Provably and Practically Efficient Granularity Control

Umut Acar
Carnegie Mellon University and Inria

Vitaly Aksenov
Inria & ITMO University

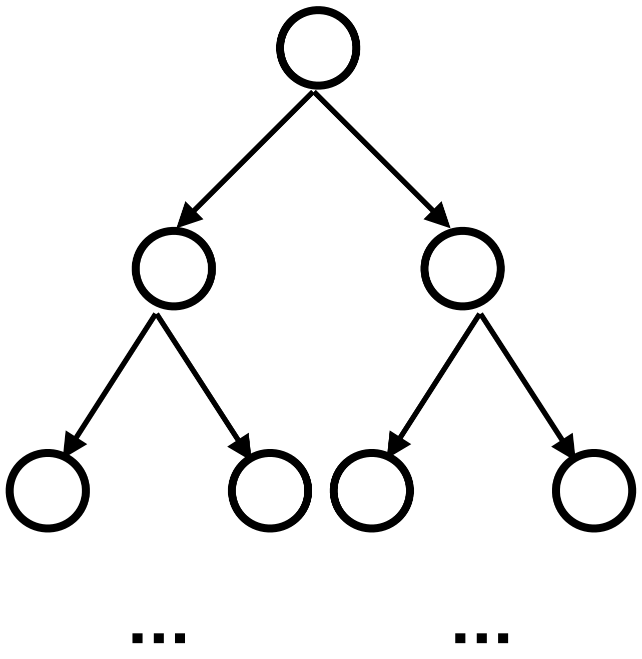
Arthur Charguéraud
Inria & University of Strasbourg, ICube

Mike Rainey
Indiana University & Inria

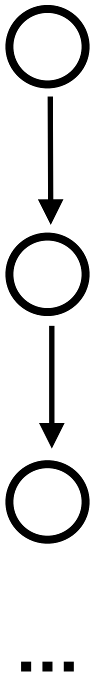
Granularity control is a balancing act

Strategies for executing fork-join programs

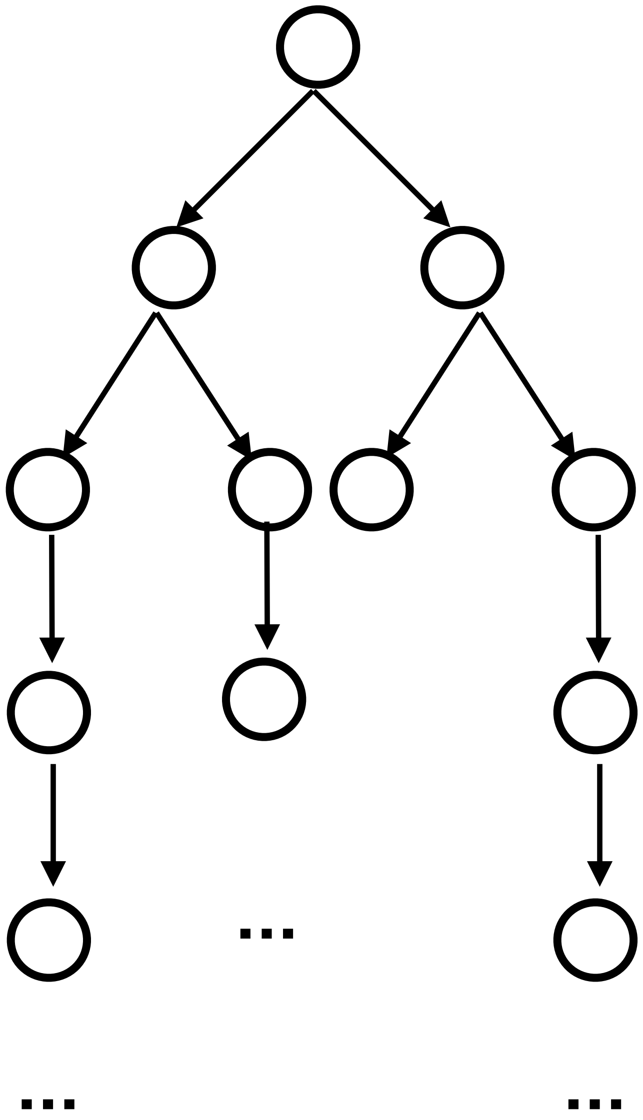
Parallelize all fork points



Sequentialize all fork points



More practical: somewhere in between



State of the art

- Expect the programmer to solve the problem by tuning the program.
- Goal: minimum-size parallel task is large enough.
- Tuning is an exponential search problem.
- Result is platform dependent code.
- Tuning generic/templated code is impractical.

Limitations of manual granularity control

```
parallel-for (i=0; i<n; i++)
  b[i] = toUpperCase(a[i])
```

Limitations of manual granularity control

```
parallel-for (i=0; i<n; i++)
  b[i] = toUpperCase(a[i])
```

```
int grain = 5000 // picked by tuning

parallel-for (i=0; i<(n+grain-1)/grain; i++)
  for (j=i*grain; j<min(n, (i+1)*grain); j++)
    b[j] = toUpperCase(a[j])
```

"sequential alternative"

Limitations of manual granularity control

```
parallel-for (i=0; i<n; i++)
  b[i] = toUpperCase(a[i])
```

```
int grain = 5000 //  picked by tuning

parallel-for (i=0; i<(n+grain-1)/grain; i++)
  for (j=i*grain; j<min(n, (i+1)*grain); j++)
    b[j] = toUpperCase(a[j])
```

“sequential alternative”

```
template <F,A,B>
void map(F f, A* a, B* b, int n)
  parallel-for (i=0; i<n; i++)
    b[i] = f(a[i])
                    map(toUpperCase, a, b, n)
                    map(someExpensiveComputation, a, b, n)
```

No single usable setting of grain for all call sites!

Related work & contribution

Main approaches to taming task-creation overheads

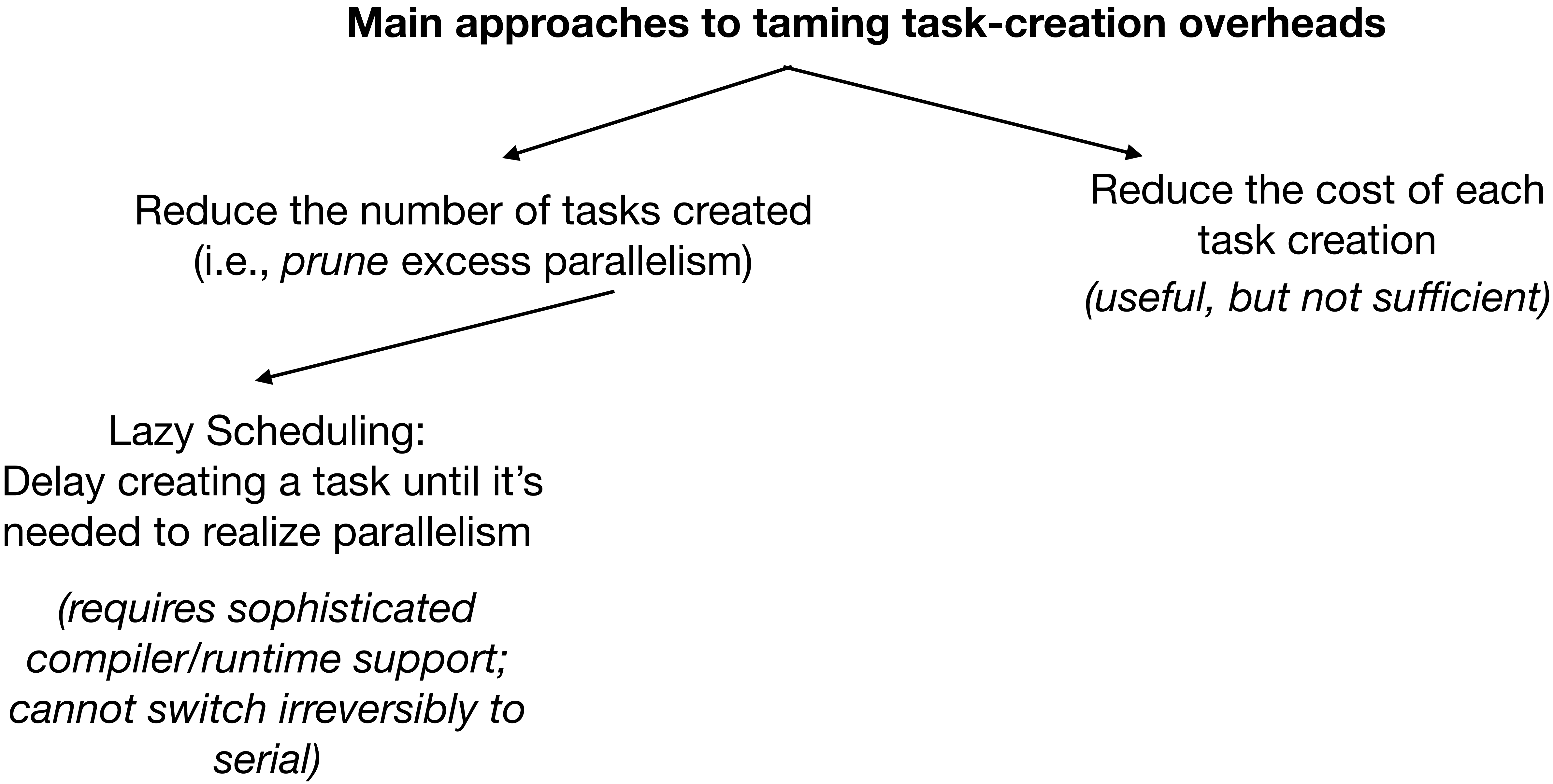
Related work & contribution

Main approaches to taming task-creation overheads

Reduce the number of tasks created
(i.e., *prune* excess parallelism)

Reduce the cost of each task creation
(useful, but not sufficient)

Related work & contribution



Main approaches to taming task-creation overheads

Reduce the number of tasks created
(i.e., *prune* excess parallelism)

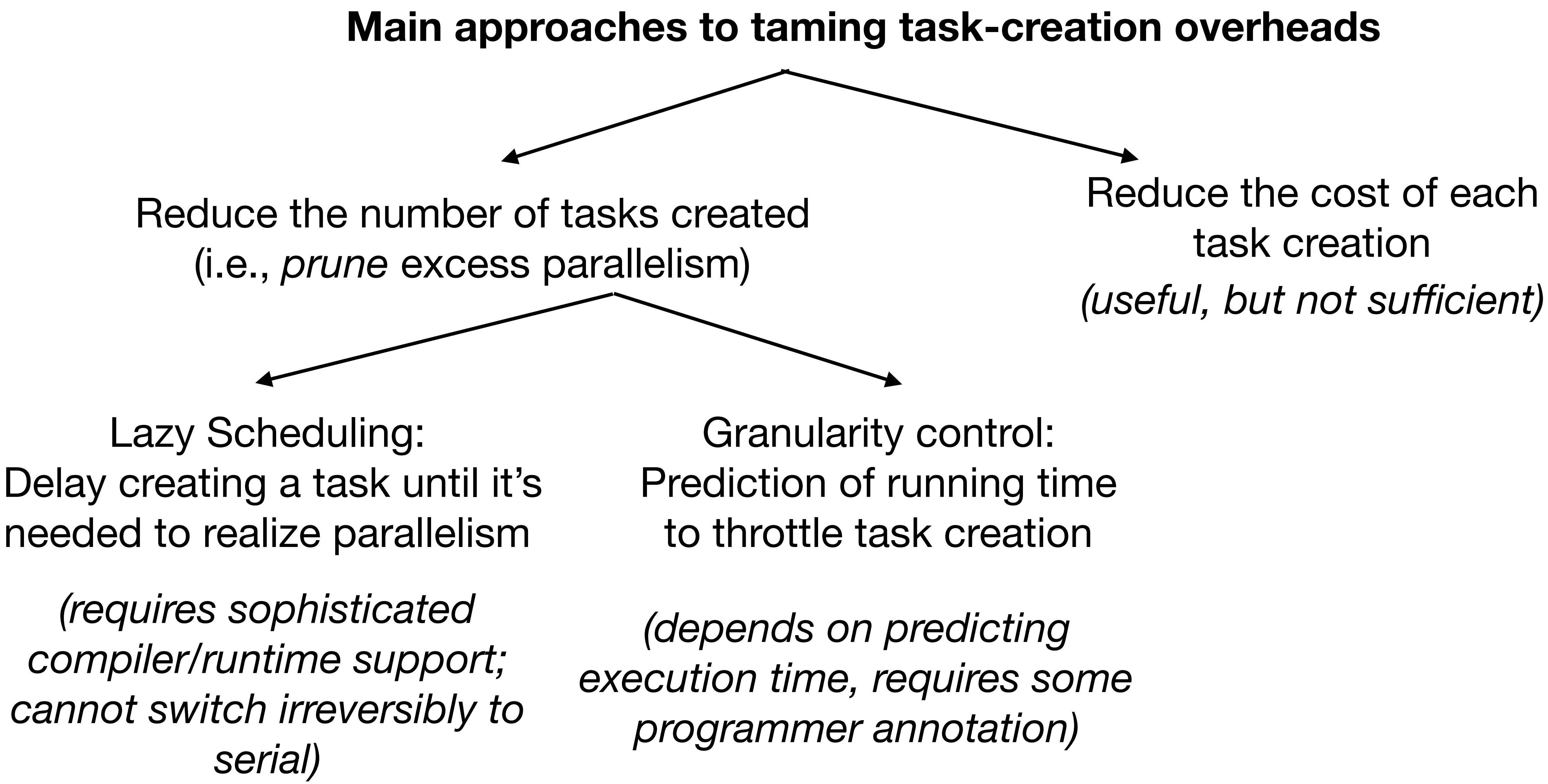
Lazy Scheduling:
Delay creating a task until it’s
needed to realize parallelism

(requires sophisticated compiler/runtime support; cannot switch irreversibly to serial)

Reduce the cost of each
task creation

(useful, but not sufficient)

Related work & contribution



Main approaches to taming task-creation overheads

Reduce the number of tasks created
(i.e., *prune* excess parallelism)

Reduce the cost of each
task creation
(useful, but not sufficient)

Lazy Scheduling:
Delay creating a task until it's
needed to realize parallelism

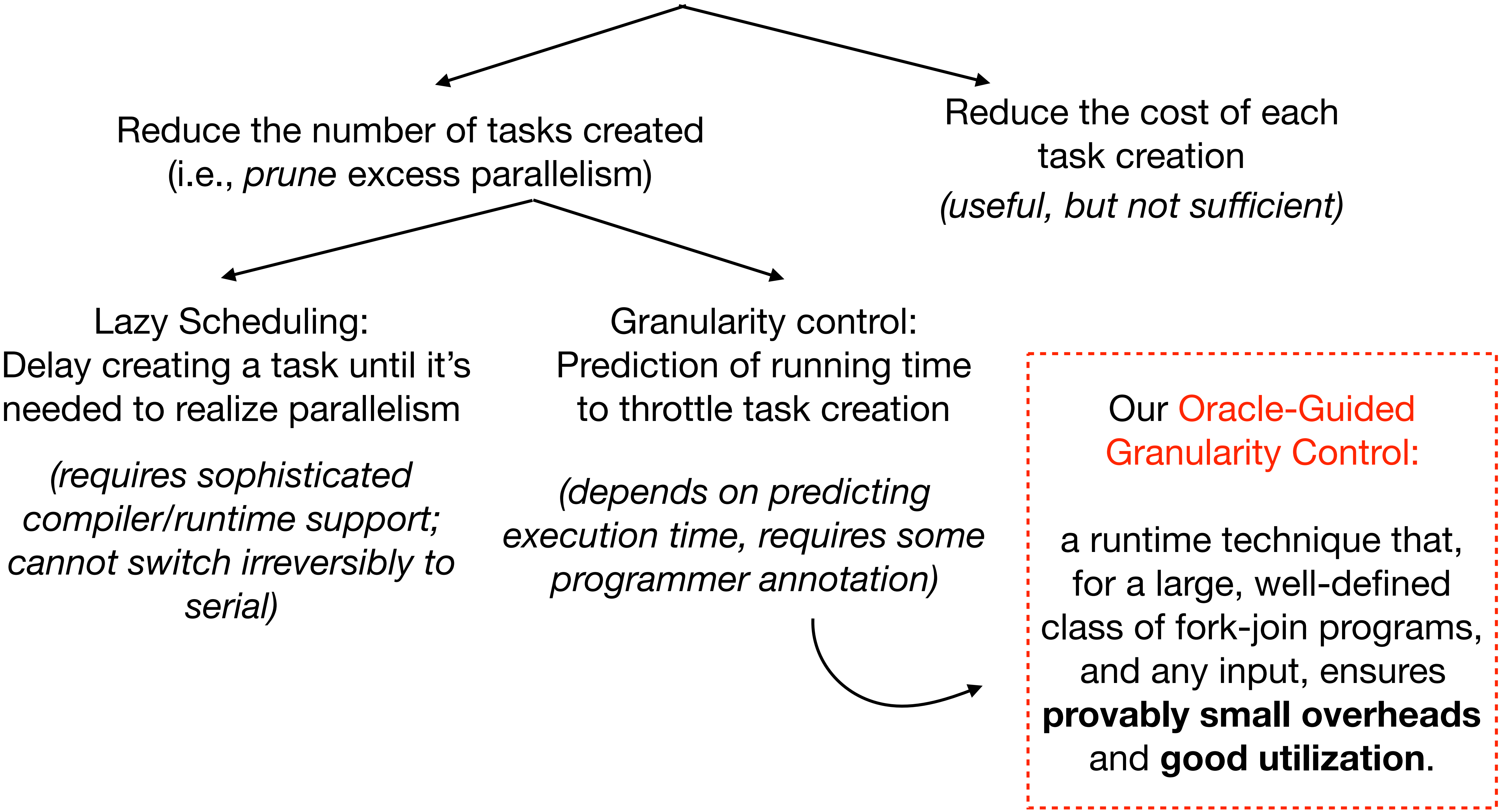
(requires sophisticated compiler/runtime support; cannot switch irreversibly to serial)

Granularity control:
Prediction of running time
to throttle task creation

(depends on predicting execution time, requires some programmer annotation)

Related work & contribution

Main approaches to taming task-creation overheads



Reduce the number of tasks created
(i.e., *prune* excess parallelism)

Reduce the cost of each
task creation

(useful, but not sufficient)

Lazy Scheduling:
Delay creating a task until it's
needed to realize parallelism

(requires sophisticated compiler/runtime support; cannot switch irreversibly to serial)

Granularity control:
Prediction of running time
to throttle task creation

(depends on predicting execution time, requires some programmer annotation)

Our Oracle-Guided
Granularity Control:

a runtime technique that, for a large, well-defined class of fork-join programs, and any input, ensures **provably small overheads** and **good utilization**.

Series-parallel guard

Our goal: lift the burden of tuning by transferring to the runtime.

We propose: (a single, new programming construct)

$\mathtt{\mathbf{spguard}}(\mathtt{F}_{\mathtt{cost}},\ \mathtt{F}_{\mathtt{par}},\ \mathtt{F}_{\mathtt{seq}})$

Abstract-cost function
e.g., $n * \log(n)$, n^2

Parallel body

Sequential body
(some code that is semantically equivalent to the parallel body)

Behavior of spguard: determine automatically, at run time, whether to run sequential or parallel body.

Example: parallel mergesort

```
Seq parallelMergesort(Seq x) {
  Seq r
  spguard([&] {
    int n = size(x)
    return n * log(n)
  }, [&] {
    if size(x) < 2
      r = x
    else
      (x1, x2) = splitInHalves(x)
      r1 = spawn parallelMergesort(x1)
      r2 =       parallelMergesort(x2)
      sync
      r = concat(r1, r2)
  }, [&] {
    r = sequentialSort(x)
  }) // end spguard
  return r
}
```

← Abstract-cost function

← Parallel body

← Sequential body

How does it predict when to sequentialize?

Our desired task size:

κ Marginal profitable task size (e.g., 25-500 μsec)

How does it predict when to sequentialize?

Our desired task size:

κ Marginal profitable task size (e.g., 25-500 μsec)

Consider an execution of `spguard`$(\mathtt{F}_{\mathtt{cost}},\ \mathtt{F}_{\mathtt{par}},\ \mathtt{F}_{\mathtt{seq}})$

For such an execution, let:

$cost\ =$ Result of cost function (i.e., $cost = \mathtt{F}_{\mathtt{cost}}\mathtt{()}$)

$work =$ Execution time across all parallel paths of body, (i.e., $\mathtt{F}_{\mathtt{par}}\mathtt{()}$ or $\mathtt{F}_{\mathtt{seq}}\mathtt{()}$).

How does it predict when to sequentialize?

Our desired task size:

κ Marginal profitable task size (e.g., 25-500 μsec)

Consider an execution of `spguard(`$\texttt{F}_{\texttt{cost}}$`, `$\texttt{F}_{\texttt{par}}$`, `$\texttt{F}_{\texttt{seq}}$`)`

For such an execution, let:

$cost$ = Result of cost function (i.e., $cost = \texttt{F}_{\texttt{cost}}\texttt{()}$)

$work$ = Execution time across all parallel paths of body, (i.e., $\texttt{F}_{\texttt{par}}\texttt{()}$ or $\texttt{F}_{\texttt{seq}}\texttt{()}$).

After it executes, we update the internal state of the spguard:

$$cost_{max} \, ,$$

which represents the largest observed $cost$ such that $work \leq \kappa$.

How does it predict when to sequentialize?

Our desired task size:

κ Marginal profitable task size (e.g., 25-500 μsec)

Consider an execution of **`spguard`**`(`$\texttt{F}_{\texttt{cost}}$`, `$\texttt{F}_{\texttt{par}}$`, `$\texttt{F}_{\texttt{seq}}$`)`

For such an execution, let:

$cost$ = Result of cost function (i.e., $cost = \texttt{F}_{\texttt{cost}}\texttt{()}$)

$work$ = Execution time across all parallel paths of body, (i.e., $\texttt{F}_{\texttt{par}}\texttt{()}$ or $\texttt{F}_{\texttt{seq}}\texttt{()}$).

After it executes, we update the internal state of the spguard:

$$cost_{max} \,,$$

which represents the largest observed $cost$ such that $work \leq \kappa$.

Sequentialize iff: $cost \leq 2 * cost_{max}$

Challenge: predicting when to sequentialize

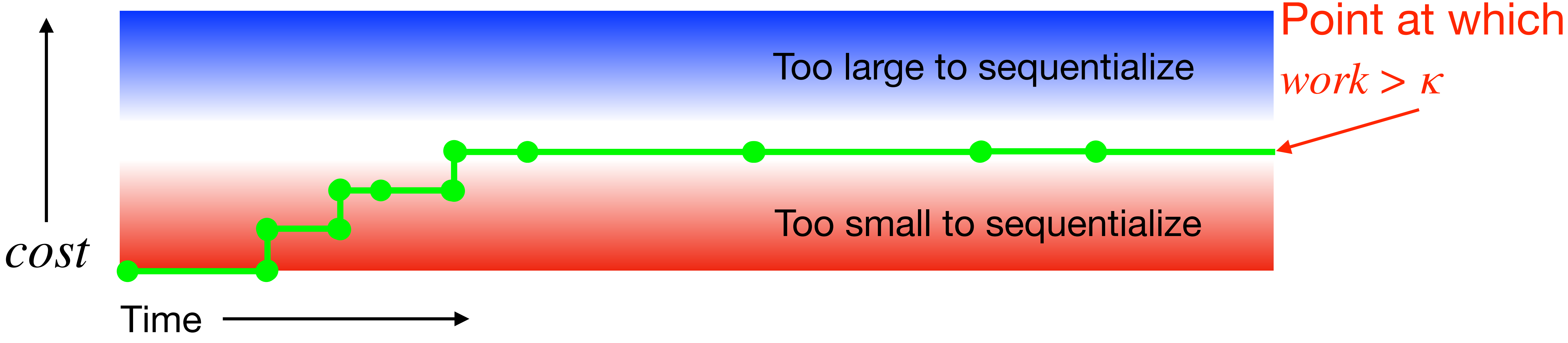
`spguard``(`$\mathtt{F}_{\mathtt{cost}}$`,` $\mathtt{F}_{\mathtt{par}}$`,` $\mathtt{F}_{\mathtt{seq}}$`)`

κ Marginal profitable task size (e.g., 25-500 μsec)

$cost =$ Result of cost function (i.e., $cost = \mathtt{F}_{\mathtt{cost}}\mathtt{()}$)

$work =$ Execution time across all parallel paths of an execution of the spguard

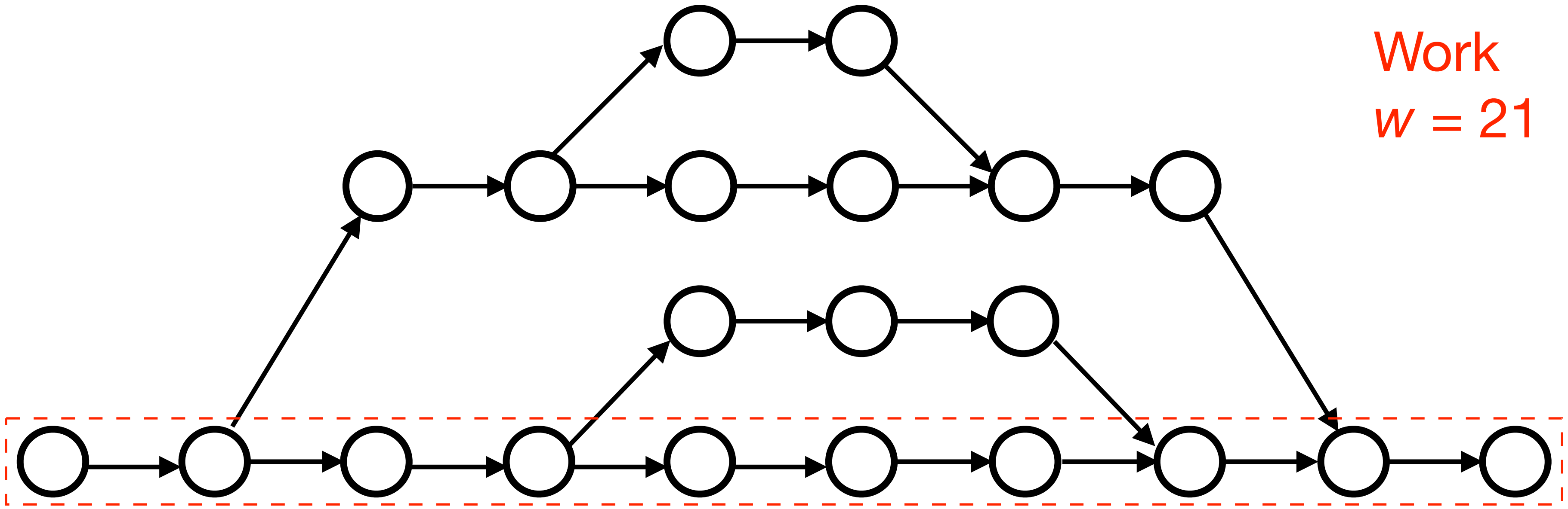
Convergence of $cost_{max}$ **:**



Cost model and bound

Work
w = total # of vertices

Span
s = length of critical path



Work-stealing bound (Blumofe & Leiserson)

For any fork-join program, the running time t_p on p cores, including the load balancing operations, but excluding task-creation overheads, is bounded as follows:

$$E[t_p] \le w/p + O(s)$$

Bound for Oracle-Guided Granularity Control

w	Work (total # vertices)
s	Span (critical-path length)
t_p	Running time of the program on p cores

We extend the model to take into account task-creation costs:

τ	Cost of creating a fiber
κ	Amount of per-task work targeted

(e.g., to ensure 5% per-task overhead, set $\kappa = 20\tau$)

Work stealing: $E[t_p] \le w/p + O(s)$

Our bound:

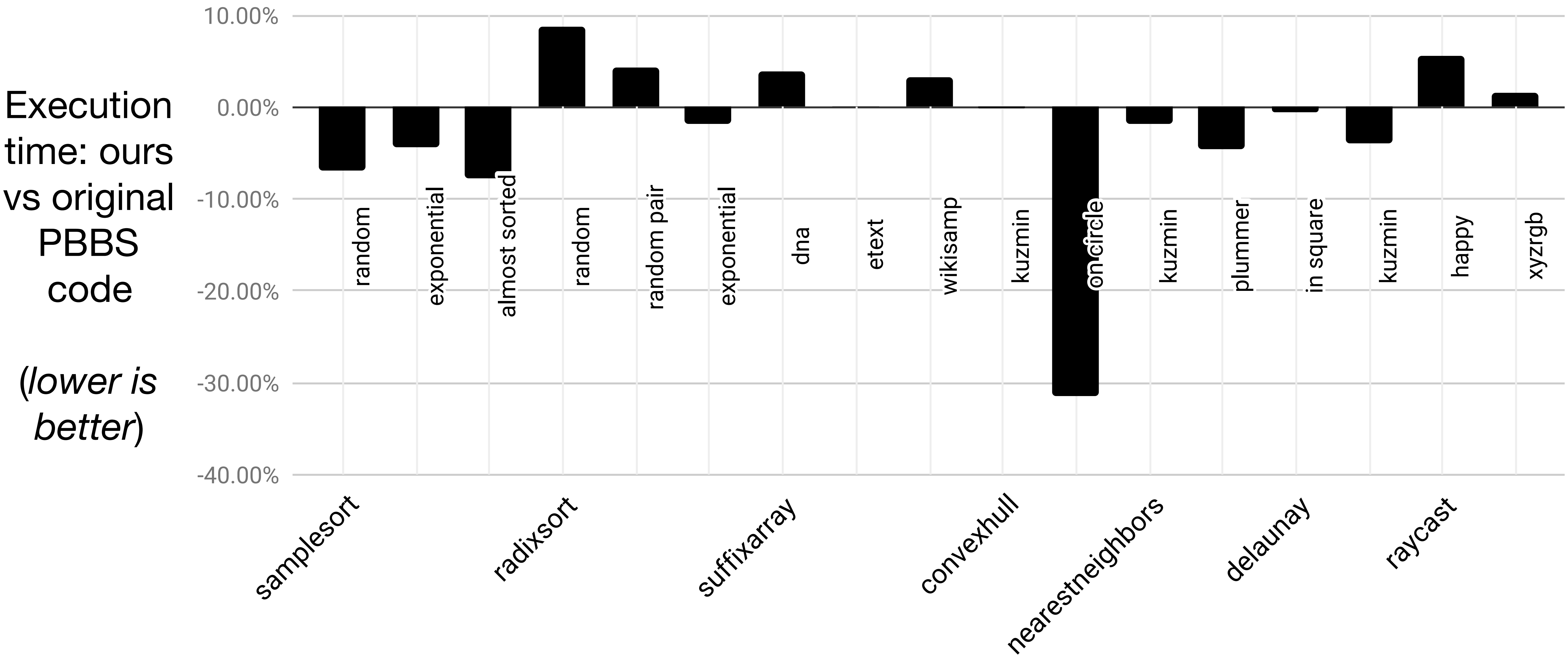
$$E[t_p] \le w/p + \underbrace{(\tau/\kappa * w/p)}_{\text{1. (e.g., 5\%)}} + O\underbrace{(\kappa/\tau * s)}_{\text{2. (e.g., 20x)}} + \underbrace{O(\log^2 \kappa)}_{\text{3. Overhead introduced by granularity controller}}$$

C++ library implementation

- Our library provides:
  - the spguard construct
  - helper functions for frequently used cost functions
  - parallel-for loops and data-parallel operations, e.g., map, reduce, prefix-scan, filter, etc.
- Our library uses Cilk Plus spawn/sync as basis, but is compatible with any fork-join language or library.
- We ported 8 benchmark codes from the Problem Based Benchmark Suite (PBBS), a collection representing irregular workloads.
- We needed to write only 24 explicit cost functions; the rest could use the default, which is linear complexity.

Benchmarking results

Our spguard automatically delivers similar or better results to manually controlled code.



40-core Intel machine with 1TB RAM

Conclusion

Formal bounds for scheduling fork join

Brent ’74, Arora et al ’98, Blumofe & Leiserson ’99, Agarwal et al ’07, Acar et al ‘11

Oracle-Guided Granularity control extends these results with analytical bounds on scheduling overheads for fork-join programs.

Lazy-scheduling methods

Mohr et al ’91, Feeley ’93, Goldstein et al ’96, Frigo et al ’98, Imam et al ’14, Tzannes et al ’14, Acar et al ‘18

Oracle-Guided Granularity Control can be implemented as a library and can switch irrevocably to serial algorithms, unlike this class of algorithms.

Prediction-based methods

Weening ’89, Pehoushek et al ’90, Lopez et al ’96, Duran et al ’08, Acar et al ’16, Iwasaki et al ’16, Shintaro et al ‘16

Oracle-Guided Granularity Control is the first in this class to have a state-of-the-art implementation and be backed by end-to-end bounds.

Conclusion

Formal bounds for scheduling fork join

Brent '74, Arora et al '98, Blumofe & Leiserson '99, Agarwal et al '07, Acar et al '11

Oracle-Guided Granularity control extends these results with analytical bounds on scheduling overheads for fork-join programs.

Lazy-scheduling methods

Mohr et al '91, Feeley '93, Goldstein et al '96, Frigo et al '98, Imam et al '14, Tzannes et al '14, Acar et al '18

Oracle-Guided Granularity Control can be implemented as a library and can switch irrevocably to serial algorithms, unlike this class of algorithms.

Prediction-based methods

Weening '89, Pehoushek et al '90, Lopez et al '96, Duran et al '08, Acar et al '16, Iwasaki et al '16, Shintaro et al '16

Oracle-Guided Granularity Control is the first in this class to have a state-of-the-art implementation and be backed by end-to-end bounds.

Thanks!