Oracle Scheduling: Controlling Granularity in Implicitly Parallel Languages

Umut Acar – **Arthur Charguéraud** – Mike Rainey

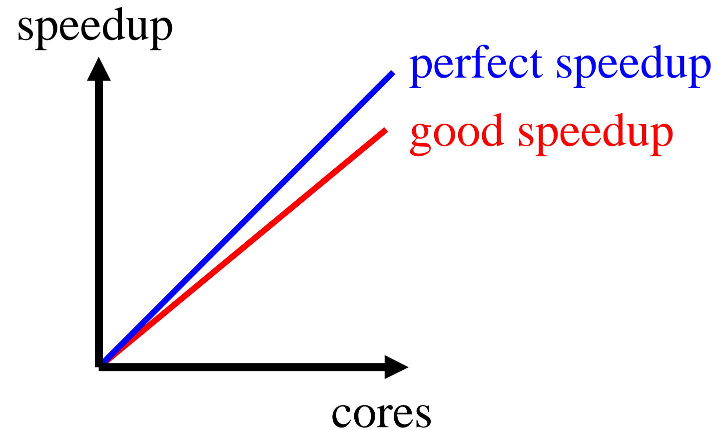
Max Planck Institute for Software Systems

OOPSLA

Portland, 2011/10/26

Speedups with multicores

Goal: get good speedups from using several cores

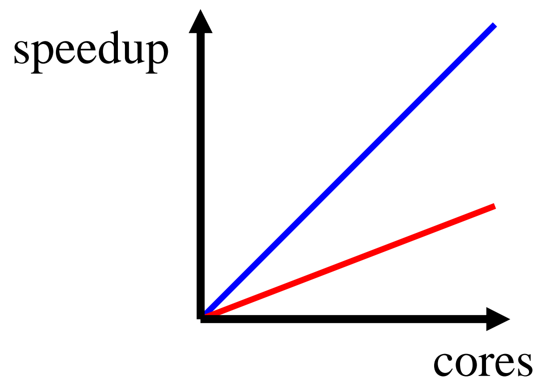


Obstacles: lack of parallelism, memory wall, scheduling overheads

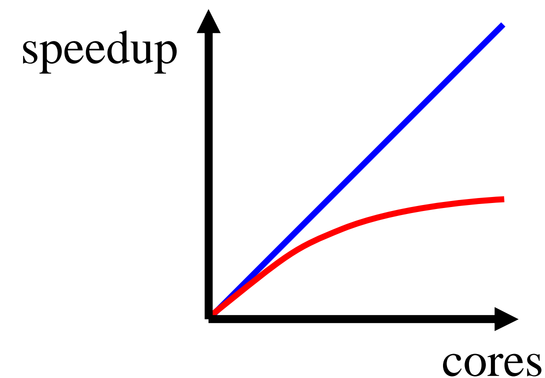
Granularity control

Scheduling overheads: they mainly depend on the number of tasks

Too many tasks:
→ large overheads



Not enough tasks:
→ limited parallelism



Granularity control: problem of finding the right size for parallel tasks

→ we propose a new approach to granularity control
based on asymptotic complexity annotations

Importance of granularity control

Sequential code:

```
int fibseq(int n) {
   if (n < 2) return 1;
   int a = fibseq(n-1);
   int b = fibseq(n-2);
   return a+b;
}
```

Parallel code:

```
int fibpar(int n) {
   if (n < 2) return 1;
   spawn int a = fibpar(n-1);
   int b = fibpar(n-2);
   sync;
   return a+b;
}
```

compute fibonnaci(45)

10 seconds on a single core

20 seconds on 42 cores

→ **1.8 billion parallel tasks created**

→ **per-task overhead of a few dozens memory accesses**

Introduction of a cutoff value

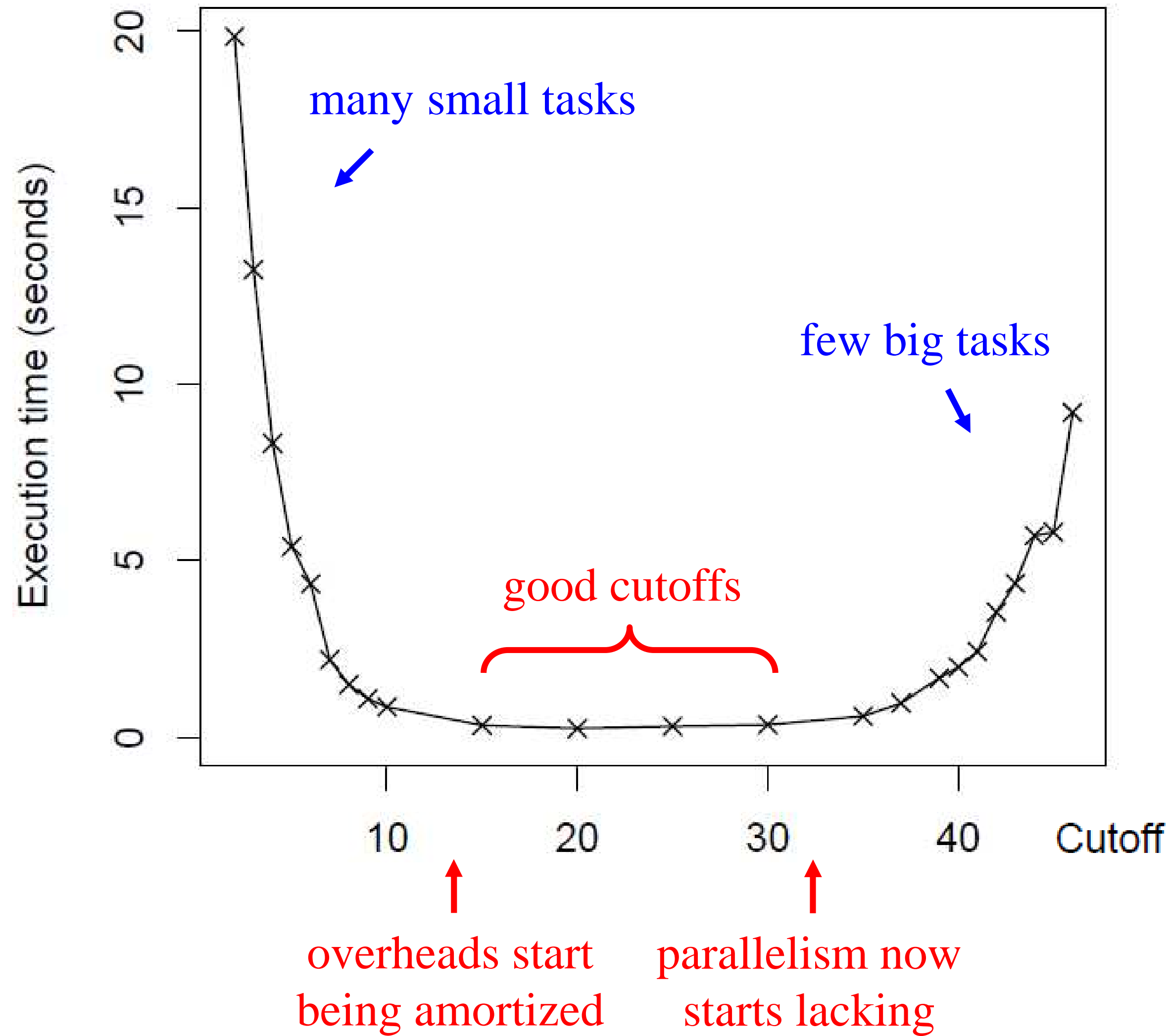
Parallel code with cutoff value:

```
int fibcut(int n) {
   if (n < cutoff)
     return fibseq(n)
   spawn int a = fibcut (n-1);
   int b = fibcut(n-2);
   sync;
   return a+b;
}
```

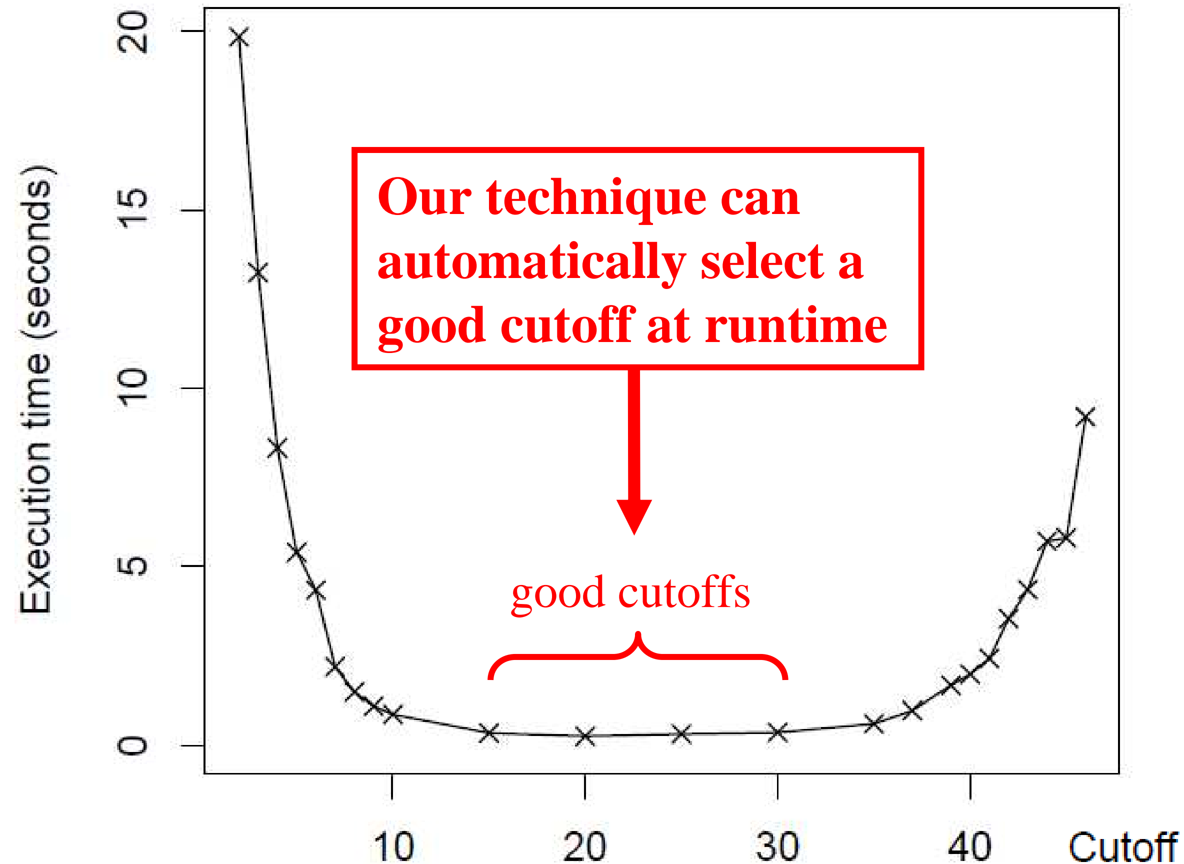
→ **What is the right value to use as cutoff?**

Execution time vs cutoff

Running fibpar(45) on 42 cores, using a work-stealing scheduler



Selection of the cutoff value



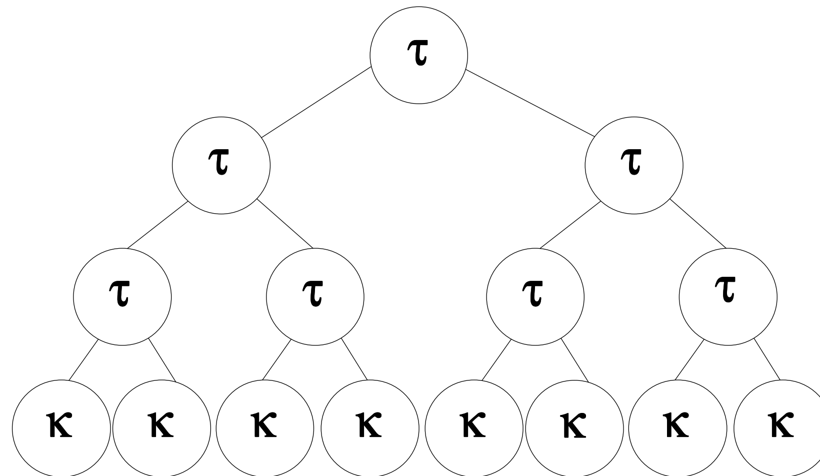
– **hard-coding a cutoff → non portable code**

– **trying all cutoffs (auto-tuning) → requires a tuning process**

Amortizing task creation overheads

Idea: **Assume that every fork costs τ. If the cutoff value leads to tasks of size $\kappa \approx 100 \cdot \tau$, then the overheads are approximately equal to 1%.**

Policy: **tasks predicted to take less than κ time are not parallelized**



Remark: κ depends on τ, which depends on the hardware and the scheduler, but not on the algorithm, contrary to auto-tuning

Theory

Brent's theorem: (task creation overheads completely ignored)

$$T_P \leq \frac{T_1}{P} + T_\infty$$

neglectable when lot
of parallel available

Our theorem: (fork operation overhead = τ, sequentialize if exec time < κ)

$$T_P \leq \left(1 + \frac{\tau}{\kappa}\right) \cdot \frac{T_1}{P} + \kappa \cdot T_\infty$$

we chose κ such
that τ/κ ≈ 1%

increased a lot but still
remains neglectable

How to predict execution times?

In addition to:

```
int fibseq(int n)        int fibpar(int n)
```

We require the user to provide an asymptotic cost function:

```
int fibcost(int n) {
   return 1.618 ** n;
}
```

We use runtime profiling to deduce the associated constant factor

Benefits:

→ complexity annotations are hardware independent

→ runtime profiling does not impose a per-algorithm tuning phase

Limitations:

→ cost functions must be cheap to evaluate

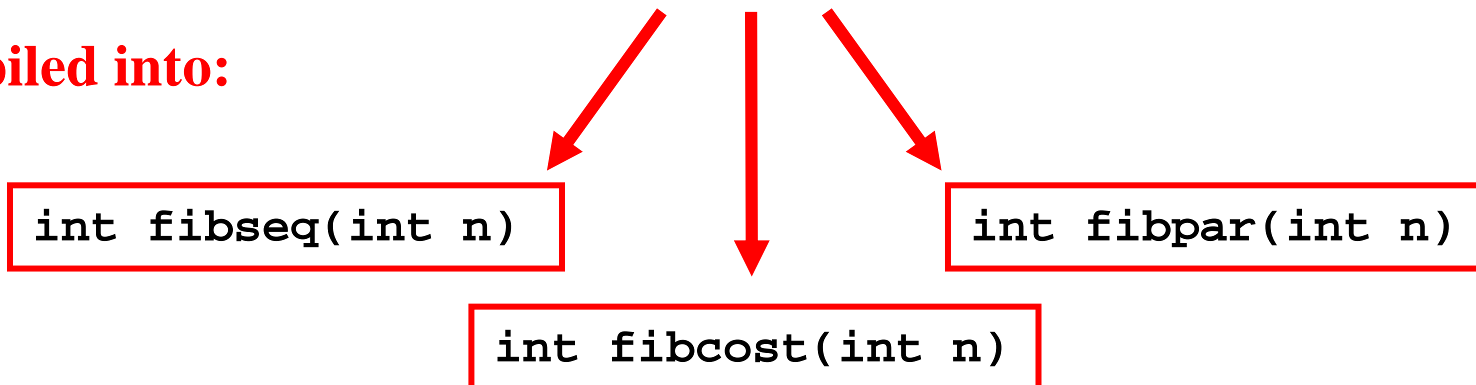
→ average complexity needs to match worst-case complexity

Code generation

source code:

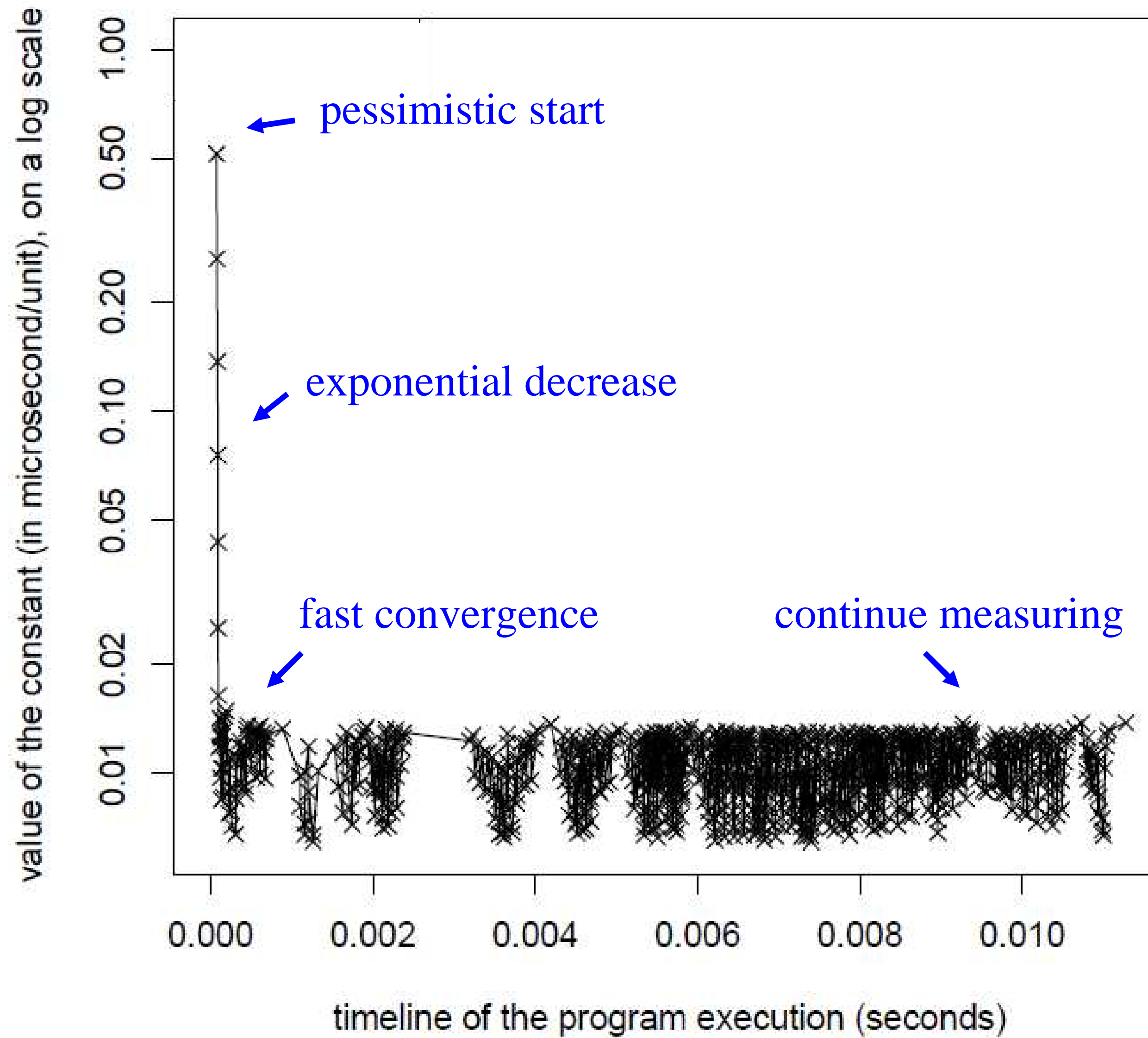
```
int fib(int n) {
   costs { return 1.618 ** n; }
   if (n < 2) return 1;
   spawn int a = fib(n-1);
   int b = fib(n-2);
   sync;
   return a+b;
}
```

compiled into:



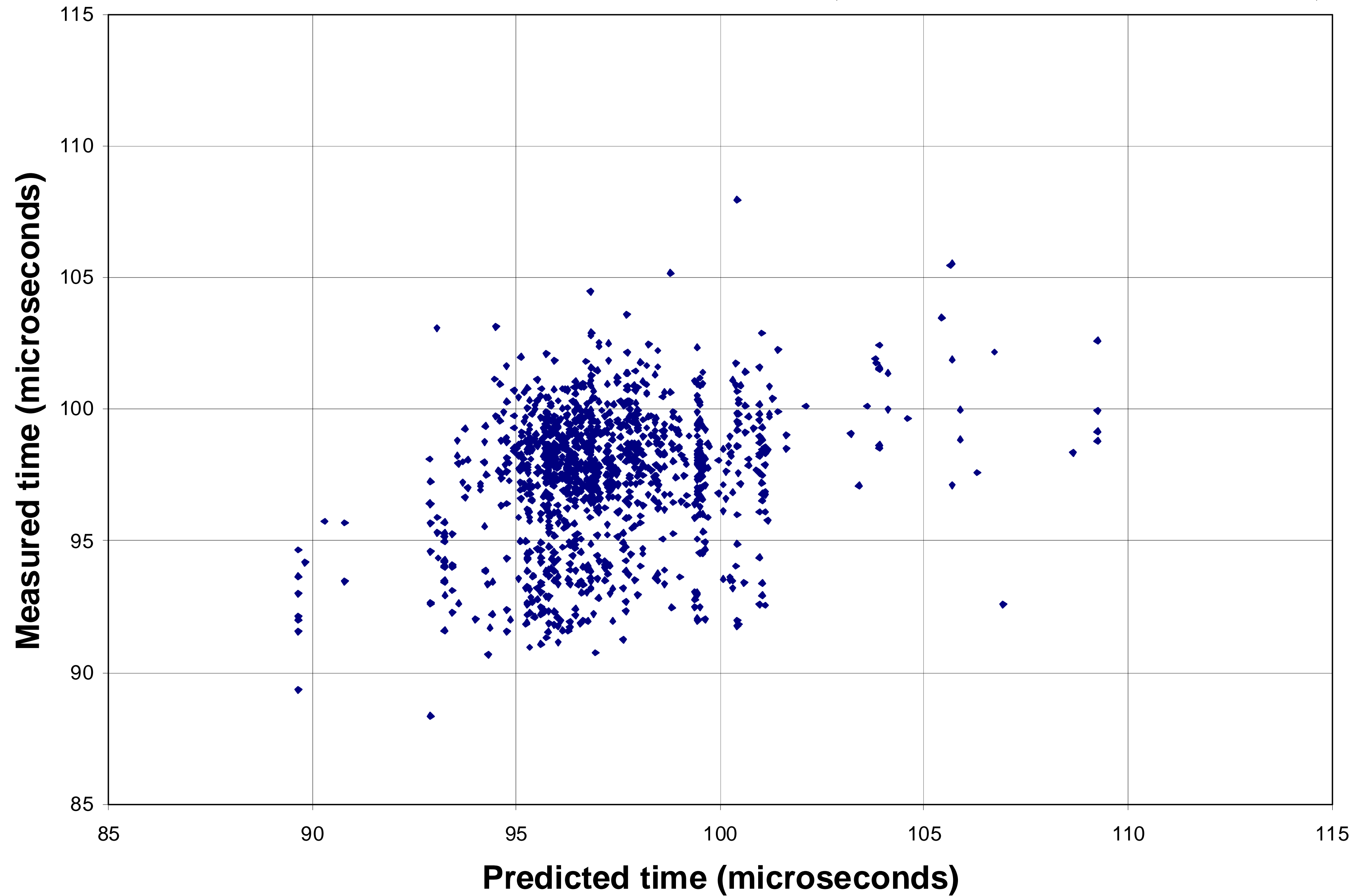
(translation implemented for the ML front-end, not yet for the C front-end)

Convergence of the constant



Accuracy of the predictions

(measured on the cilksort benchmark)



Theory, generalized model

- let ϕ be the cost of making a time prediction and a time measure
- let μ be the maximal error factor for predictions
- let γ the max ratio between two time predictions (γ=2 for most programs)

$$T_P \leq \left(1 + \frac{\mu(\tau + \gamma\phi)}{\kappa}\right) \cdot \frac{T_1}{P} + (\kappa\mu + \phi + 1) \cdot T_\infty$$

just a few percent — relatively small

Example:

1% — 2% of the first term

τ = 100 ns

ϕ = 200 ns

κ = 100,000 ns (= 0.1ms)

$\mu = 2$

$\gamma = 2$

$P = 30$

$T_1 = 10^9 \cdot 10$ns

$T_\infty = 30$

Benchmarks

Benchmarks: quickhull, quicksort, barnes-hut, dense matrix multiply, sparse matrix multiply, KMP string search, Bellman-Ford algorithm, ...

Examples of complexity functions:

```
return 1.618 ** n

return n * log n

return n ** 3

return high - low

return prefixsum[high] - prefixsum[low]
```

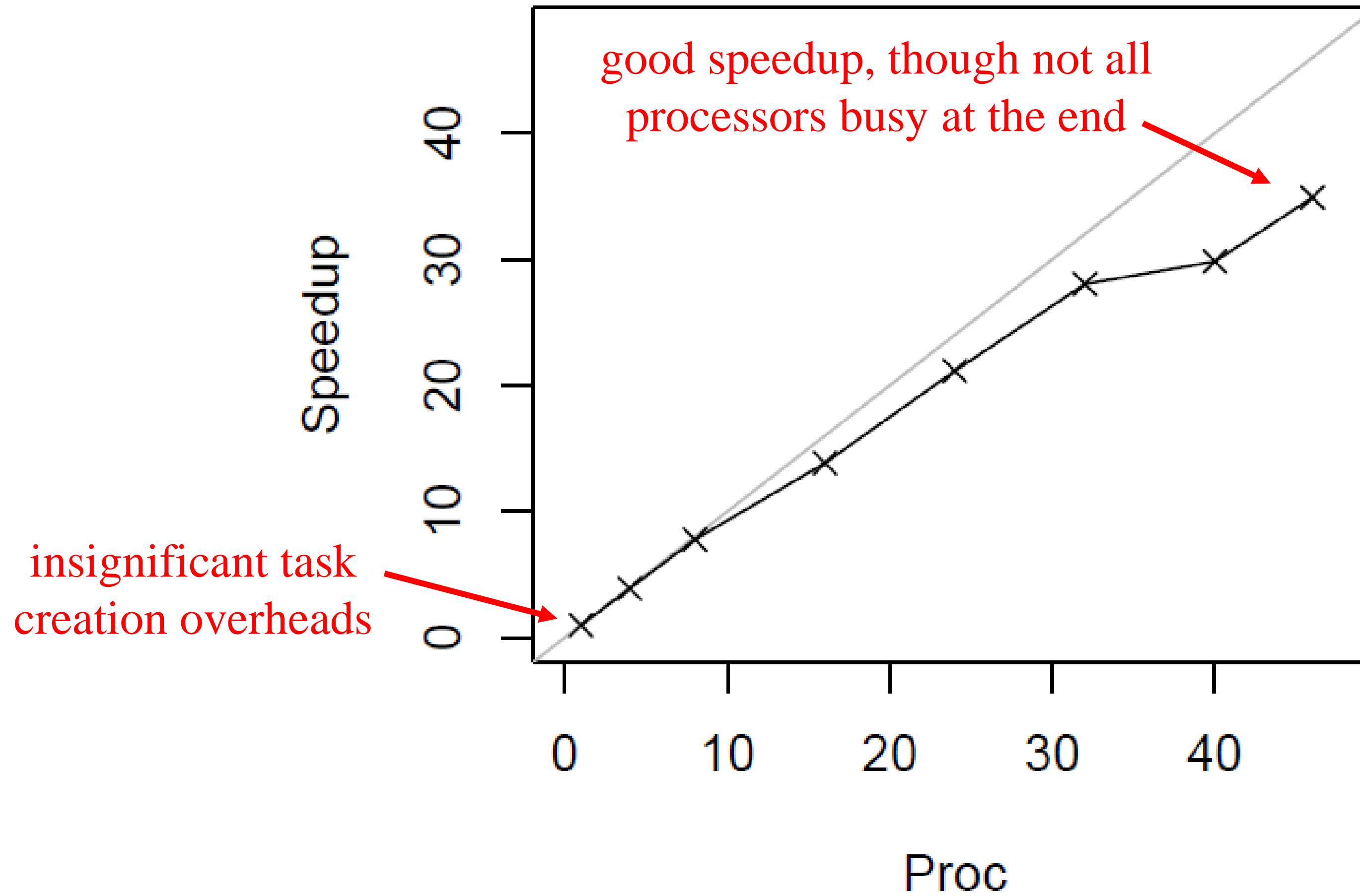
Results:

→ appropriate cutoff values are selected

→ the overheads do not exceed a few percents

Speedup curve: fib of 45

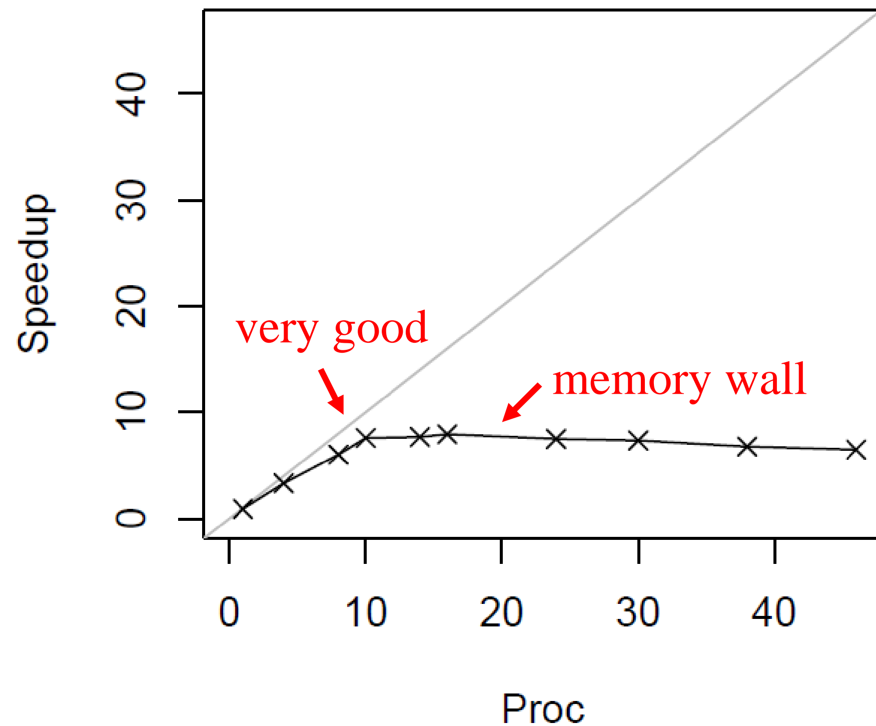
→ selected cutoff = 20



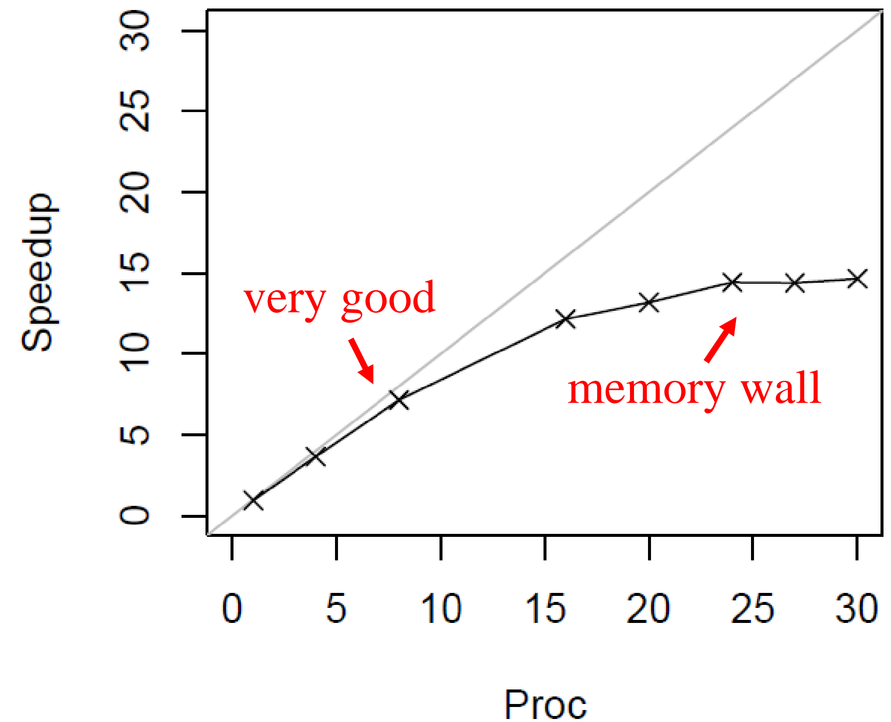
Speedup curve: cilksort on 10^8 integers

→ selected cutoff ≈ 13,000 items

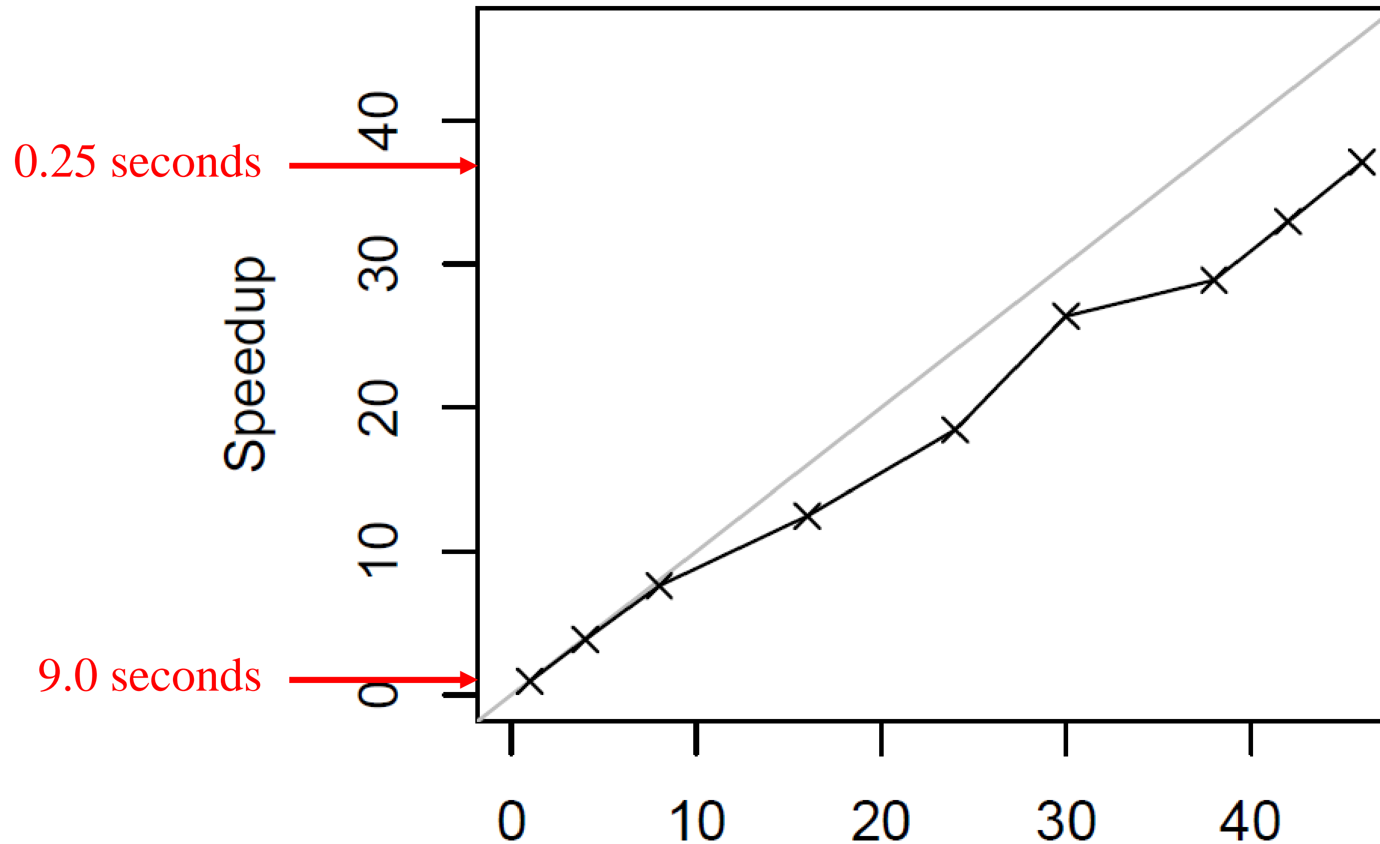
AMD, NUMA, 8 nodes with 6 cores each, 2Ghz



INTEL, UMA, 4 nodes with 8 cores each, 2Ghz



Speedup curve: KMP on 10^9 chars



→ **speedups achieved without tuning phase nor hardcoding of the cutoff**

Conclusion

1) asymptotic complexity annotations + runtime profiling
$\rightarrow$ enable execution time predictions

2) sequentializing all tasks that are predicted to be small
$\rightarrow$ ensure that task creation overheads are well amortized

Good granularity control with little effort!

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