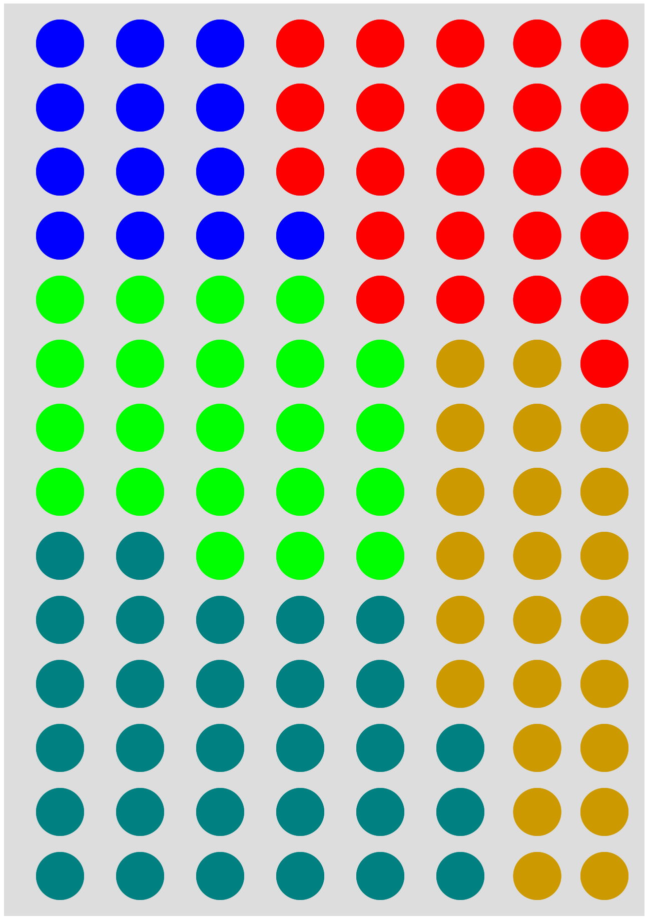
Adaptive Scheduling with Parallelism Feedback

Kunal Agrawal

Collaborators: Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson

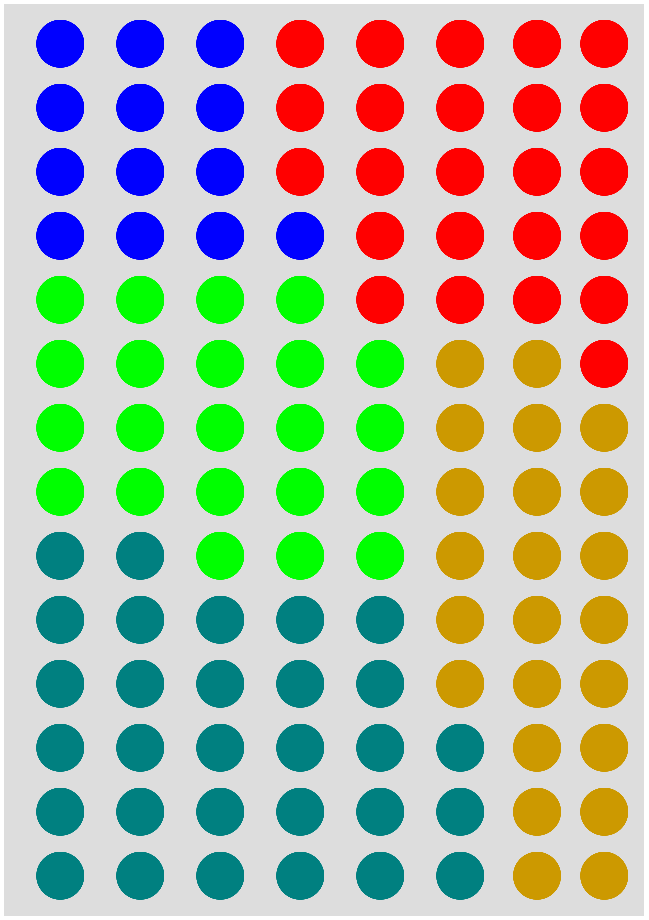
Adaptive Multiprocessor Scheduling

- Many jobs space-share a large multiprocessor, entering and leaving the system dynamically. The number of processors available to the job may, therefore, change during execution and the jobs must *adapt* to these changes.

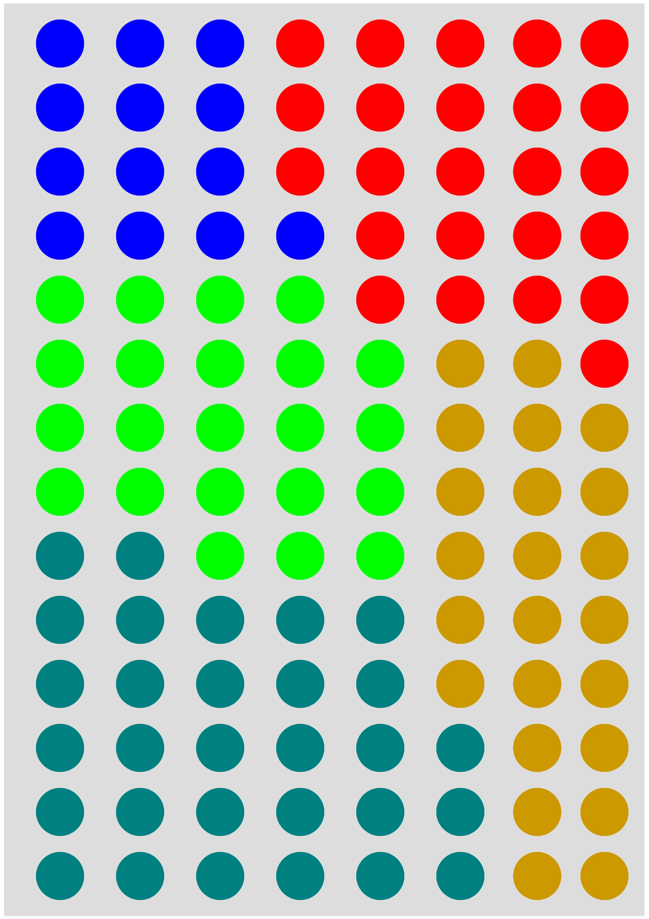


Adaptive Multiprocessor Scheduling

- Many jobs space-share a large multiprocessor, entering and leaving the system dynamically. The number of processors available to the job may, therefore, change during execution and the jobs must *adapt* to these changes.
- The parallelism of a job may change during execution. The jobs do not know their future parallelism.

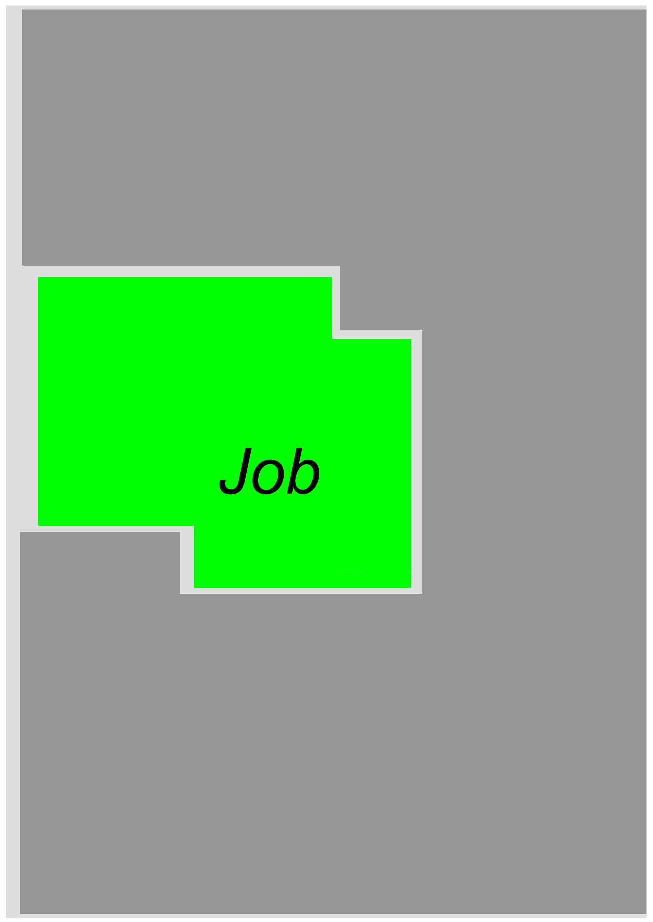


A Job and Its Environment



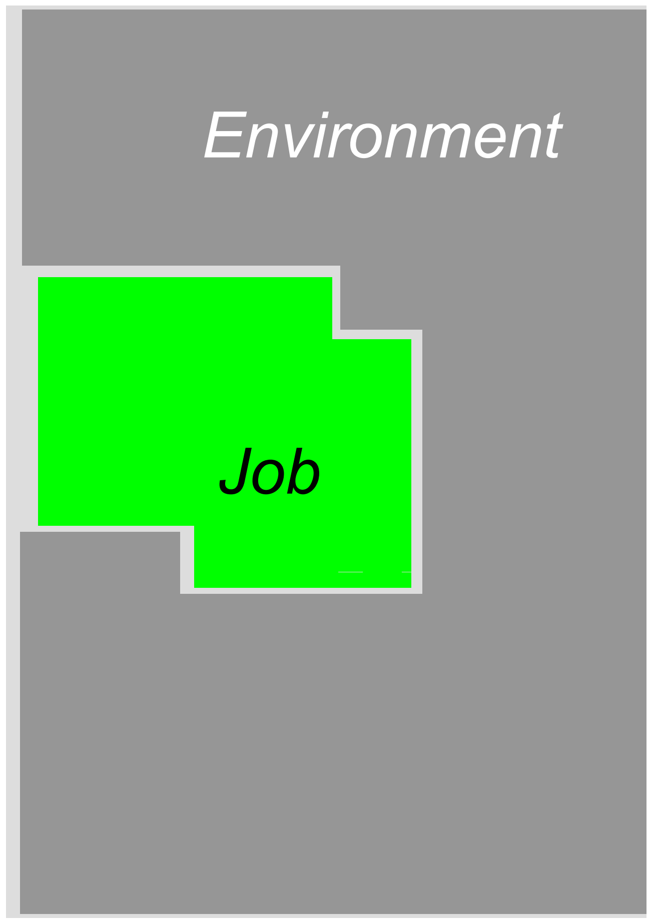
A Job and Its Environment

- Focus on a single job.



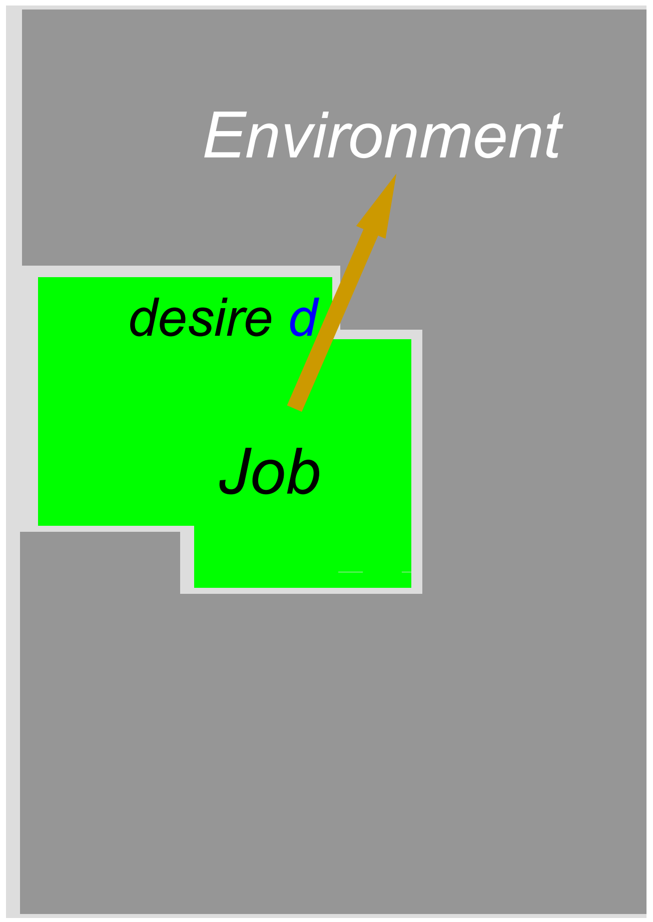
A Job and Its Environment

- Focus on a single job.
- The *environment* is an abstract entity that captures the behavior of the other jobs in the system and the allocation policy of the job scheduler.



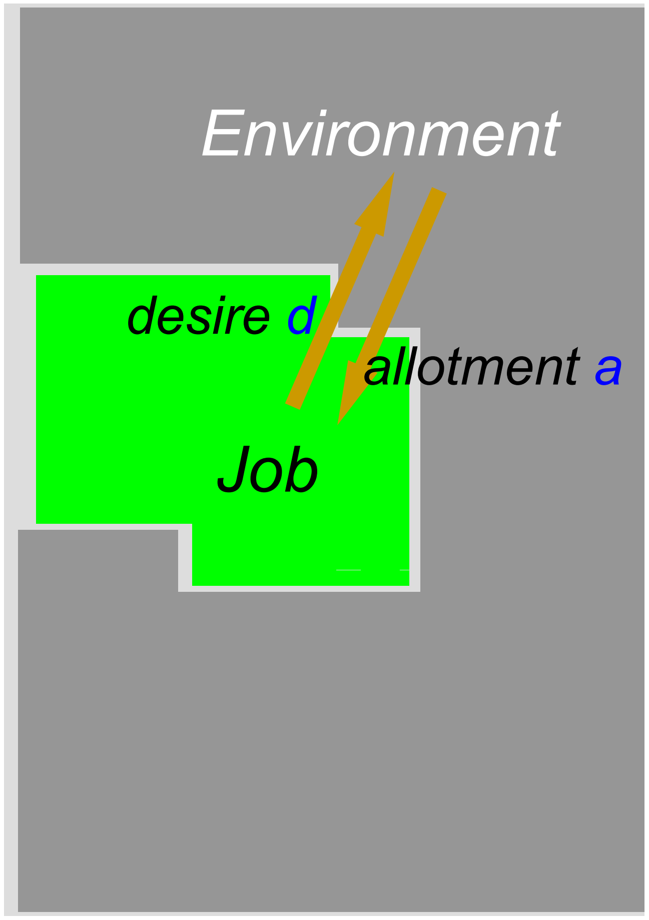
A Job and Its Environment

- Focus on a single job.
- The *environment* is an abstract entity that captures the behavior of the other jobs in the system and the allocation policy of the job scheduler.
- Between scheduling *quanta*, the job provides *parallelism feedback* to the environment by requesting the *desired* number of processors for the next quantum.



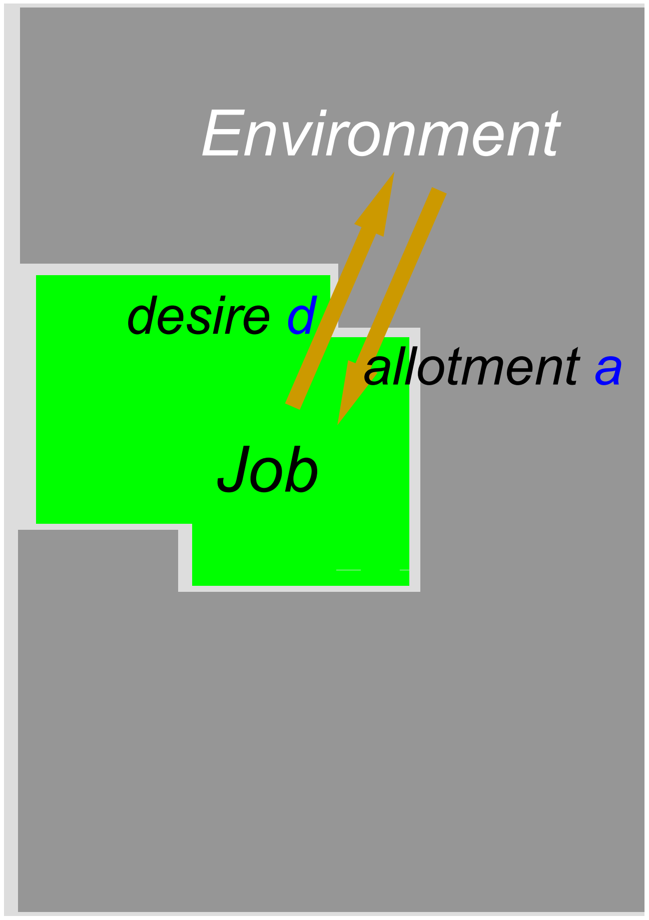
A Job and Its Environment

- Focus on a single job.
- The *environment* is an abstract entity that captures the behavior of the other jobs in the system and the allocation policy of the job scheduler.
- Between scheduling *quanta*, the job provides *parallelism feedback* to the environment by requesting the *desired* number of processors for the next quantum.
- The environment *allots* processors to the job. The *allotment* for a quantum is always less than or equal to the desire.



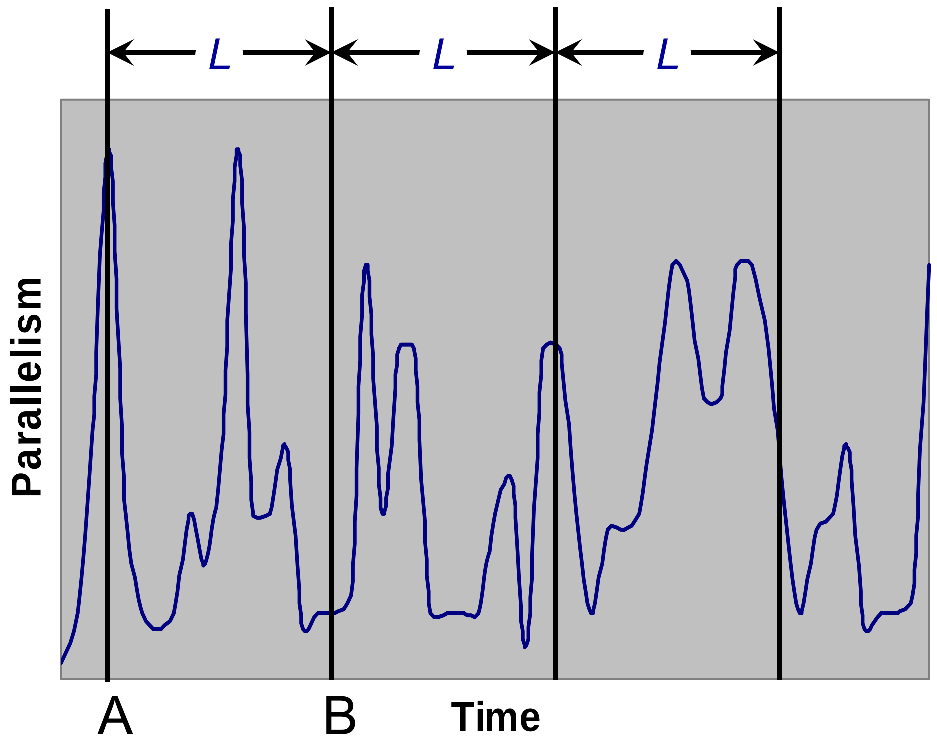
A Job and Its Environment

- Focus on a single job.
- The *environment* is an abstract entity that captures the behavior of the other jobs in the system and the allocation policy of the job scheduler.
- Between scheduling *quanta*, the job provides *parallelism feedback* to the environment by requesting the *desired* number of processors for the next quantum.
- The environment *allots* processors to the job. The *allotment* for a quantum is always less than or equal to the desire.
- The allotment remains unchanged during a quantum, and the *quantum length* L is long enough to amortize the overheads due to communication and reallocation.



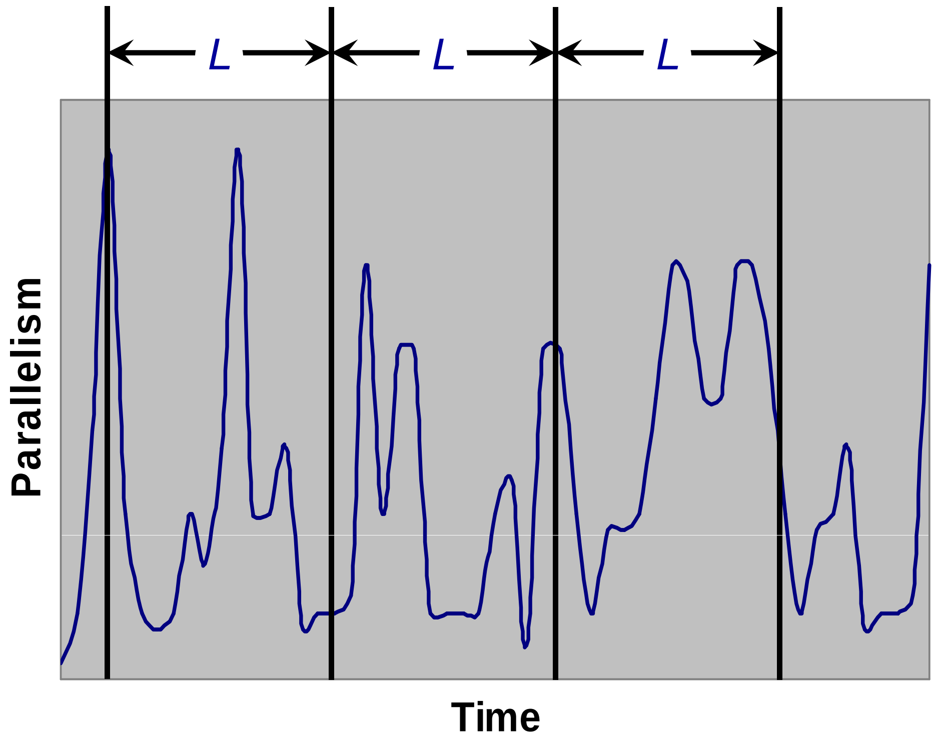
How To Provide Feedback

- The job does not know its future parallelism.
- The job's parallelism may change during the quantum. Therefore, using the *instantaneous parallelism* as the desire might be ineffective.
- For example, if the job uses instantaneous parallelism to estimate desire, then at time A, the job will ask for more processors than it needs. On the other hand, at time B, the job will ask for fewer processors than it needs.



Completion Time and Waste

- If the job requests (and receives) too few processors, then it runs slowly.
- If the job requests (and receives) too many processors, then it wastes processor cycles that other jobs could have used more effectively.



- An effective parallelism feedback algorithm tries to reduce both *completion time* and *waste* of the job.

Contributions

A provably good algorithm for providing parallelism feedback.

- If this algorithm is combined with *greedy scheduling* [G69, B72], the job completes quickly and waste few processor cycles, even if the environment is *adversarial*. We call the combination *A-GREEDY*.
- Similarly, if it is combined with work-stealing [BL94], we call the combination *A-STEAL*.
- To analyze A-GREEDY and A-STEAL, we introduce a new analytical technique called *trim analysis*.
- Preliminary experiments using simulations indicate that advantages of parallelism feedback become apparent on moderately to heavily loaded large machines.

Theoretical Results

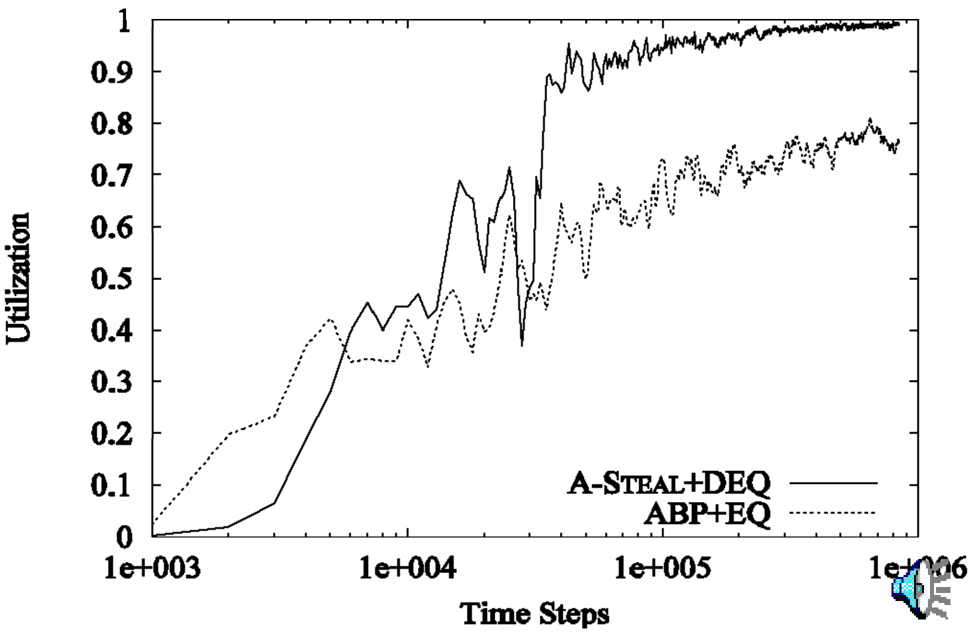
THEOREM: Consider a job with *work* T_1 and *span* (*critical-path length*) T_∞ running on P-processor machine and scheduling quanta of length L. A-GREEDY (and A-STEAL) guarantees that the job

- wastes $O(T_1)$ processor cycles and
- attains linear speedup on all but $O(T_\infty + L \lg P)$ time steps.

The proofs use *trim analysis*, which ignores a few of data points and obtains tight bounds on the remainder.

Experimental Results

- We simulated a large multiprogrammed multiprocessor and used synthetic jobs to assess the performance of A-STEAL.
- A-STEAL provides nearly perfect linear speedup and wastes less than 20% of the allotted processor cycles.
- We compared the utilization provided by A-STEAL and *ABP*, a work-stealing scheduler that does not employ parallelism feedback. When many synthetic jobs share a large server, and jobs arrive and leave dynamically, A-STEAL consistently provided higher utilization than ABP for a variety of job mixes.



Outline

- Introduction
- The Feedback Algorithm
- Greedy Scheduling
- Adversarial Environment and Trim Analysis
- Analysis Idea
- Conclusions

The Feedback Algorithm

It is a multiplicative-increase, multiplicative-decrease algorithm for providing parallelism feedback to the environment.

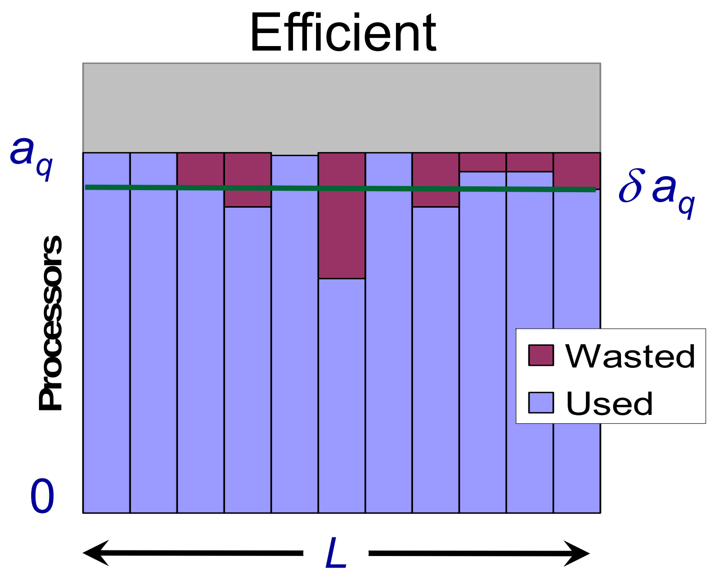
At the beginning of quantum q, it uses the job's recent history to provide parallelism feedback to the environment:

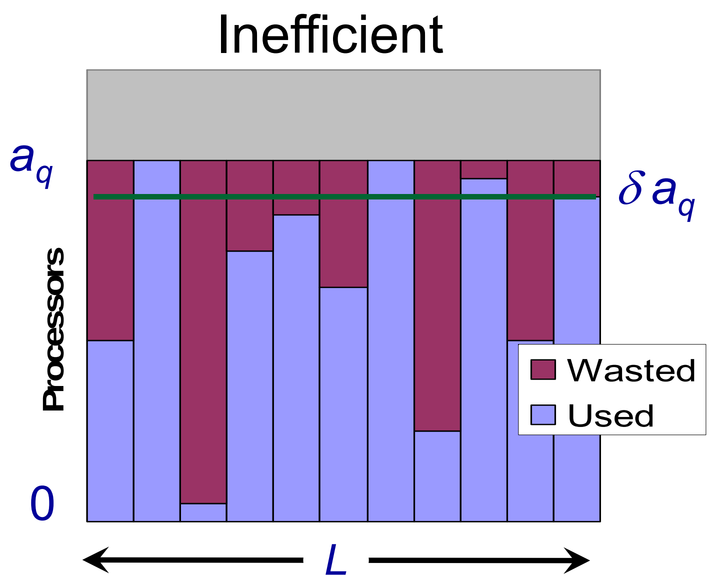
- desire d_{q-1} in quantum $q-1$;
- allotment a_{q-1} in quantum $q-1$;
- *usage* u_{q-1} — the number of processor cycles used effectively (not wasted) in quantum $q-1$.

Utilization Parameter

The *utilization parameter* $\delta < 1$ is an input to the algorithm, which provides a threshold for determining whether a particular quantum is efficient or inefficient, e.g., $\delta = 90\%$.

- Quantum q is *efficient* if $u_q \geq L\,\delta a_q$.
- Quantum q is *inefficient* if $u_q < L\,\delta a_q$.





Details of the Algorithm

```
FEEDBACK$(q, \delta)$
1. if $q = 1$
2.    then $d_q \leftarrow 1$
3. elseif $u_{q-1} < L\delta a_{q-1}$
4.    then $d_q \leftarrow d_{q-1}/2$
5. elseif $a_{q-1} = d_{q-1}$
6.    then $d_q \leftarrow 2d_{q-1}$
7. else $d_q \leftarrow d_{q-1}$
8. Report $d_q$ as the desire for quantum $q$.
```

Details of the Algorithm

```
FEEDBACK($q, \delta$)
1. if $q = 1$
2.    then $d_q \leftarrow 1$
3. elseif $u_{q-1} < L\delta a_{q-1}$
4.    then $d_q \leftarrow d_{q-1}/2$
5. elseif $a_{q-1} = d_{q-1}$
6.    then $d_q \leftarrow 2d_{q-1}$
7. else $d_q \leftarrow d_{q-1}$
```

Initial desire is $d_1 = 1$.

Details of the Algorithm

```
FEEDBACK($q, \delta$)
1. if $q = 1$
2.    then $d_q \leftarrow 1$
3. elseif $u_{q-1} < L\delta a_{q-1}$
4.    then $d_q \leftarrow d_{q-1}/2$
5. elseif $a_{q-1} = d_{q-1}$
6.    then $d_q \leftarrow 2d_{q-1}$
7. else $d_q \leftarrow d_{q-1}$
```

Initial desire is $d_1 = 1$.

Too much waste.

Job overestimated the desire.

Decrease the desire.

Details of the Algorithm

```
FEEDBACK($q, \delta$)
1. if $q = 1$
2.    then $d_q \leftarrow 1$
3. elseif $u_{q-1} < L \delta a_{q-1}$
4.    then $d_q \leftarrow d_{q-1}/2$
5. elseif $a_{q-1} = d_{q-1}$
6.    then $d_q \leftarrow 2d_{q-1}$
7. else $d_q \leftarrow d_{q-1}$
```

Initial desire is $d_1 = 1$.

Too much waste.

Decrease the desire.

Small waste + large allotment.

Job effectively utilized all the processors requested and speculates that it may be able to use more.

Increase the desire.

Details of the Algorithm

```
FEEDBACK($q, \delta$)
1. if $q = 1$
2.    then $d_q \leftarrow 1$
3. elseif $u_{q-1} < L \delta a_{q-1}$
4.    then $d_q \leftarrow d_{q-1}/2$
5. elseif $a_{q-1} = d_{q-1}$
6.    then $d_q \leftarrow 2 d_{q-1}$
7. else $d_q \leftarrow d_{q-1}$
```

Initial desire is $d_1 = 1$.

Too much waste.
Decrease the desire.

Small waste + large allotment.
Increase the desire.

Small waste + small allotment.
The job was allotted few processors, but used all of them effectively.
Maintain the desire.

Details of the Algorithm

```
FEEDBACK($q, \delta$)
1. if $q = 1$
2.     then $d_q \leftarrow 1$
3. elseif $u_{q-1} < L\delta a_{q-1}$
4.     then $d_q \leftarrow d_{q-1}/2$
5. elseif $a_{q-1} = d_{q-1}$
6.     then $d_q \leftarrow 2d_{q-1}$
7. else $d_q \leftarrow d_{q-1}$
```

Initial desire is $d_1 = 1$.

Too much waste.
Decrease the desire.

Small waste + large allotment.
Increase the desire.

Small waste + small allotment.
Maintain the desire.

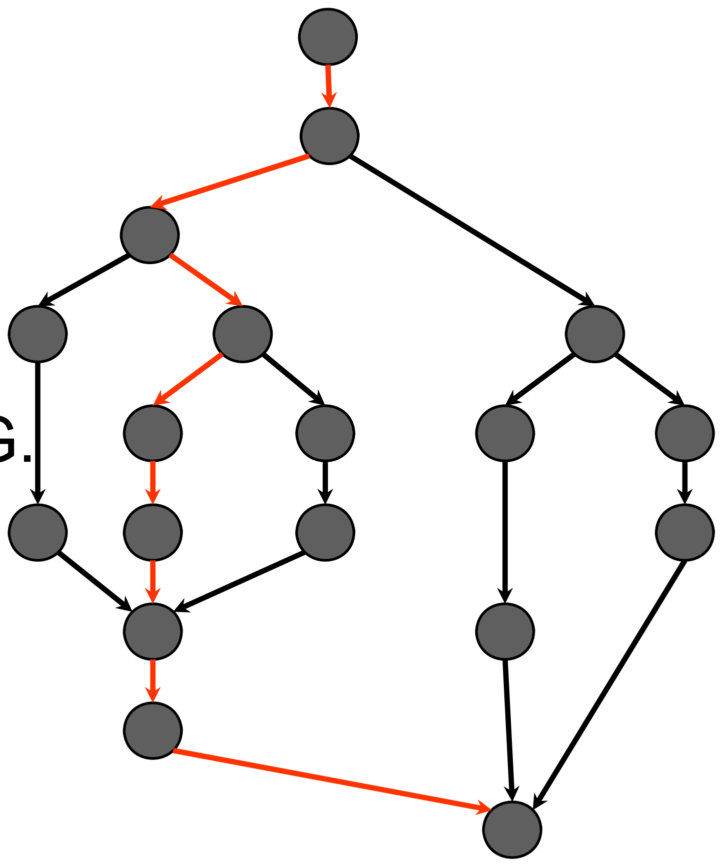
It provides simple and effective parallelism feedback.

Outline

- Introduction
- The Feedback Algorithm
- Greedy Scheduling
- Adversarial Environment and Trim Analysis
- Analysis Idea
- Conclusions

Job Model

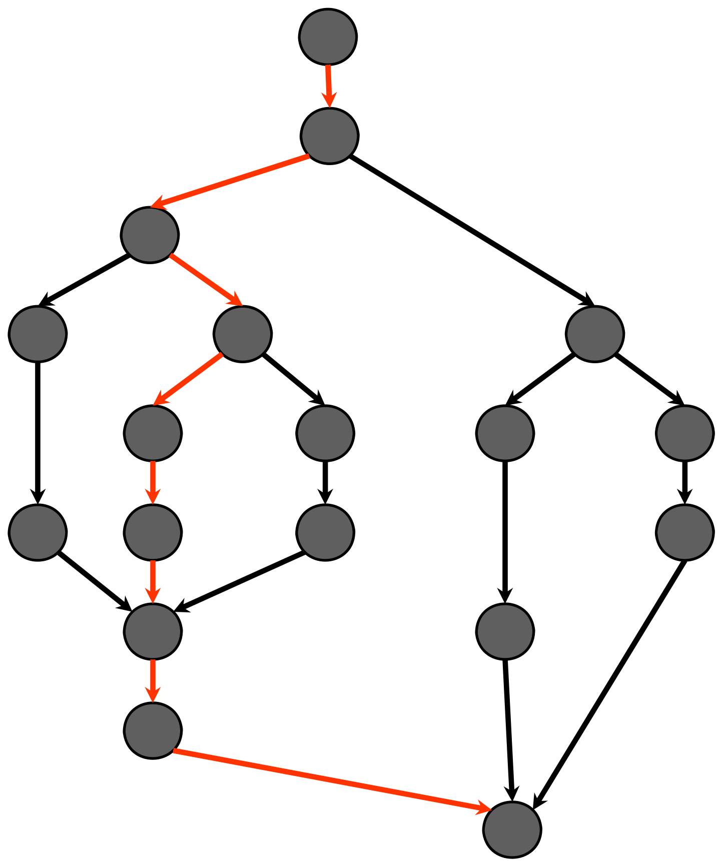
- DAG
  - Node: Unit time task.
  - Edge: Dependence between tasks.
  - A task becomes *ready* when all its predecessors have been executed.
- T_1: *Work*
  - Number of nodes (tasks) in the DAG.
  - Time to execute on one processor.
- T_∞: *Span (Critical Path Length)*
  - Length of the longest path in the DAG.
  - Time to execute on infinite number of processors.



Greedy Scheduling

During quantum q, A-GREEDY schedules the tasks on a_q allotted processors greedily.

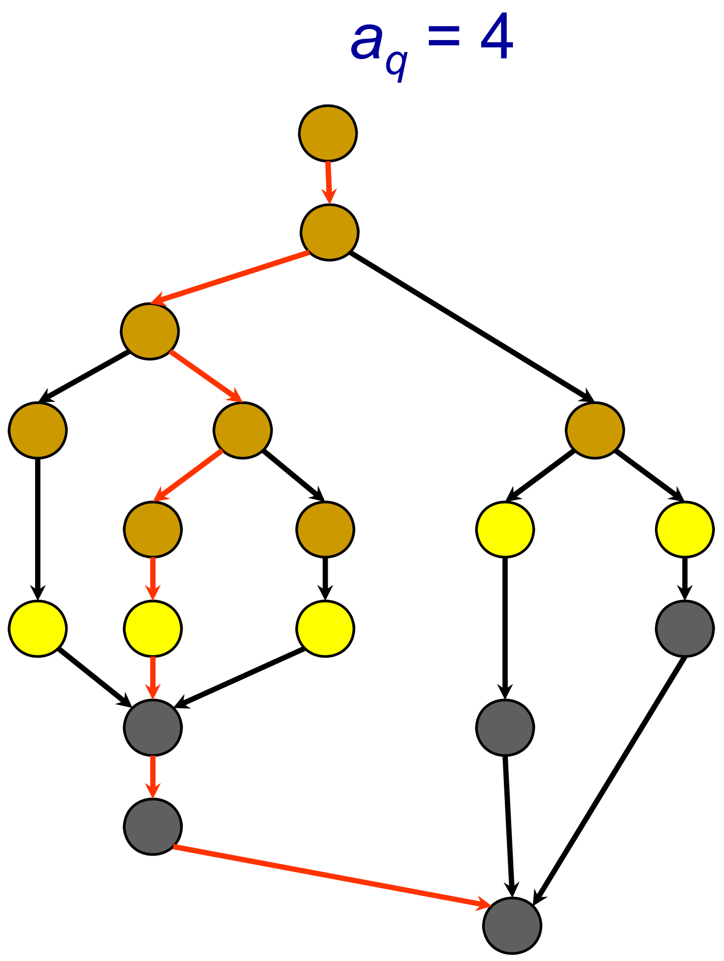
$a_q = 4$



Greedy Scheduling

During quantum q, A-GREEDY schedules the tasks on a_q allotted processors greedily.

- *Case 1: (complete step)* If at least a_q tasks are ready, execute any a_q of them.

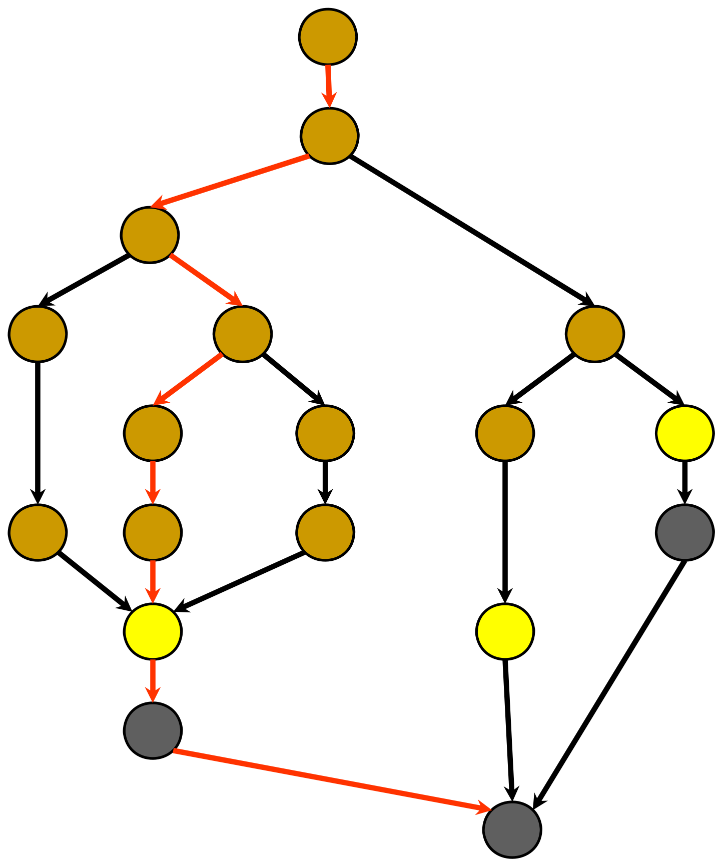


Greedy Scheduling

During quantum q, A-GREEDY schedules the tasks on a_q allotted processors greedily.

- *Case 1: (complete step)* If at least a_q tasks are ready, execute any a_q of them.

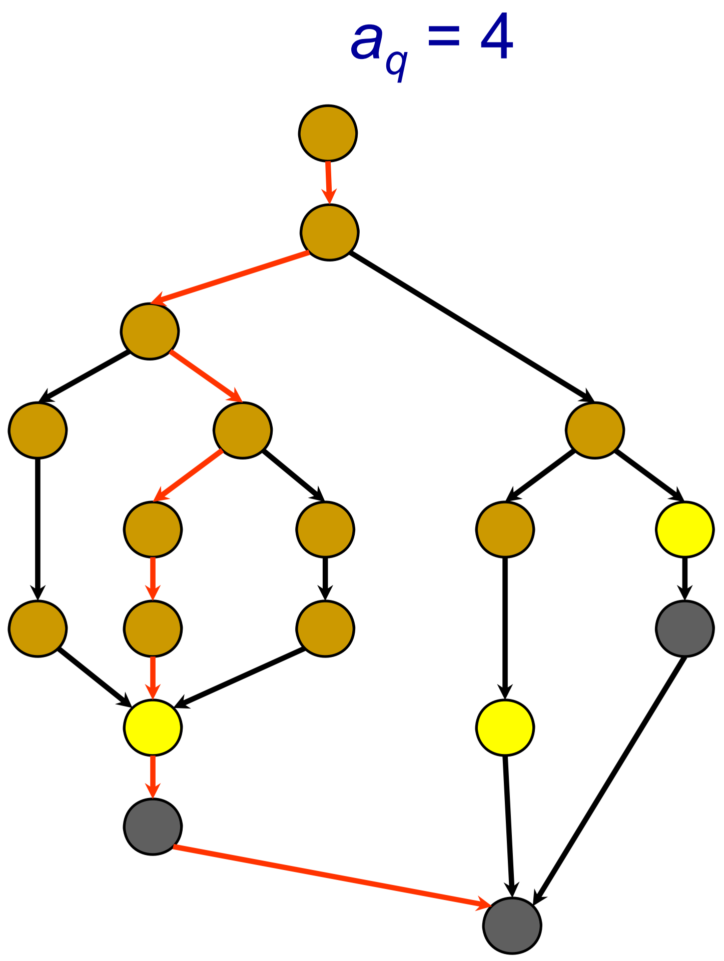
$a_q = 4$



Greedy Scheduling

During quantum q, A-GREEDY schedules the tasks on a_q allotted processors greedily.

- *Case 1:* (*complete step*) If at least a_q tasks are ready, execute any a_q of them.
- *Case 2*: (*incomplete step*) If less than a_q tasks are ready, execute all of them.

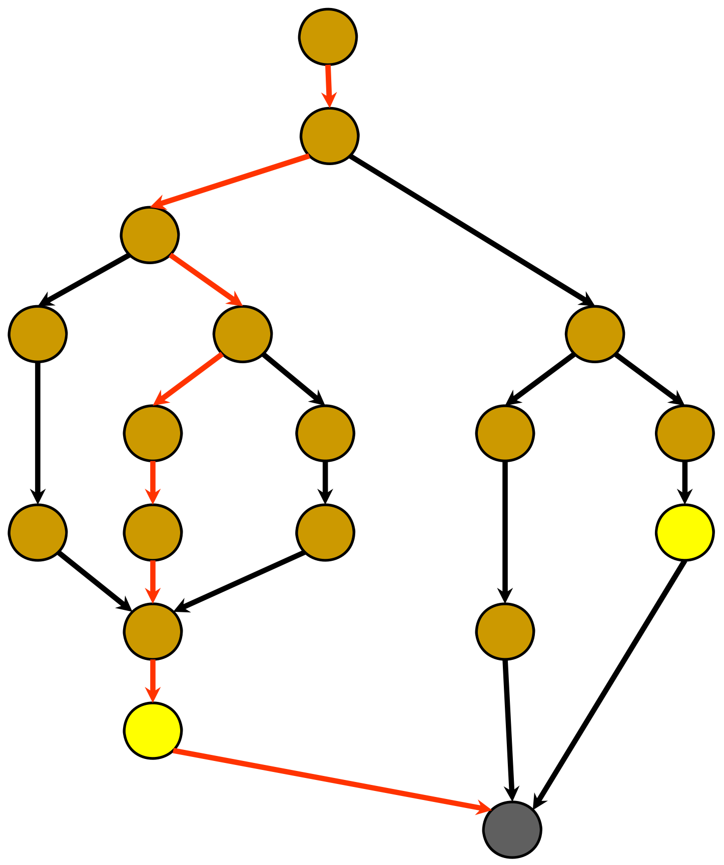


Greedy Scheduling

During quantum *q*, A-GREEDY schedules the tasks on a_q allotted processors greedily.

- *Case 1:* (*complete step*) If at least a_q tasks are ready, execute any a_q of them.
- *Case 2*: (*incomplete step*) If less than a_q tasks are ready, execute all of them.

$a_q = 4$



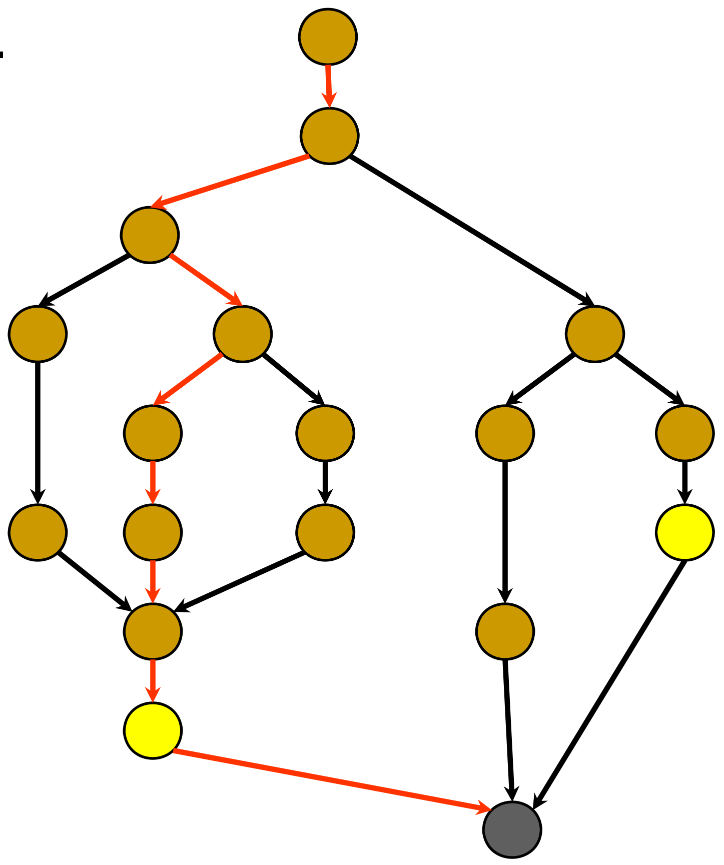
Nonadaptive Greedy Scheduling

The allotment is fixed at P processors.

Lower Bound: Completion time
$T \geq \max\{T_1/P, T_\infty\}$

Greedy scheduling guarantees
$T \leq T_1/P + T_\infty$. [G69, B72]

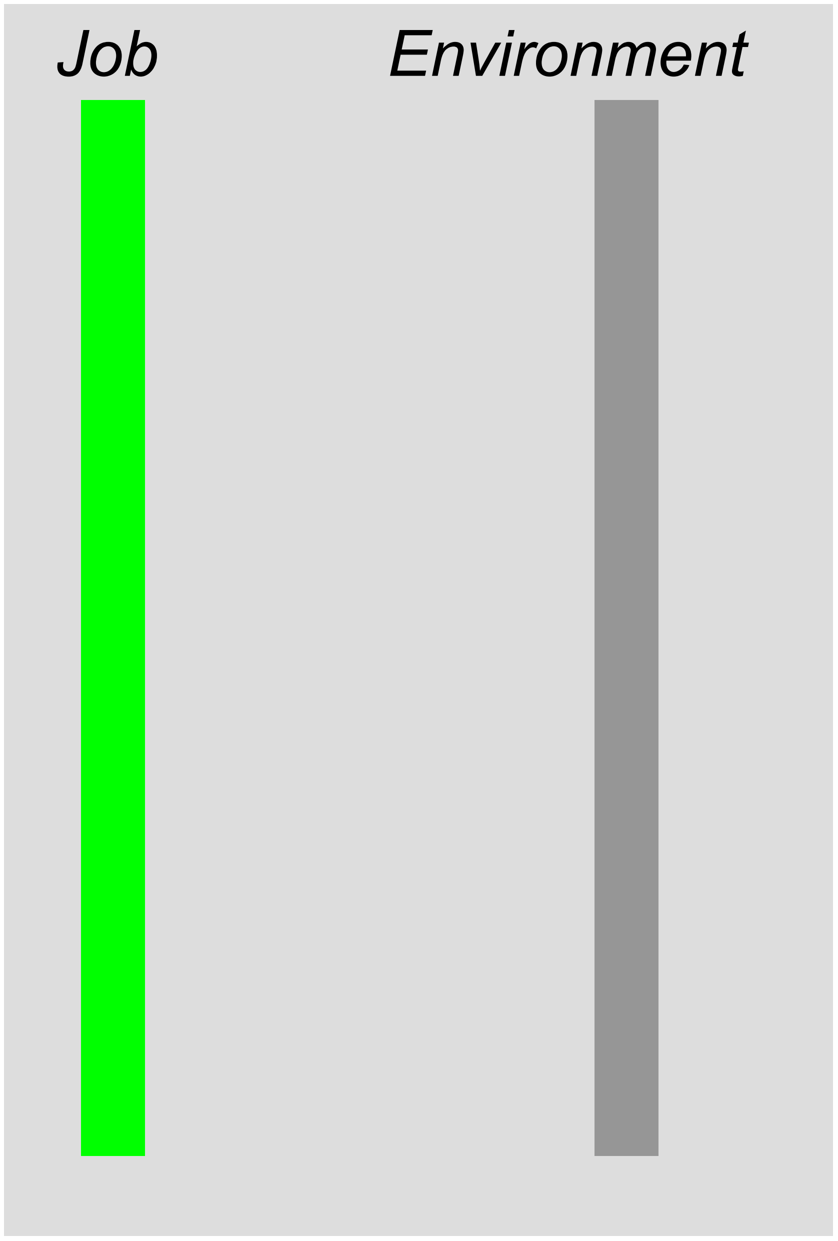
- The completion time is within 2 of optimal.
- If the job is parallel enough, then $T_1/P << T_\infty$. The completion time is almost optimal (linear speedup).



Outline

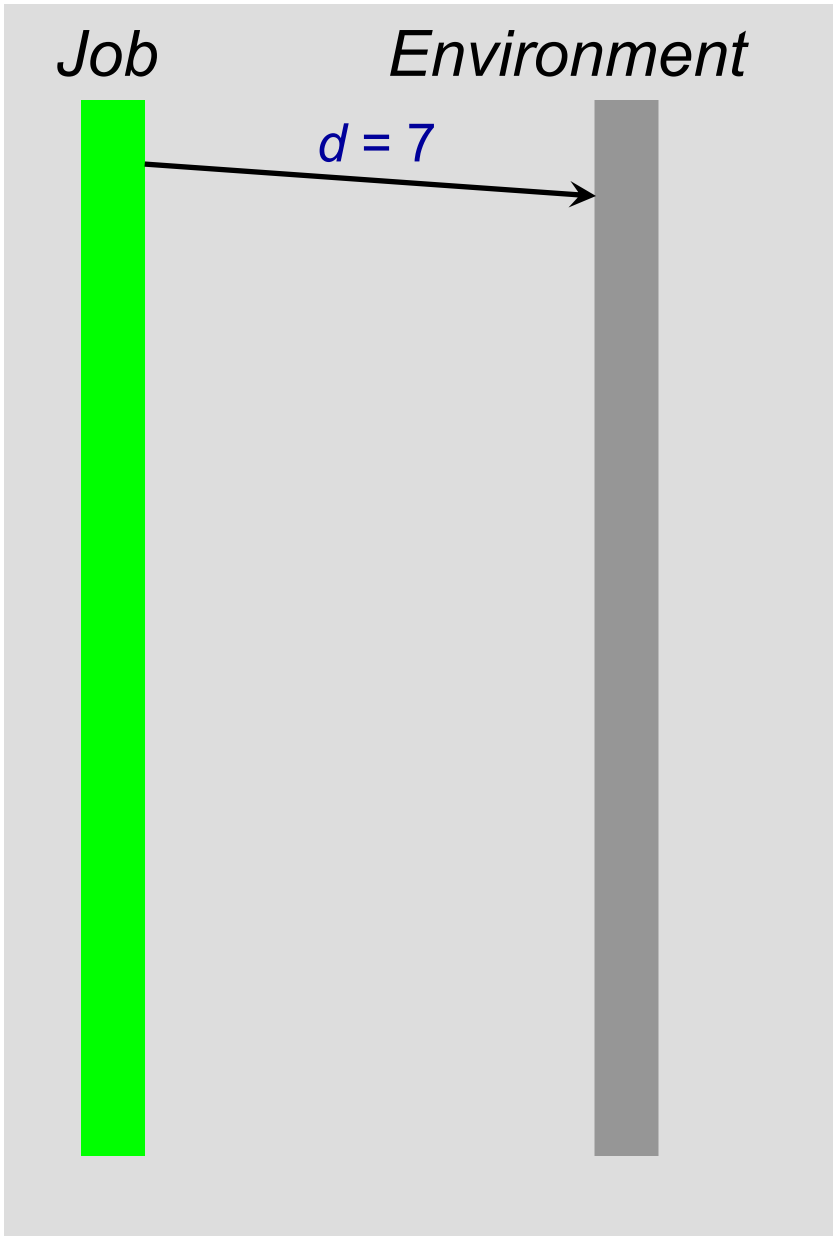
- Introduction
- The Feedback Algorithm
- Greedy Scheduling
- Adversarial Environment and Trim Analysis
- Analysis Idea
- Conclusions

Processor Availability



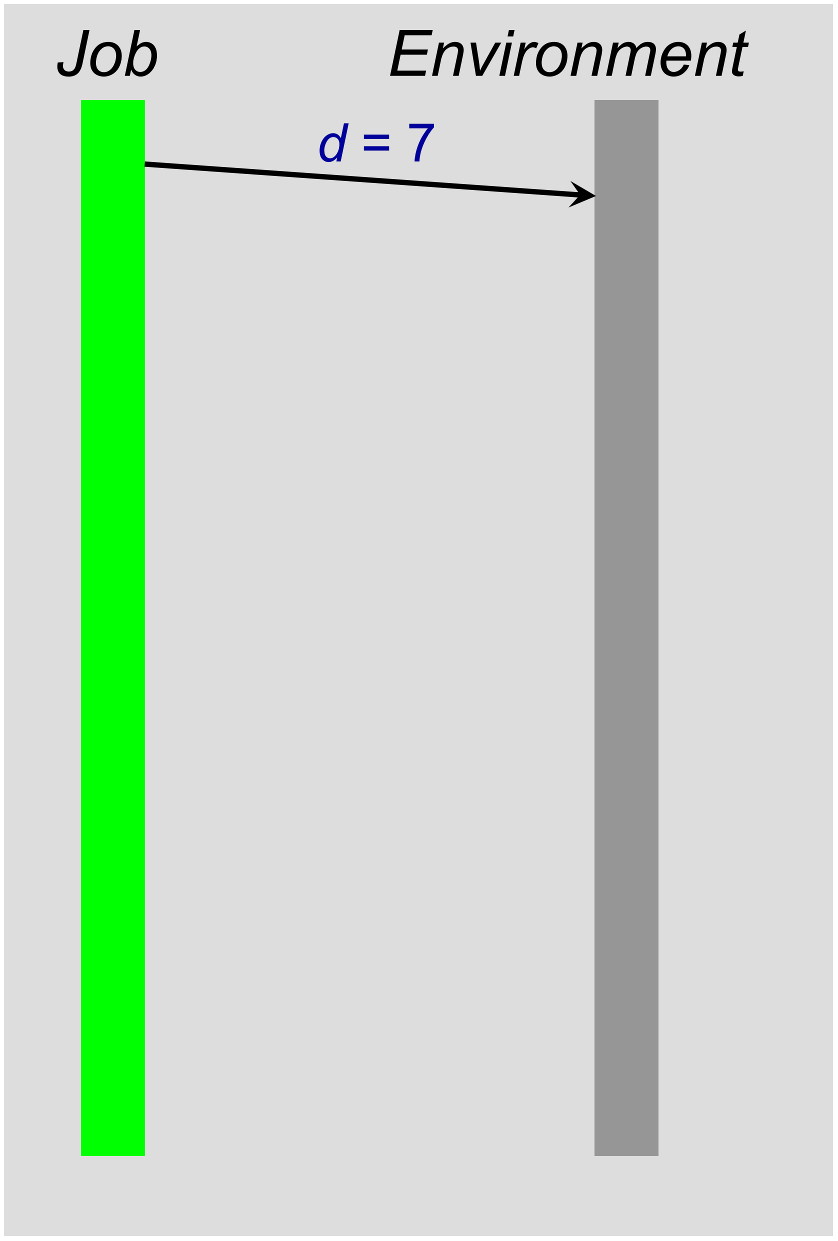
Processor Availability

- Between quanta, the job interacts with its environment to decide its allotment for the next quantum.



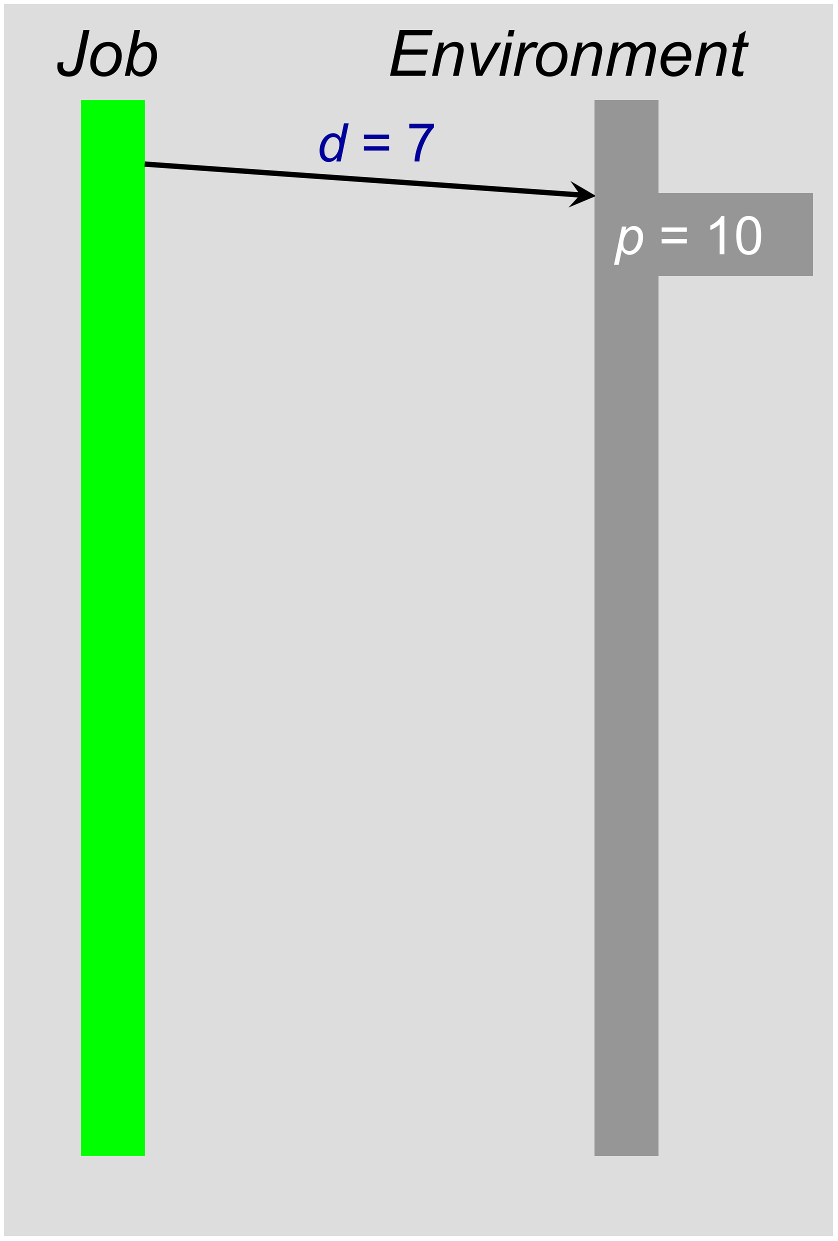
Processor Availability

- Between quanta, the job interacts with its environment to decide its allotment for the next quantum.
- The environment represents the other jobs in the system, the allocation policy of the job scheduler, etc.



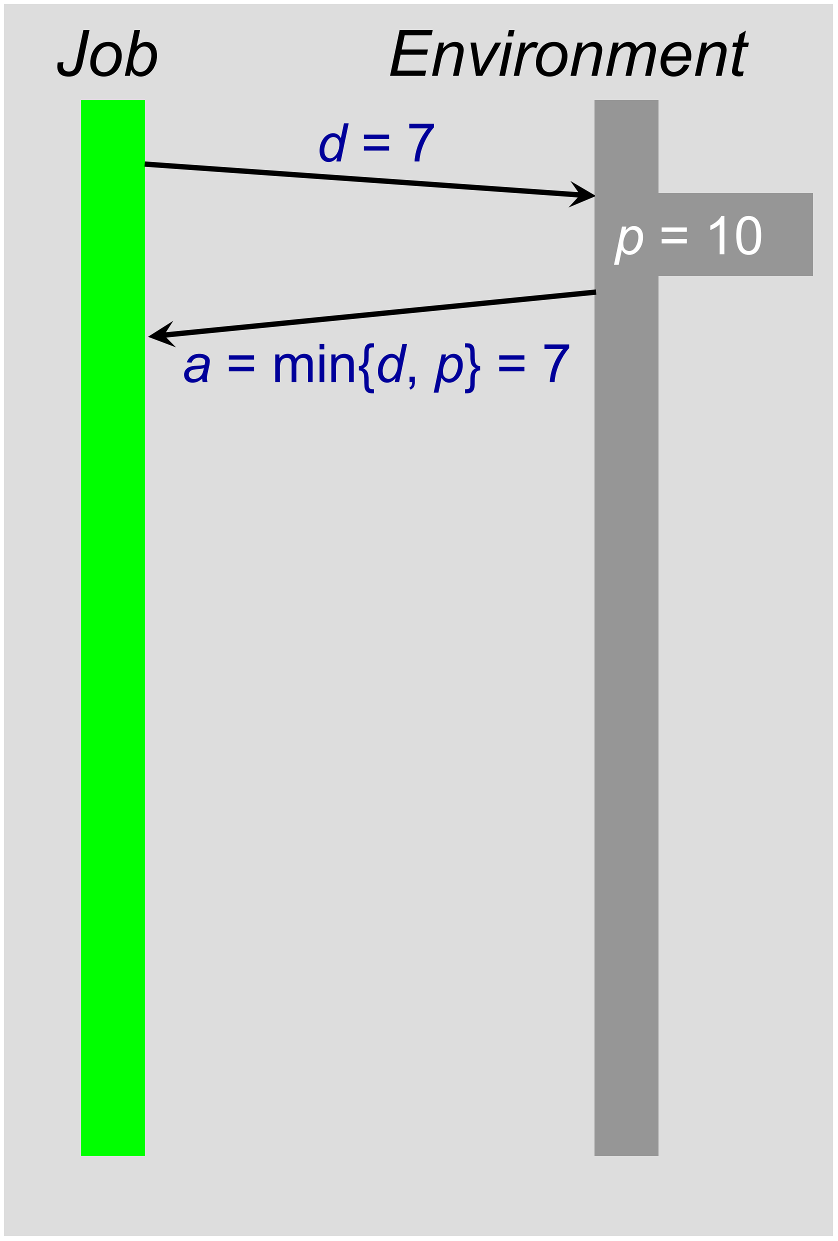
Processor Availability

- Between quanta, the job interacts with its environment to decide its allotment for the next quantum.
- The environment represents the other jobs in the system, the allocation policy of the job scheduler, etc.
- By whatever means, the environment determines the *processor availability* *p* for the job.



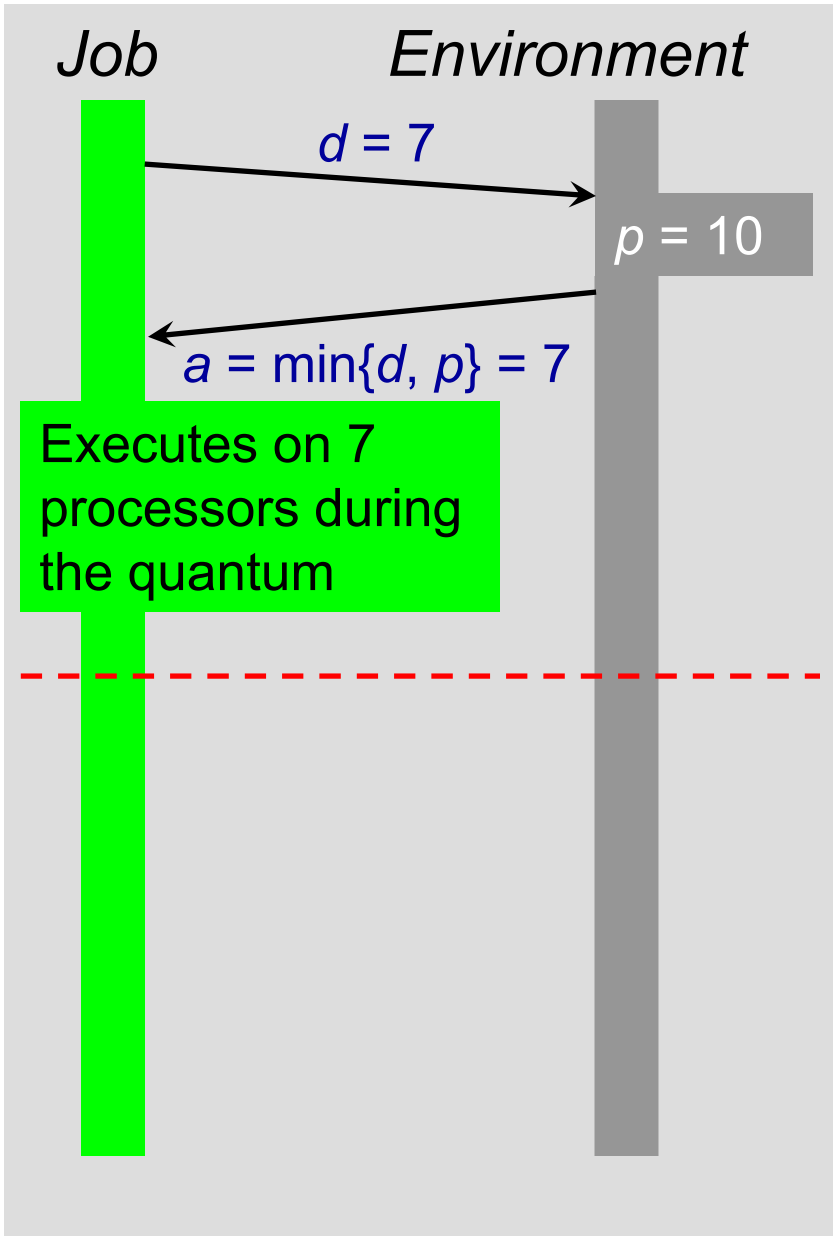
Processor Availability

- Between quanta, the job interacts with its environment to decide its allotment for the next quantum.
- The environment represents the other jobs in the system, the allocation policy of the job scheduler, etc.
- By whatever means, the environment determines the *processor availability* p for the job.
- The allotment of the job is always the minimum of the desire and the processor availability.



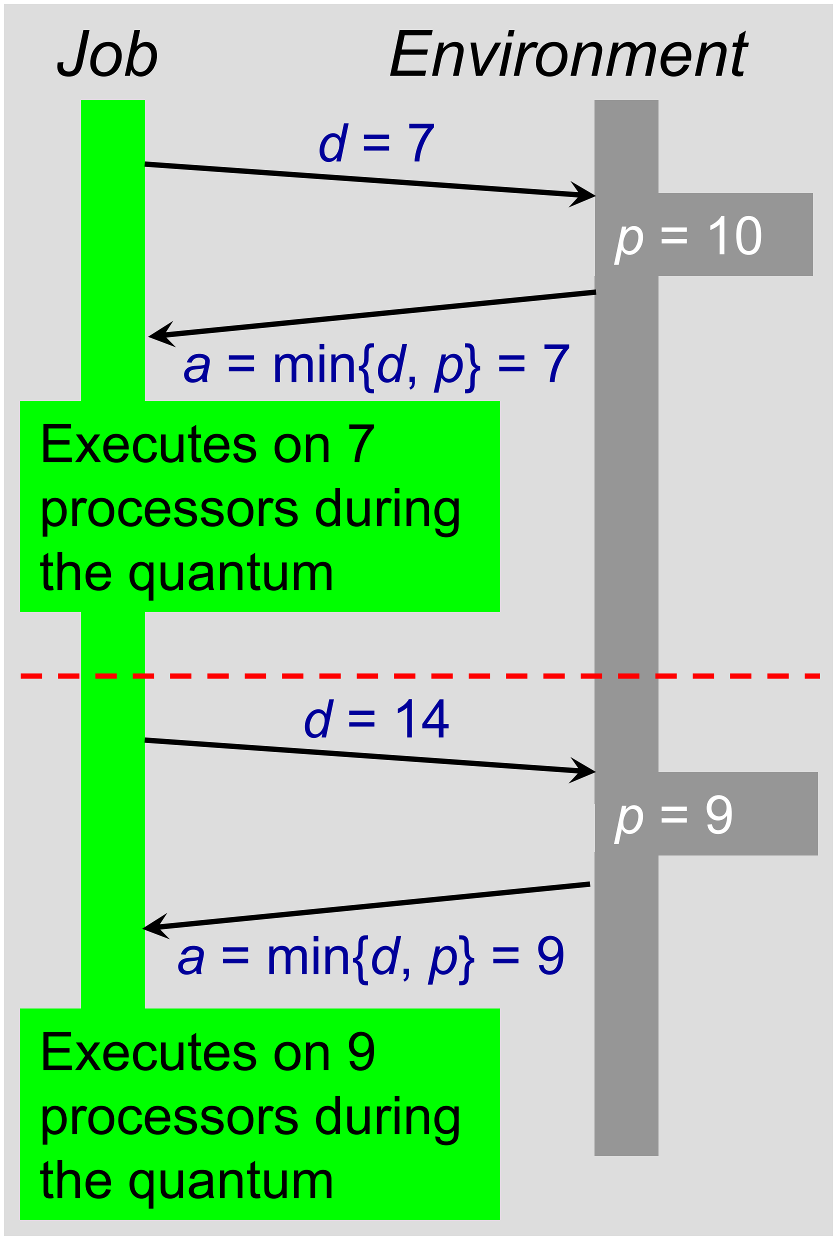
Processor Availability

- Between quanta, the job interacts with its environment to decide its allotment for the next quantum.
- The environment represents the other jobs in the system, the allocation policy of the job scheduler, etc.
- By whatever means, the environment determines the *processor availability* p for the job.
- The allotment of the job is always the minimum of the desire and the processor availability.



Processor Availability

- Between quanta, the job interacts with its environment to decide its allotment for the next quantum.
- The environment represents the other jobs in the system, the allocation policy of the job scheduler, etc.
- By whatever means, the environment determines the *processor availability* p for the job.
- The allotment of the job is always the minimum of the desire and the processor availability.



The Environment

The completion time of the job depends on its environment's processor availability.

- For example, if the environment decides that the processor availability of the job is always 1, then the job runs slowly, no matter what its parallelism.

Therefore, we analyze the completion time relative to the availability provided by the environment.

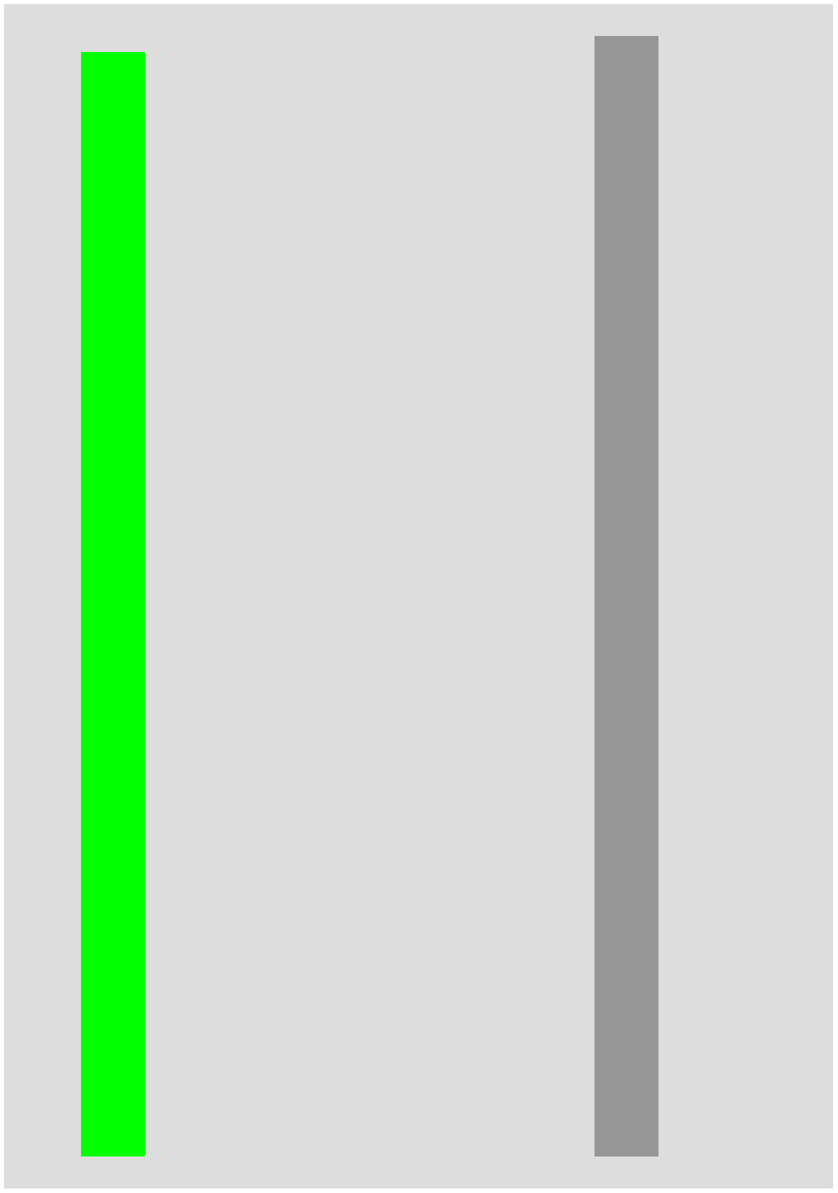
We assume that the environment is an *adversary* of the job.

- Thus, our results apply to *any* environment.

We would like to prove that A-GREEDY provides speedup with respect to (proportional to) the *mean availability* P_{mean}.

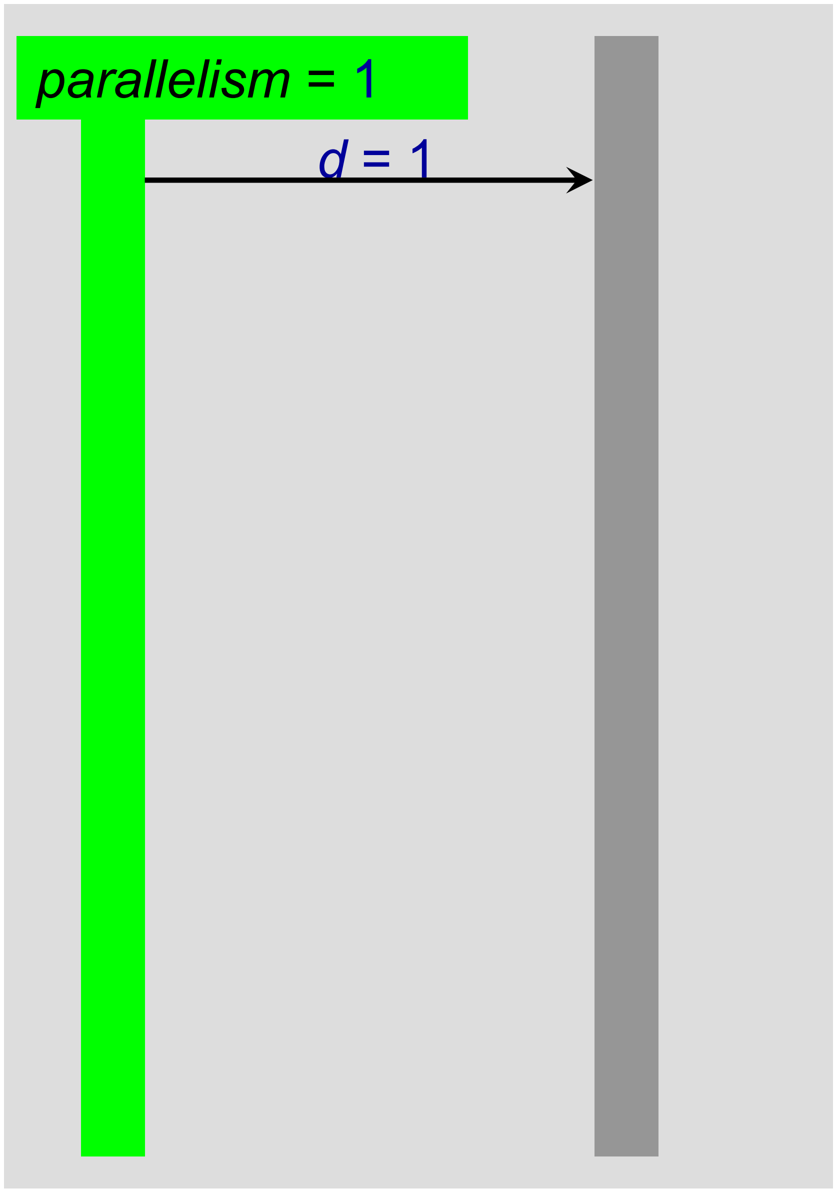
Power of the Adversary

PROBLEM: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.



Power of the Adversary

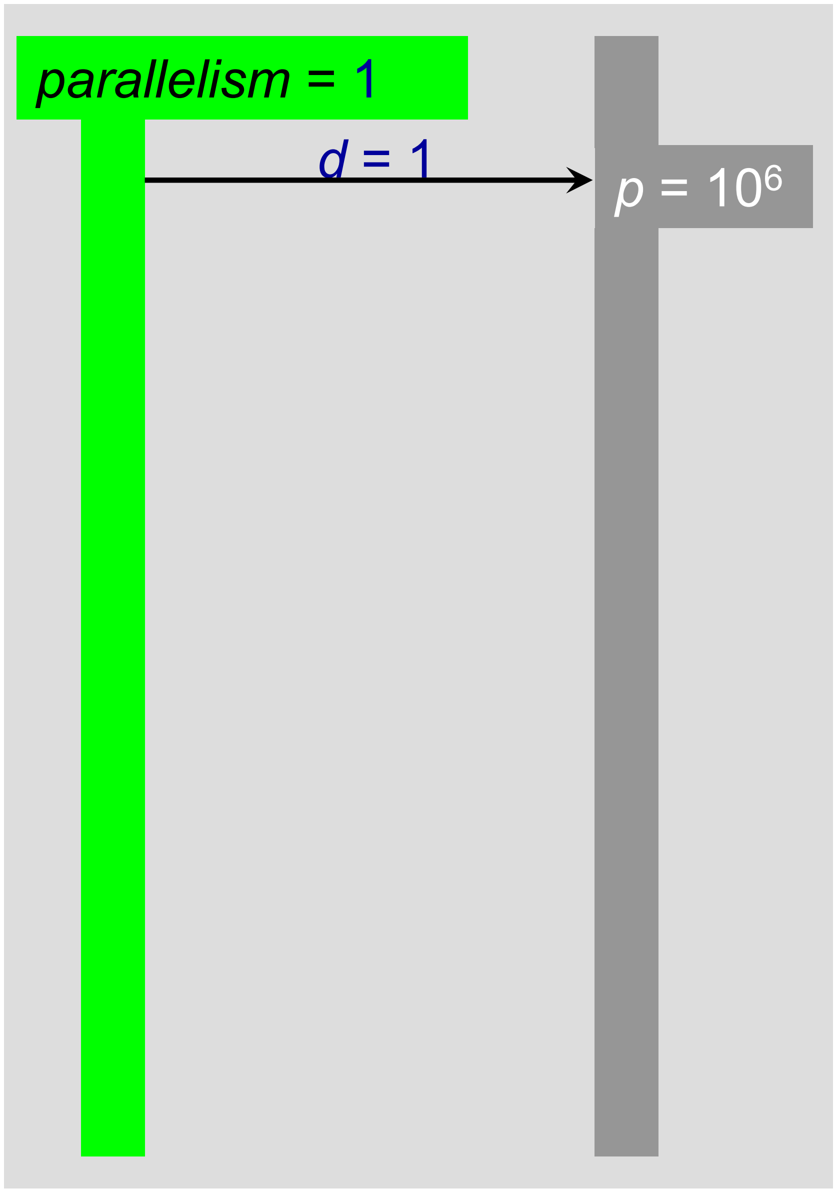
Problem: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.



Power of the Adversary

PROBLEM: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.

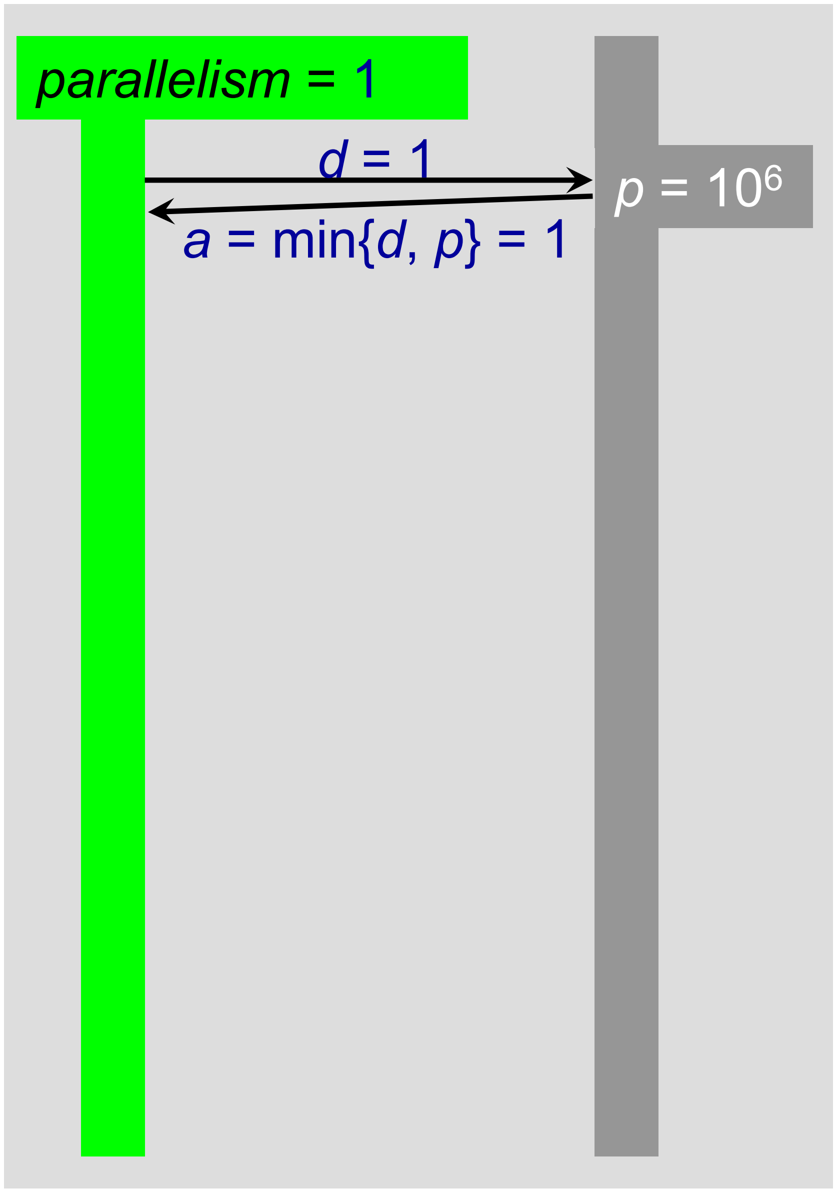
- When a job has small parallelism and requests few processors, the environment makes the availability huge.



Power of the Adversary

Problem: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.

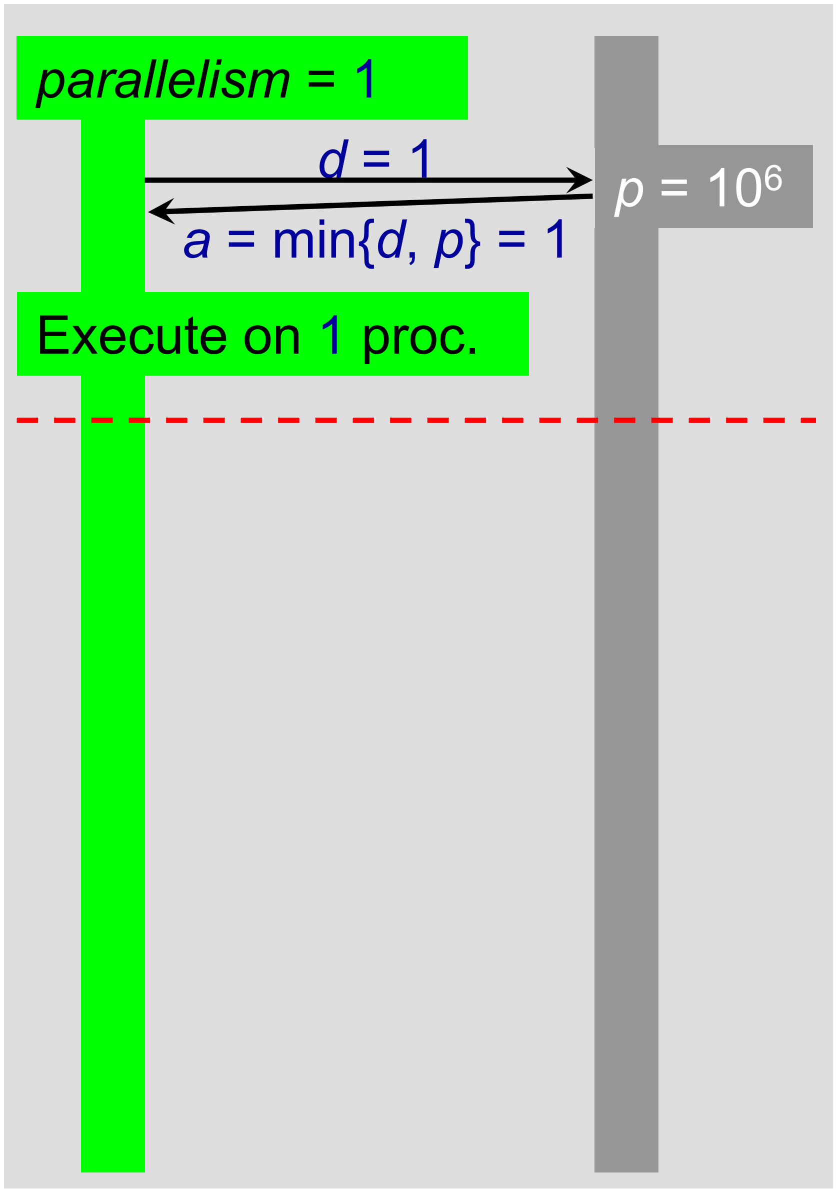
- When a job has small parallelism and requests few processors, the environment makes the availability huge.



Power of the Adversary

PROBLEM: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.

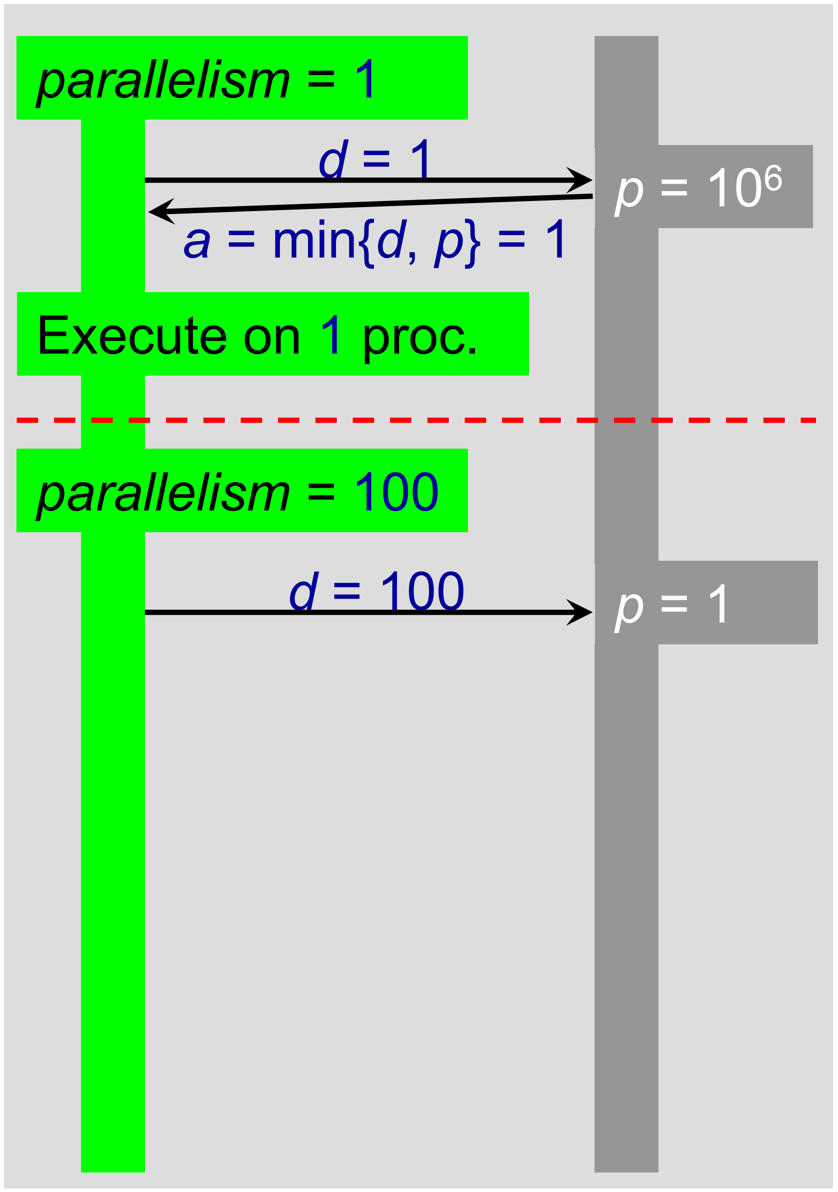
- When a job has small parallelism and requests few processors, the environment makes the availability huge.



Power of the Adversary

PROBLEM: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.

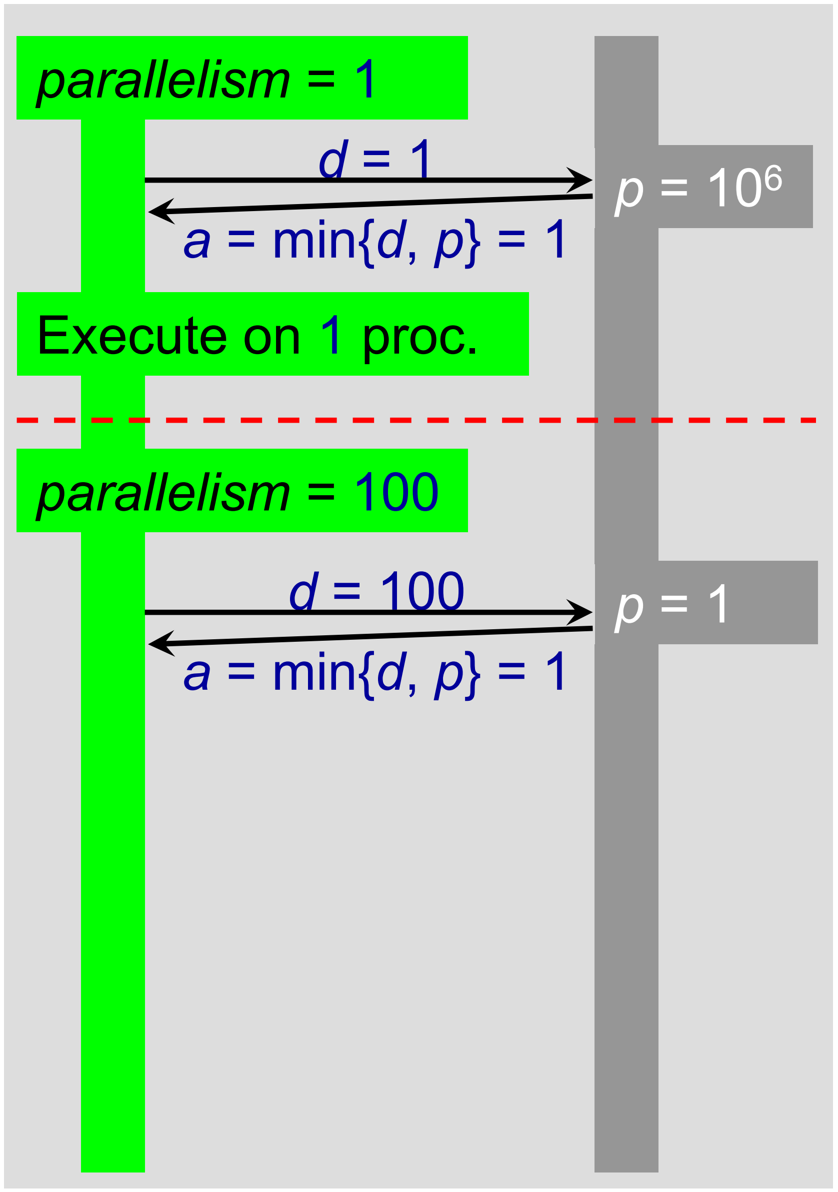
- When a job has small parallelism and requests few processors, the environment makes the availability huge.
- When a job has large parallelism, the environment makes the availability small.



Power of the Adversary

PROBLEM: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.

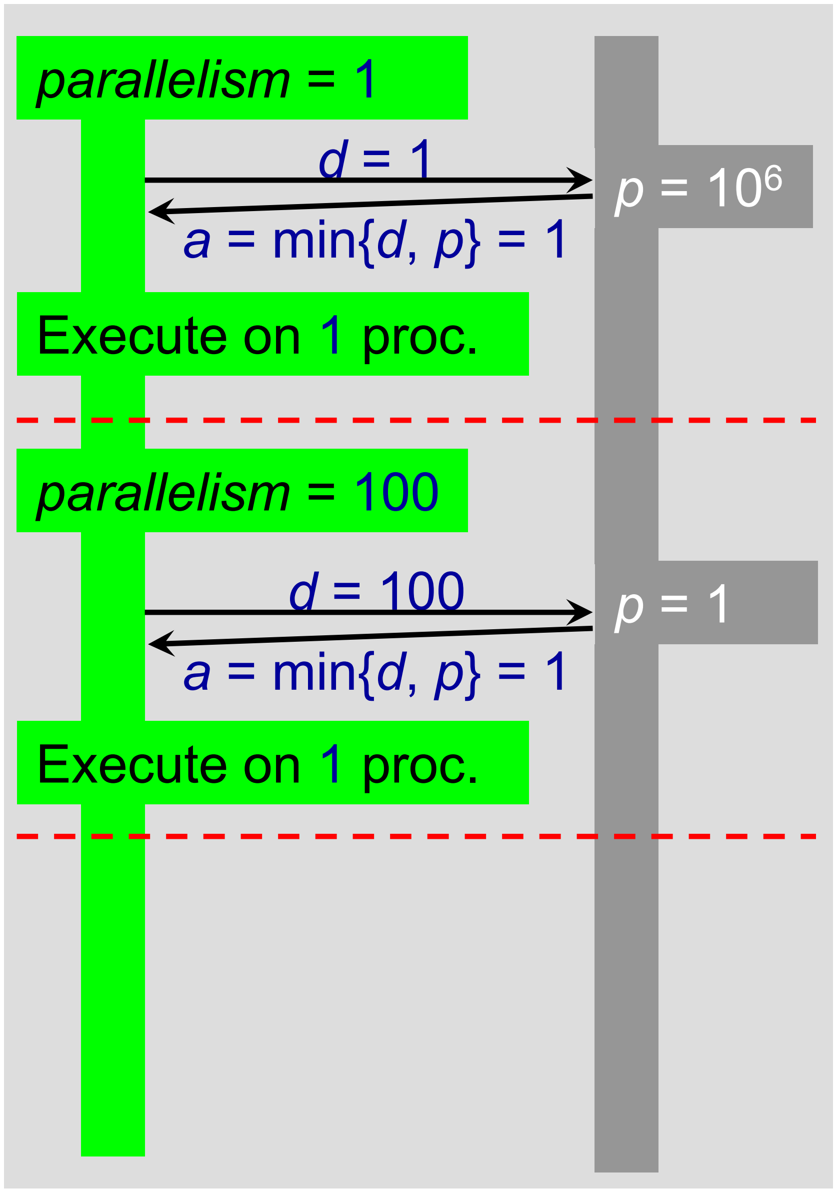
- When a job has small parallelism and requests few processors, the environment makes the availability huge.
- When a job has large parallelism, the environment makes the availability small.



Power of the Adversary

PROBLEM: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.

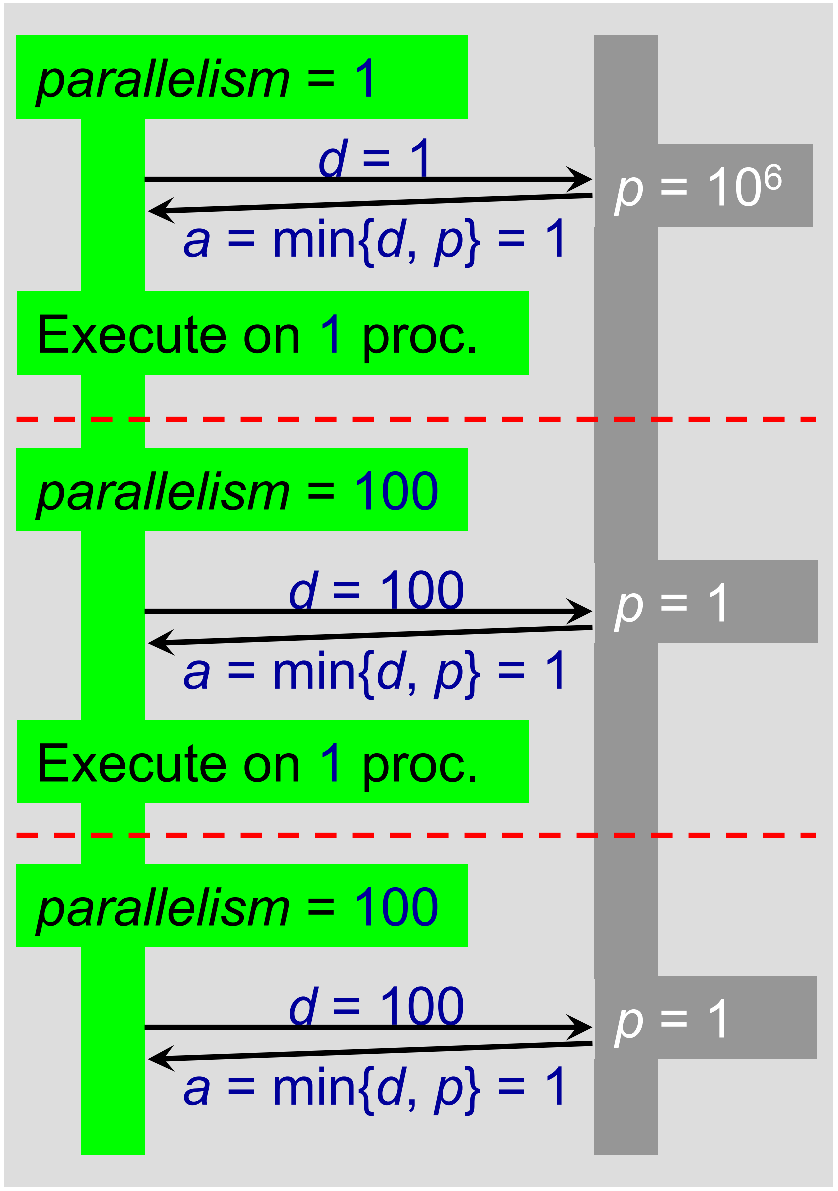
- When a job has small parallelism and requests few processors, the environment makes the availability huge.
- When a job has large parallelism, the environment makes the availability small.



Power of the Adversary

PROBLEM: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.

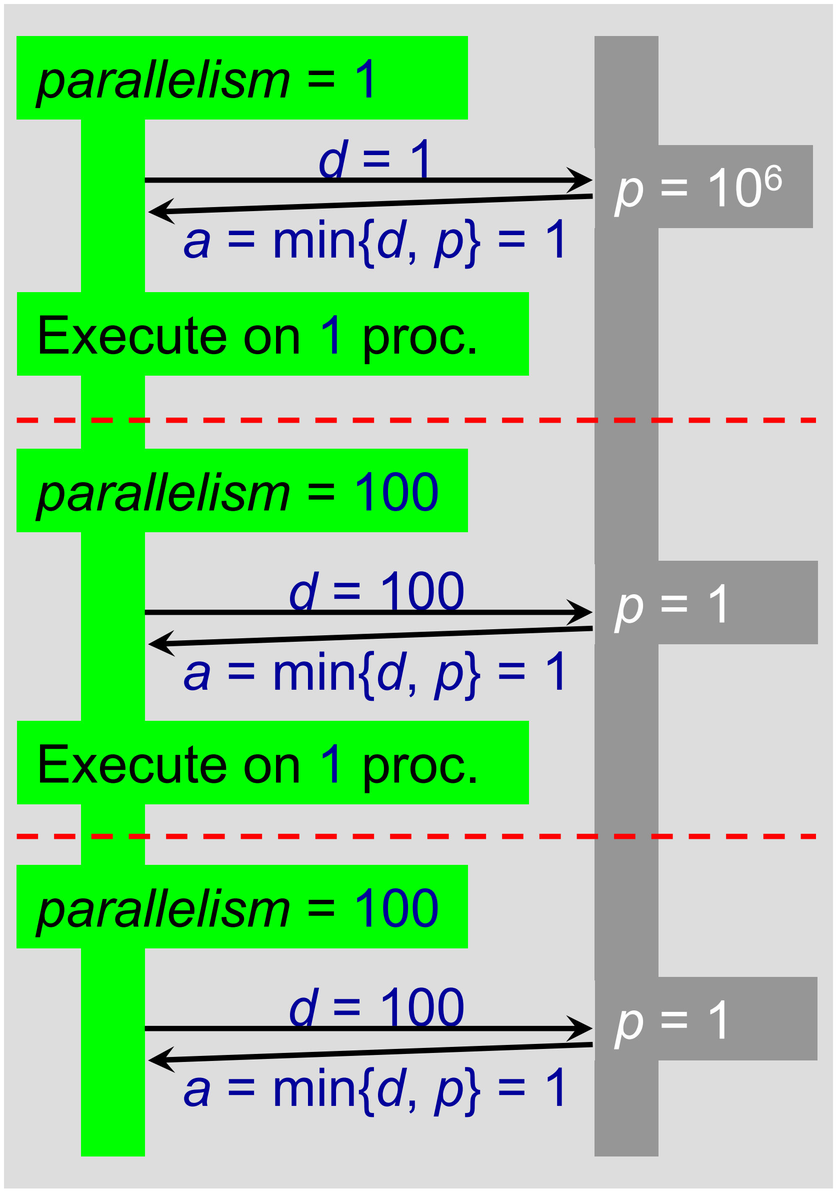
- When a job has small parallelism and requests few processors, the environment makes the availability huge.
- When a job has large parallelism, the environment makes the availability small.



Power of the Adversary

PROBLEM: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.

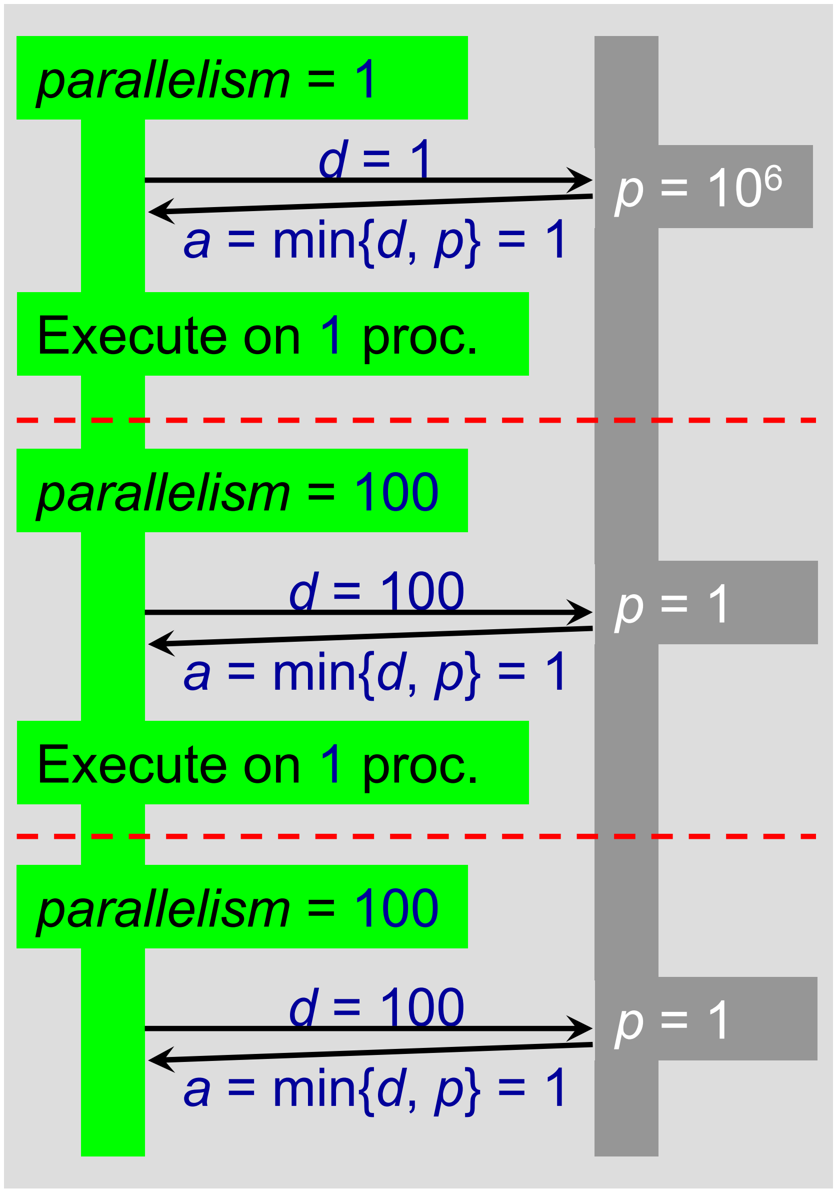
- When a job has small parallelism and requests few processors, the environment makes the availability huge.
- When a job has large parallelism, the environment makes the availability small.
- Therefore, the job's allotment is always small.



Power of the Adversary

PROBLEM: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.

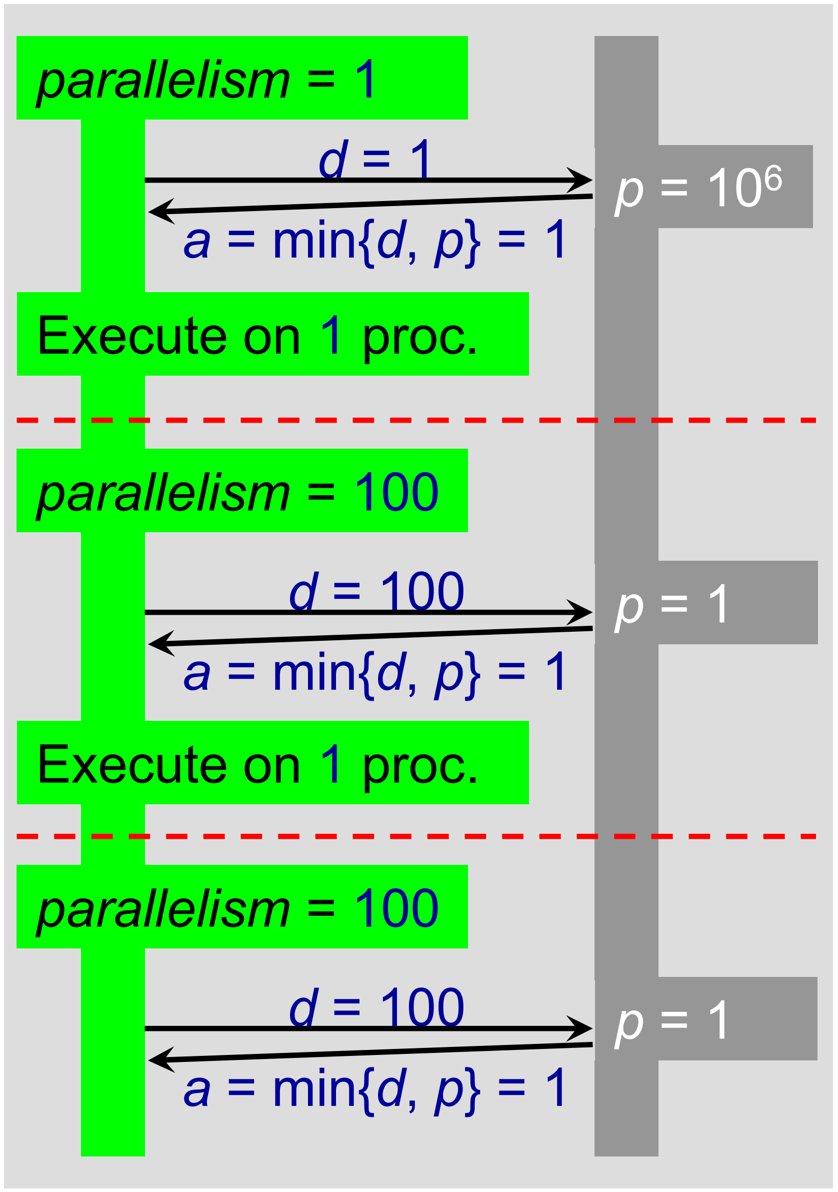
- Since the availability is huge on some quanta, the *mean availability* P_{mean} is large.



Power of the Adversary

PROBLEM: Even *omniscient* parallelism feedback cannot guarantee good speedup with respect to the *mean availability* P_{mean} against an adversary.

- Since the availability is huge on some quanta, the *mean availability* P_{mean} is large.
- But the availability is large only on time steps when the job isn't able to use the available processors. Therefore, it runs virtually serially.

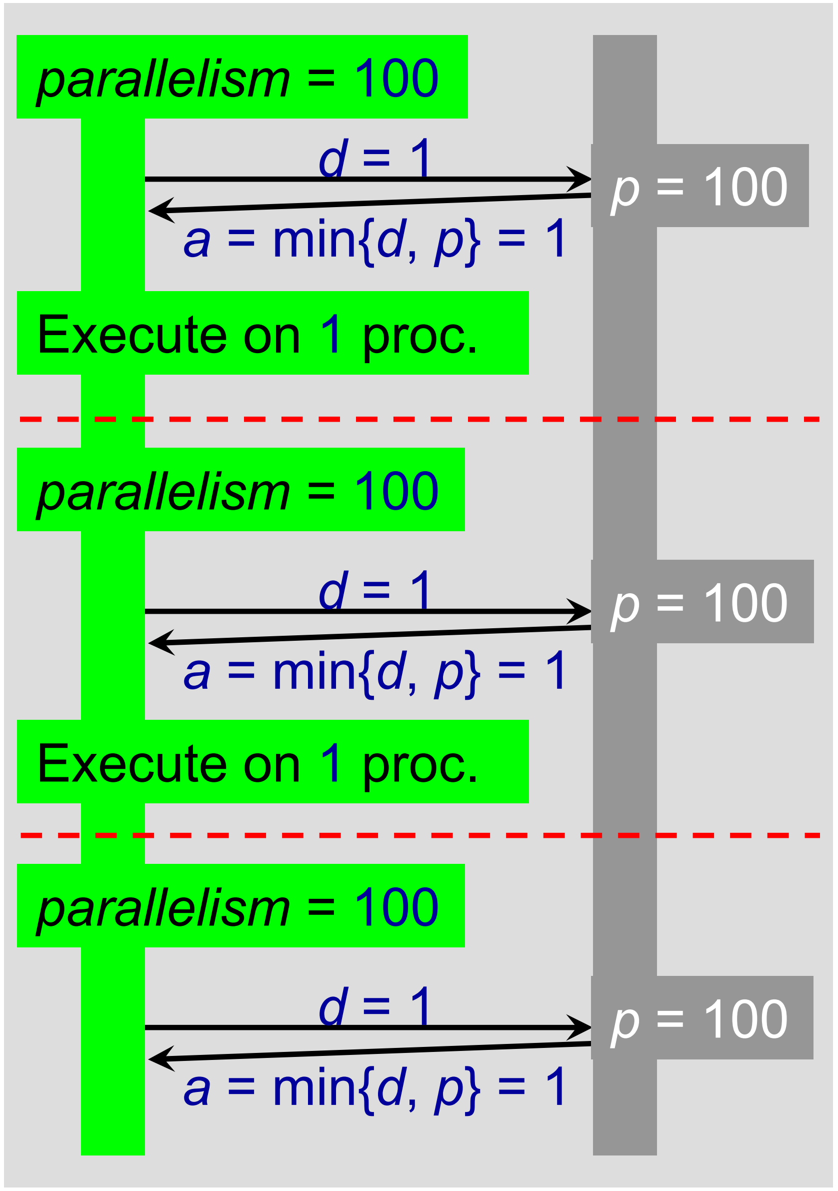


Mean Allotment Does Not Work

Try to get speedup with respect to the mean allotment. Here is a trivial algorithm.

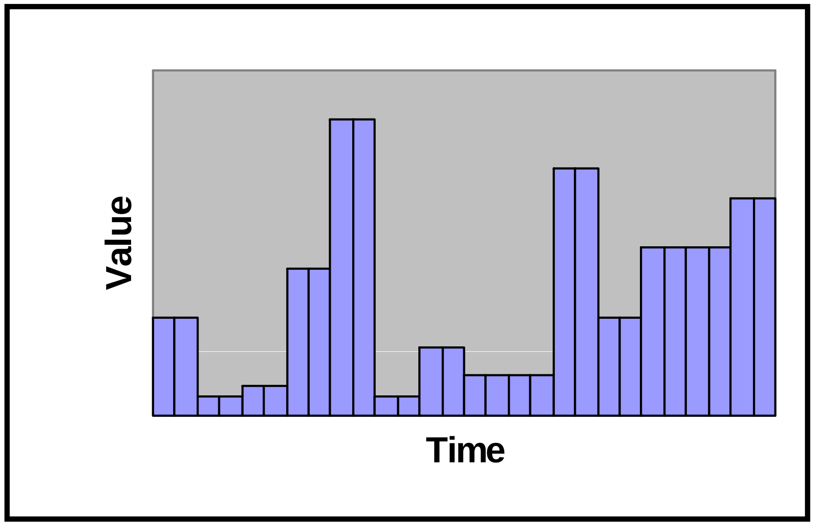
- Always requests 1 processor.
- The job achieves optimal completion time with respect to the mean allotment.
- And the job does not waste any processor cycles.

PROBLEM: The algorithm provides no useful parallelism feedback.



R-Trimmed Mean

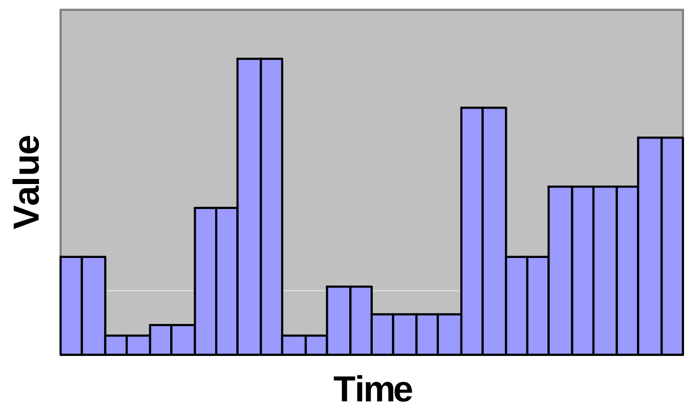
R-trimmed mean $S_{\mathrm{trim}(R)}$ of a time series S is computed as follows:

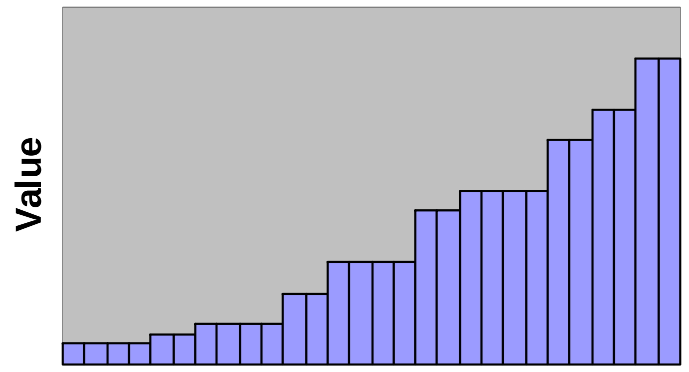


R-Trimmed Mean

R-trimmed mean $S_{\text{trim}(R)}$ of a time series S is computed as follows:

- Sort the data in the set by value.

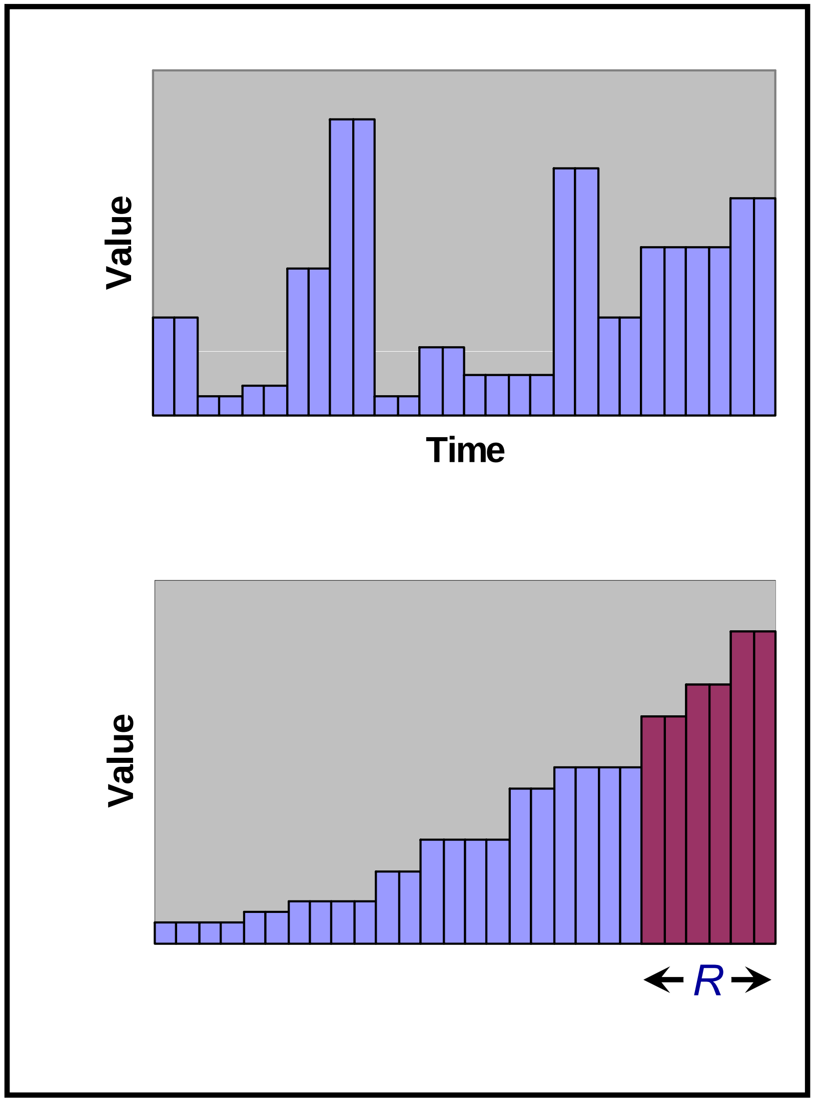




R-Trimmed Mean

R-trimmed mean $S_{\text{trim}(R)}$ of a time series S is computed as follows:

- Sort the data in the set by value.
- Ignore the *R* *largest* values.

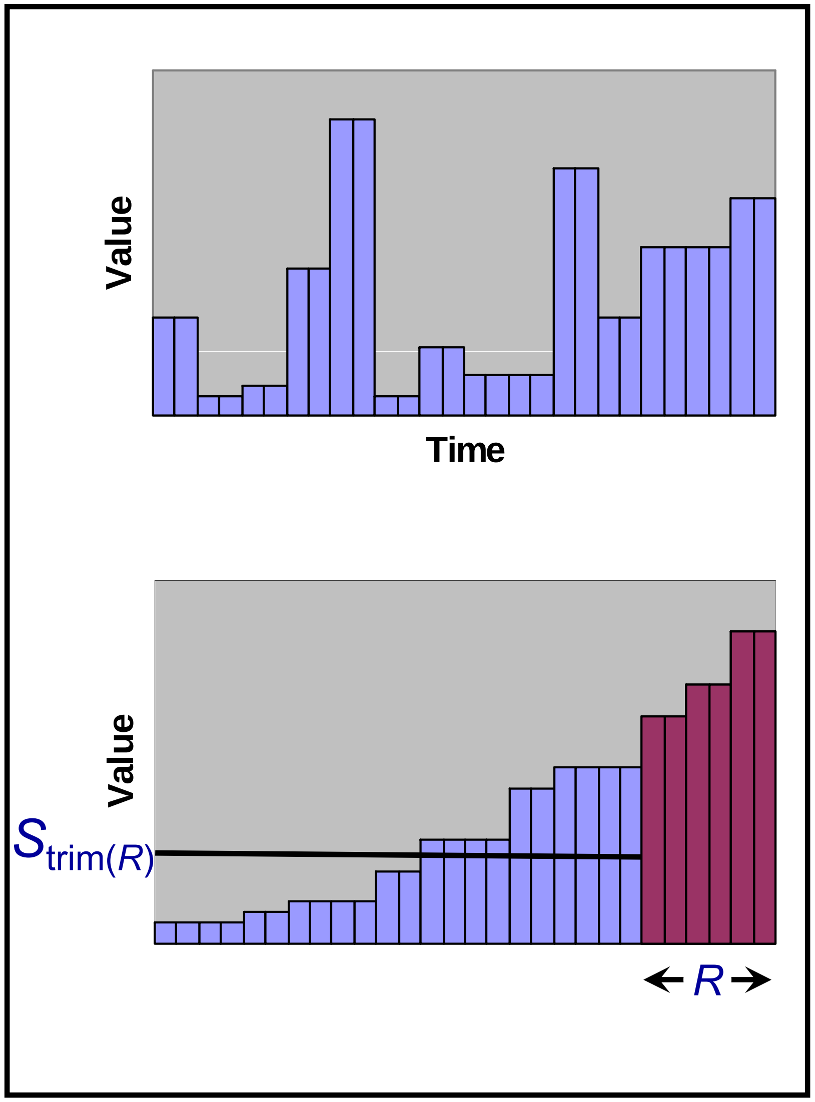


R-Trimmed Mean

R-trimmed mean $S_{\text{trim}(R)}$ of a time series S is computed as follows:

- Sort the data in the set by value.
- Ignore the R *largest* values.
- Compute the mean of the remaining.

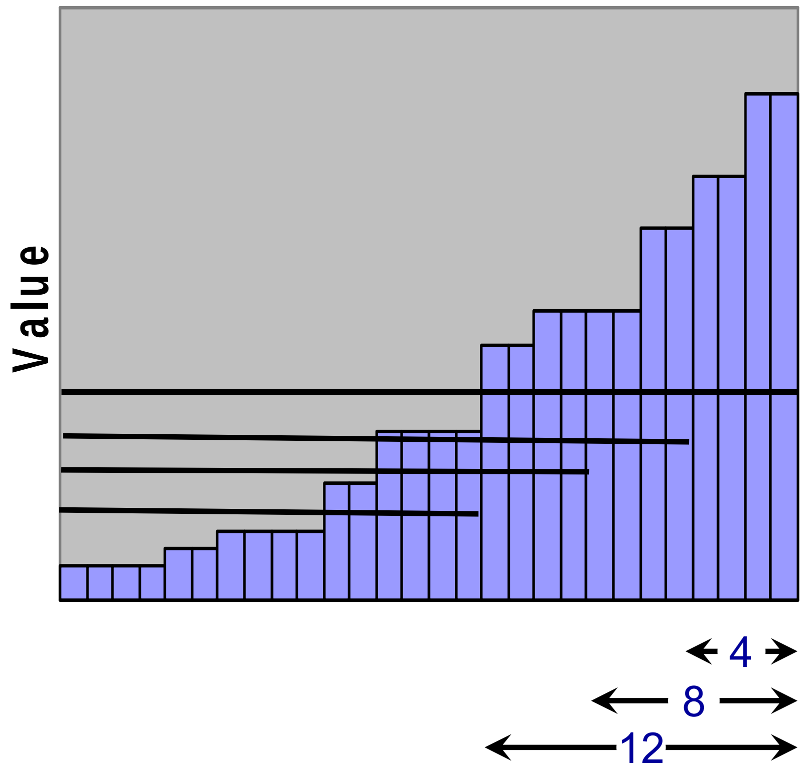
In general, as R *increases*, R-trimmed mean $S_{\text{trim}(R)}$ *decreases*.



Trim Analysis

IDEA: *Trim* (ignore) the *R* steps with the highest availability.

- Against an adversarial environment, no parallelism feedback algorithm can guarantee linear speedup with respect to 0-trimmed mean availability.
- A good parallelism feedback algorithm should provide linear speedup with respect to *R*-trimmed mean availability, where *R* is small.



Outline

- Introduction
- The Feedback Algorithm
- Greedy Scheduling
- Adversarial Environment and Trim Analysis
- Analysis Idea
- Conclusions

Three Types of Quanta

```
FEEDBACK($q, \delta$)
1. if $q = 1$
2.    then $d_q \leftarrow 1$
3. elseif $u_{q-1} < L \delta a_{q-1}$
4.    then $d_q \leftarrow d_{q-1}/2$
5. elseif $a_{q-1} = d_{q-1}$
6.    then $d_q \leftarrow 2d_{q-1}$
7. else $d_q \leftarrow d_{q-1}$
```

Too much waste.
Inefficient Quanta.

Small waste + large allotment.
Efficient and Satisfied Quanta.

Small waste + small allotment.
Efficient and Deprived Quanta.

Analysis Idea — Waste

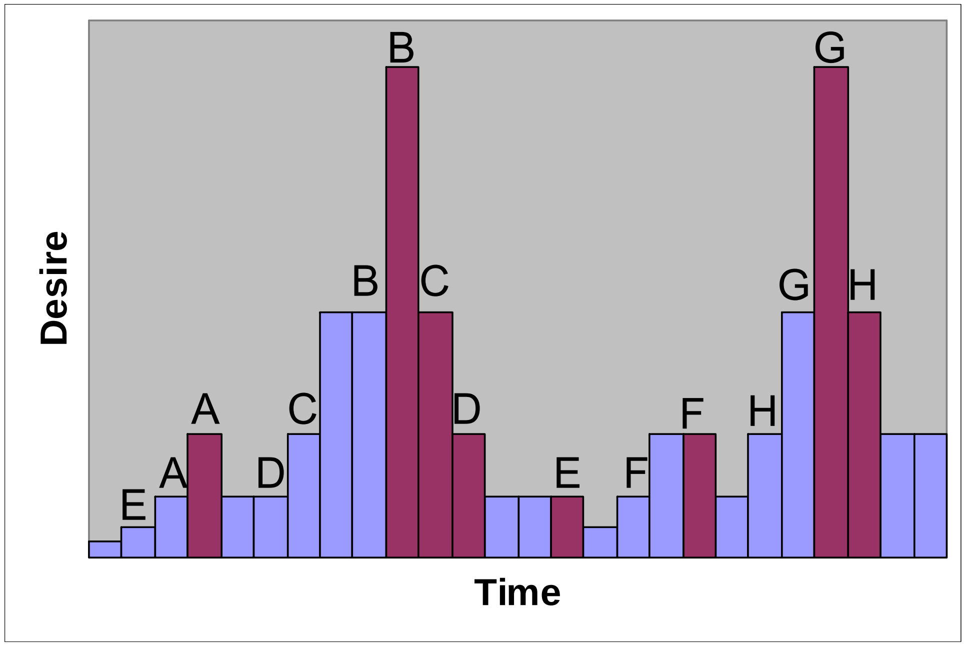
Theorem: A job with work T_1 wastes at most $O(T_1)$ processor cycles.

Analysis Idea — Waste

THEOREM: A job with work T_1 wastes at most $O(T_1)$ processor cycles.

PROOF:

- If there is a inefficient quantum r with desire d_r, there is an earlier efficient quantum s with desire $d_s = d_r/2$.

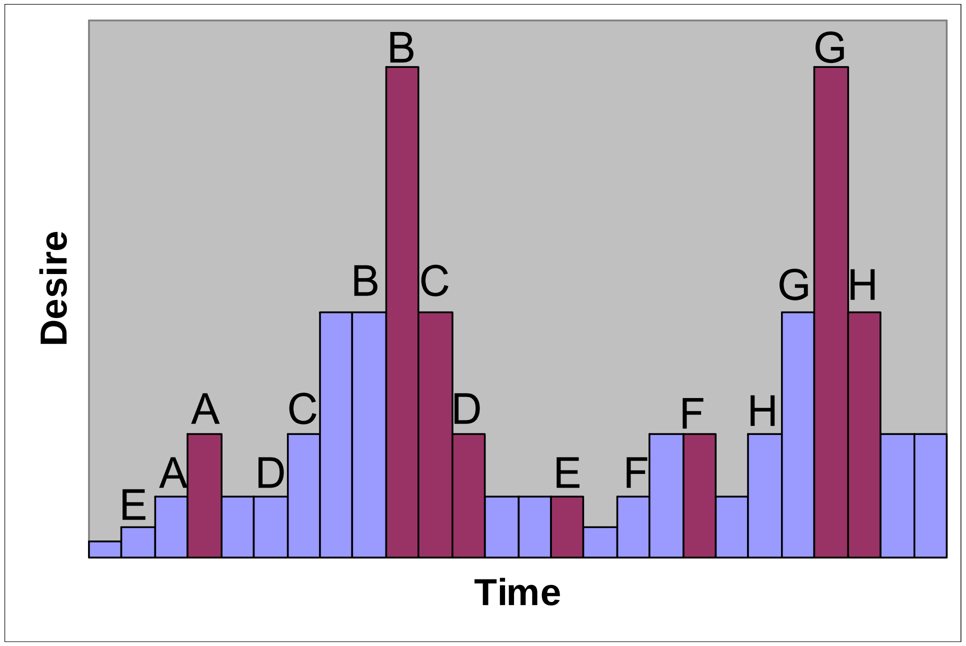


Analysis Idea — Waste

THEOREM: A job with work T_1 wastes at most $O(T_1)$ processor cycles.

PROOF:

- If there is a inefficient quantum r with desire d_r, there is an earlier efficient quantum s with desire $d_s = d_r/2$.
- We amortize the waste during quantum r to the work done during quantum s.



Analysis Idea — Time

THEOREM: Consider a job with work T_1 and span T_∞ running on a P-processor machine with quantum length L and let $R = O(T_\infty + L \lg P)$. A-GREEDY guarantees that the job completes in time $T \leq O(T_1/P_{\text{trim}(R)} + T_\infty + L \lg P)$.

Analysis Idea — Time

THEOREM: Consider a job with work T_1 and span T_∞ running on a P-processor machine with quantum length L and let $R = O(T_\infty + L \lg P)$. A-GREEDY guarantees that the job completes in time $T \leq O(T_1/P_{\text{trim}(R)} + T_\infty + L \lg P)$.

PROOF IDEA: Bound the number of the three types of quanta.

FEEDBACK (q, δ)

```
3. elseif $u_{q-1} < L\delta a_{q-1}$
4.    then $d_q \leftarrow d_{q-1}/2$
```

Too much waste.
Inefficient Quanta.

```
5. elseif $a_{q-1} = d_{q-1}$
6.    then $d_q \leftarrow 2d_{q-1}$
```

Small waste + large allotment.
Efficient and Satisfied Quanta.

```
7. else $d_q \leftarrow d_{q-1}$
```

Small waste + small allotment.
Efficient and Deprived Quanta.

Special Case: $L = 1$ and $\delta = 1$

THEOREM: Consider a job with work T_1 and span T_∞ running on a P-processor machine with $L = 1$ and $R = 2T_\infty + \lg P + 1$. A-GREEDY completes the job in at most $T_1/P_{\text{trim}(R)} + 2T_\infty + \lg P + 1$ time steps.

PROOF IDEA:

FEEDBACK$(q, 1)$

```
3. elseif $u_{q-1} < a_{q-1}$
4.    then $d_q \leftarrow d_{q-1}/2$
```

Too much waste.
Incomplete step

```
5. elseif $a_{q-1} = d_{q-1}$
6.    then $d_q \leftarrow 2d_{q-1}$
```

Small waste + large allotment.
Complete and satisfied step

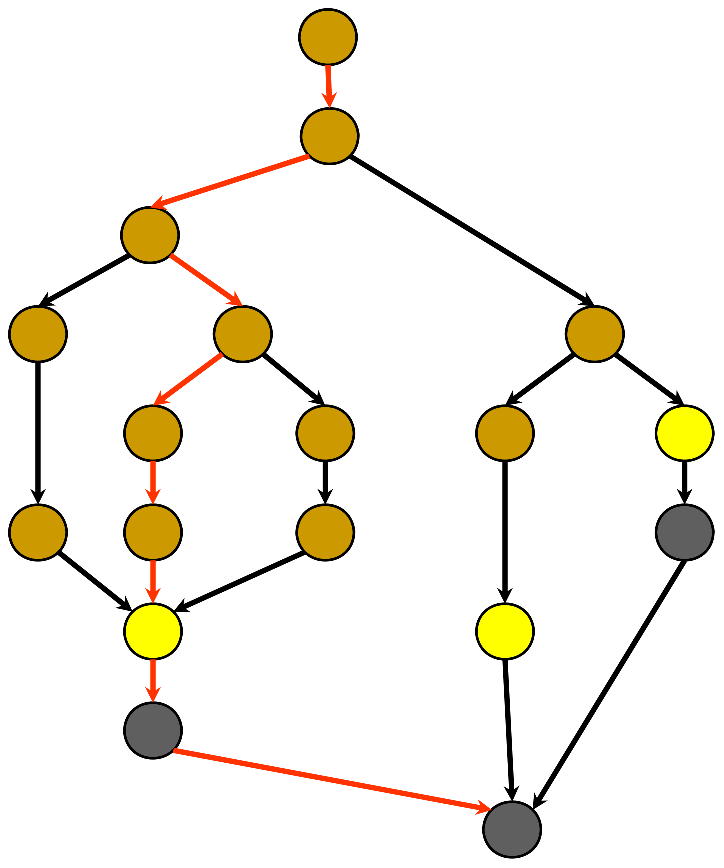
```
7. else $d_q \leftarrow d_{q-1}$
```

Small waste + small allotment.
Complete and deprived step

Incomplete Steps

On an incomplete step, A-GREEDY executes all the ready threads, and therefore reduces the remaining critical path by 1.

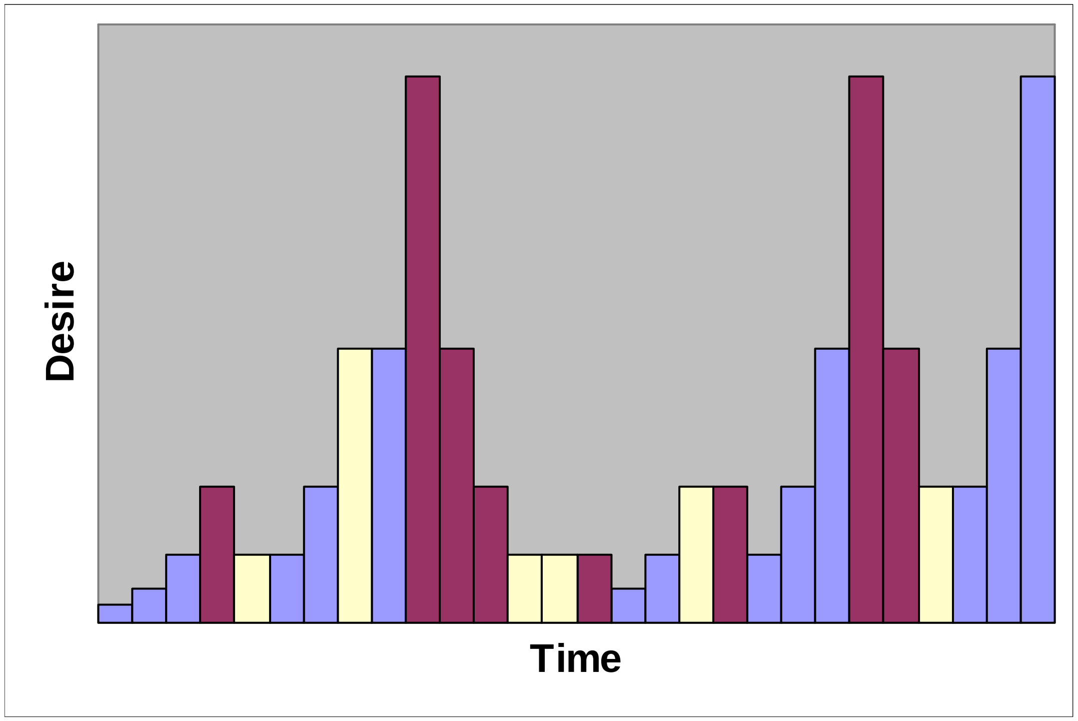
LEMMA 1: There are at most T_∞ incomplete steps.



Complete and Satisfied Steps

- A-GREEDY doubles the desire after complete and satisfied steps, and halves the desire after incomplete steps.

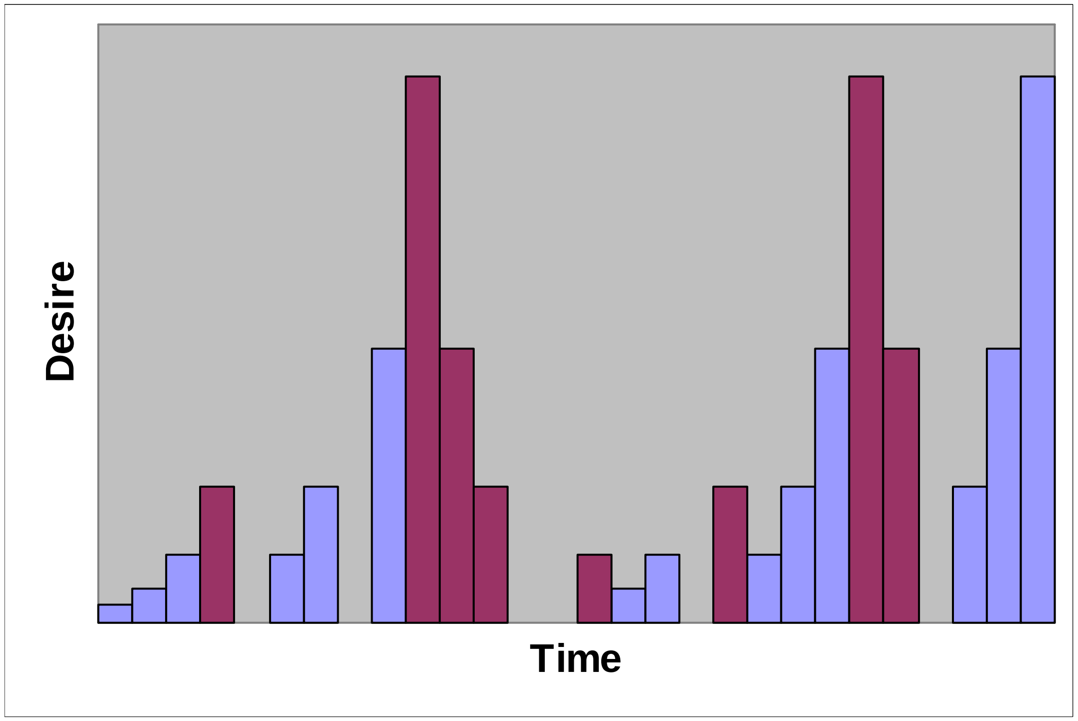
🟪	Incomplete
🟦	Comp. and sat.
🟨	Comp. and dep.



Complete and Satisfied Steps

- A-GREEDY doubles the desire after complete and satisfied steps, and halves the desire after incomplete steps.

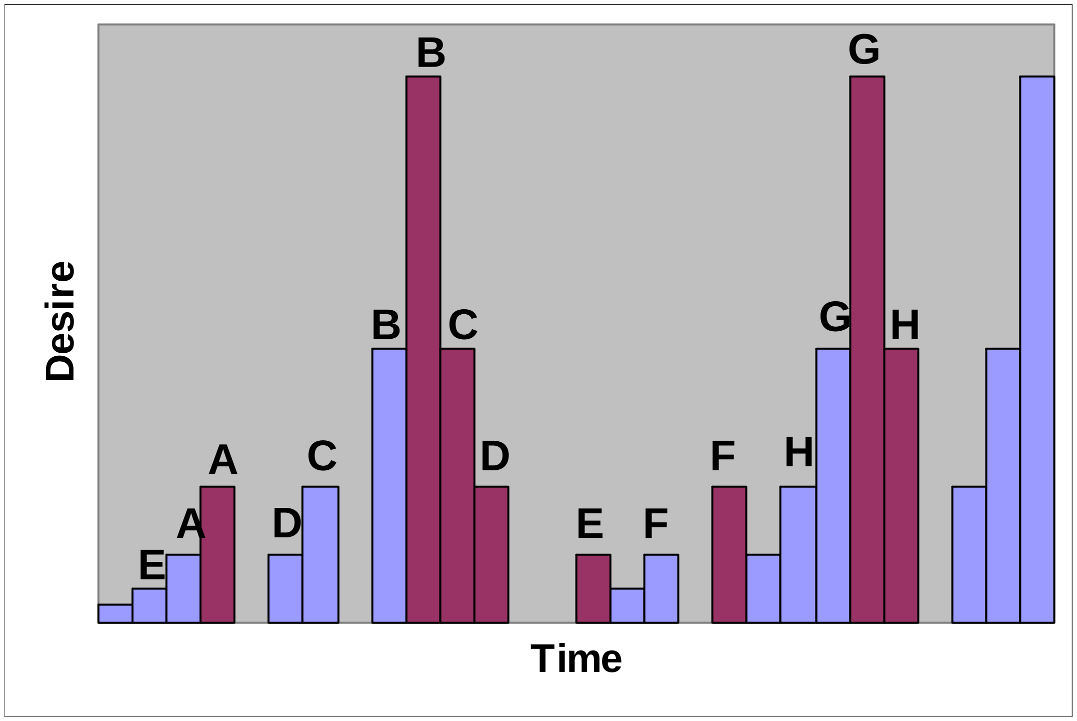
🟪	**Incomplete**
🟦	**Comp. and sat.**
🟨	**Comp. and dep.**



Complete and Satisfied Steps

- A-GREEDY doubles the desire after complete and satisfied steps, and halves the desire after incomplete steps.
- Most complete and satisfied steps are amortized against incomplete steps.

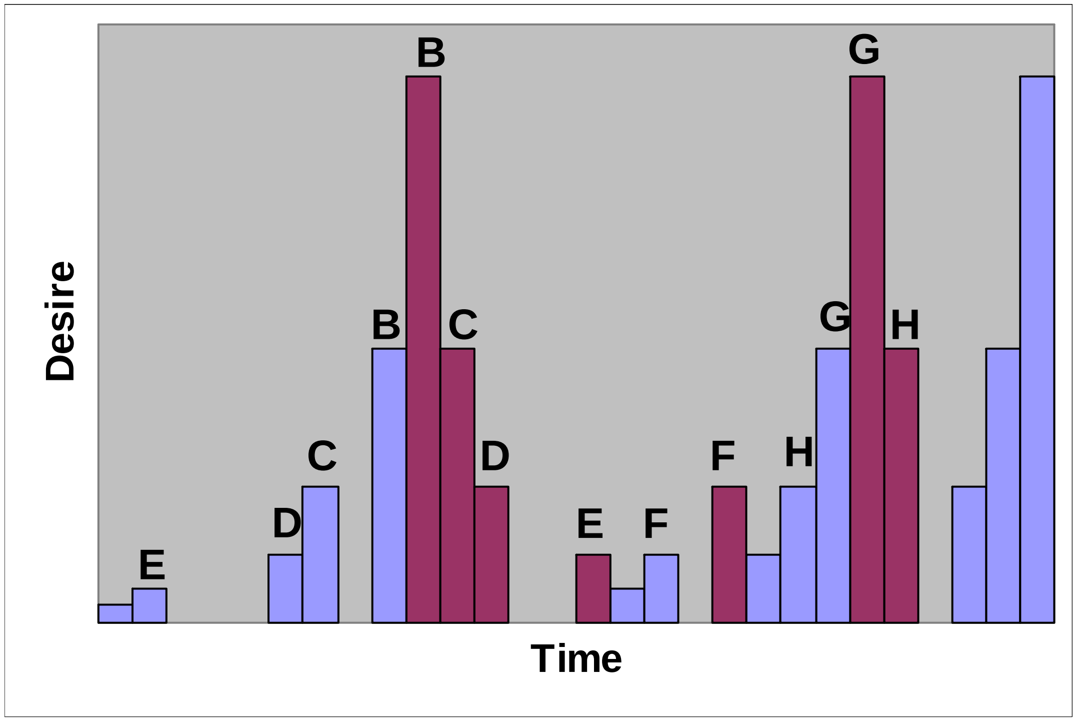
🟪	Incomplete
🟦	Comp. and sat.
⬜	Comp. and dep.



Complete and Satisfied Steps

- A-GREEDY doubles the desire after complete and satisfied steps, and halves the desire after incomplete steps.
- Most complete and satisfied steps are amortized against incomplete steps.

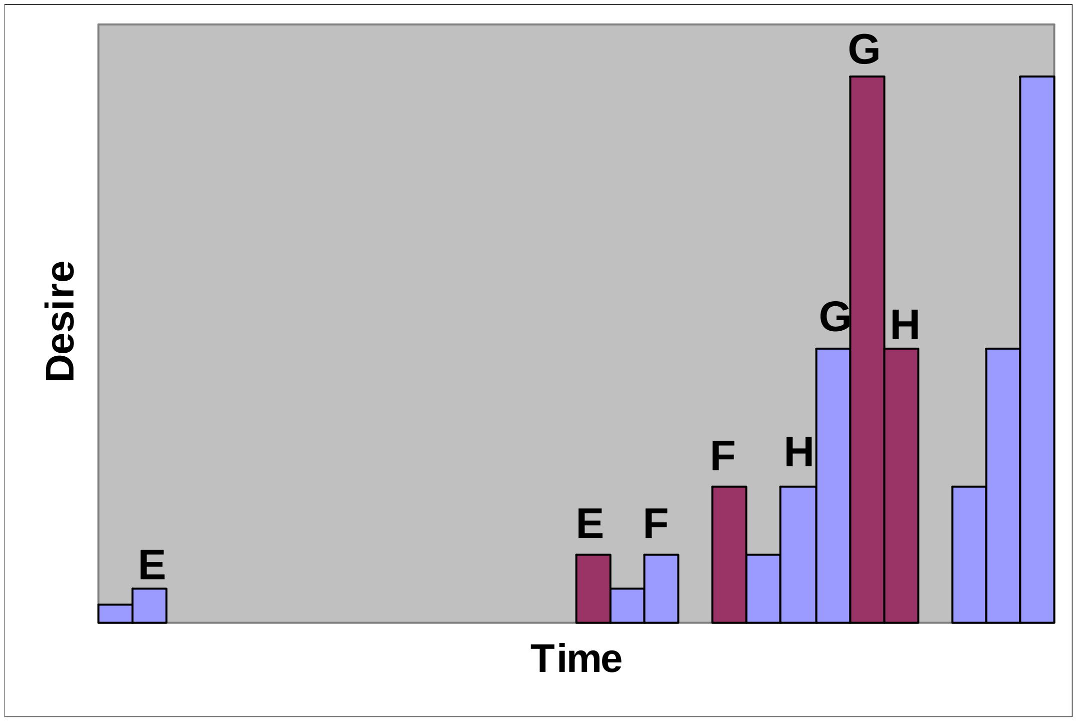
🟪	Incomplete
🟦	Comp. and sat.
⬜	Comp. and dep.



Complete and Satisfied Steps

- A-GREEDY doubles the desire after complete and satisfied steps, and halves the desire after incomplete steps.
- Most complete and satisfied steps are amortized against incomplete steps.

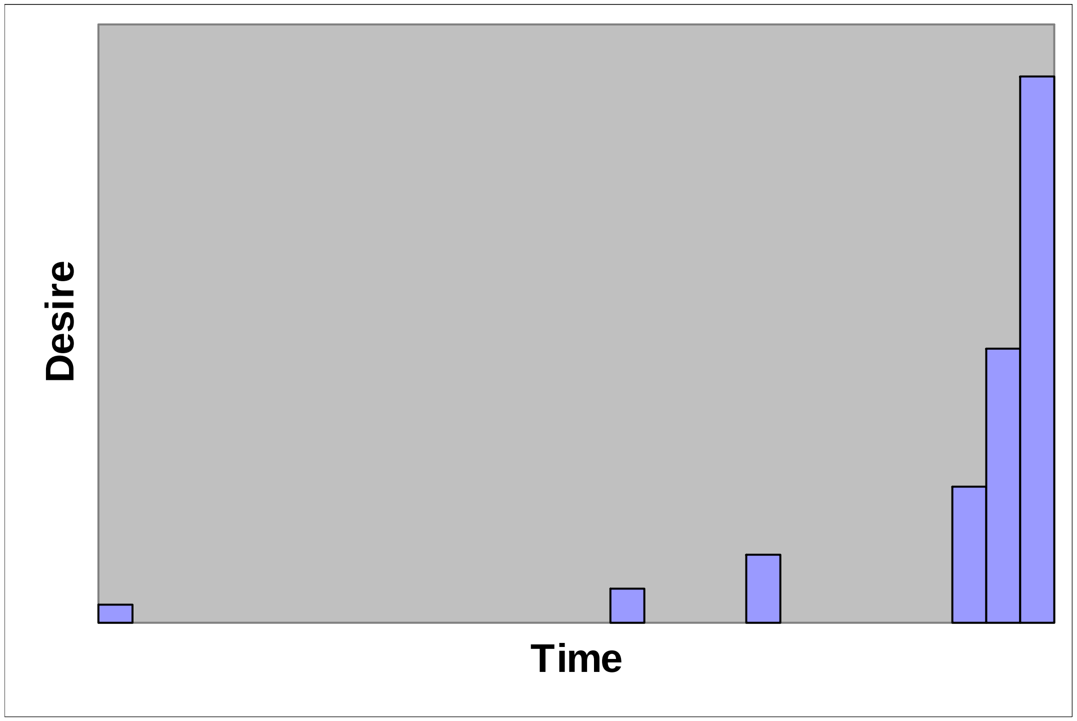
🟪	**Incomplete**
🟦	**Comp. and sat.**
⬜	**Comp. and dep.**



Complete and Satisfied Steps

- A-GREEDY doubles the desire after complete and satisfied steps, and halves the desire after incomplete steps.
- Most complete and satisfied steps are amortized against incomplete steps.

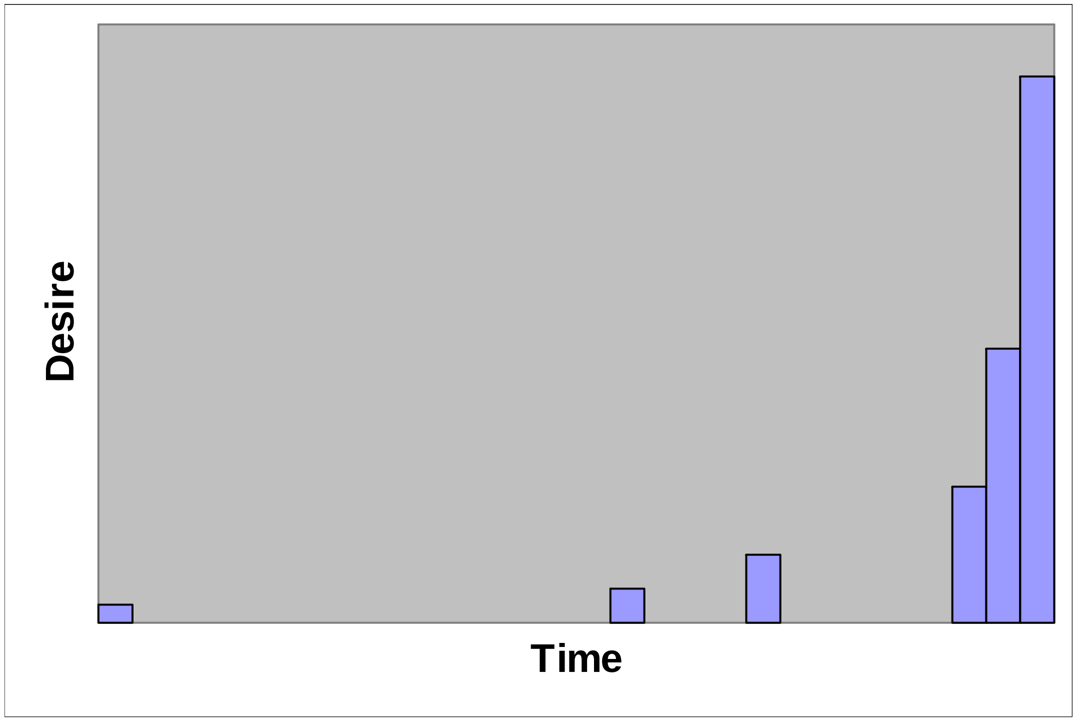
🟪	Incomplete
🟦	Comp. and sat.
⬜	Comp. and dep.



Complete and Satisfied Steps

- A-GREEDY doubles the desire after complete and satisfied steps, and halves the desire after incomplete steps.
- Most complete and satisfied steps are amortized against incomplete steps.
- LEMMA 2: If there are r incomplete steps, there are at most $r + \lg P + 1$ complete and satisfied steps.

🟥	Incomplete
🟦	Comp. and sat.
⬜	Comp. and dep.



Complete and Deprived Steps

- On deprived steps, $a_q = \min\{d_q, p_q\} < d_q \Rightarrow a_q = p_q$.
- On complete steps, $u_q = a_q$.
- Let S be the set of complete and deprived steps, and $P_S = (1/|S|) \sum_S p_q$ be the mean availability on these steps.
- The total number of tasks executed is T_1. Therefore,

$$\begin{aligned} T_1 &\ge \textstyle\sum_S u_q \\ &= \textstyle\sum_S p_q \\ &= |S|\, P_S. \end{aligned}$$

- LEMMA 3: The total number of complete and deprived steps is $|S| \le T_1/P_S$.

Special Case: $L = 1$ and $\delta = 1$

FEEDBACK$(q, 1)$

```
3. elseif $u_{q-1} < a_{q-1}$
4.     then $d_q \leftarrow d_{q-1}/2$
```

#incom. $\le T_\infty$

```
5. elseif $a_{q-1} = d_{q-1}$
6.     then $d_q \leftarrow 2d_{q-1}$
```

#com&sat. $\le$ #incom. $+ \lg P + 1$

$\le T_\infty + \lg P + 1$

```
7. else $d_q \leftarrow d_{q-1}$
```

#com&dep. $= |S| \le T_1/P_S$

Special Case: $L = 1$ and $\delta = 1$

FEEDBACK $(q, 1)$

```
3. elseif $u_{q-1} < a_{q-1}$
4.     then $d_q \leftarrow d_{q-1}/2$
```

#incom. $\leq T_\infty$

```
5. elseif $a_{q-1} = d_{q-1}$
6.     then $d_q \leftarrow 2d_{q-1}$
```

#com&sat. $\leq$ #incom. $+ \lg P + 1$

$\leq T_\infty + \lg P + 1$

```
7. else $d_q \leftarrow d_{q-1}$
```

#com&dep. $= |S| \leq T_1/P_S$

Completion time $T \leq T_1/P_S + 2T_\infty + \lg P + 1$.

Special Case: $L = 1$ and $\delta = 1$

FEEDBACK$(q, 1)$

```
3. elseif $u_{q-1} < a_{q-1}$
4.    then $d_q \leftarrow d_{q-1}/2$
```

#incom. $\leq T_\infty$

```
5. elseif $a_{q-1} = d_{q-1}$
6.    then $d_q \leftarrow 2d_{q-1}$
```

#com&sat. $\leq$ #incom. $+ \lg P + 1$
$\leq T_\infty + \lg P + 1$

```
7. else $d_q \leftarrow d_{q-1}$
```

#com&dep. $= |S| \leq T_1/P_S$

Completion time $T \leq T_1/P_S + 2T_\infty + \lg P + 1$.

Trim incomplete and complete & satisfied steps.

Special Case: $L = 1$ and $\delta = 1$

FEEDBACK $(q, 1)$

```
3. elseif $u_{q-1} < a_{q-1}$
4.     then $d_q \leftarrow d_{q-1}/2$
```

#incom. $\leq T_\infty$

```
5. elseif $a_{q-1} = d_{q-1}$
6.     then $d_q \leftarrow 2d_{q-1}$
```

#com&sat. $\leq$ #incom. $+ \lg P + 1$
$\leq T_\infty + \lg P + 1$

```
7. else $d_q \leftarrow d_{q-1}$
```

#com&dep. $= |S| \leq T_1/P_S$

Completion time $T \leq T_1/P_S + 2T_\infty + \lg P + 1$.

Trim incomplete and complete & satisfied steps.

Set $R = 2T_\infty + \lg P + 1$.

Special Case: $L = 1$ and $\delta = 1$

FEEDBACK$(q, 1)$

```
3. elseif $u_{q-1} < a_{q-1}$
4.    then $d_q \leftarrow d_{q-1}/2$
```

#incom. $\leq T_\infty$

```
5. elseif $a_{q-1} = d_{q-1}$
6.    then $d_q \leftarrow 2d_{q-1}$
```

#com&sat. $\leq$ #incom. $+ \lg P + 1$
$\leq T_\infty + \lg P + 1$

```
7. else $d_q \leftarrow d_{q-1}$
```

#com&dep. $= |S| \leq T_1/P_S$

Completion time $T \leq T_1/P_S + 2T_\infty + \lg P + 1$.

Trim incomplete and complete & satisfied steps.

Set $R = 2T_\infty + \lg P + 1$.

Therefore, $P_S \geq P_{\text{trim}(R)}$, and $T_1/P_S \leq T_1/P_{\text{trim}(R)}$.

Special Case: $L = 1$ and $\delta = 1$

FEEDBACK $(q, 1)$

```
3. elseif $u_{q-1} < a_{q-1}$
4.     then $d_q \leftarrow d_{q-1}/2$
```

\#incom. $\leq T_\infty$

```
5. elseif $a_{q-1} = d_{q-1}$
6.     then $d_q \leftarrow 2d_{q-1}$
```

\#com&sat. $\leq$ #incom. $+ \lg P + 1$
$\leq T_\infty + \lg P + 1$

```
7. else $d_q \leftarrow d_{q-1}$
```

\#com&dep. $= |S| \leq T_1/P_S$

Completion time $T \leq T_1/P_S + 2T_\infty + \lg P + 1$.

Trim incomplete and complete & satisfied steps.

Set $R = 2T_\infty + \lg P + 1$.

Therefore, $P_S \geq P_{\text{trim}(R)}$, and $T_1/P_S \leq T_1/ P_{\text{trim}(R)}$.

Completion time $T \leq T_1/ P_{\text{trim}(R)} + 2T_\infty + \lg P + 1$.

Back to the General Case

```
FEEDBACK($q$, $\delta$)
3. elseif $u_{q-1} < L\delta a_{q-1}$
4.     then $d_q \leftarrow d_{q-1}/2$
5. elseif $a_{q-1} = d_{q-1}$
6.     then $d_q \leftarrow 2d_{q-1}$
7. else $d_q \leftarrow d_{q-1}$
```

#ineff. $\le T_\infty/(L(1-\delta))$

#eff&sat. $\le$ #ineff. $+ \lg P$

$\le T_\infty/(L(1-\delta)) + \lg P$

#eff&dep. $= |S| \le T_1/(L\,\delta\,P_S)$

Back to the General Case

Feedback (q, δ)

```
3. elseif $u_{q-1} < L\delta a_{q-1}$
4.     then $d_q \leftarrow d_{q-1}/2$
5. elseif $a_{q-1} = d_{q-1}$
6.     then $d_q \leftarrow 2d_{q-1}$
7. else $d_q \leftarrow d_{q-1}$
```

#ineff. $\le T_\infty/(L(1-\delta))$

#eff&sat. $\le$ #ineff. $+ \lg P$

$\le T_\infty/(L(1-\delta)) + \lg P$

#eff&dep. $= |S| \le T_1/(L\,\delta\,P_S)$

Trim all steps in inefficient and efficient & satisfied quanta.

Set $R = 2T_\infty/(1-\delta) + L\lg P$.

We get $P_S \ge P_{\text{trim}(R)}$ and $T_1/P_S \le T_1/P_{\text{trim}(R)}$.

Therefore, $T \le T_1/(\delta P_{\text{trim}(R)}) + 2T_\infty/(1-\delta) + L\lg P$.

Closer Look at the Completion Time

$T = O(T_1/P_{\text{trim}(R)} + T_\infty + L \lg P)$ where $R = O(T_\infty + L \lg P)$.

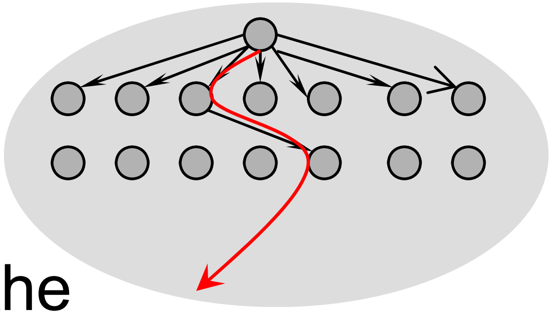
Lower Bound: $T_L > \max\{T_1/P_{\text{mean}}, T_\infty\}$

Closer Look at the Completion Time

$T = O(T_1/P_{\text{trim}(R)} + T_\infty + L \lg P)$ where $R = O(T_\infty + L \lg P)$.

Lower Bound: $T_L > \max\{T_1/P_{\text{mean}}, T_\infty\}$

Jobs with large parallelism,
$T_1/P_{\text{trim}(R)} >> T_\infty + L \lg P$.



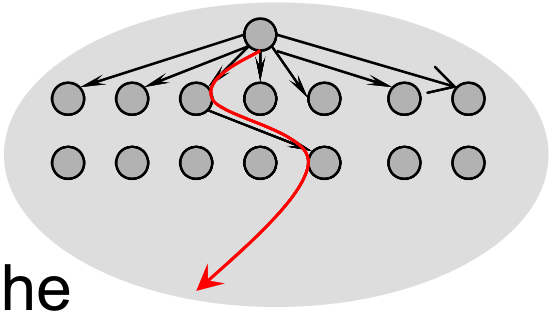
- The number of time steps trimmed is a small fraction of the total number of time steps.
- For "nice" availability profiles, $P_{\text{trim}(R)} \sim P_{\text{mean}}$. $T = O(T_1/P_{\text{mean}})$.

Closer Look at the Completion Time

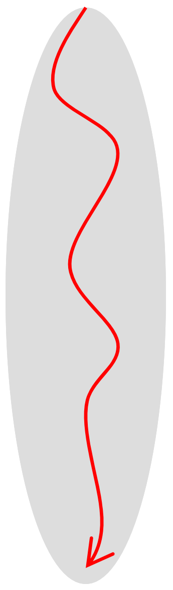
$T = O(T_1/P_{\text{trim}(R)} + T_\infty + L \lg P)$ where $R = O(T_\infty + L \lg P)$.

Lower Bound: $T_L > \max\{T_1/P_{\text{mean}}, T_\infty\}$

Jobs with large parallelism,
$T_1/P_{\text{trim}(R)} >> T_\infty + L \lg P$.



- The number of time steps trimmed is a small fraction of the total number of time steps.
- For "nice" availability profiles, $P_{\text{trim}(R)} \sim P_{\text{mean}}$. $T = O(T_1/P_{\text{mean}})$.



Jobs with small parallelism,

$T = O(T_\infty + L \lg P)$.

- Except for very short jobs, we have $T_\infty >> L \lg P$. In this case, $T = O(T_\infty)$.

A-STEAL Bounds

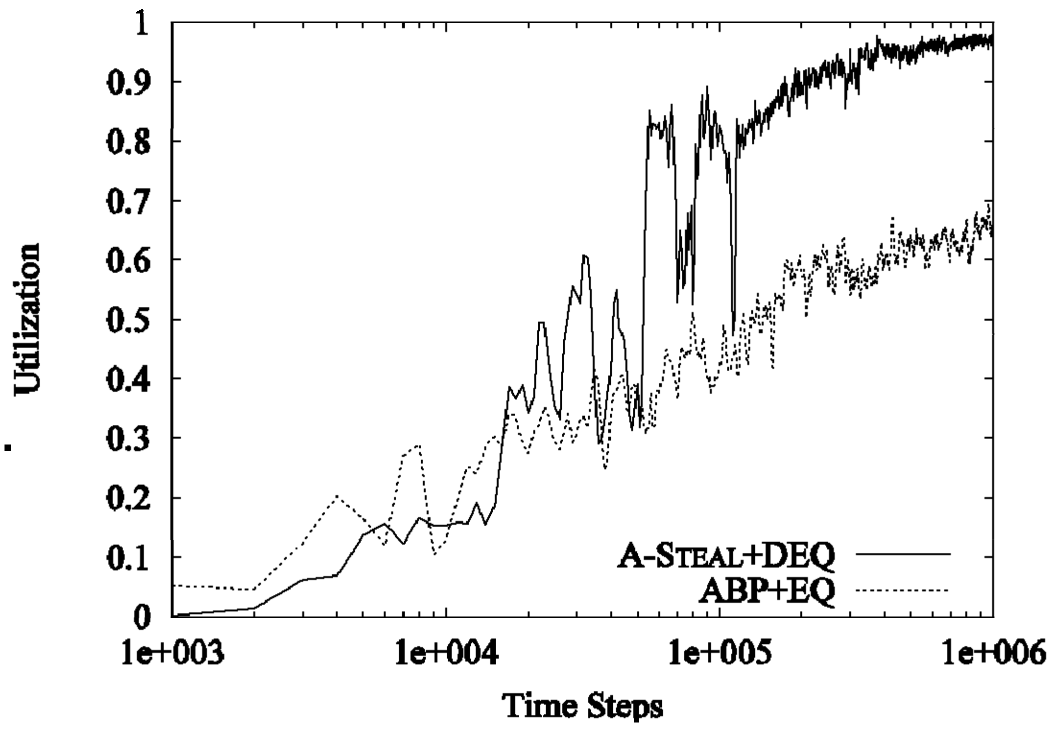
THEOREM: Consider a job with work T_1 and span T_∞ running on a P-processor machine with quantum length L and let $R = O(T_\infty + L \lg P + L \ln 1/\varepsilon)$. A-STEAL guarantees that the job

- completes the job in $O(T_1/P_{\text{trim}(R)} + T_\infty + L \lg P + L \ln 1/\varepsilon)$ time steps with probability $(1-\varepsilon)$, and
- wastes at most $O(T_1)$ processor cycles.

More Results

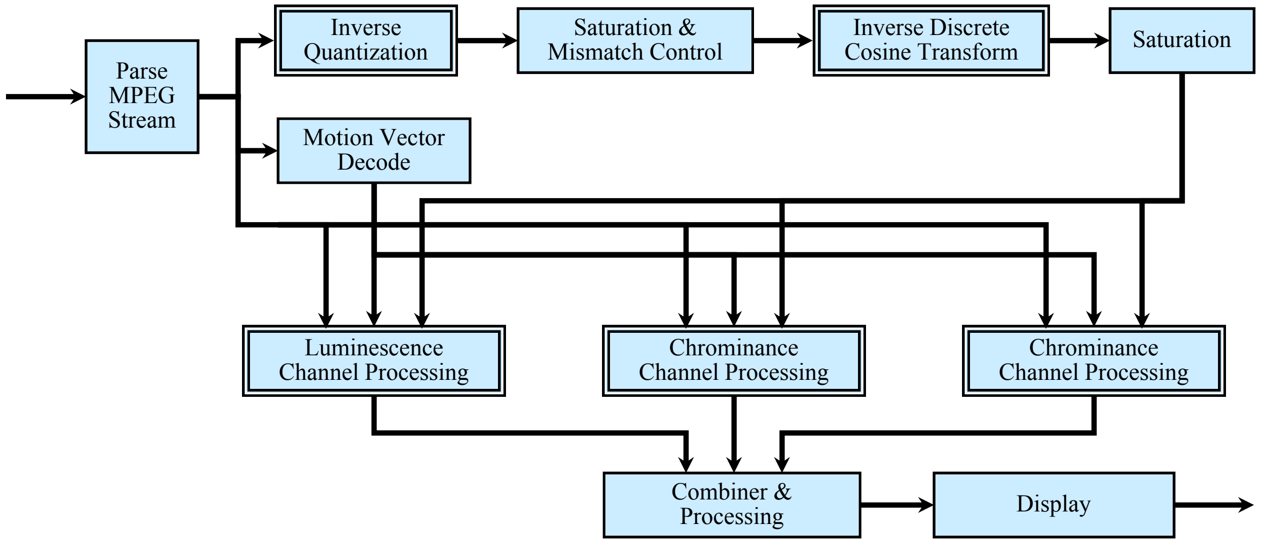
- The *job scheduler* allocates processors to individual jobs. The combination of A-GREEDY (or A-STEAL) and a job scheduler that implements *dynamic equipartitioning* (or *round-robin*) is constant-competitive with optimal scheduler with respect to both *makespan* and *mean completion time*.

- Extensive simulations show that A-STEAL should work well in practice.



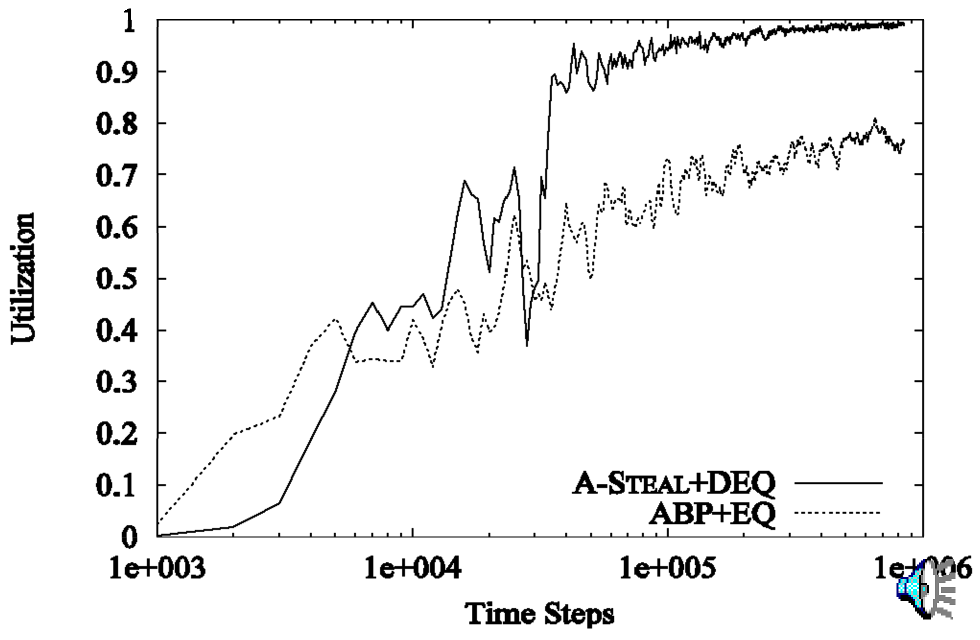
Future Work

- Does trim analysis apply in other domains?
- Is A-GREEDY or A-STEAL provably good with incomplete information about the job's utilization history?
- How do you provide parallelism feedback when the individual jobs are not independent?


Parse MPEG Stream
Inverse Quantization
Saturation & Mismatch Control
Inverse Discrete Cosine Transform
Saturation
Motion Vector Decode
Luminescence Channel Processing
Chrominance Channel Processing
Chrominance Channel Processing
Combiner & Processing
Display

Experimental Results

- We simulated a large multiprogrammed multiprocessor and used synthetic jobs to assess the performance of A-STEAL.
- A-STEAL provides nearly perfect linear speedup and wastes less than 20% of the allotted processor cycles.
- We compared the utilization provided by A-STEAL and *ABP*, a work-stealing scheduler that does not employ parallelism feedback. When many synthetic jobs share a large server, and jobs arrive and leave dynamically, A-STEAL consistently provided higher utilization than ABP for a variety of job mixes.



The Simulation Environment

The simulated large multiprogrammed multiprocessors, and the simulation environment is built using Desmo-J.

- We simulated the execution of synthetic jobs.
- The jobs are scheduled using work-stealing. On each discrete time step, a processer can complete either a work-cycle, a steal-cycle or a mug-cycle.
- We compared A-STEAL or *ABP*. ABP, presented by Arora, Blumofe and Plaxton, is an adaptive work-stealing scheduler that does not provide parallelism feedback.

- The work, critical path, and the rate of change of parallelism of the jobs can be changed by changing w_1, w_2 and h.

Job Schedulers

The *job scheduler* allocates processors to individual jobs.

- An *equipartitioning* (EQ) job scheduler simply allots the same number of processors to all the jobs in the system.
- A *dynamic equipartitioning* (DEQ) is a dynamic version of the equipartitioning job scheduler, which allots equal number of processors to all jobs with the constraint that no job gets more processors than it desires. Thus, it requires parallelism feedback.
- For *profile-based* job schedulers, we pre-computed the sequence of processor availabilities for each quantum of the job's execution using workload archives [Feitelson].

Utilization Experiment

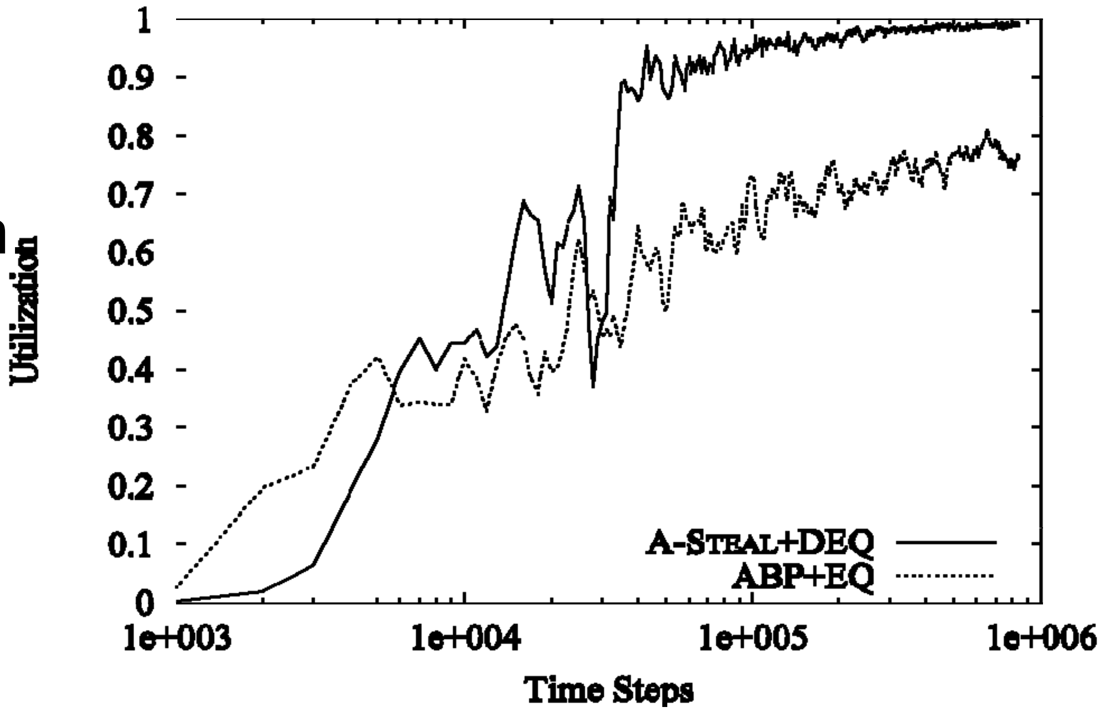
Comparing the utilization provided by A-Steal and ABP on a simulated 1000 processor server, where jobs enter dynamically with mean inter-arrival time of 1000 time steps.

We considered 9 sets of jobs with the three distributions on each parallelism and critical path.

- Uniform distribution
- Heavy tailed I:

  Pr{*x*} ~1/*x*.
- Heavy tailed II:

  Pr{*x*} ~ 1/√*x*.

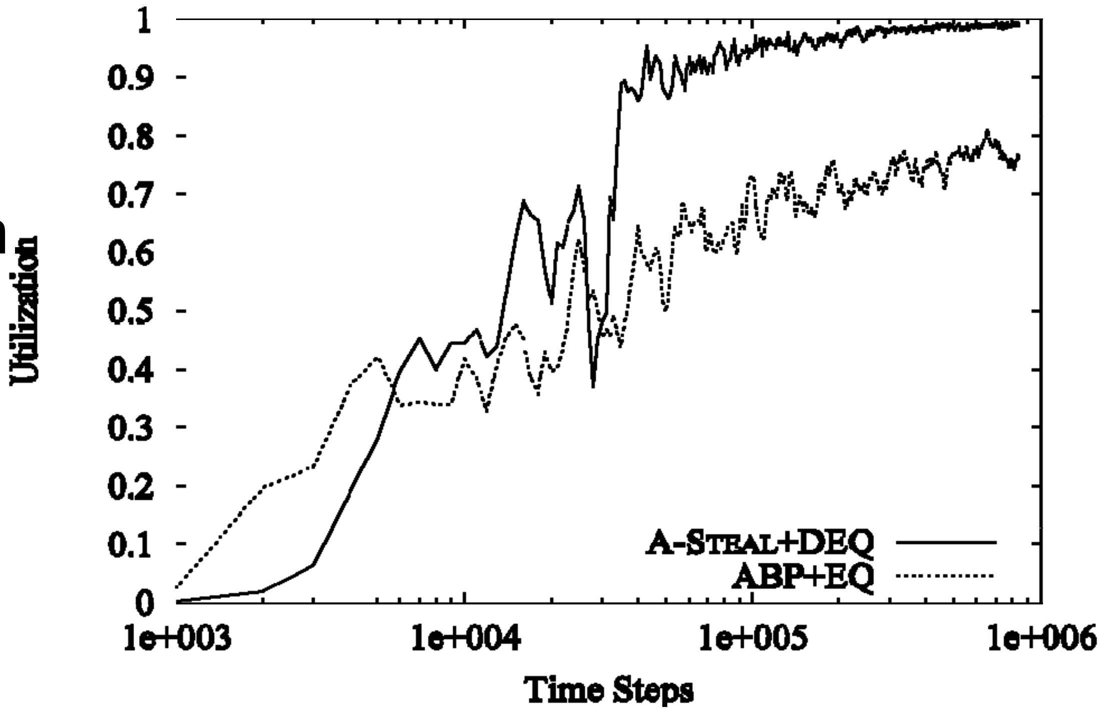


Utilization Experiment

Comparing the utilization provided by A-Steal and ABP on a simulated 1000 processor server, where jobs enter dynamically with mean inter-arrival time of 1000 time steps.

We considered 9 sets of jobs with the three distributions on each parallelism and critical path.

- Uniform distribution
- Heavy tailed I: Pr{*x*} ~1/*x*.
- Heavy tailed II: Pr{*x*} ~ 1/√*x*.

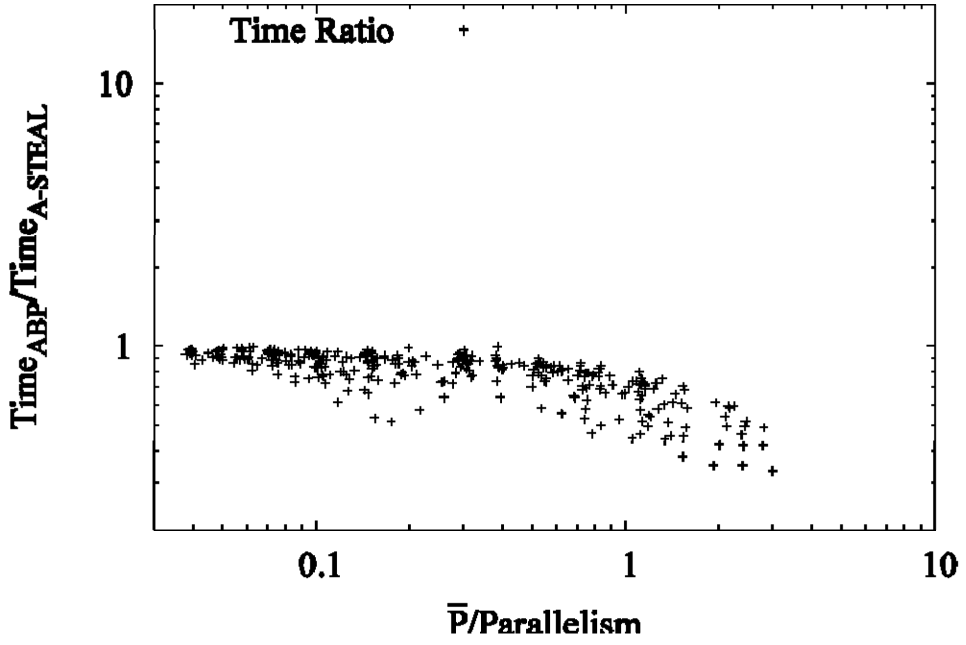


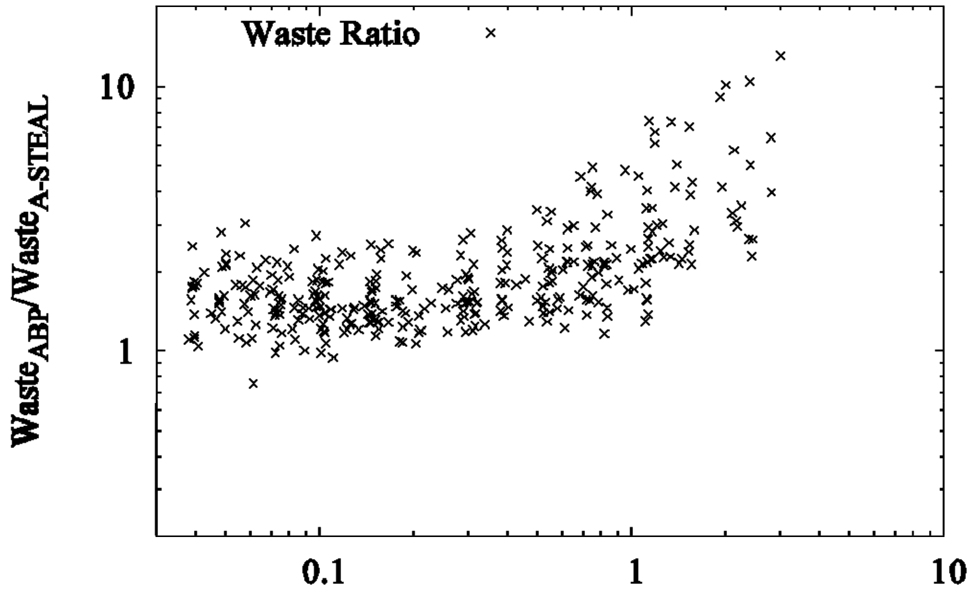
A-STEAL consistently provides higher utilization. The mean completion time of the jobs is about 50% faster using A-STEAL

Time-Waste Experiments I

Comparing the completion time and waste on a simulated $P = 128$ processor machine using predetermined availability profiles with mean availability $\overline{P} = 30,60$.

- A-STEAL wastes fewer processor cycles, since it uses parallelism feedback to control excess allotment.
- But A-STEAL completes the jobs slightly slower.



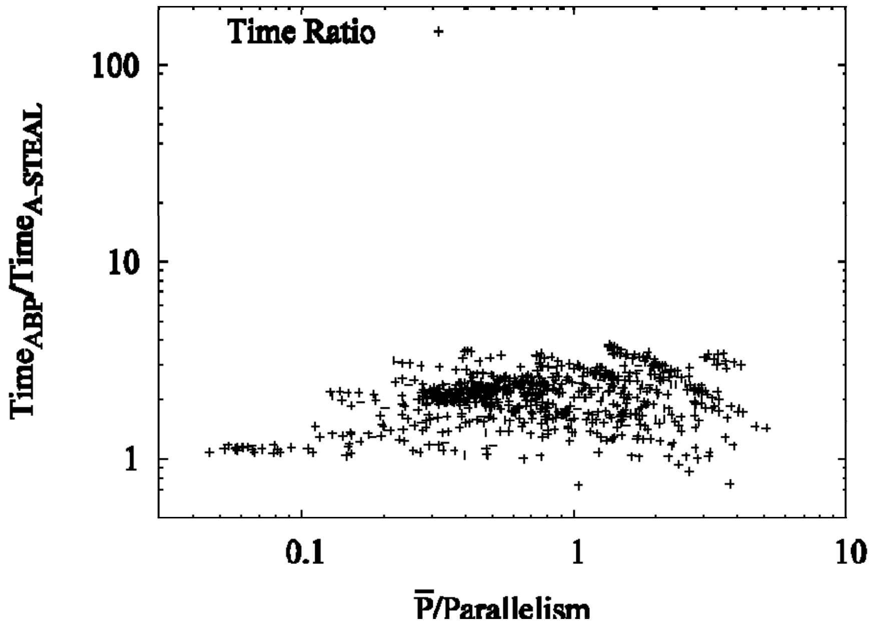


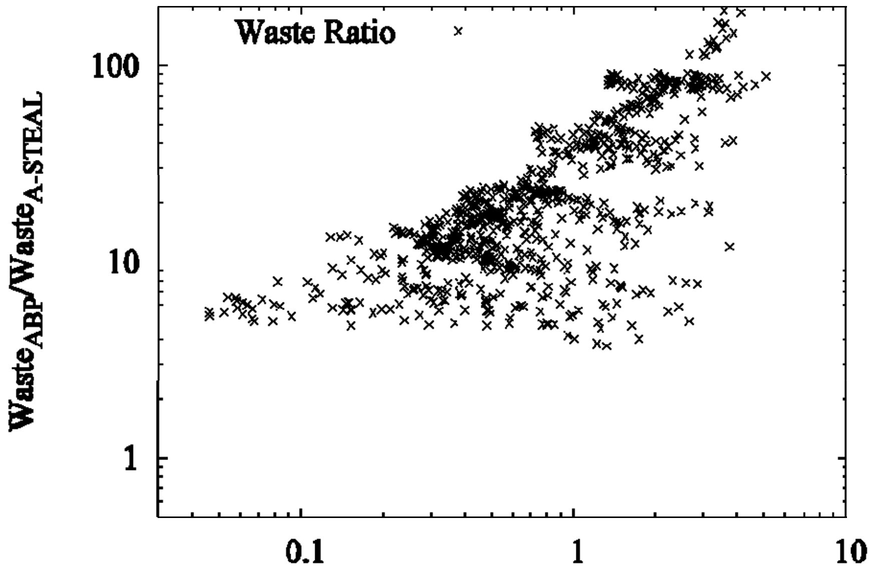
Time-Waste Experiment II

This experiment is similar to the previous one, except that we ran the jobs on a larger ($P = 512$ processor) machine.

- Again, A-STEAL wastes fewer processor cycles.
- Paradoxically, in this case A-STEAL also completes faster.

A-STEAL may be a better option on heavily loaded large machines where each job gets a small fraction of the total processors.


Time Ratio
$\text{Time}_{\text{ABP}}/\text{Time}_{\text{A-STEAL}}$
$\overline{P}$/Parallelism


Waste Ratio
×
$\text{Waste}_{\text{ABP}}/\text{Waste}_{\text{A-STEAL}}$