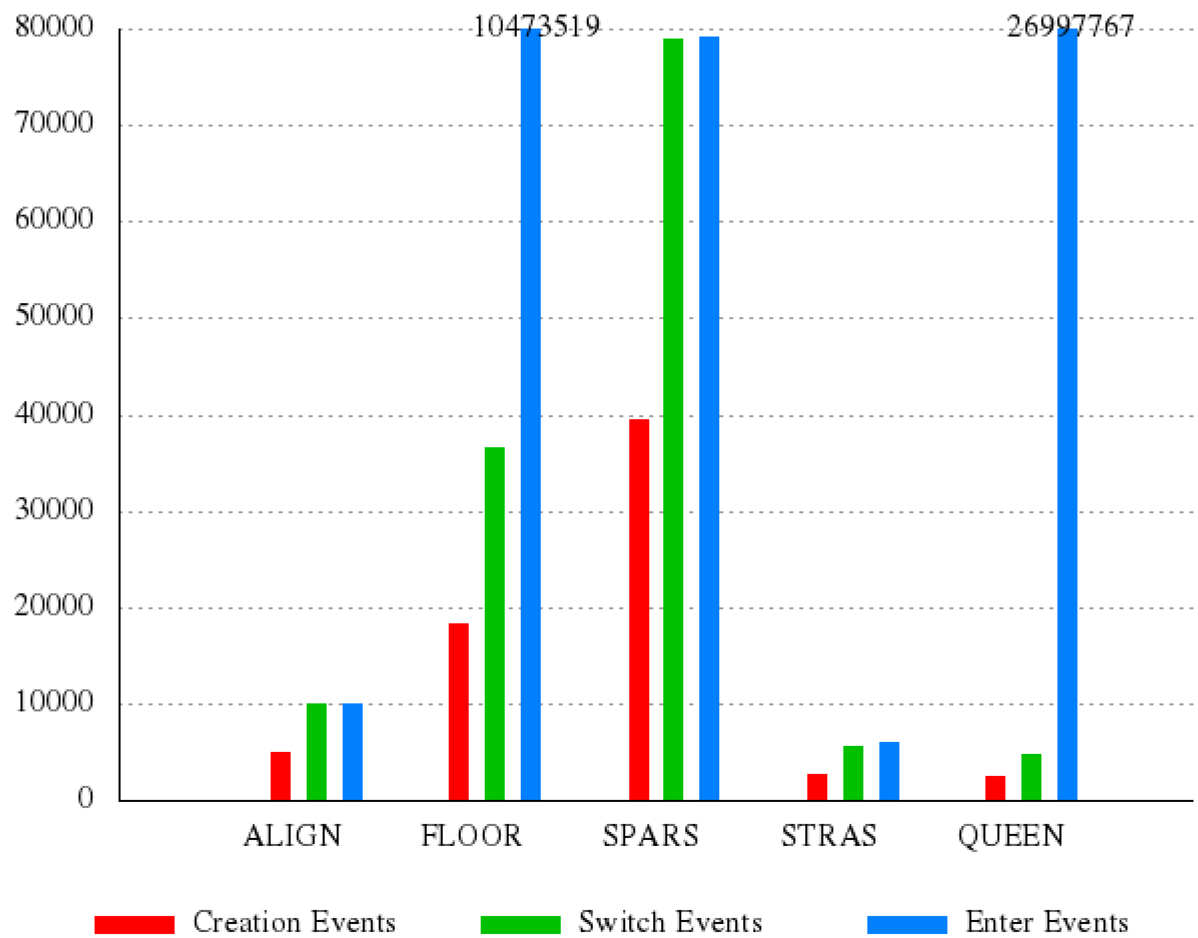


Figure from Youssef Hatem: “Critical pat Analysis of Parallel Applications Using OpenMP Tasks”

JÜLICH
FORSCHUNGSZENTRUM

Thanks for your attention