Chapter 6

The Quantitative Partitioning Problem

6.1 Theoretical Aspects

- Let be

  T(1) the execution time on one processor

  T(p) the execution time on a p processor system
- The gain by parallel computing is expressed by

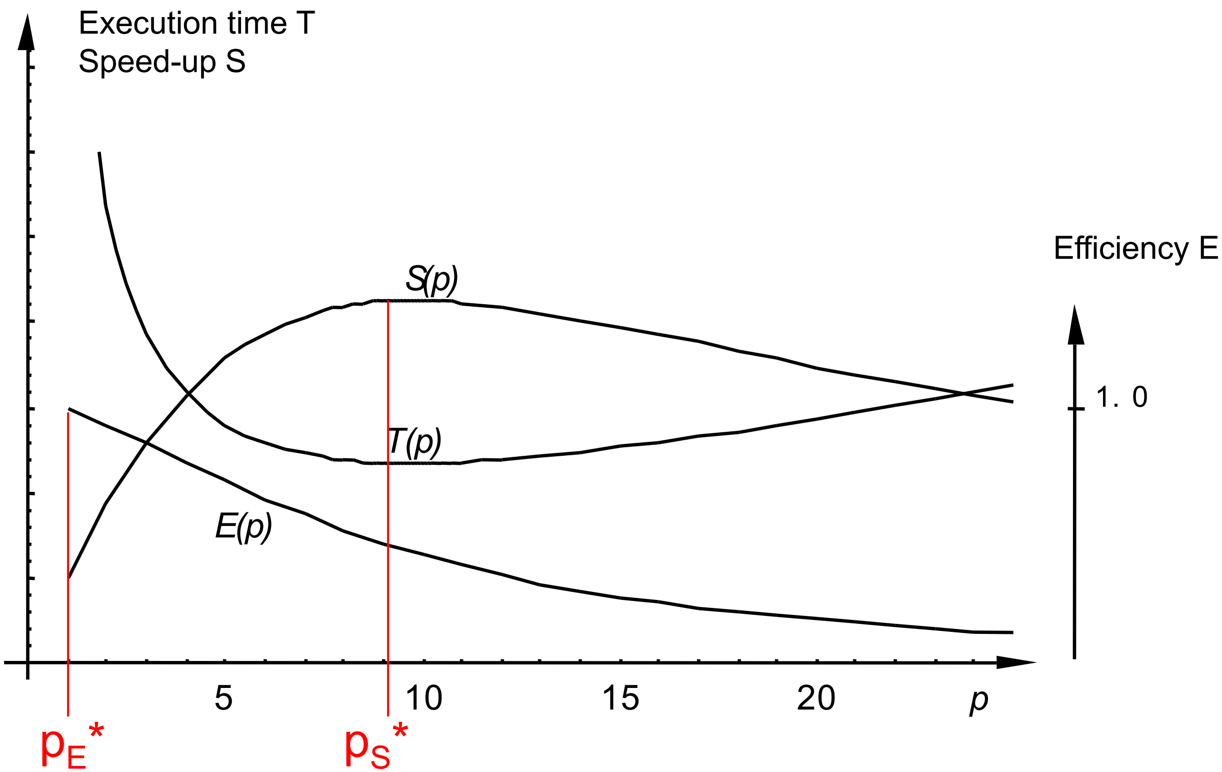
  $S(p) := T(1) / T(p)$ *Speed-up*

- Normalizing the Speed-up by dividing by the number *p* of processors is defined as the **efficiency**:

  $E(p) := S(p) / p$ *Efficiency*

Conflict of interests

- Cost minimization (Minimizing execution time or maximizing speed-up, respectively)
- Benefit maximization (Maximization of efficiency)



Speed-up efficiency

- Compromise in conflict of interests – Optimization of Cost-Benefit-Ratio:
- **Speed-Up Efficiency** η (Benefit at unit cost)

$$\eta(p) = \frac{E(p)}{T(p)} T(1) = E(p) \cdot S(p) = \frac{S(p)^2}{p}$$

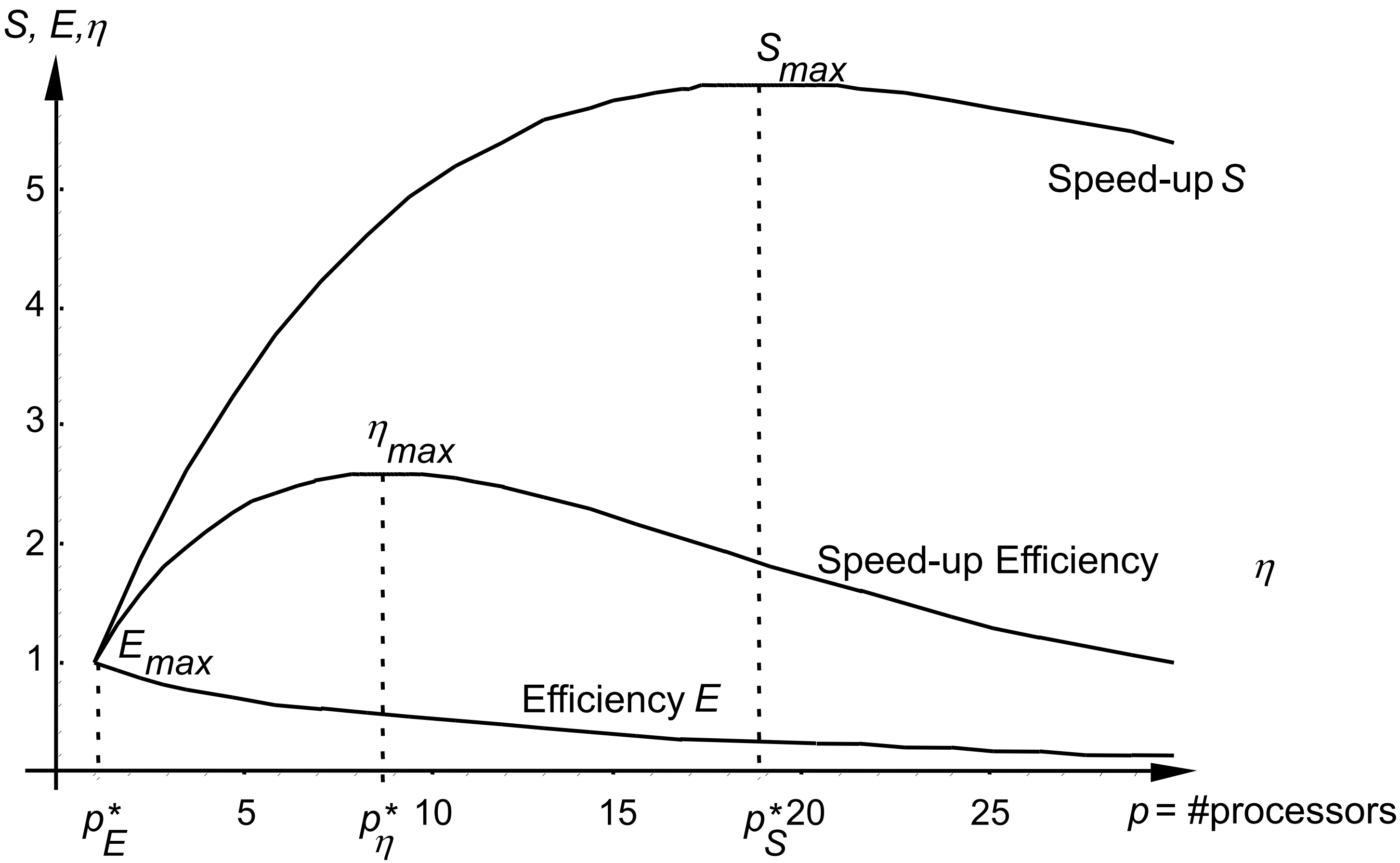
- Considering $\eta(p)$ as a two times differentiable function of a continuous p, we find a maximum at p_η*.

$$\frac{d\eta}{dp}\left(p_\eta^*\right) = 0 \quad \text{with} \quad \frac{d^2\eta}{dp^2}\left(p_\eta^*\right) < 0$$

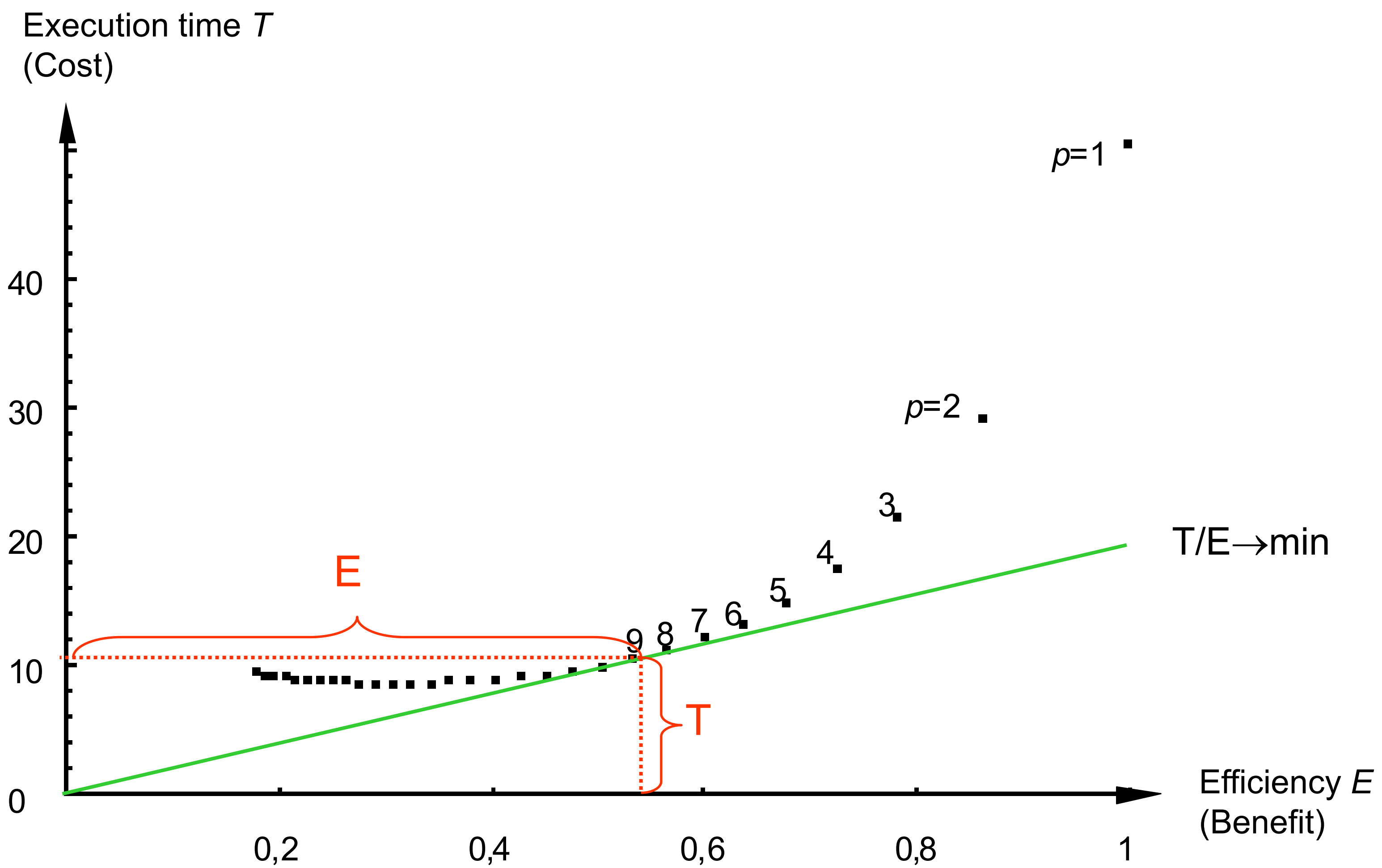
  p_η* is called **processor working set** and indicates the number of processors that minimizes the cost-benefit ratio *T*/*E*.

- $\eta(p)$ is sometimes also called **Power**.

Speed-up efficiency



The „Knee" in the Cost-Benefit-Function



Optimal Number of Processors

Depending on the general goal, there is a specific optimal number of processors p_{opt} for each program:

- Maximization of throughput and thus of the efficiency:

  Optimal number is $p_{opt} = p_E{}^* = 1$ for all programs

  Caution: This is only true if processors behave independent from each other. This is not given in most multi-core systems as cores share resources (cache, memory bandwidth, power, …) and therefore influence each other. Here, detailed evaluation is necessary.
- Minimization of execution time (Maximization of Speed-up):

  Optimal number is $p_{opt} = p_S{}^*$ individually for each program
- Maximization of the speed-up efficiency:

  Optimal number is $p_{opt} = p_\eta{}^*$ individually for each program

6.2 Static Partitioning

- Given:
  - A set *M* of parallel programs, with known processor demand *p(i)* and execution time *T(i) = T(p(i))*.
  - Either *p* and *T* are firmly specified for each program or we know the speed-up function of the programs and calculate for each program *i* the optimal demand $p_{opt}(i)$ and the resulting execution time $T(p_{opt})$.

- Problem:
  - Find a schedule for the *M* programs, such that the total execution time (makespan) is minimized.

Definitions

- Let A = (A_1, A_2, ..., A_M) be the sequence of requests (programs), p the number of available processors, $p(i)$ the number of processors demanded by A_i and $T(i)$ the execution time of A_i.
- **A schedule** S is a mapping of start times $t(i)$ to requests (programs) A_i.
- Schedule S is called **valid**, if at each point in time the sum of all occupied processors does not exceed p.
- $T(S) = max\ \{t(i)+T(i)\}$ is the **length** of the schedule, also called **makespan**.

$$U(S) = \frac{1}{p \cdot T(S)} \sum_{i=1}^{M} p(i) \cdot T(i)$$

is the machine utilization under schedule S

$$W(S) = \frac{1}{M} \sum_{i=1}^{M} t(i)$$

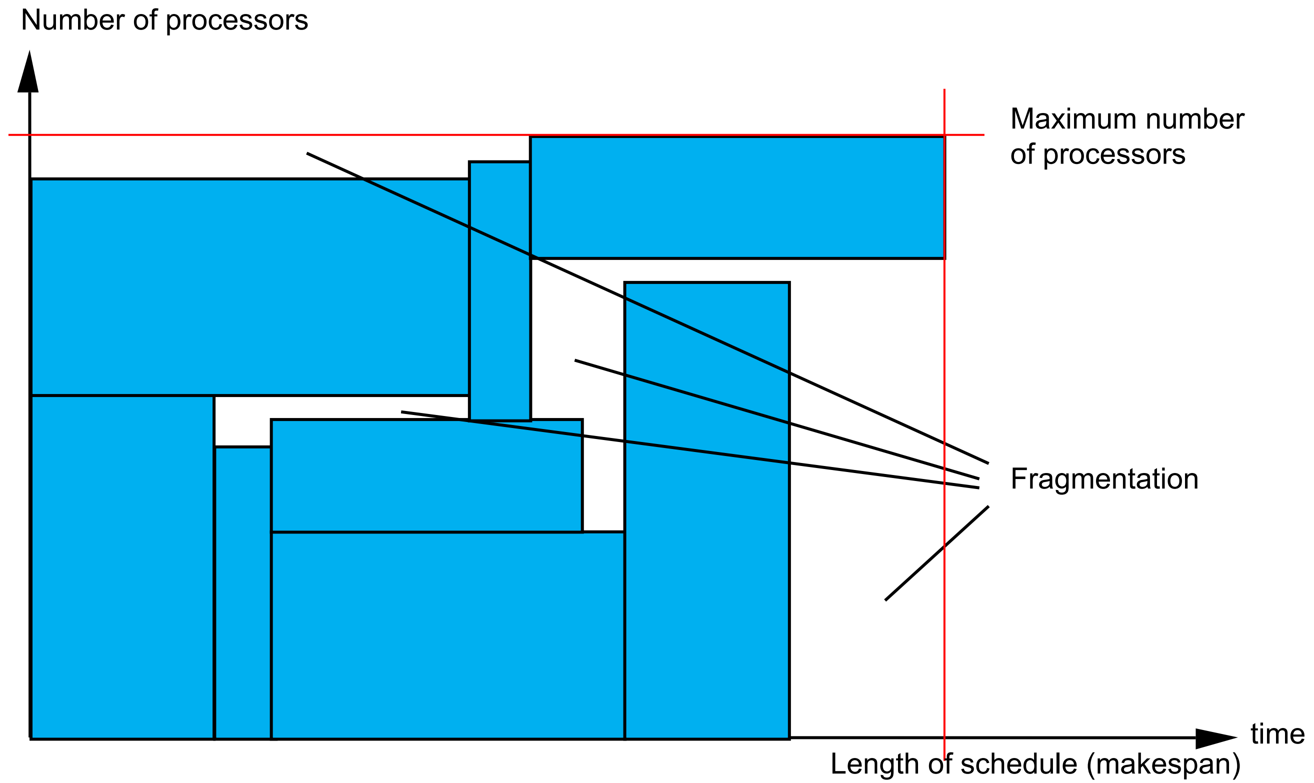
is the mean waiting time

$$R(S) = \frac{1}{M} \sum_{i=1}^{M} \left(t(i) + T(i)\right)$$

is the mean response time

Interpretation as 2D-Bin-Packing-Problem

- Programm *i* is represented as rectangle with edge lengths p_{opt} and $T(p_{opt})$.
- Goal: Find a placement of the rectangles such that the maximum number of processors is not exceeded and the makespan is minimized.



2D-Bin-Packing

- The problem is NP-complete.
- Heuristic approaches are:
  - FCFS: The requests are processed in the order of arrival.
  - FFDH (First Fit Decreasing Height): The requests are ordered according to their execution times (decreasing).
  - FFIH (First Fit Increasing Height): The requests are ordered according to their execution times (increasing).
- Example sequence

i	1	2	3	4	5	6	7	8	9	10
p(i)	16	256	16	256	32	128	32	128	64	64
T(i)	25	50	10	5	20	40	20	10	15	30

FCFS

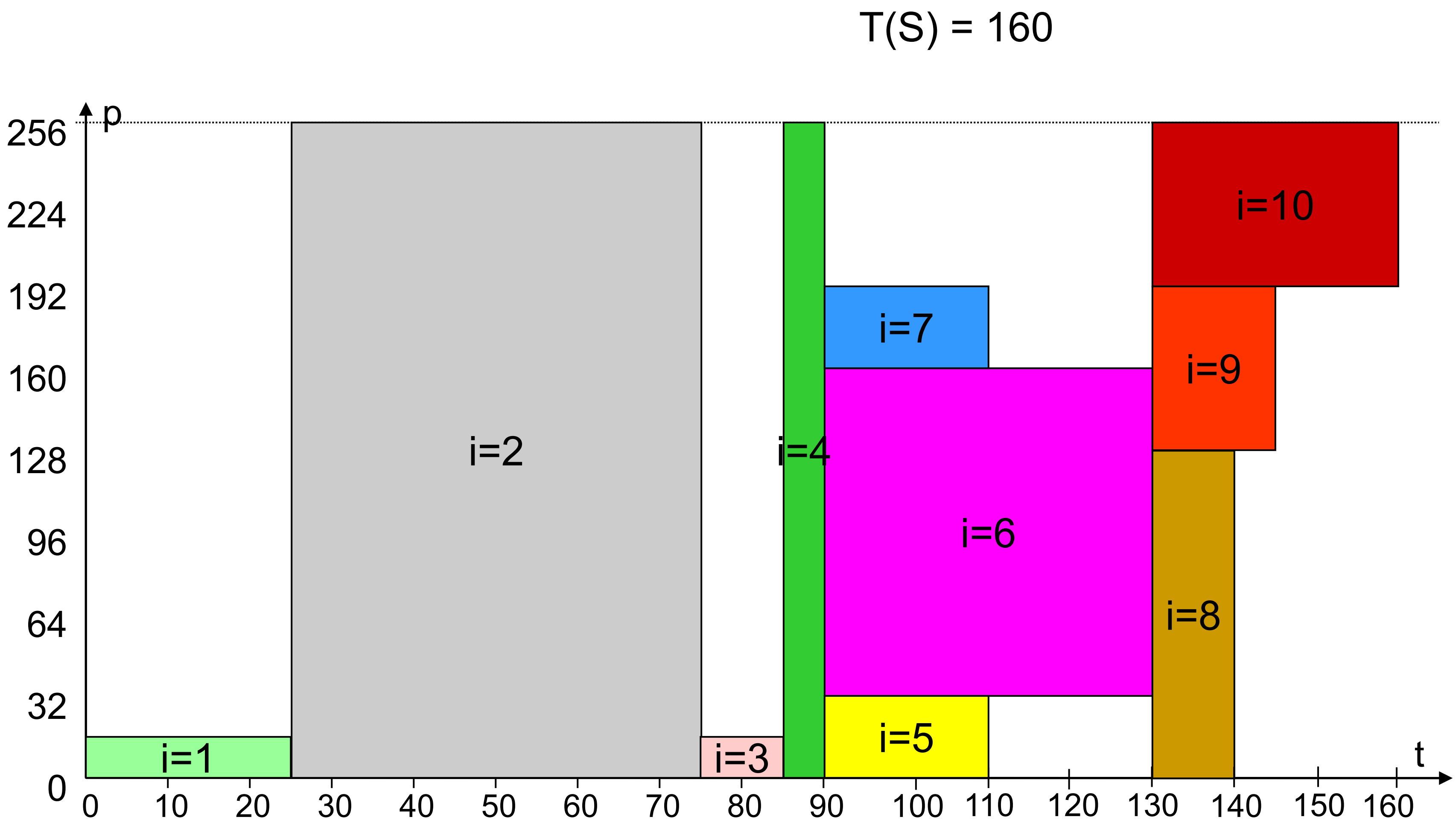
- Procedure:
  - sort requests according to arrival
  - schedule A_1 for $t=0$
  - schedule next requests $A_2, A_3, \ldots, A_k$ also for $t=0$, as long as

$$\sum_{i=1}^{k} p(i) \leq p$$

  - if not, start a new scheduling level beginning at

$$t(k+1) := \max_{i=1}^{k} \{T(i)\}$$

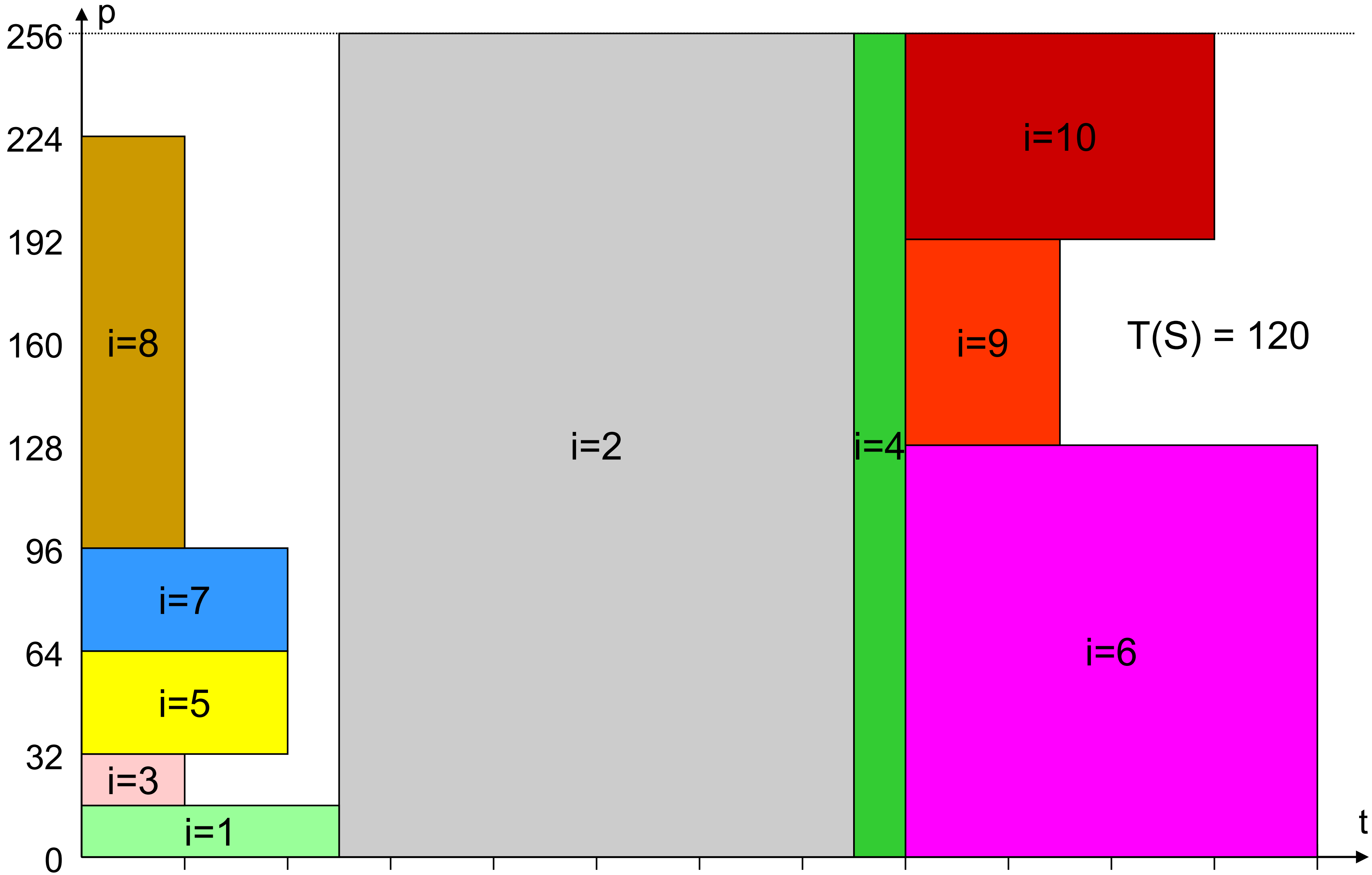
FCFS: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10



Backfilling

- Pure FCFS leads to high fragmentation.
- „Backfilling“ can improve this:
- To fill up a scheduling level not only the next request, but all requests in the queue are considered. That means smaller requests that still fit in will be preferred.

FCFS-Backfilling:
1, 3, 5, 7, 8, 2, 4, 6, 9, 10

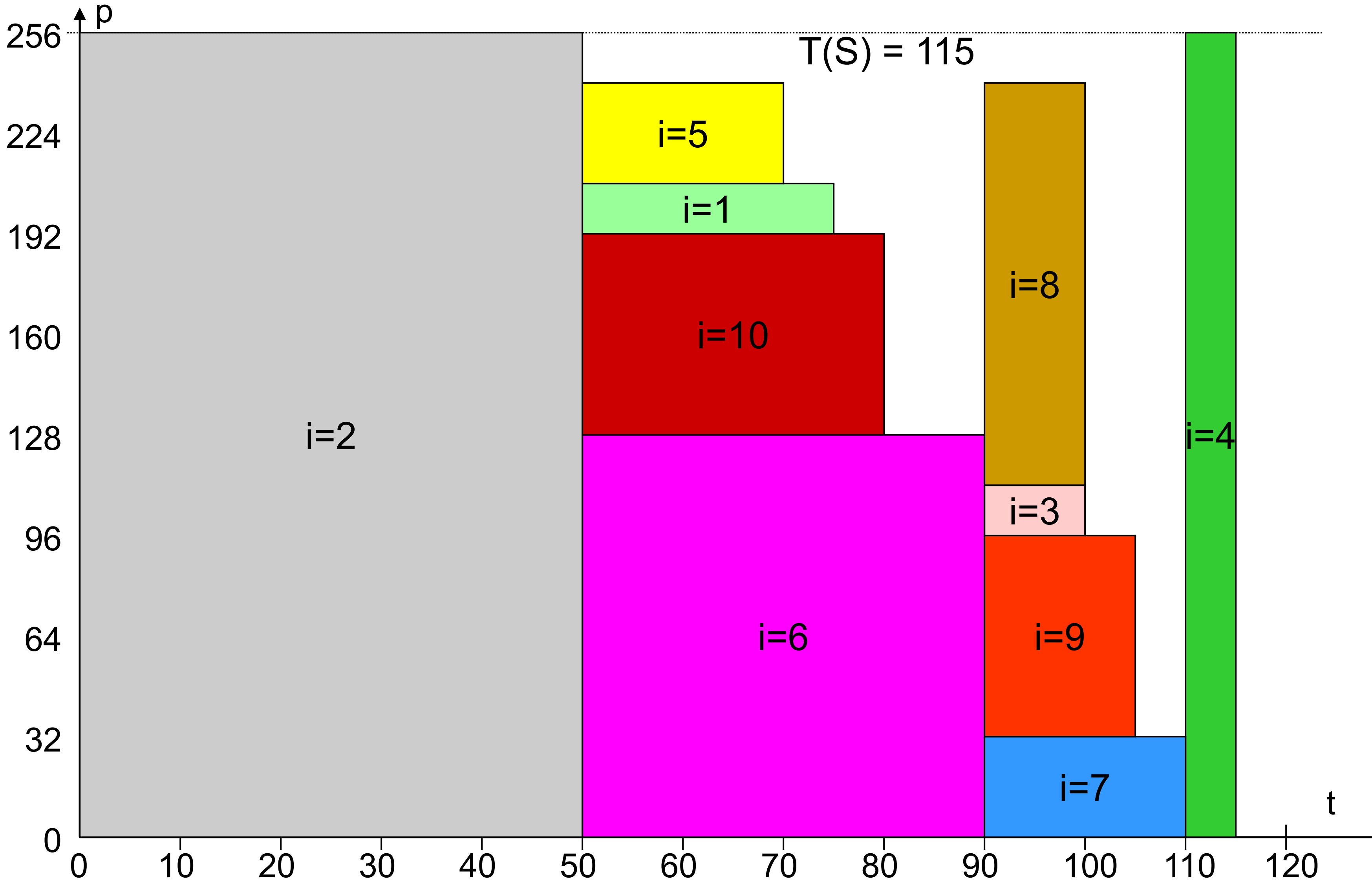


First Fit Decreasing Height (FFDH)

- Procedure:
  - Rectangles are left adjusted in the respective scheduling level.
  - Rectangles are sorted by decreasing execution time *T(i).*
  - Starting with the empty schedule and scheduling level t = 0 the rectangles are put onto each other until the next one does not fit in, since we reached the ceiling.
  - Than we start the next scheduling level.
- Theoretical result :
  - Let be T_{max} the longest execution time of a request.
  - Let be $T(S_{opt})$ the length of the optimal schedule.
  - Let be $T(S_{FFDH})$ the length of the schedule found by FFDH.
  - Than the following upper bound holds:

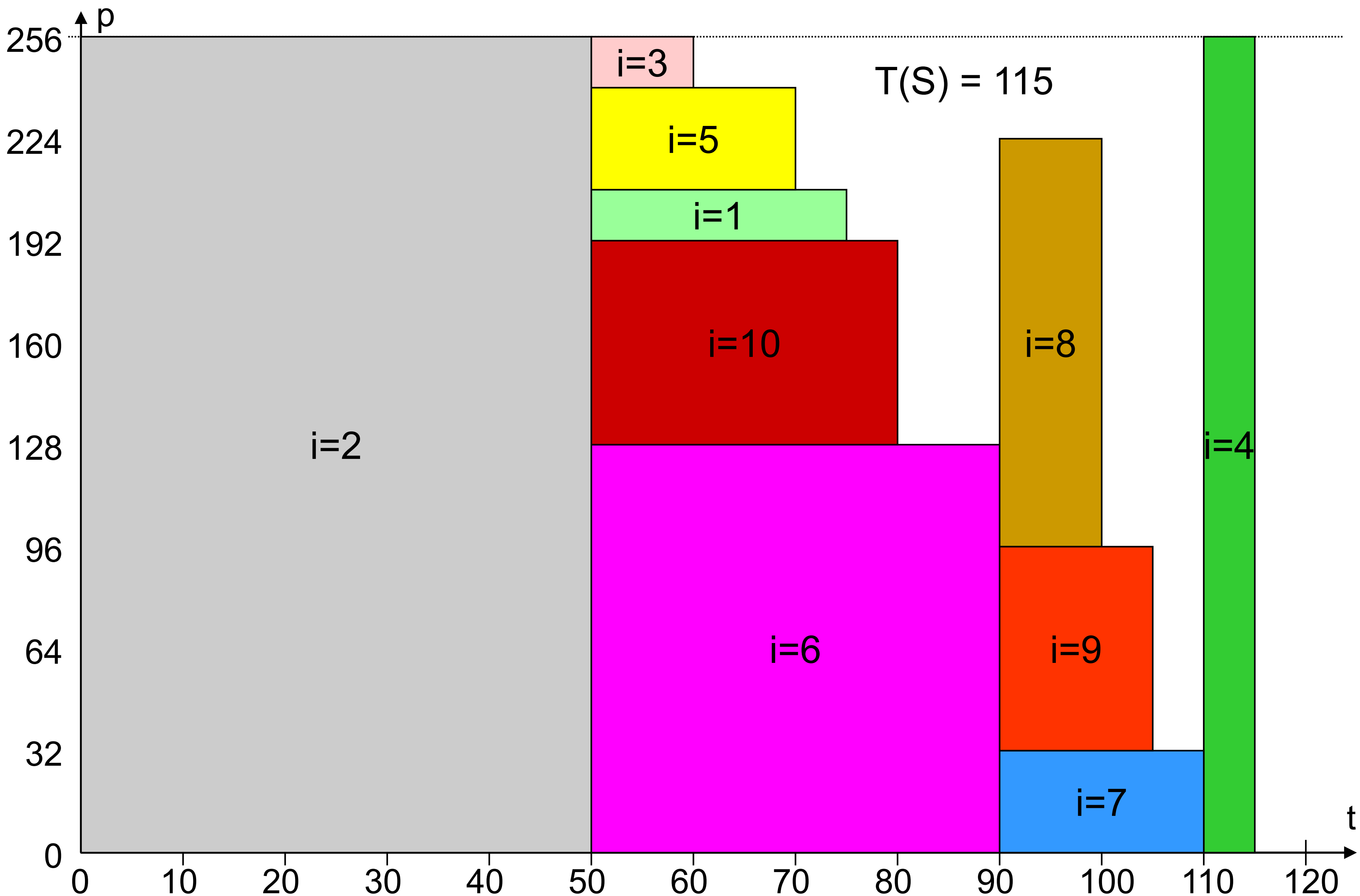
$$T(S_{FFDH}) \leq 1{,}7\ T(S_{opt}) + T_{max}$$

FFDH: 2, 6, 10, 1, 5, 7, 9, 3, 8, 4



FFDH-Backfilling:
2, 6, 10, 1, 5, 3, 7, 9, 8, 4

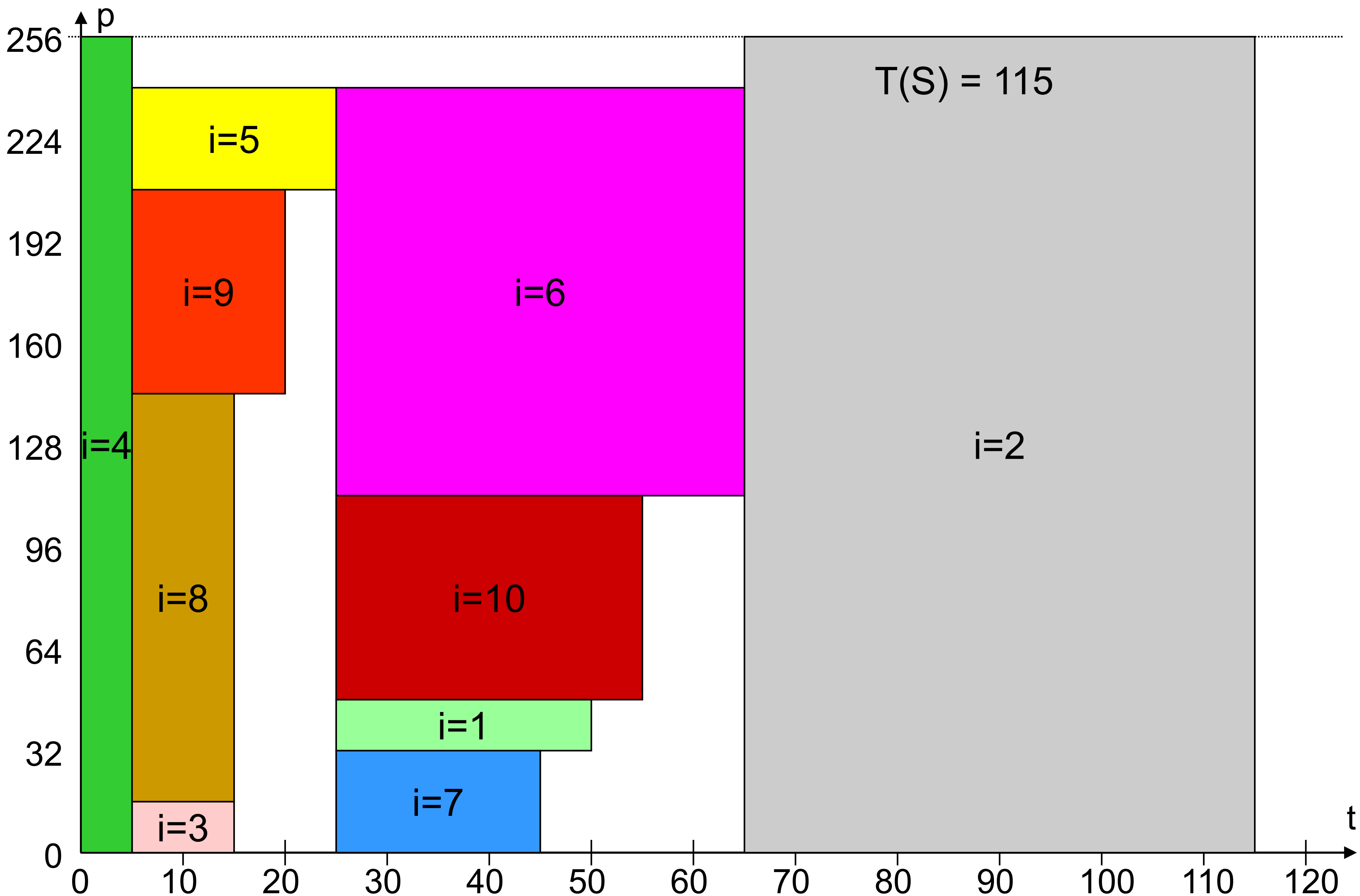
Freie Universität Berlin



First Fit Increasing Height (FFIH)

- Procedure:
  - Like FFDH, but with opposite sorting direction: shortest jobs next.
- Similar fragmentation and schedule length as FFDH
- Shorter mean waiting time (corresponds to Shortest Job Next)

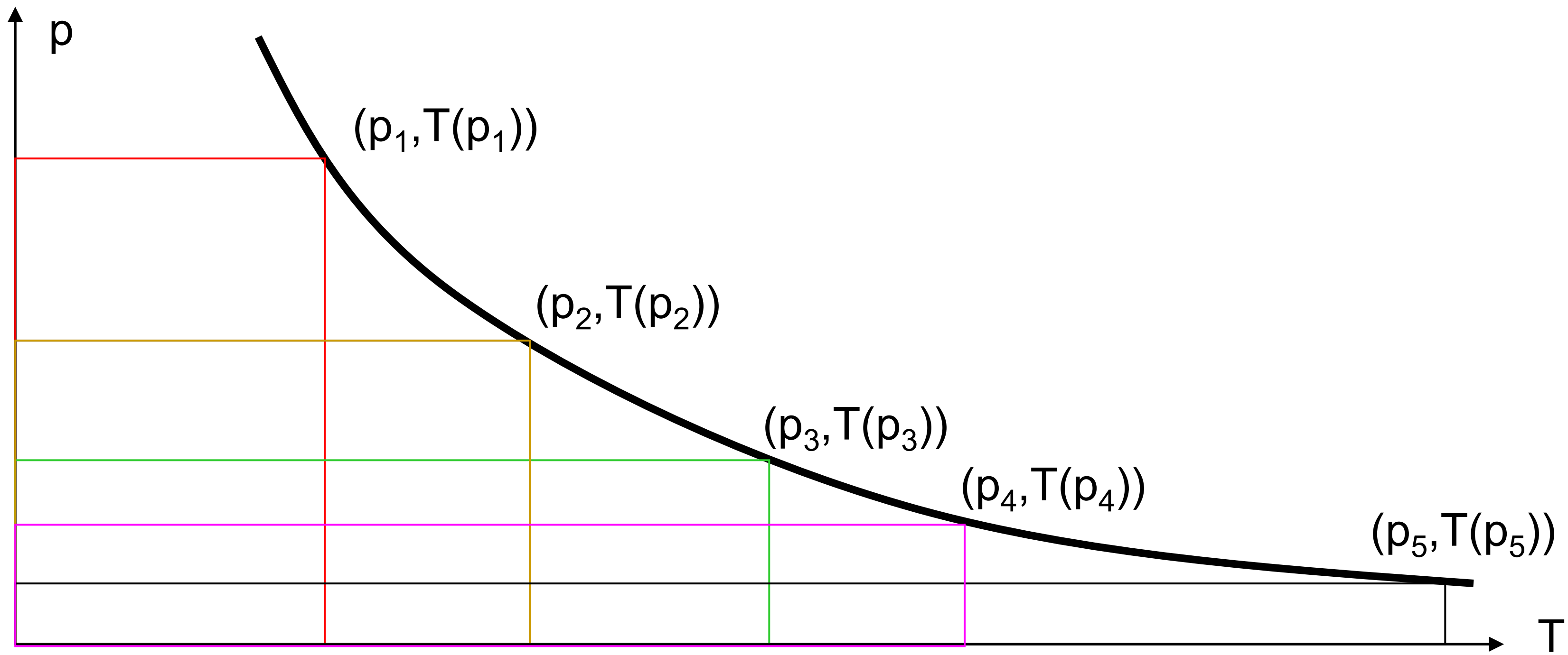
FFIH



p
256
224
192
160
128
96
64
32
0
T(S) = 115
i=5
i=9
i=6
i=4
i=2
i=8
i=10
i=1
i=7
i=3
t
0
10
20
30
40
50
60
70
80
90
100
110
120

Malleable (moldable) Rectangles

- Another degree of freedom for a scheduler arises when we take into account that in most cases a program can be started even without having p_{opt} processors available.
- (The rectangles can be considered malleable)



6.3 Semi-dynamic Allocation

- Given:
  - Dynamic set of programs, fed by an (usually stochastic) arrival process.
  - $p_f(t)$ number of free processors at time t
  - $W(t)$ set of programs that already arrived at time t but are still waiting for allocation

    We assume that $W(t)$ is ordered according to the order of arrival (FIFO queue).

Strategies

- FIFO bzw. FCFS
  - Let i be index of the first program in the queue. If $p(i) \leq p_f(t)$, $p(i)$ processors are allocated to the program.
  - Drawback: Larger numbers of processors may be unused only because the request at the front of the queue is currently not satifiable.
- First-Fit
  - The queue is scanned beginning at the front until a request j is found that can be satisfied ($p(j) \leq p_f(t)$).
- Best-Fit
  - The queue is completely scanned until a request j is found for which the following minimum condition is true:

$$\min_{j \in W(t) \wedge p(j) \leq p_f(t))} \{ p_f(t) - p(j) \}$$

Strategies

- Best-Fit-Set
  - Goal: Find a subset of requests, the sum of which matches the number of free processors $p_f(t)$ as close as possible, i.e. a subset $M \subseteq W(t)$, such that

$$p_f(t) - \Sigma\, p(j) \quad \rightarrow \quad \min$$

    where

$$\Sigma\, p(j) \leq p_f(t)$$

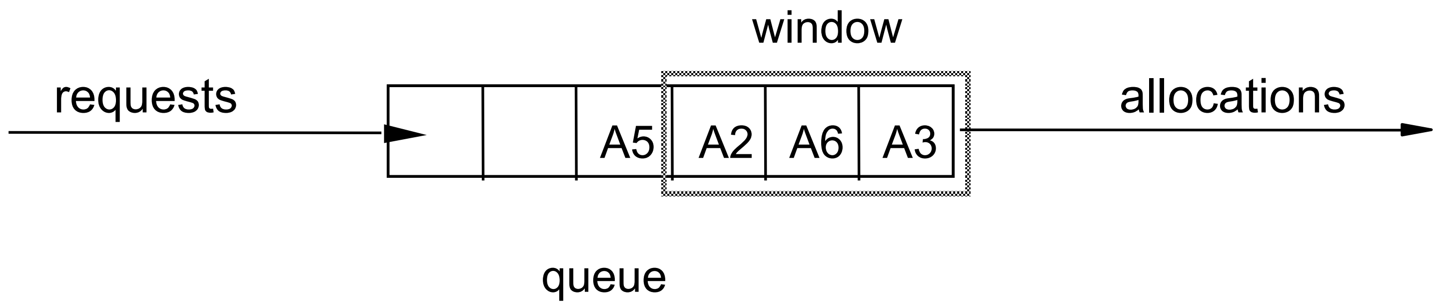
    Remark: The problem is apparently again a "Bin-packing-Problem" and therefore NP-complete.

- All strategies except FIFO hold the danger of starvation: A large request at the front of the queue could be ignored forever.

Selection strategies

- Window

  To reduce the overhead, we can limit the search for a candidate in the queue to a window of size *L*, i.e. only the first *L* positions of the queue are considered.

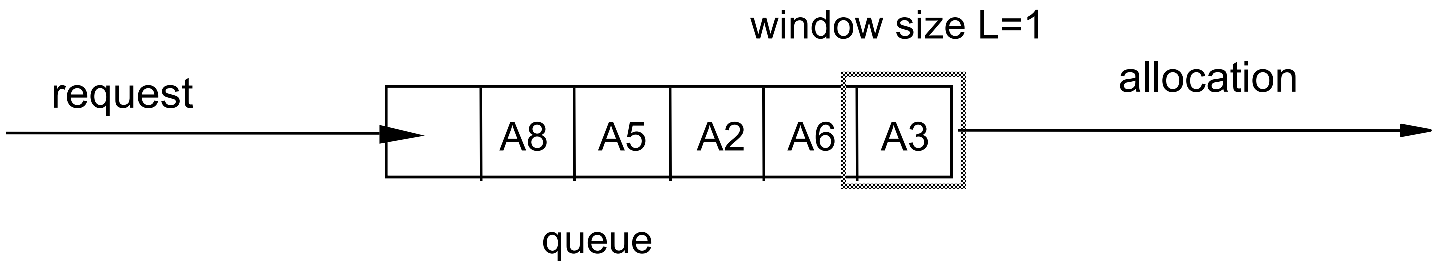


Solution of the starvation problem

- If we use a dynamic window size, we can solve the starvation problem of large requests (with First-Fit-Request or Best-Fit-Request).
- Let L_{max} *be the* maximum window size (initial value).
- At each successful allocation the window size is updated according to:

$$L := \begin{cases} L-1, & \text{if } L > 1 \text{ and the request at the head of the queue is skipped.} \\ L_{max}, & \text{otherwise} \end{cases}$$

- By doing so, the window size shrinks to 1 when the foremost request has been passed over L-1 times. In this case, this first request must be selected since it is the only one in the window.



- For L approaching 1 Best-Fit-Request and First-Fit-Request converge to FCFS.

6.4 Dynamic Partitioning

- While up to now the programs were given a fixed number of processors for the whole runtime, we are considering the case that processors are allocated to and withdrawn from a program dynamically (before runtime).
- Basic idea: allocate an additional processor to a program so that the highest speed-up gain is achieved.
- Given: p processors and M programs with their speed-up-functions $S(i,k)$, $i = 1,\dots,M$; $k=1,\dots,p$)
- Goal: Find a quantitative partitioning $p(i)$ such that

$$\sum_{i=1}^{M} S\left(i, p(i)\right) \rightarrow max \qquad \text{with} \qquad p = \sum_{i=1}^{M} p(i)$$

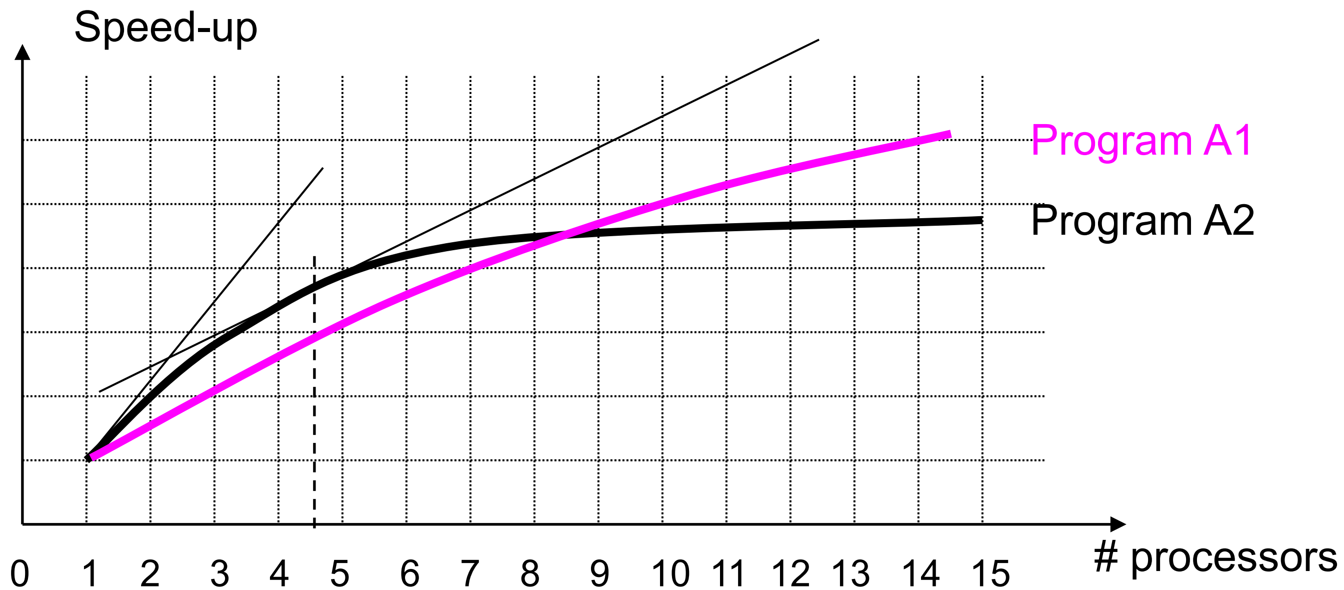
  - Maximization of the sum of speed-ups indirectly also minimizes the sum of execution times and maximizes the throughput.

Dynamic Partitioning

The processors are incrementally allocated to the programs.

The program with the highest speed-up-increase (first derivative) gets an additional processor.

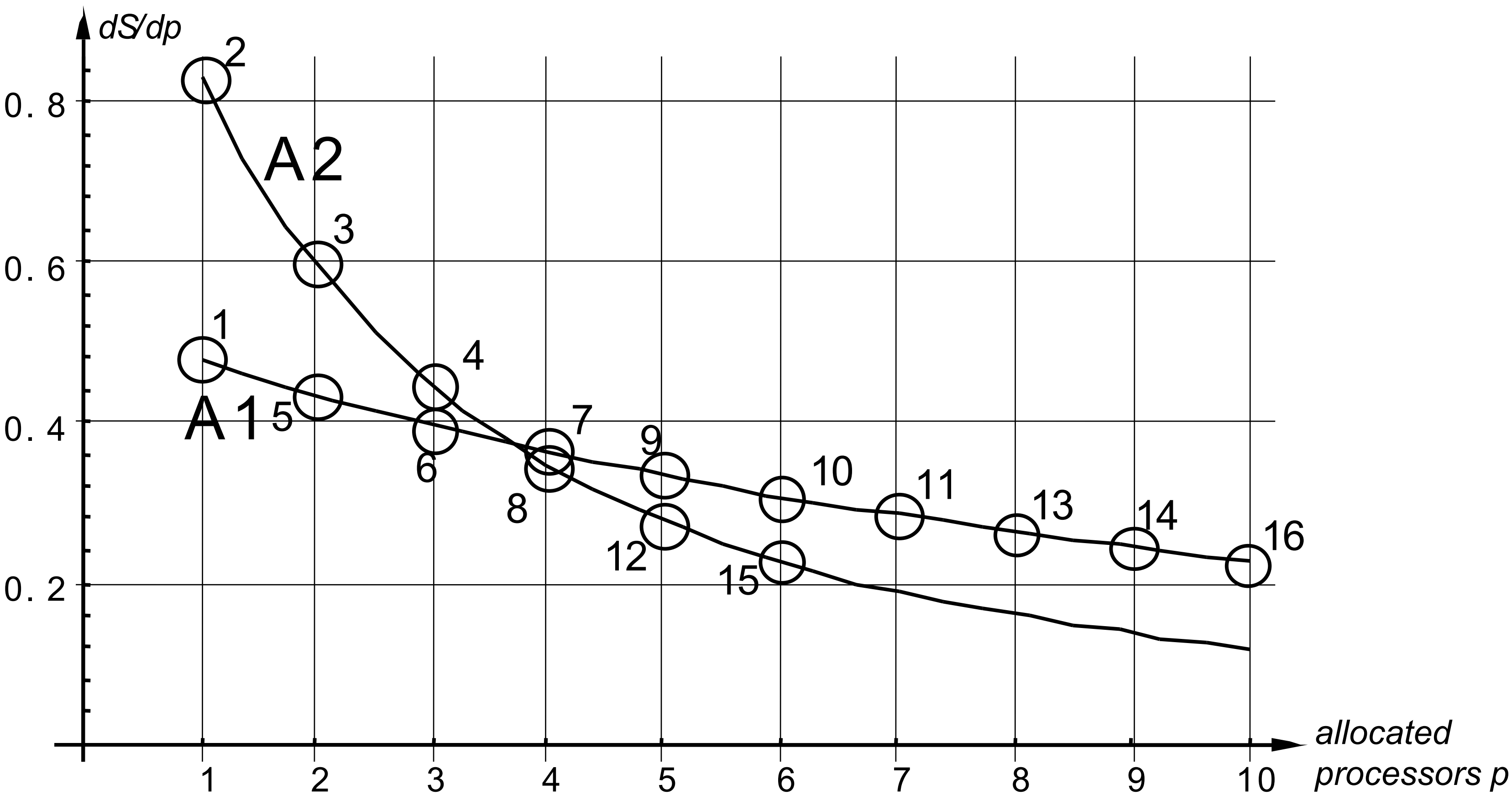
Which program gets how many processors?



Quantitative Dynamic Partitioning

`available ← p`	All processors available.
for `i ← 1 to m` **do**	All programs.
`    p(i) ← 1`	Minimal allocation (Initialization).
`    DS(i) ← S(i,p(i)+1)-S(i,p(i))`	Calculate differential Speed-up
end `for`	(derivative).
`DS_list ← sort_descending({i,DS(i)})`	Sort programs according to Speed-up derivative.
while `available > 0` **do**	All processors are being allocated.
`    x ← first(DS-list)`	Program with steepest speed-up growth p(i) is being selected
`    remove(DS(x),DS_list)`	and removed from list.
`    p(x) ← p(x)+1`	its no. of alloc. proc. is incremented
`    DS(x) ← S(x,p(x)+1)-S(x,p(x))`	the speed-up derivative for this
`    insert(DS(x), DS_list)`	new value is recomputed and sorted and reinserted into the list
`    available ← available - 1`	
end `while`	

Example



dS/dp
0. 8
0. 6
0. 4
0. 2
A2
A1
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
1
2
3
4
5
6
7
8
9
10
allocated
processors p

Further References

- E. Shmueli and D. G. Feitelson, *Backfilling with lookahead to optimize the performance of parallel job scheduling*''. In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.), Springer-Verlag, LNCS 2862. pp. 228-251, 2003
- Skovira, J. et. al.: *The EASY-LoadLeveler API Project*, LNCS 1162, pp.41-47, 1996
- Keleher,J. et al.: *Attacking the bottlenecks of backfilling schedulers*. Cluster Computing 3(4), pp.245-254, 2000