

Oracle Scheduling: Controlling Granularity in Implicitly Parallel Languages

Arthur Charguéraud

with Umut Acar and Mike Rainey

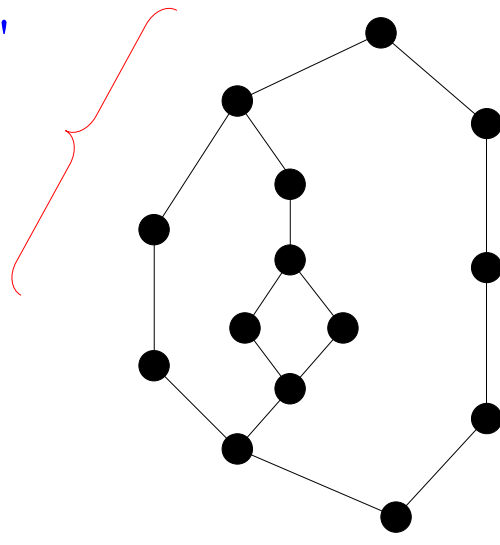
Quick summary (1/2)

Parallelism is expressed through parallel tuples

(| t1, t2 |)

Computations are represented using series-parallel DAGs

W : "work"
= nb nodes



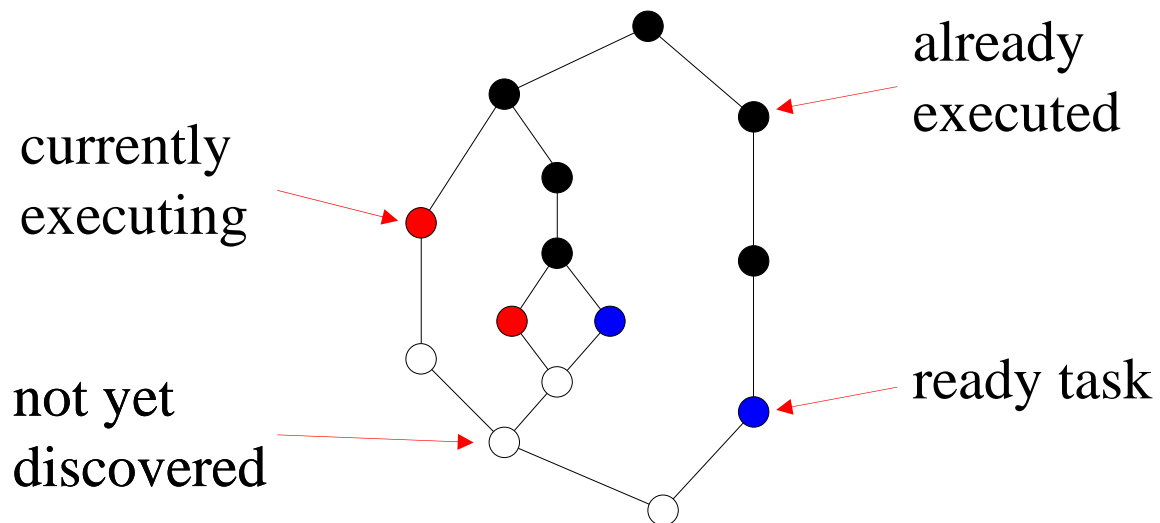
S : "span"
= length of
critical path

Average parallelism: $\bar{P} = W/S$ (equal to the maximum speedup)

Parallelism is plenty when $\bar{P} \gg P$ (i.e. "parallel slackness")

Quick summary (2/2)

A **ready task** is a task that is waiting to be executed



A **greedy scheduler** never leaves a processor idle if there is a ready task

Brent's theorem: any greedy scheduler achieves

$$T_P \leq W/P + S$$

where T_P is the execution time on P processors

Plan of this talk



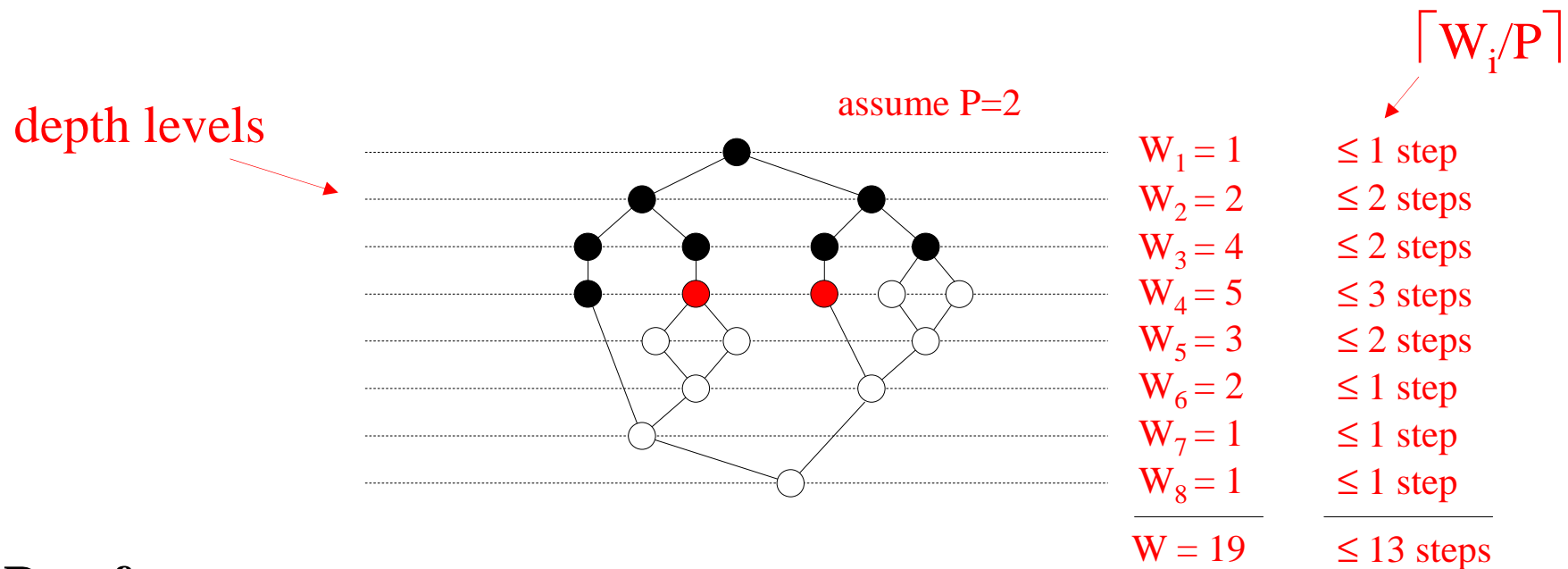
- **Brent's theorem and generalization**
- **Granularity control using an oracle**
- **Implementation**
- **Results**

BFS achieves Brent's theorem

Consider a breadth-first scheduling policy for executing tasks

Let us prove that this policy achieves Brent's bound: $T_P \leq W/P + S$

→ lay out the SP-dag according to the depth of tasks from the root



Proof:

$$T_P \leq \sum_{i=1}^S \left\lceil \frac{W_i}{P} \right\rceil \leq \sum_{i=1}^S \left(\frac{W_i}{P} + 1 \right) \leq \frac{\sum_{i=1}^S W_i}{P} + S \leq \frac{W}{P} + S$$

Limitation of unit costs

In the model used so far, every task has a unit cost

In reality, tasks may have very different costs:

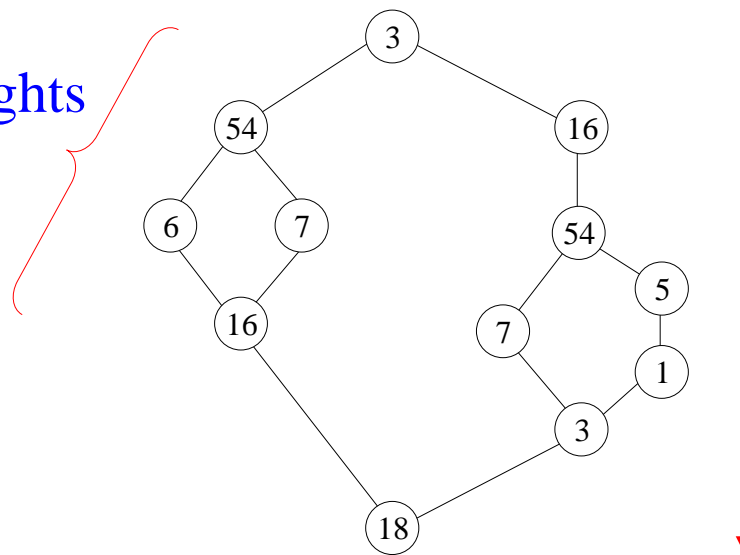
basic operation	1 cycle
memory read, cache hit	10 cycles
memory read, cache miss	100 cycles
function call	100 cycles
fork/join operation	500 cycles
cost of a steal	> 1000 cycles

Note: it is not correct to encode a task of size 2 as a sequential composition of two unit tasks, because "preemption" is not allowed: the scheduler cannot suspend the task in the middle of its execution

Weighted SP-dags

We thus generalize the model to have a cost for every task:

W : "work"
= sum of the weights



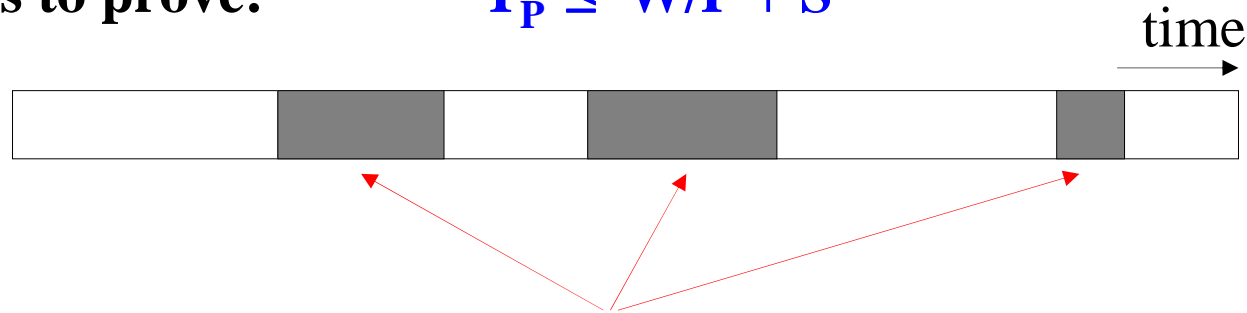
S : "span"
= weight of the
heaviest path

Question: does Brent's theorem still hold in this generalized model?

Proof of the generalized Brent's theorem

The goal is to prove:

$$T_p \leq W/P + S$$



Divide the execution time: $T_p = T^{\text{full}} + T^{\text{partial}}$ where

- T^{full} : the amount of time during which all processors are busy
- T^{partial} : the time during which at least one processor is idle

*) During time T^{full} exactly P processors are working, thus executing a total of $P \cdot T^{\text{full}}$ work, which cannot exceed the total work W . Thus,

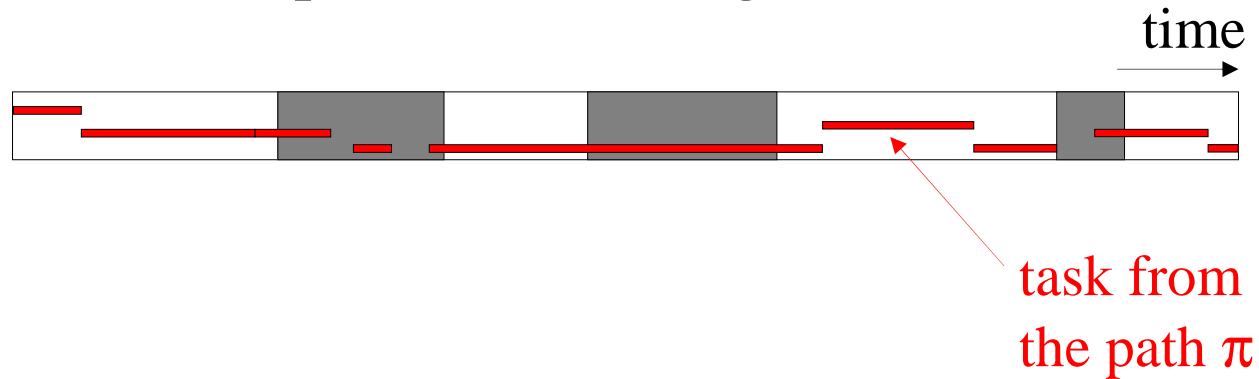
$$T^{\text{full}} \leq W/P$$

*) So, there remains to establish

$$T^{\text{partial}} \leq S$$

Proving the inequality on T^{partial}

Idea of the proof: find a path π in the SP-dag such that covers T^{partial}
→ in the sense that, for any time at which not all the processors are busy, a task from the path π is executing



Once we find such a path, we can conclude as follows:

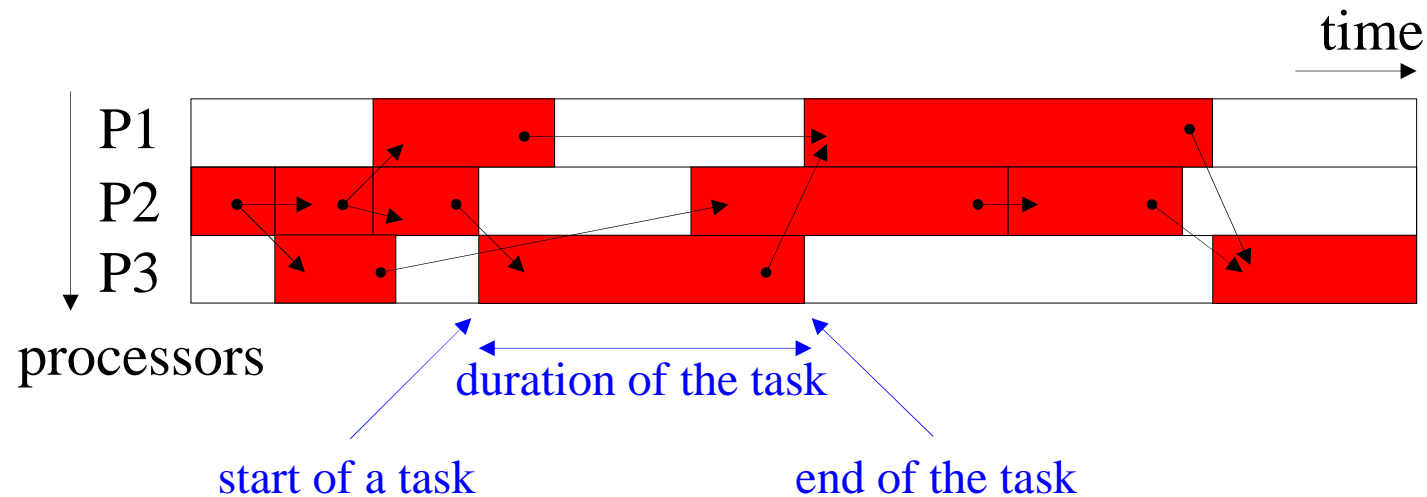
$$\begin{aligned} T^{\text{partial}} &\leq \text{weight of the path } \pi \\ &\leq \text{maximum weight of a path} \\ &= S \end{aligned}$$

Graham diagrams

A Graham diagram can be used to represent a scheduling of tasks

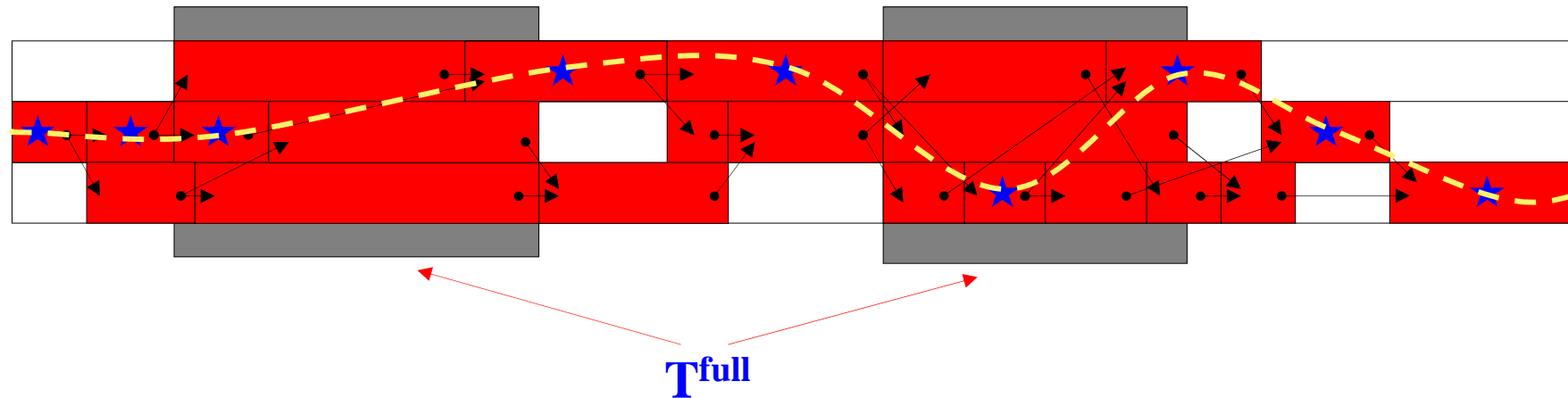
→ blocks indicate scheduling of a task on a given processor

→ arrows indicate task dependencies: they go from left to right



Construction of a path covering T^{partial}

Construct the path π that covers T^{partial} backwards from the end



- *) Task starting in a full-activity period (gray background)
 - must depend on another task immediately preceding it
- *) Task starting in a partial-activity period (white background)
 - must depend on a task executing at the beginning

Summary

For a SP-dag with work **W** and span **S**, any greedy scheduler achieves:

$$\begin{aligned} T_P &= T^{\text{full}} + T^{\text{partial}} \\ &\leq W/P + \text{weight of a particular path } \pi \\ &\leq W/P + \text{maximum weight of a path} \\ &\leq W/P + S \end{aligned}$$

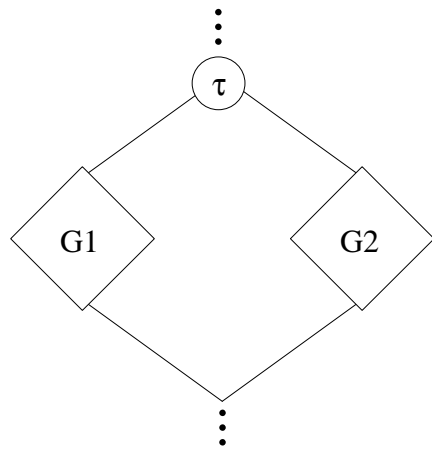


- **Brent's theorem and generalization**
- **Granularity control using an oracle**
- **Implementation**
- **Results**

Scheduling decisions

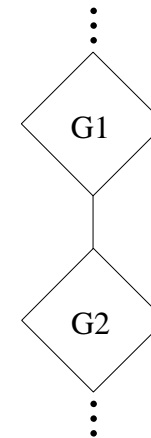
On a parallel tuple (| t_1, t_2 |), the scheduler has two options:

1) Create two parallel tasks, execute the fork and the join
→ this induces an extra cost "scheduling cost" τ



$$W = W_1 + W_2 + \tau$$
$$S = \max(S_1, S_2) + \tau$$

2) Turn the tuple into a sequential tuple (t_1, t_2)
→ this reduces the amount of parallelism available

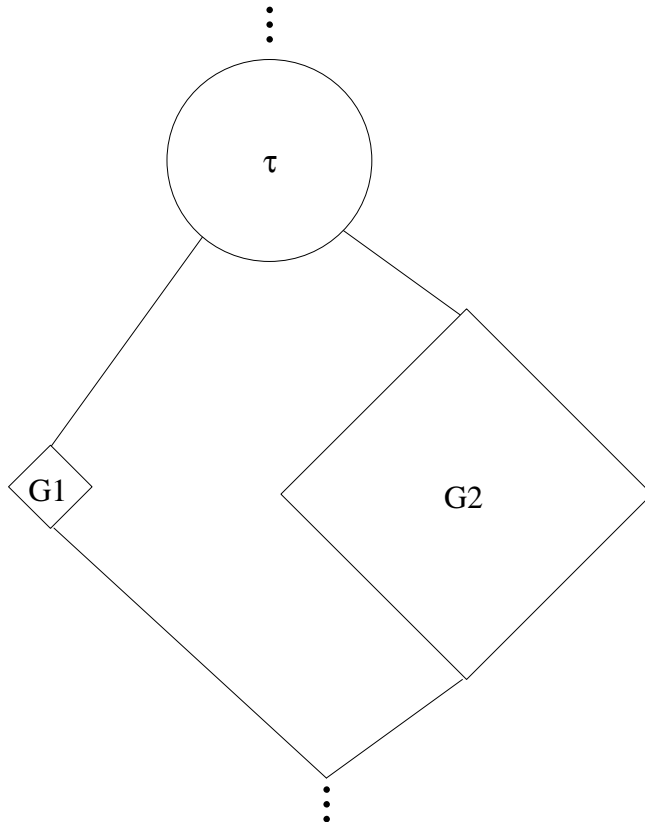


$$W = W_1 + W_2$$
$$S = S_1 + S_2$$

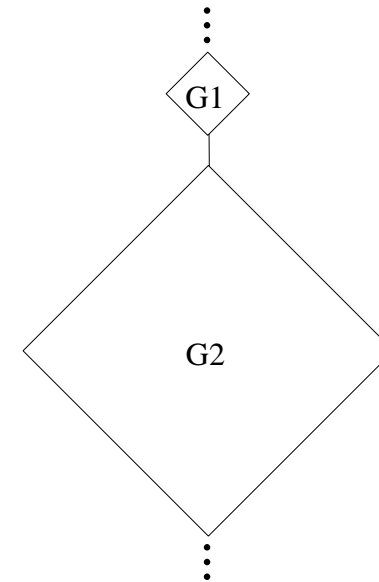
Case of a small branch

If one of the two branches of the parallel tuple (| t_1 , t_2 |) involves less than τ work, then it is clearly better to sequentialize

Run in parallel:



Run in sequence:

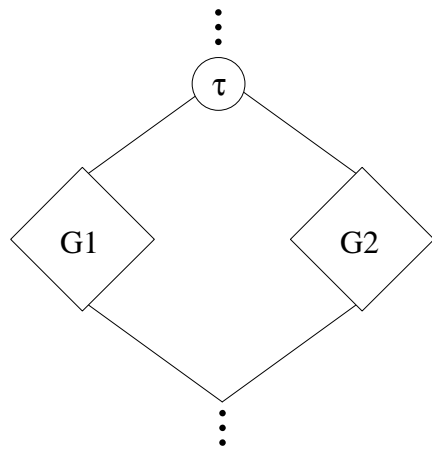


smaller work and
smaller span

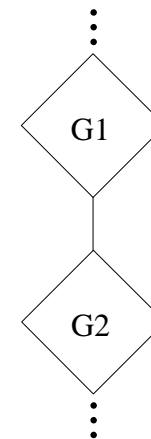
Scheduling policy for granularity control

On a parallel tuple (| t_1, t_2 |), distinguish two cases:

If both tasks involve more than κ work, schedule a parallel tuple



If one branch involves less than κ work, schedule a sequential tuple



where κ is some constant greater than τ

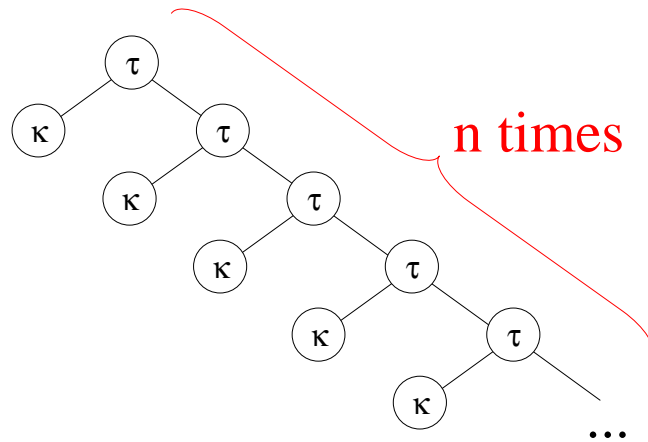
Question 1: how much can the span increase, at most?

Question 2: can we get a bound on the total time spent on scheduling?

Question 3: what is an appropriate choice for the value of κ ?

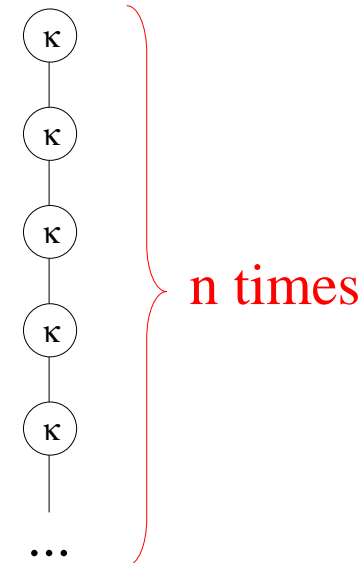
Maximum increase in depth

Original SP-dag:



$$S \approx n \tau$$

SP-dag produced by the policy:

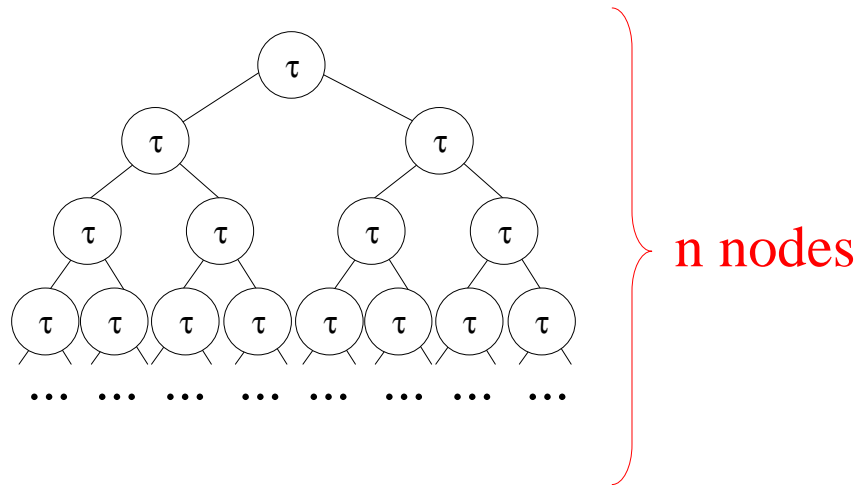


$$S \approx n \kappa$$

In general, the span can be multiplied by up to a factor κ / τ

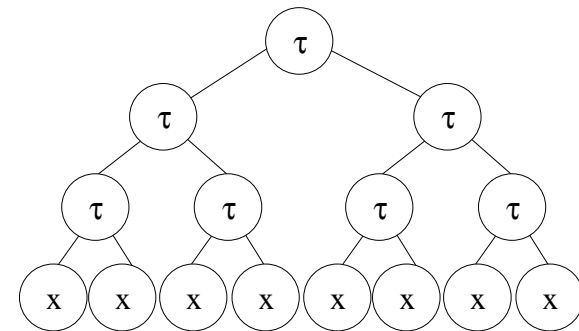
Typical increase in depth

Original SP-dag:



$$S = n \tau$$

SP-dag produced by the policy:



only the leaves are sequentialized;
they have work x , with $\kappa \leq x \leq 2\kappa$

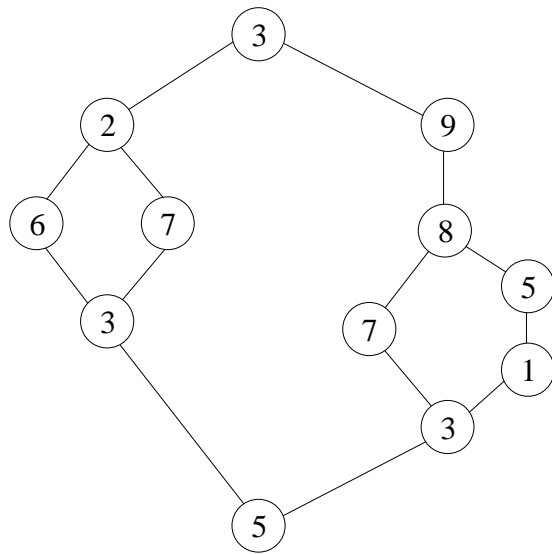
$$S \leq n \tau + 2\kappa$$

In relatively-balanced computations, the span typically only augments by a constant (instead of being multiplied by a constant)

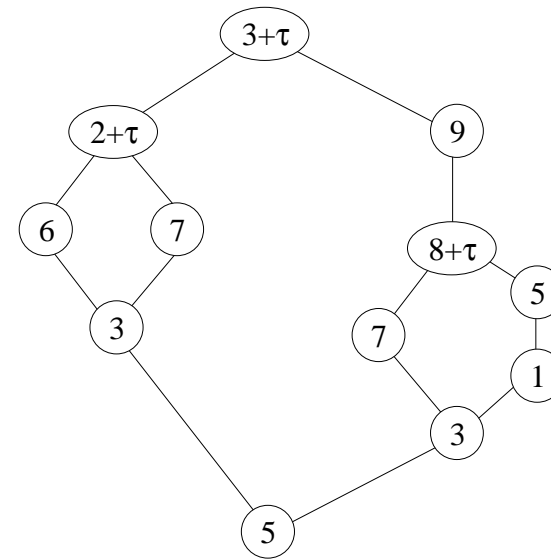
Raw work/ span v.s. total work/span

To investigate the total scheduling costs, we define two versions of work and span: one version without scheduling costs and one version with

w : raw work
 s : raw span



W : total work
 S : total span

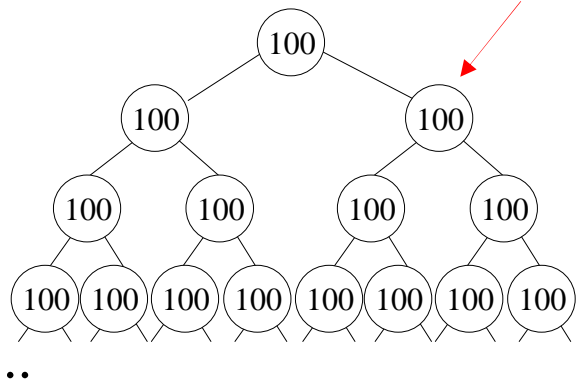


In the example: $W = w + 3\tau$, and $S = s + 2\tau$

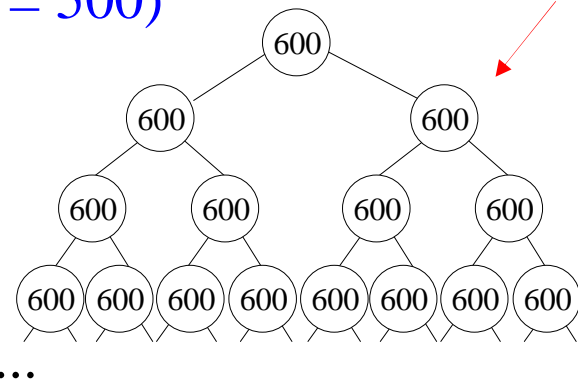
Maximum increase in total work

For a divide-and-conquer algorithm with recursive calls in parallel, if we do not control granularity, then we typically have the following:

Raw costs:



Total costs:
($\tau = 500$)



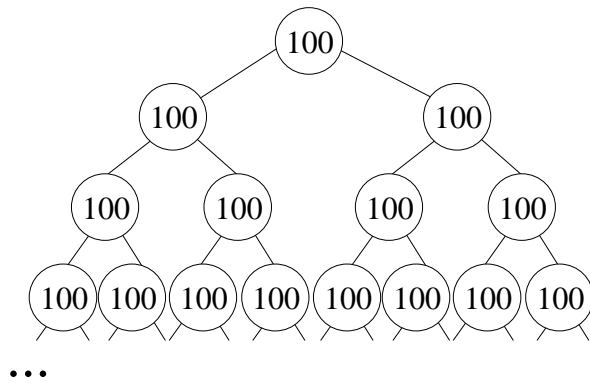
(note: we will only draw the top half of the SP-dag from now on)

Total costs is multiplied by 6; this means that we need at least 6 processors in order to possibly outperform a run by a single processor!

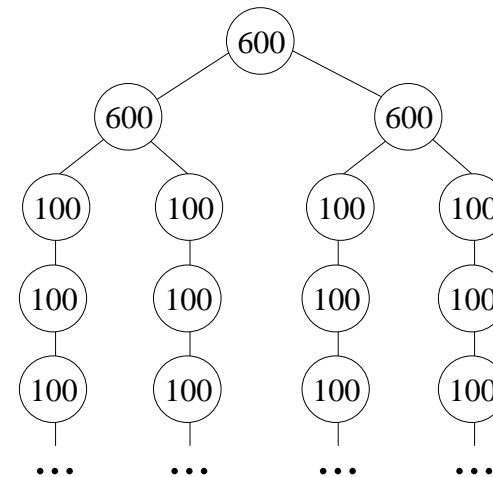
Effect of granularity control

When leaves become small enough, they get executed sequentially

Raw costs:



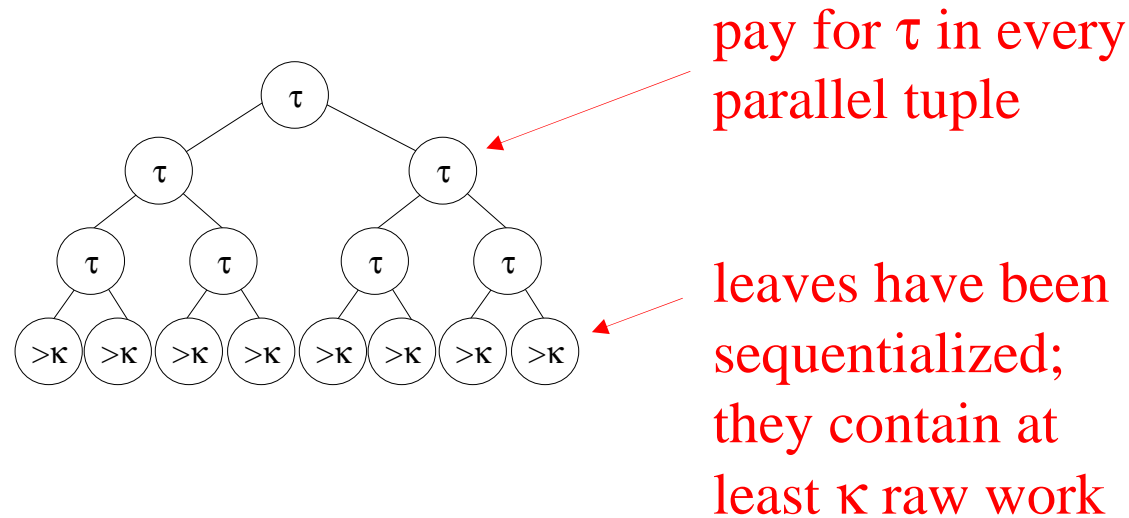
Total costs: (for $\tau = 500$)



With granularity control, we only pay for τ on the upper nodes

Bound in the scheduling costs: example

Shape of a typical SP-dag after our scheduling policy has been applied:



We pay for τ on every node

The number of nodes is roughly equal to the number of leaves

Since leaves involve at least κ work, there are at most w/κ leaves

So the total scheduling cost does not exceed $\tau \cdot w/\kappa$

So, $W \leq (1 + \tau/\kappa) \cdot w$ (the cost τ gets amortized on leaves of size κ)

→ If we take $\kappa = 20 \cdot \tau$ then the scheduling overhead is less than 5%.

Bound in the scheduling costs: formally

For any SP-dag, we can prove by induction the following inequality:

$$W \leq w + \tau \left\lfloor \frac{(w - \kappa)^+}{\kappa + 1} \right\rfloor$$

where x^+ is equal to x if $x > 0$ or 0 otherwise

maximal number of leaves in a SP-dag of raw work w where sub-dags of less than κ work have been sequentialized

We can then derive the same bound as on the previous slide:

$$W \leq w + \tau \cdot \frac{w}{\kappa} \leq \left(1 + \frac{\tau}{\kappa}\right) \cdot w$$



- **Brent's theorem and generalization**
- **Granularity control using an oracle**
- **Implementation**
- **Results**

Implementation

Question 1: how to estimate the sequential running time of a task?

→ combine complexity annotations and runtime profiling

Question 2: how to produce code implementing the scheduling policy?

→ perform a source-to-source translation

Question 3: how to pick an appropriate value for κ ?

→ measure the value of τ for the target machine

Question 4: how to extend the theory to model the cost of the oracle?

→ the theory generalizes under a reasonable assumption

Complexity annotations

We require the programmer to annotate every function with an expression that computes the asymptotic complexity of the raw work

```
let qsort t =  
  costs (let n = size t in n * log n);  
  let p = first t  
  let (t1,t2,t3) = partition p t  
  let (u1,u3) = (| qsort t1, qsort t3 |)  
  Node (u1, Node (t2, u3))
```

Basic translation

```
let qsort_cnst =  
  ref (.. value of the constant ..)
```

assume for now that we
know the constant

```
let qsort_cost t =  
  let n = size t in n * log n
```

the cost function

```
let qsort t =  
  let p = first t  
  let (t1,t2,t3) = partition p t  
  let (u1,u3) =  
    let size1 = !qsort_cnst * qsort_cost t1  
    let size2 = !qsort_cnst * qsort_cost t2  
    if (size1 > κ) and (size2 > κ)  
      then (| qsort t1, qsort t3 |)  
      else ( qsort t1, qsort t3 )  
  Node (u1, Node (t2, u3))
```

task size prediction

scheduling condition

Measuring sequential runs

```
let qsort t =  
  ...
```

if a branch is small, then it
should get executed sequentially

```
let size1 = !qsort_cnst * qsort_cost t1
```

```
let branch1 =
```

```
  if (size1 > κ)
```

```
    then (fun () -> qsort t1)
```

```
    else (fun () ->
```

```
      let x = time()
```

```
      qsort t1
```

```
      let y = time()
```

```
      report qsort_cnst size1 (y-x))
```

for sequential run, measure
the execution time, and
then report the measure to
update the constant

```
let size2 = ...
```

```
let branch2 = ...
```




```
if (size1 > κ) and (size2 > κ)
```


```
  then ( | branch1(), branch2() | )
```

```
  else ( branch1(), branch2() )
```

execute the branches

Optimization of the code for sequential run

```
let qsort_par t =  definition of the parallel version
  ...
  let size1 = !qsort_cnst * qsort_cost t1
  let branch1 =
    if size1 > κ
    then (fun () -> qsort_par t1)  for a big task, call the
    parallel version
    else (fun () ->
      let x = time()
      qsort_seq t1  for a small task, call the
      sequential version
      let y = time()
      report qsort_cnst size1 (y-x))
  ...

let qsort_seq t =  definition of the sequential version
  let p = first t
  let (t1,t2,t3) = partition p t
  let (u1,u3) = (qsort_seq t1, qsort_seq t3)
  Node (u1, Node (t2, u3))
```

Implementation of constant estimators

Naive idea #1: use one shared data structure to store the constant

Problem: high cost due to frequent write in a shared memory cell

Naive idea #2: use one different constant per processor

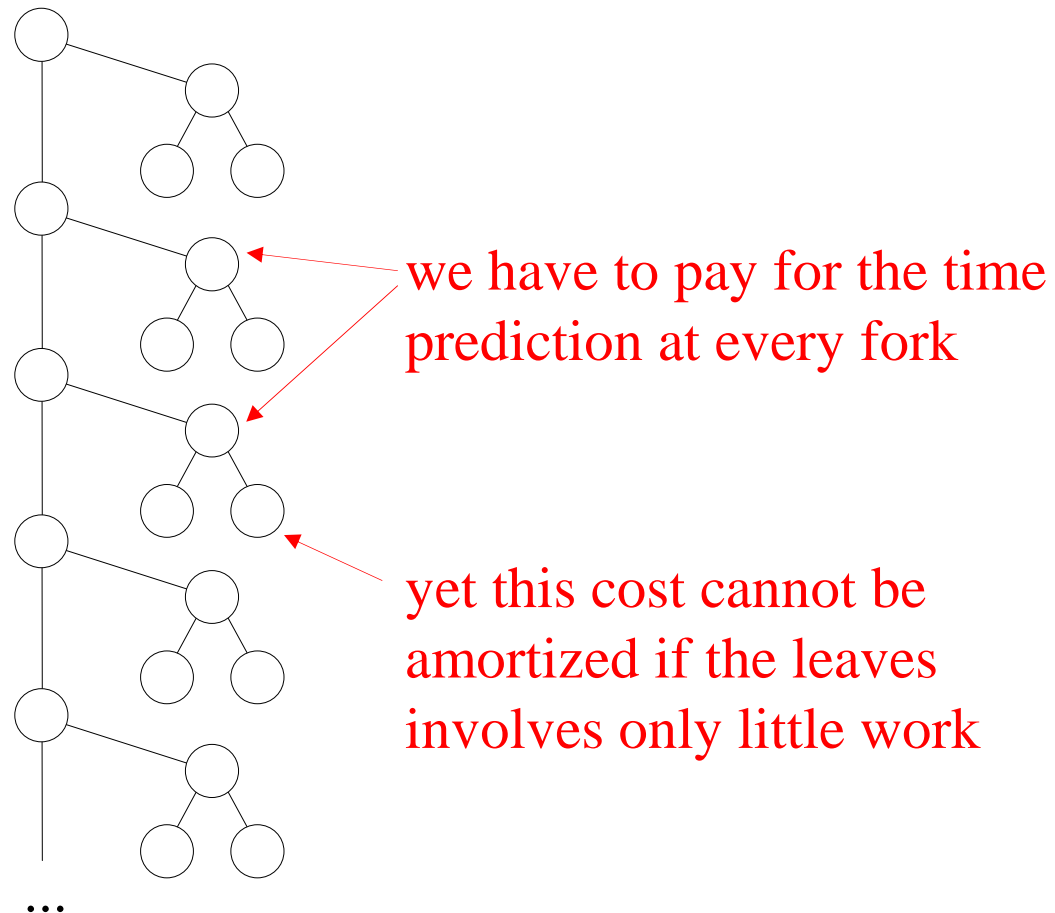
Problem: takes a lot more time to get accurate predictions

Our approach:

- one shared memory cell where every processor read the constant
- one local structure for each processor accumulating some statistics
- every $10 \cdot P$ measures, the processor reports its value to the main cell
- "reporting" consists in updating the cell with a weighted average

Example where time predictions are costly

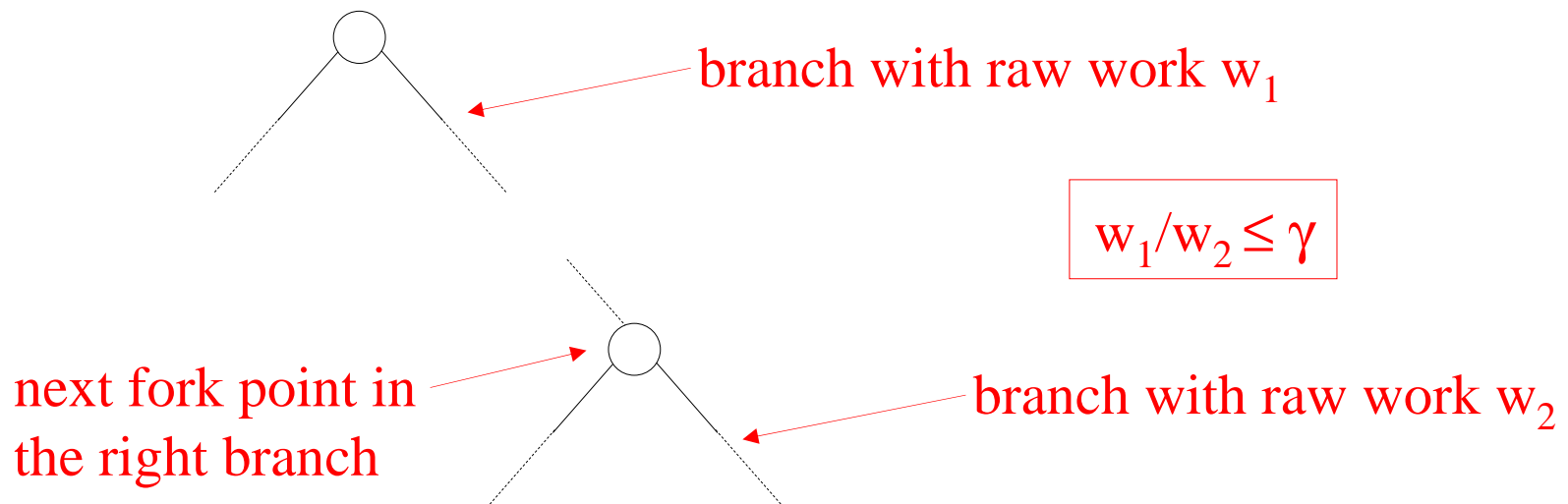
Most of the exec. time could get spent on computing time predictions...



We need to make the assumption that the work does not decrease too much between two successive time predictions

Regularity in parallel programs

We want to ensure that the raw work does not decrease by more than a constant factor between two successive time predictions

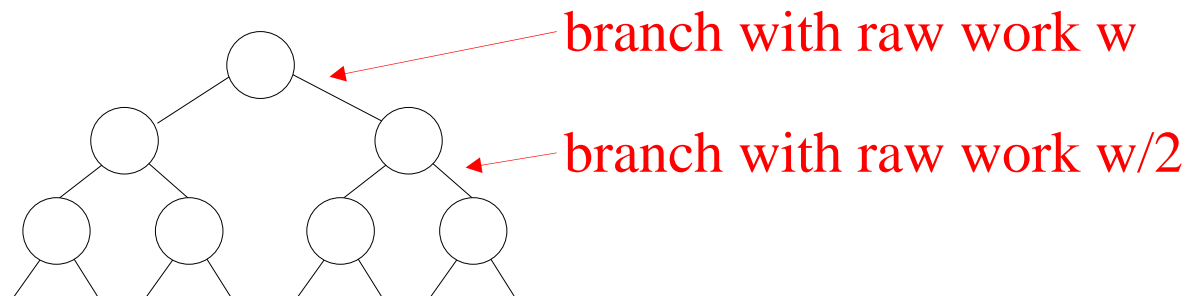


Formally:

A program is **γ -regular** if for any two branches of work w_1 and w_2 , one immediately nested in the other, we have either $w_2 \geq w_1 / \gamma$ or $w_2 \gg \kappa$

Example of regularity

*) **Regularity of balanced binary trees is 2:**



*) **Regularity of typical $n \log n$ divide and conquer is also about 2:**

$$(2n \log 2n) / (n \log n) = 2 * (\log n + 1) / (\log n) = 2 * (1 + 1 / \log n) \approx 2$$

Note: to amortize the scheduling costs, we actually only require that programs be regular on average, so we can tolerate some imbalance

Formalizing the cost and errors of the oracle

Assume that:

- a time prediction and a time measure induce a constant cost ϕ
- the time predictions are correct up to a multiplicative factor μ
- recursive calls make the work decrease by no more than a factor γ
(i.e. the program is γ -regular)

Then:

- span increases at most by a factor $\kappa \cdot \mu + \phi$
- every τ cost gets amortized on at least κ/μ work
- every ϕ cost gets amortized on at least $\kappa/(\mu\gamma)$ work

Our grand theorem:

$$T_P \leq \left(1 + \frac{\tau}{\kappa/\mu} + \frac{\phi}{\kappa/(\mu\gamma)} \right) \frac{w}{P} + (1 + \kappa\mu + \phi) d$$

Choice for the cutoff

We want to ensure that scheduling costs do not exceed 5%

To that end, it suffices to pick the smallest κ satisfying:

$$\frac{\tau}{\kappa/\mu} + \frac{\phi}{\kappa/(\mu\gamma)} \leq 5\%$$

Thus, we define:

$$\kappa = \frac{\tau\mu + \phi\mu\gamma}{5\%}$$

Example of concrete figures:

$\tau = 90$ nano-seconds

$\mu = 2$

$\phi = 180$ nano-seconds

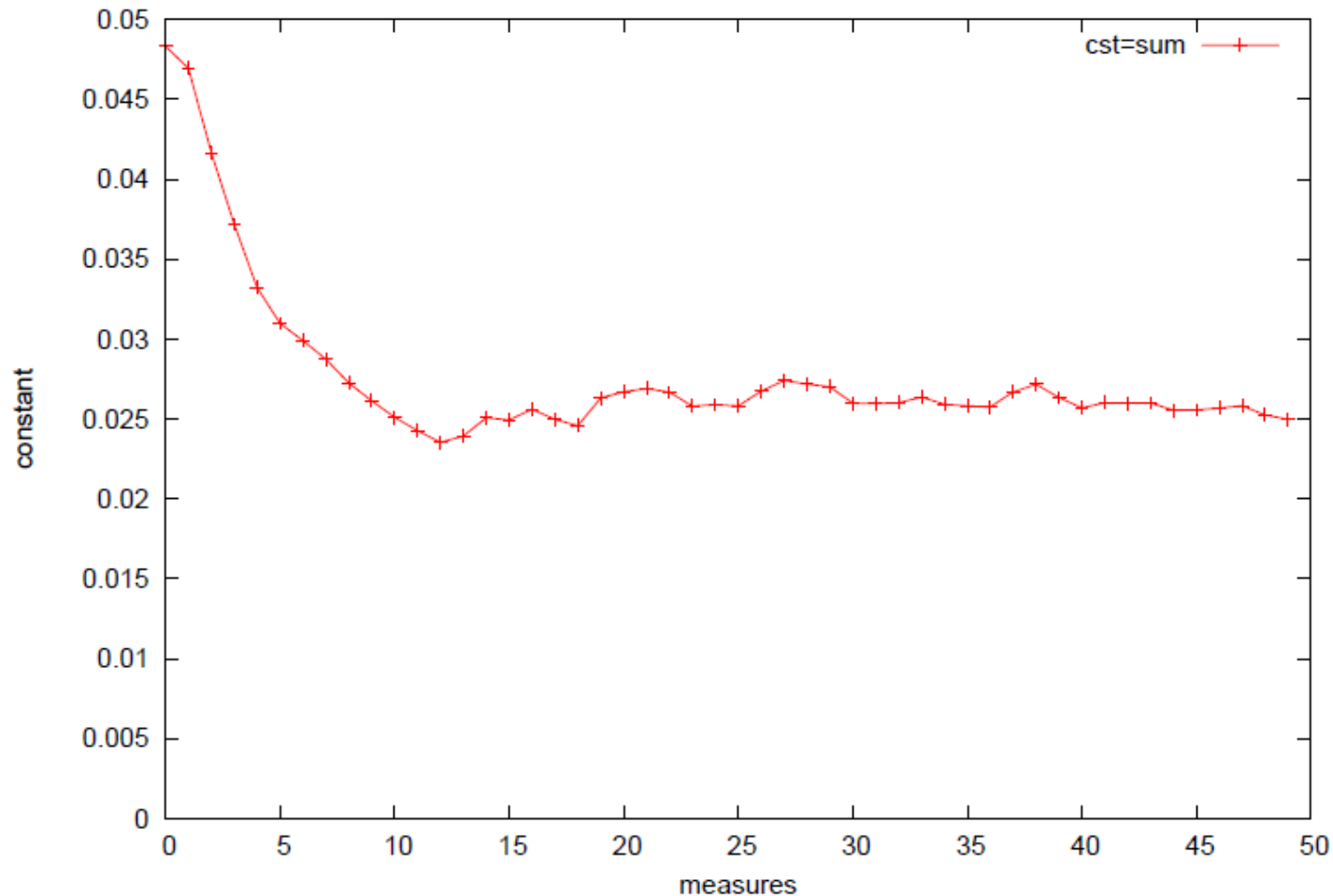
$\gamma = 3$

→ $\kappa = 25\,000$ nano-seconds (0.025 milli-seconds)

- **Brent's theorem and generalization**
- **Granularity control using an oracle**
- **Implementation**
- **Results**

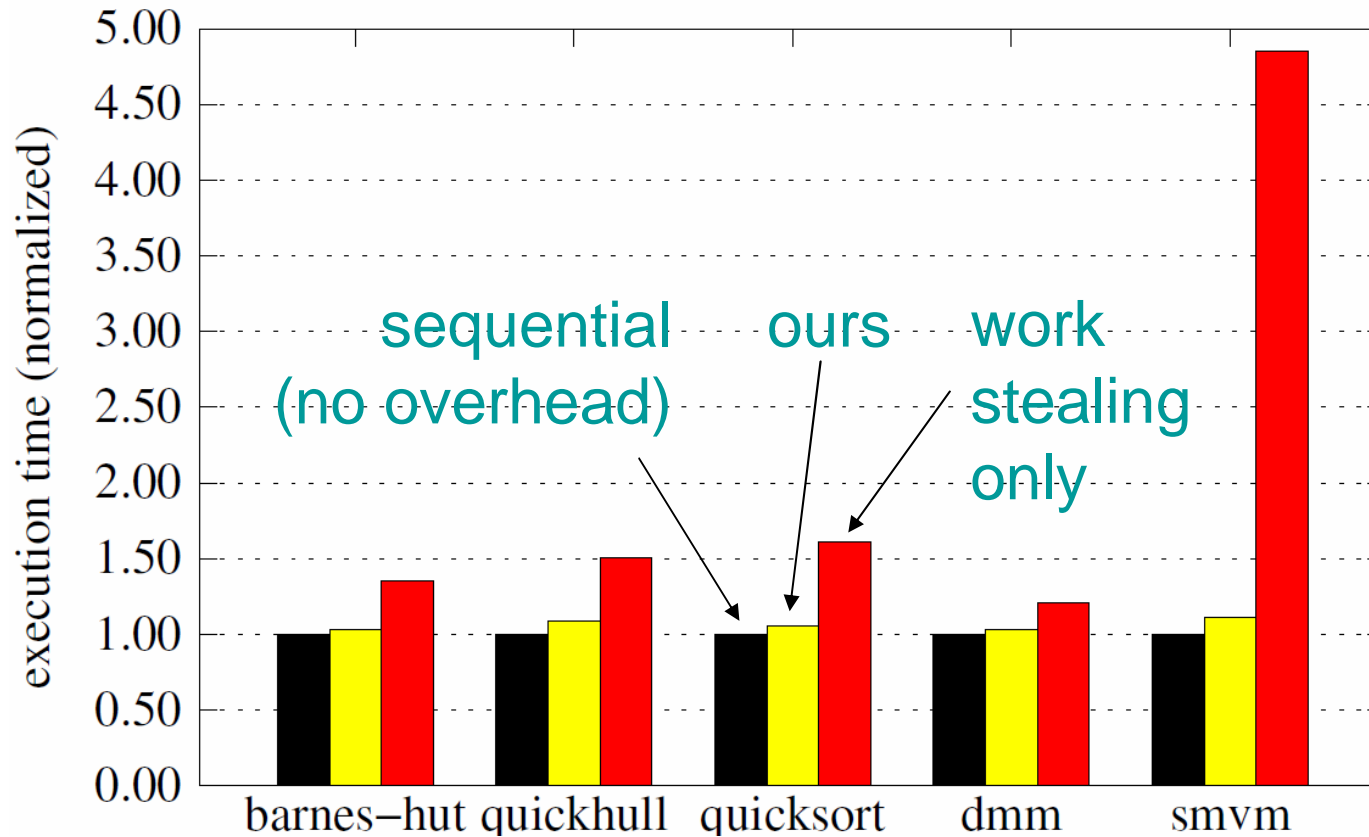


Convergence of the measure of constants



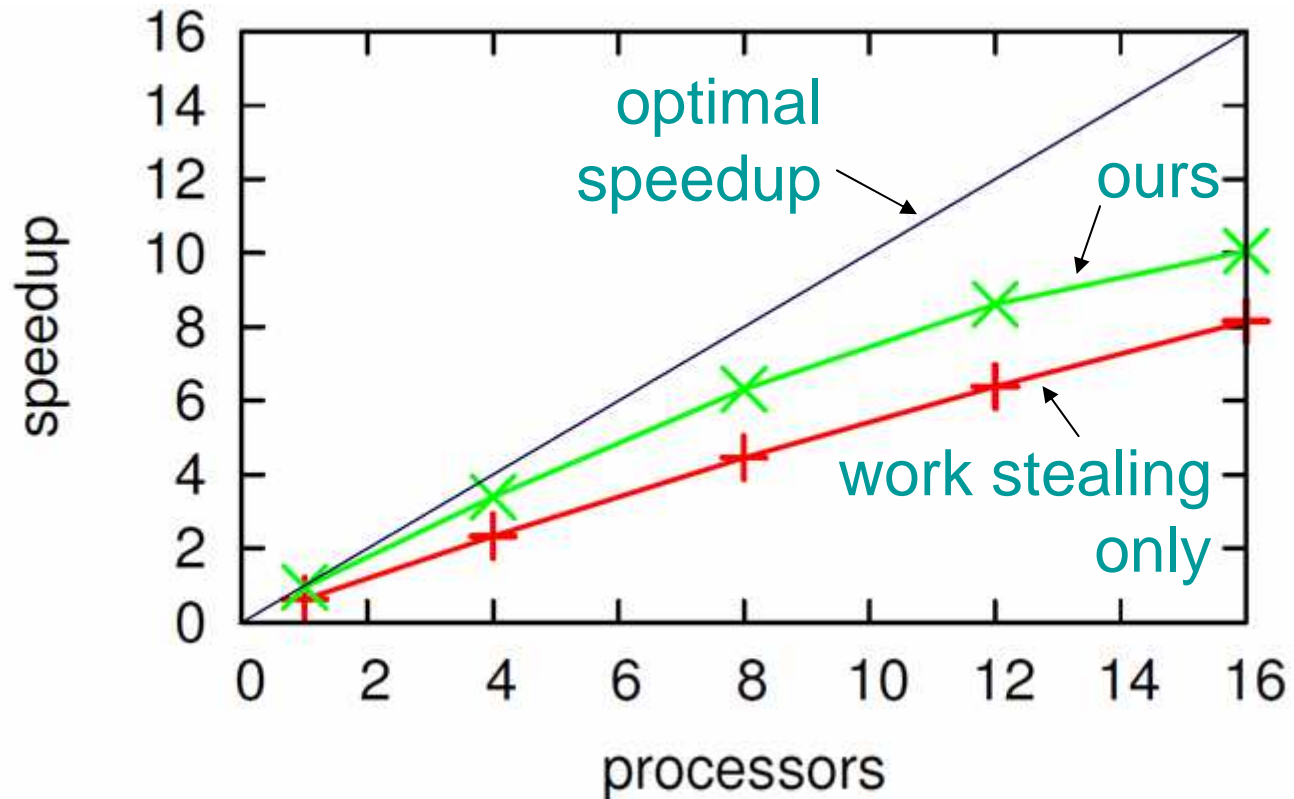
After a few measures, the constant converges to a pretty accurate value

Execution time on a single processor



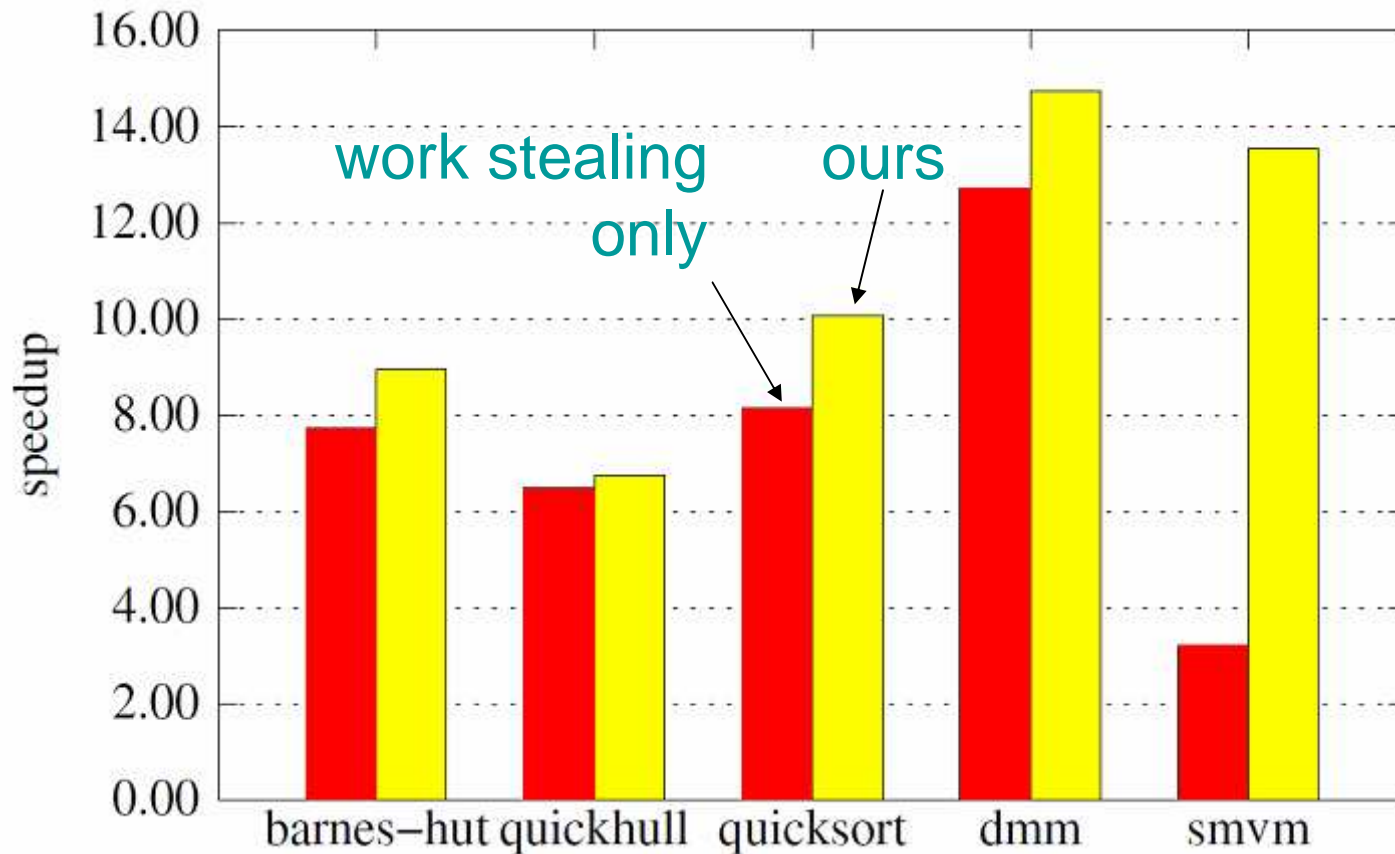
With oracle scheduling, overheads do not exceed 13% of the sequential execution time, whereas with work stealing it could be as much as 380%

Speedup curve for quicksort



Oracle scheduling appears to scale up with the number of processors as well as work stealing, yet with a better constant factor

Speedup on 16 processors



By reducing the task creation overheads, oracle scheduling improves over work stealing and achieves 6x to 15x speedups on 16 processors

Conclusion

- 1) Using asymptotic complexity annotations and profiling, we are able to find out whether a task is big or small**
- 2) By sequentializing small tasks, we ensure that we amortize the cost of the forks without increasing the span too much**
- 3) Combining this with the generalization of Brent's theorem, we get a provably good method that can be used on top of any greedy scheduler**

→ For further information, read our paper:

Oracle Scheduling: Controlling Granularity in Implicitly Parallel Languages
Umut A. Acar, Arthur Charguéraud and Mike Rainey
OOPSLA, April 2011