Theory and practice of granularity control

Umut Acar
Carnegie Mellon University / Inria

Vitaly Aksenov
Inria

Arthur Charguéraud
Inria & LRI Université Paris Sud, CNRS

Anna Malova

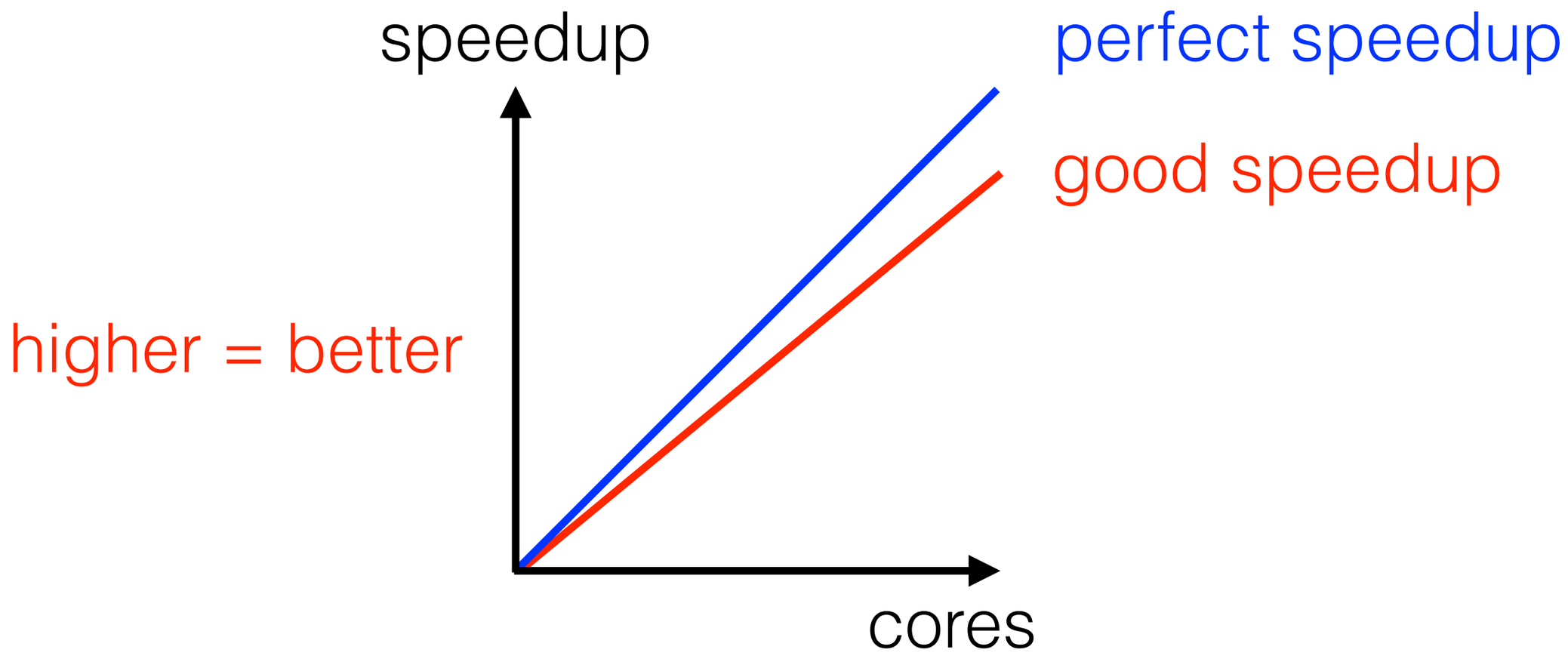
Mike Rainey
Inria

Marc Pasqualetto

Gallium seminar
September 2016

Speedups with multicores

Goal: using several cores, achieve good speedups compared to fast sequential code

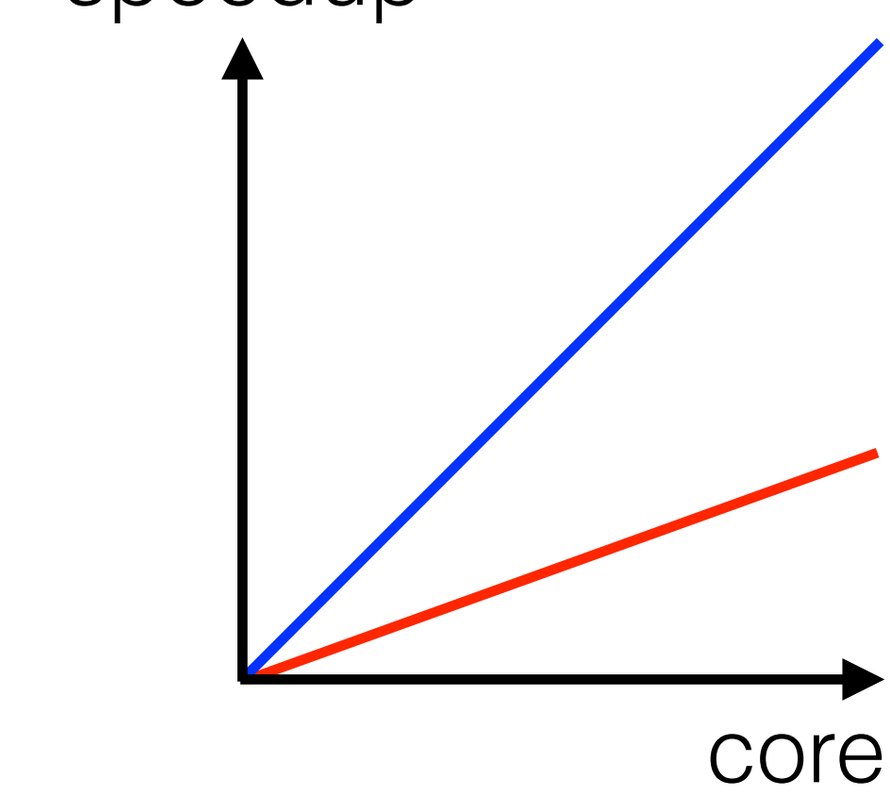


Obstacles: lack of parallelism, memory wall, scheduling overheads

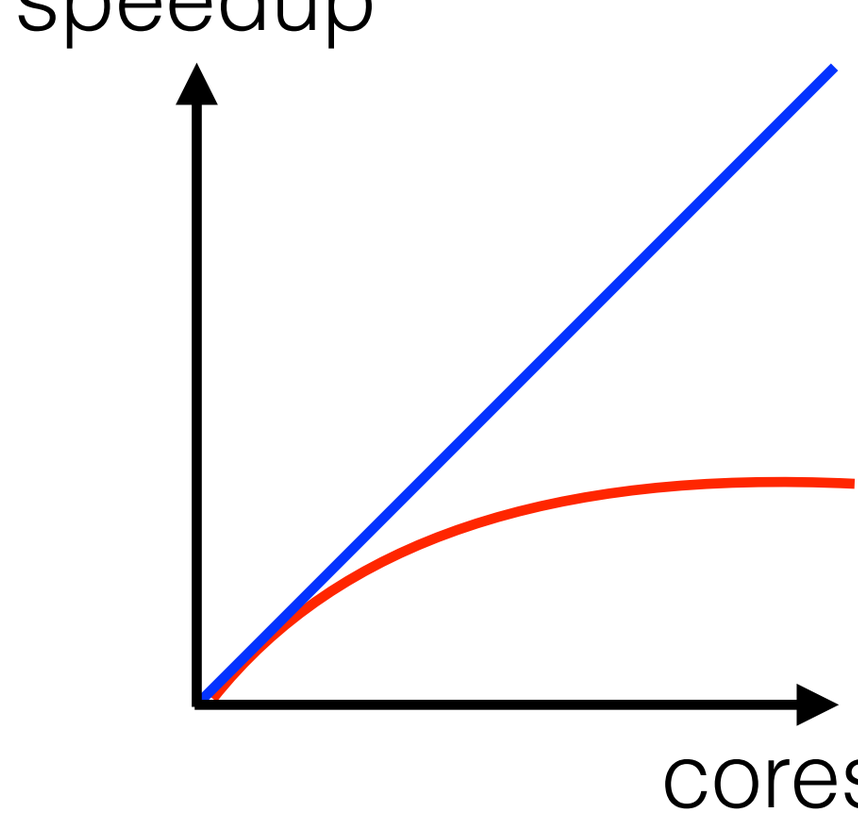
Granularity control

Scheduling overheads: they mainly depend on the number of threads

too many threads ⇒ large overheads



too few threads ⇒ limited parallelism



Granularity control: the problem of balancing between these two extremes

Importance of granularity control

Sequential code:

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

Parallel code:

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

Time to compute 45th Fibonacci number

10 seconds on a single core

20 seconds on 42 cores

- 1.8 billion parallel threads created
- per-thread overhead of a few dozens memory accesses

Introduction of a threshold

Parallel code with threshold:

```
int fibthresh(int n)
  if (n <= threshold)
    return fibseq(n)
int a = spawn fibthresh(n-1)
int b = fibthresh(n-2)
sync
return a+b
```

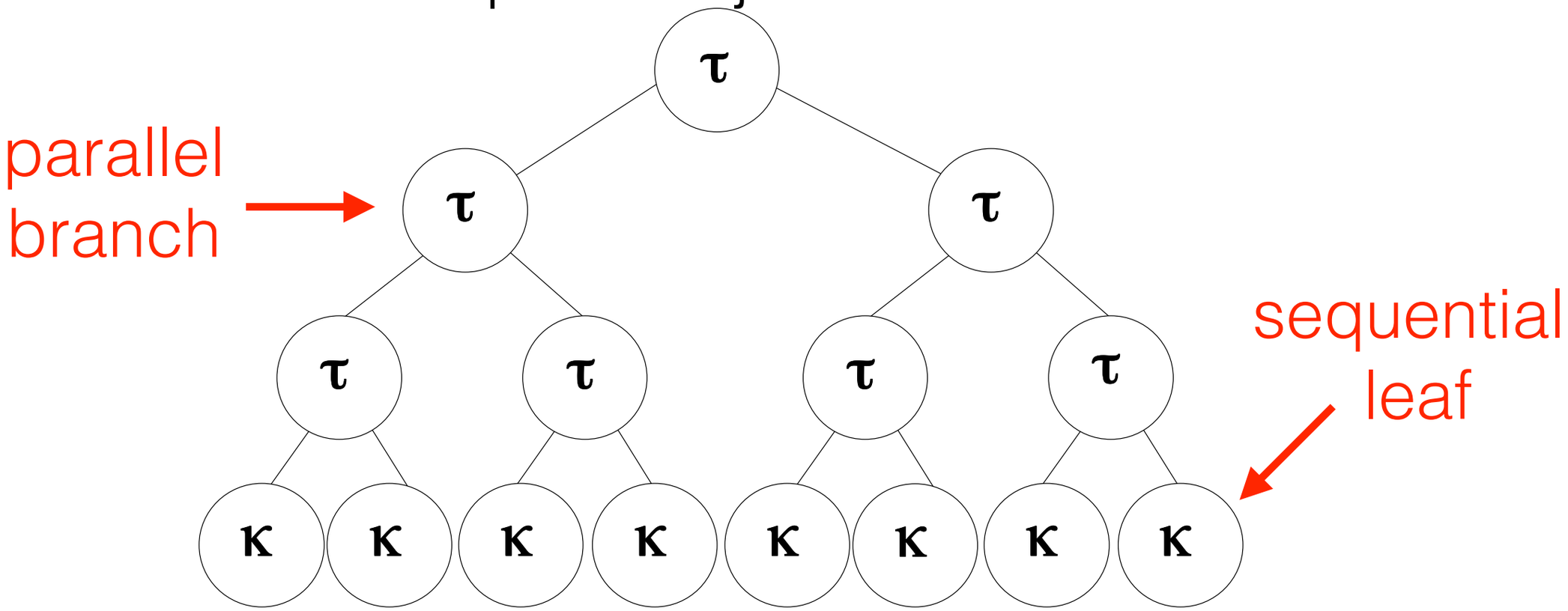
What is the right value to use as threshold?

Selection of the threshold

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

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

sample fork-join call tree



Theory, assuming ideal oracle

Brent's theorem: (thread-creation costs completely ignored)

$$T_P \leq T_1/P + T_\infty$$

negligible when a lot of parallelism is available

Our theorem: (thread-creation cost = τ, sequentialize if running time < κ)

$$T_P \leq (1 + \tau/\kappa)T_1/P + \kappa T_\infty$$

we choose κ such that $\tau/\kappa \cong 1\%$

term is increased but remains negligible

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)

$$T_P \leq (1 + \mu(\tau+\gamma\phi)/\kappa)T_1/P + (\kappa\mu+\phi+1)T_\infty$$

just a few percent — relatively small

Example:

$\tau = 100$ ns
$\varphi = 200$ ns
$\kappa = 100{,}000$ ns

1%

$\mu = 2$
$\gamma = 2$
$P = 30$

2% of first term

$T_1 = 10^9 \times 10$ ns
$T_\infty = 30$

How we predict execution times for a real program on a real machine

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 1.68^n
```

We allocate one data structure, `fibprof`, that stores profiling data.

Report the time `t` that elapsed during given call `fib(n)`:

```
fibprof.report(t / fibcost(n))
```

Predict running time `t` for given call `fib(n)`:

```
t <- fibprof.predict(n)
```

How we make prediction robust

core

time

	A	B	C
0	M	I	I
1	M	I	W
2	W	W	M
3	W	W	M
4	W	I	W
5	W	W	W
6	M	W	I
7	M	M	I
8	I	M	W

…

W	Working
I	Idle
M	Measured call

Measured calls measure running time of serial calls and report profiling data

Experimental evaluation

Benchmarks: 10 codes written by other researchers (PBBS benchmark suite of Blelloch et al)

Inputs: > 3 different inputs for each benchmark

Machine: 40 Intel Xeon cores @ 2Ghz / 1TB RAM

Platform: C++ / Cilk Plus

Examples of complexity functions:

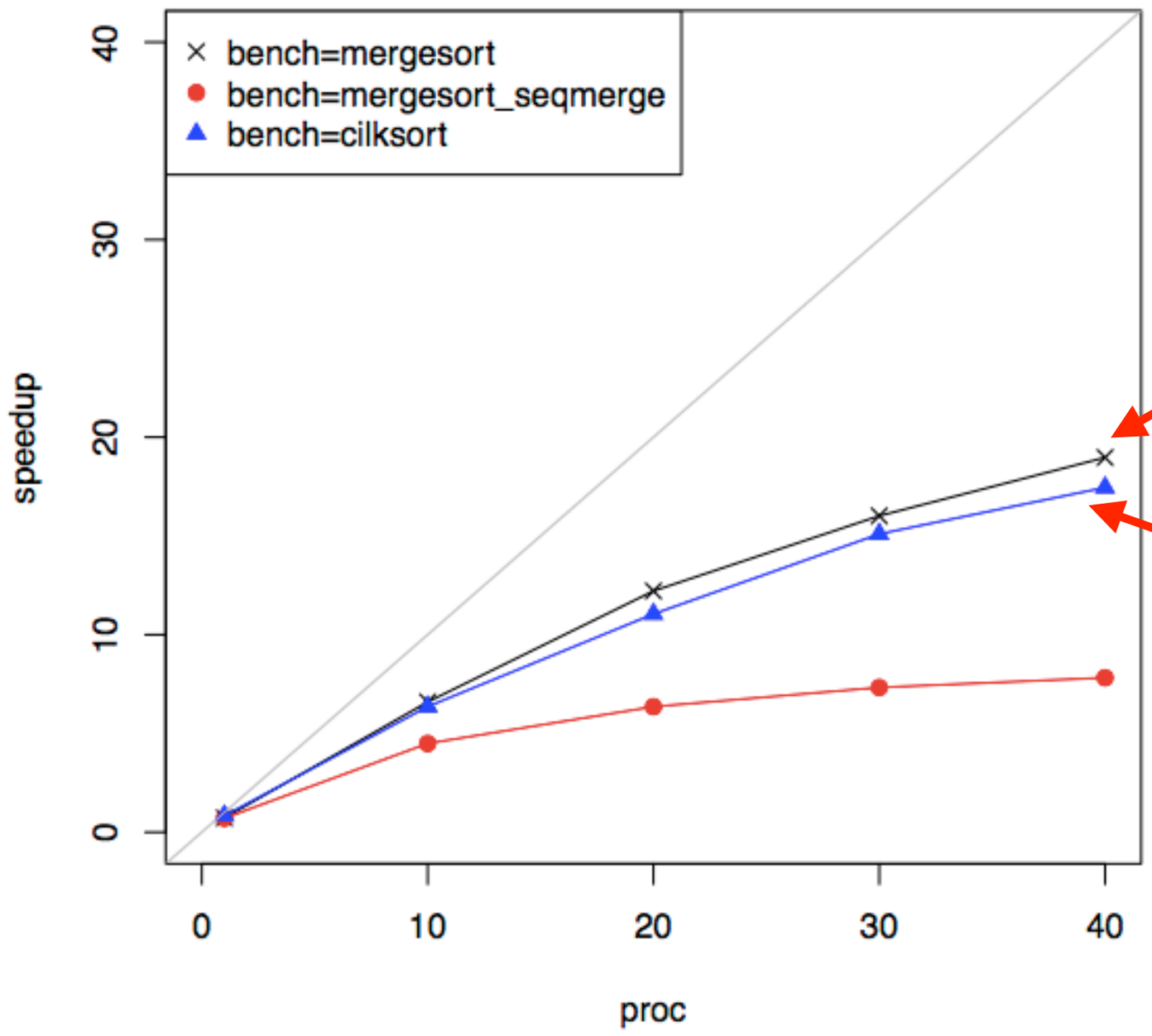
```
return 1.618 ** n
return n * log n
return n ** 3
return high - low
return prefixsum[high] - prefixsum[low]
```

Benchmarking results

	-30% —-20%	-20% — -10%	≈	+20% —+10%	+30% —+20%
comparison sort			X		
radix sort		X	X		
suffix array		X	X		
convex hull	X	X	X		
nearest neighbors		X	X	X	
ray cast			X	X	X
delaunay triangulation			X		
delaunay refine			X		
bfs			X	X	
remove duplicates			X		

% speedup of oracle-guided over original

Our technique being used in lecture material and by undergraduates in CMU algorithms lab



our mergesort with automatic granularity control

MIT mergesort with manual granularity control

Summary

Strengths:

- Many parallel codes can readily benefit from automatic granularity control.
- By switching to purely sequential code at the leaves, we can leverage on sequential optimizations.

Weaknesses:

- Irregular, nested parallelism is sometimes challenging because of complexity function.
- Complexity functions for higher-order functions are challenging, e.g., what is the complexity function for a map?
- Sometimes a suitable complexity function does not exist, e.g., string matching.
- Our technique is currently limited to fork join codes.

High-performance graph traversal

- In a *graph traversal*, computation proceeds from one vertex to the next through the edges in the graph.
- Improved performance for graph traversal means improved performance for many other algorithms.
- The main challenge is coping with irregularity in graphs.
- In this work, we present a new algorithm
  - to perform fast traversal over large, in-memory directed graphs
  - using a (single, dedicated) multicore system
  - achieving:
    - analytical bounds showing work-efficiency and high-parallelism, and
    - an implementation that outperforms state-of-the-art codes (almost always)

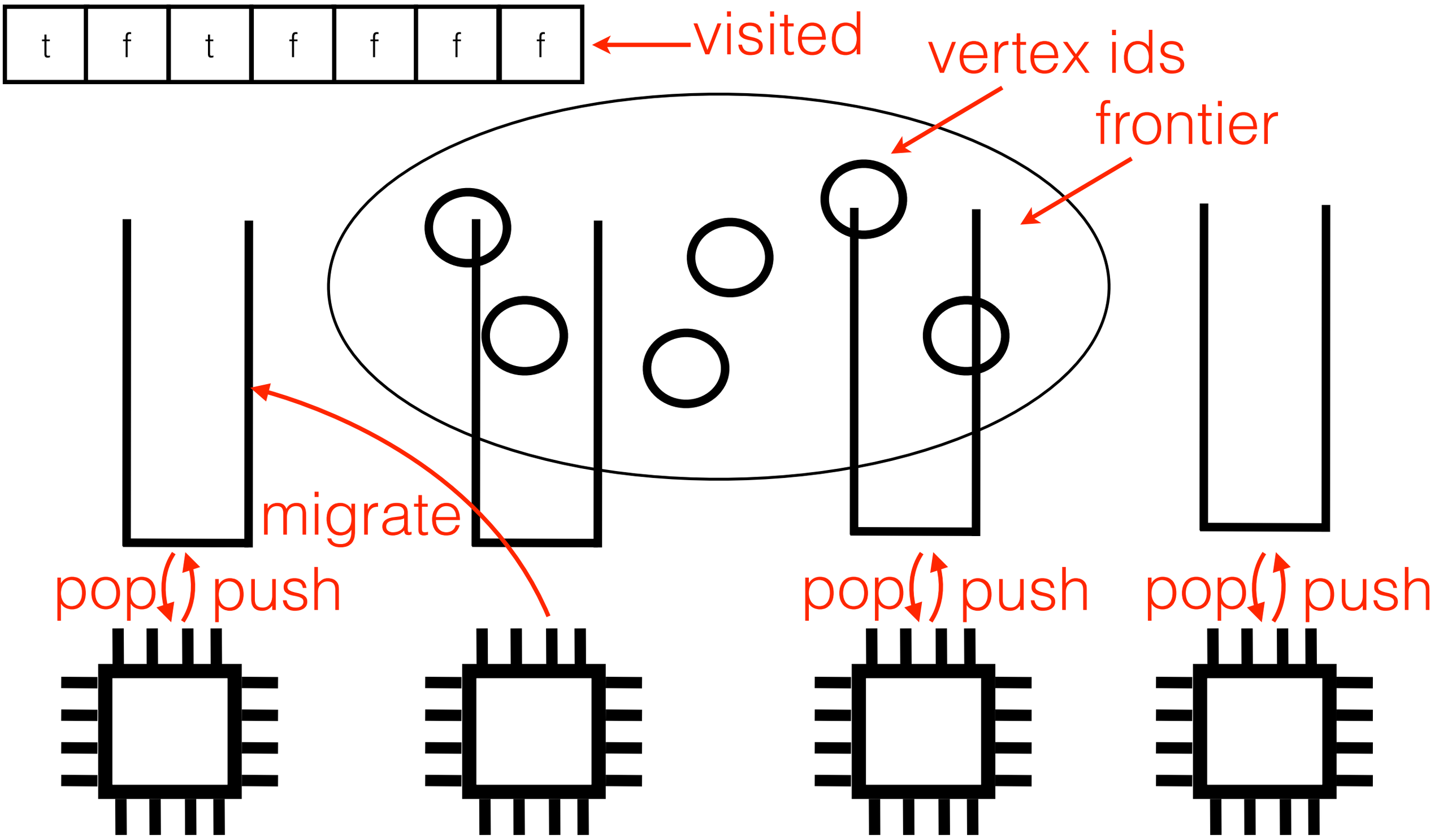
Motivation

- Most of the recent attention in the research literature on graph traversal is paid to parallel BFS.
- Why parallel BFS but not parallel DFS?
    - Parallel DFS with strict ordering is known to be P-complete (i.e., hard to parallelize).
- However, loosely ordered, parallel DFS:
    - relaxes the strict DFS ordering slightly
    - achieves a high degree of parallelism
    - has many applications, e.g.,
        - reachability analysis & graph search
        - parallel garbage collection (Jones et al 2011), etc…
        - KLA graph-processing framework (Harshvardhan et al 2014)
- When feasible, Pseudo DFS is preferred because it is usually faster than the alternatives.

Pseudo DFS (PDFS)

- Input:
  - directed graph and ID of source vertex
- Output:
  - the set of vertices connected by a path to the source vertex

PDFS


t
f
t
f
f
f
f
visited
vertex ids
frontier
migrate
pop()
push
pop()
push
pop()
push

PDFS vs. PBFS

Synchronization

- PDFS is *asynchronous*:
  - Each core traverses independently from its frontier.
- PBFS is *level synchronous*:
  - Cores traverse the graph level by level, in lock step, synchronizing between every two levels.

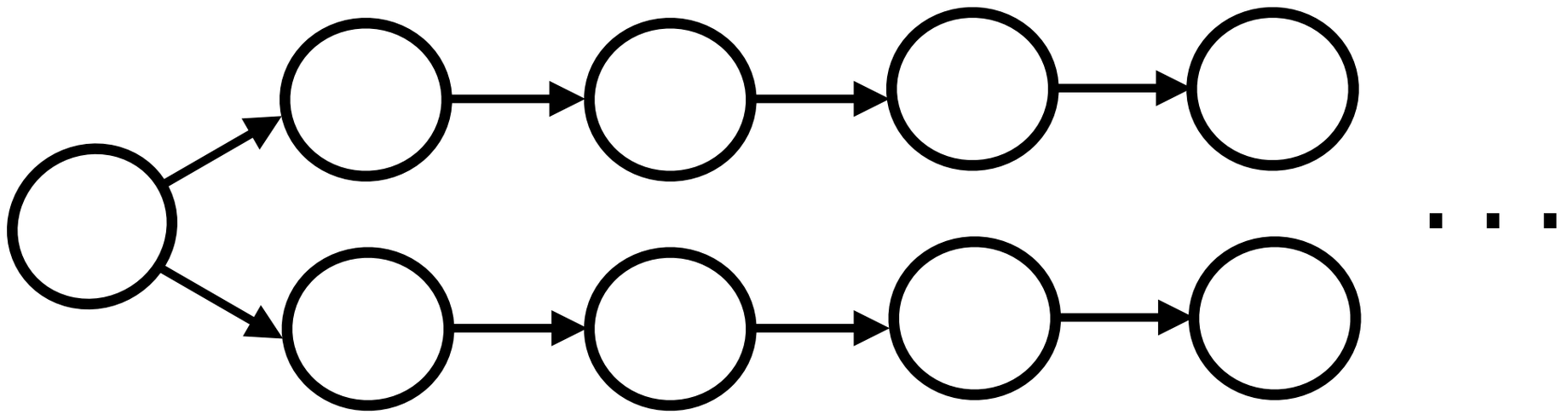
Data locality

- DFS is preferred in parallel GC.
  - e.g., mark sweep
- Why?
  - DFS visits heap objects in the order in which objects were allocated.

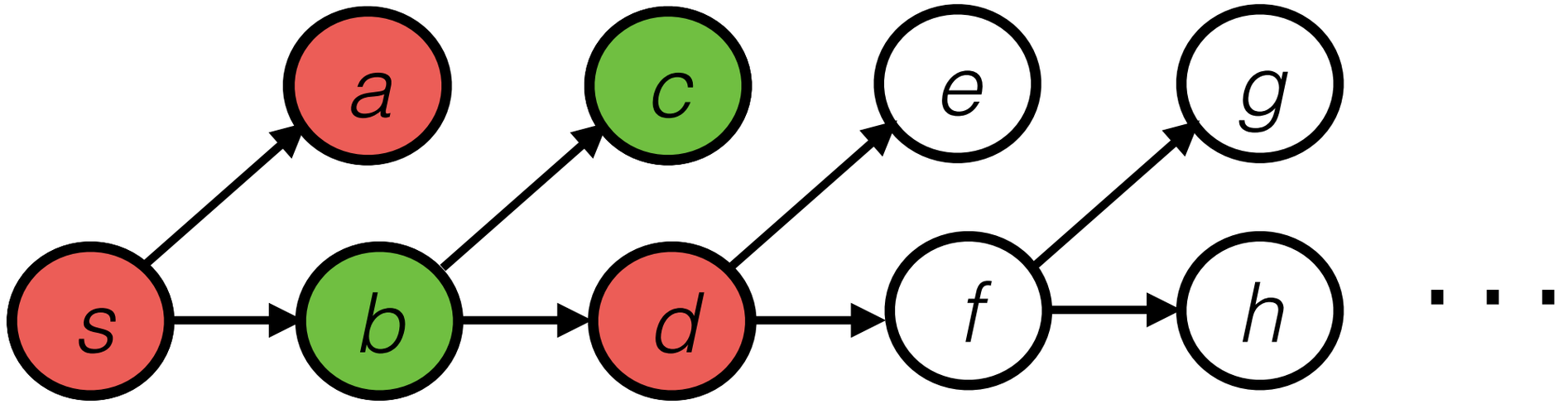
The granularity-control challenge

- The key tradeoff is between:
  - the cost to pay for migrating some chunk of work, and
  - the benefit of parallelizing the migrated work
- Migrate too often, it's too slow; too infrequently, it's too slow.
- Granularity control is a particular challenge for PDFS because, when you migrate a piece of frontier, you have little information about how much work you're giving away.

Example in favor of aggressively sharing work



Example against sharing work



Granularity control by batching vertices

- A *batch* is a small, fixed-capacity buffer that stores part of the frontier.
- In batching, each work-stealing queue stores pointers to batches of vertices.
- Idea: use batches to amortize the cost of migrating work.
- Previous state of the art for PDFS:
  - Batching PDFS (Cong et al 2008)
  - Parallel mark-sweep GC (Endo 1997 and Seibert 2010)
- No batching PDFS so far guarantees against worst-case behavior.

Our work

Central question:

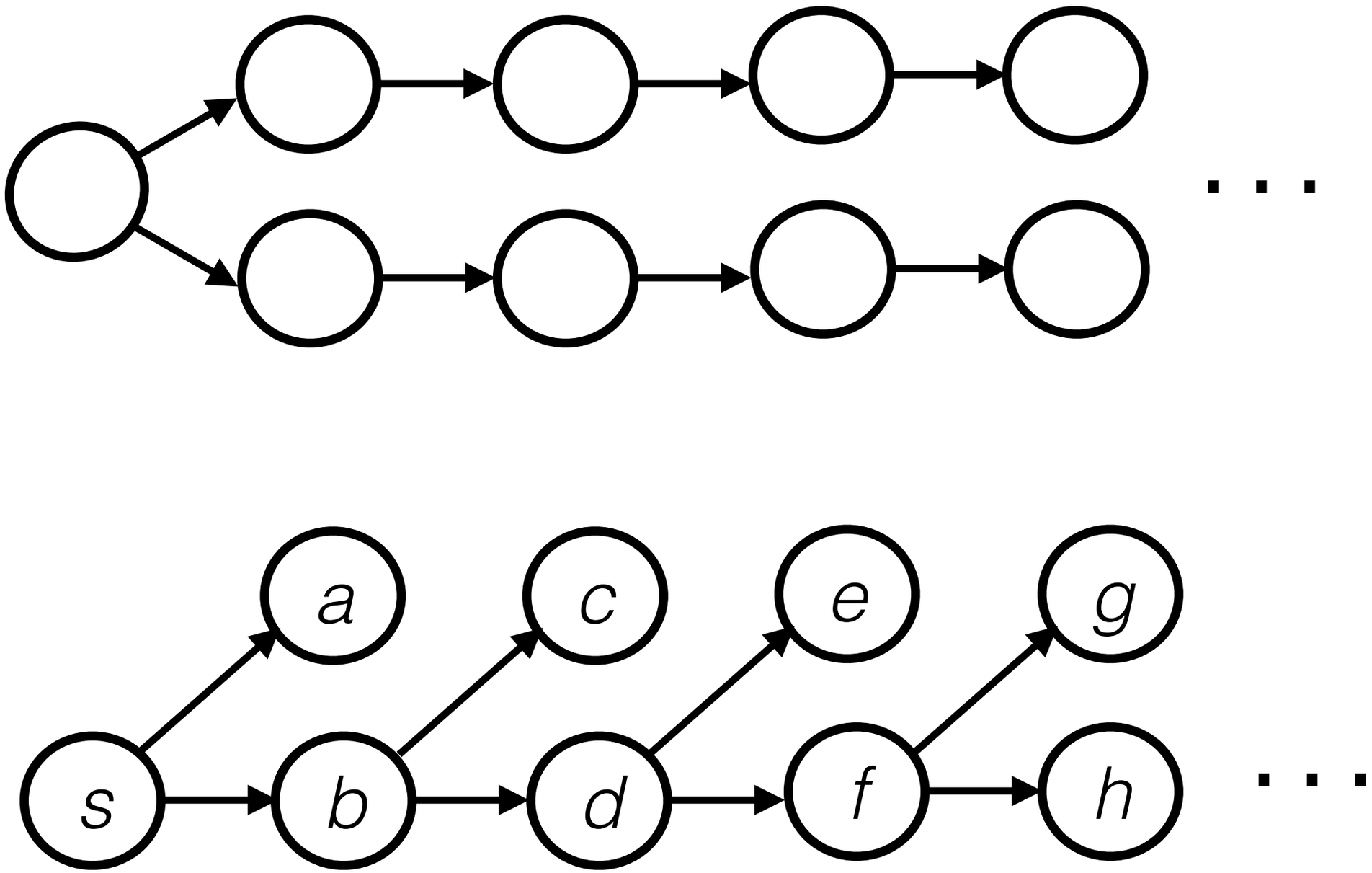
Can we bring to PDFS the analytical and empirical rigor that has been applied to PBFS, but keep the benefits of a DFS-like traversal?

- We present a new PDFS algorithm.
- In a realistic cost model:
  - We show that our PDFS is *work efficient*:
    - Running time on a single core is the same as that of serial DFS, up to constant factors.
  - We show that our PDFS is highly parallel.
- In experiments on a machine with 40 cores, we show the following.
  - Our PDFS outperforms alternative algorithms across many of a varied set of input graphs.
  - Our PDFS can exploit data locality like sequential DFS.

Our solution to granularity control

- Migration of work is realized by message passing.
  - Each core regularly polls the status of a cell (in RAM).
  - When core C_1 requests work from C_2, C_1 writes its ID into the cell owned by C_2.
  - Each core owns a private frontier.
- Our granularity control technique: when receiving a query, a core shares its frontier only if one of the following two conditions is met:
  - The frontier is larger than some fixed constant, K.
  - The core has treated at least K edges already
- The setting for K can be picked once based (solely) on the characteristics of the machine.

Why is our granularity-control technique effective?



Our PDFS algorithm

Tuning parameters:

- *K*: positive integer controlling the eagerness of work sharing
- *D*: positive integer controlling the frequency of polling

Each core does:

- if my frontier is empty
  - repeatedly query random cores until finding work
- else
  - handle an incoming request for work
  - process up to *D* edges:
    - for each edge ending at vertex *v*
      - if this core wins the race to claim *v*, push outgoing neighbors of *v* into the frontier
      - remove *v* from the frontier

To handle a work request, a core does:

- if frontier contains at least *K* edges or has at least two edges and has treated at least *K* edges since previously sending work:
  - transfer half of the local frontier to the frontier of the hungry core
- notify the hungry core

Analytical bounds

Theorem 1
The number of migrations is $3m/K$.

Shows that each work migration is amortized over at least $K/3$ edges.

Theorem 2
The total amount of work performed is linear in the size of the input graph.

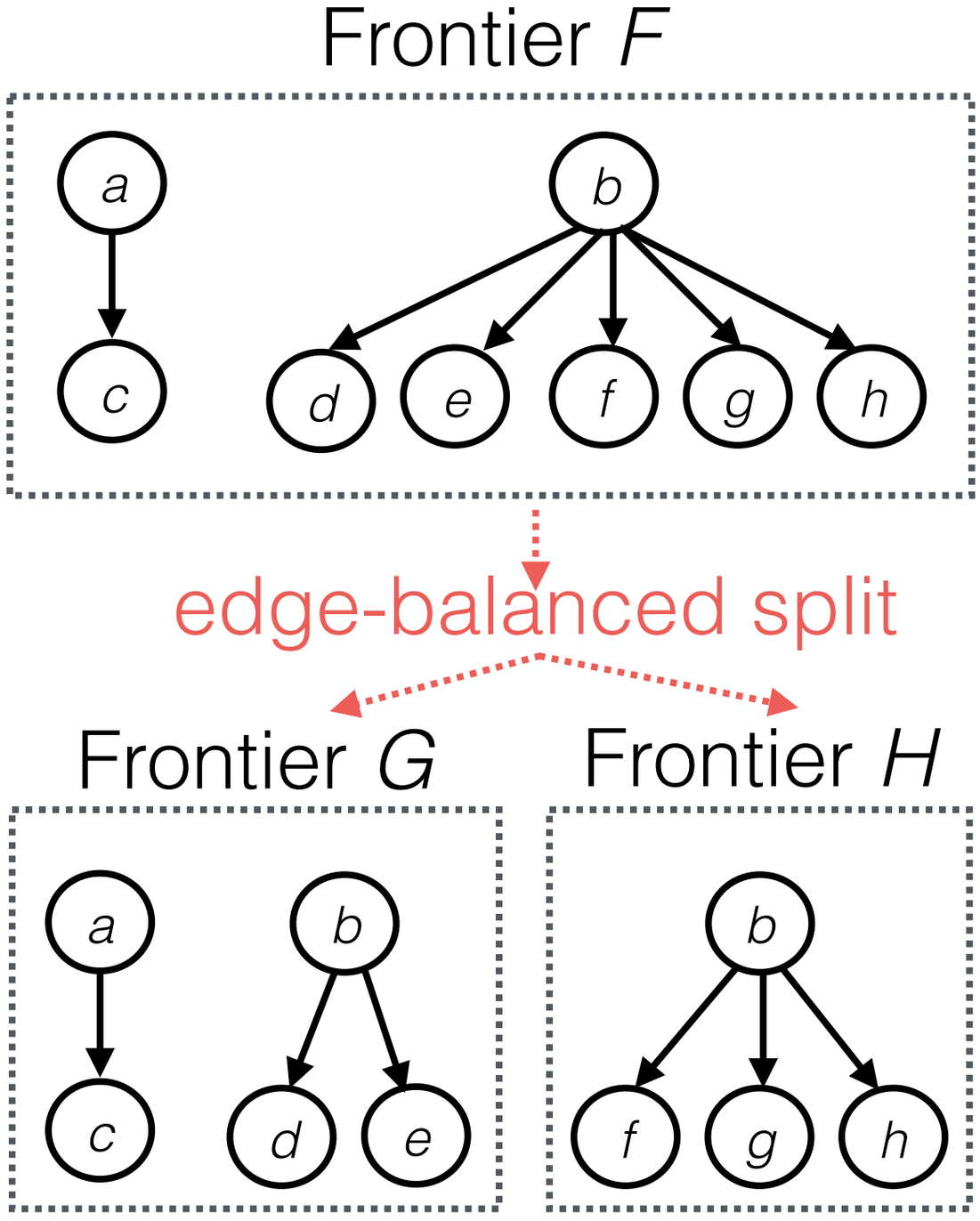
Shows that all polling and communication costs are well amortized.

Theorem 3
Each work query is matched by a response in $O(D + \log n)$ time.

Shows that the algorithm can achieve almost every opportunity for parallelism.

Our frontier data structure

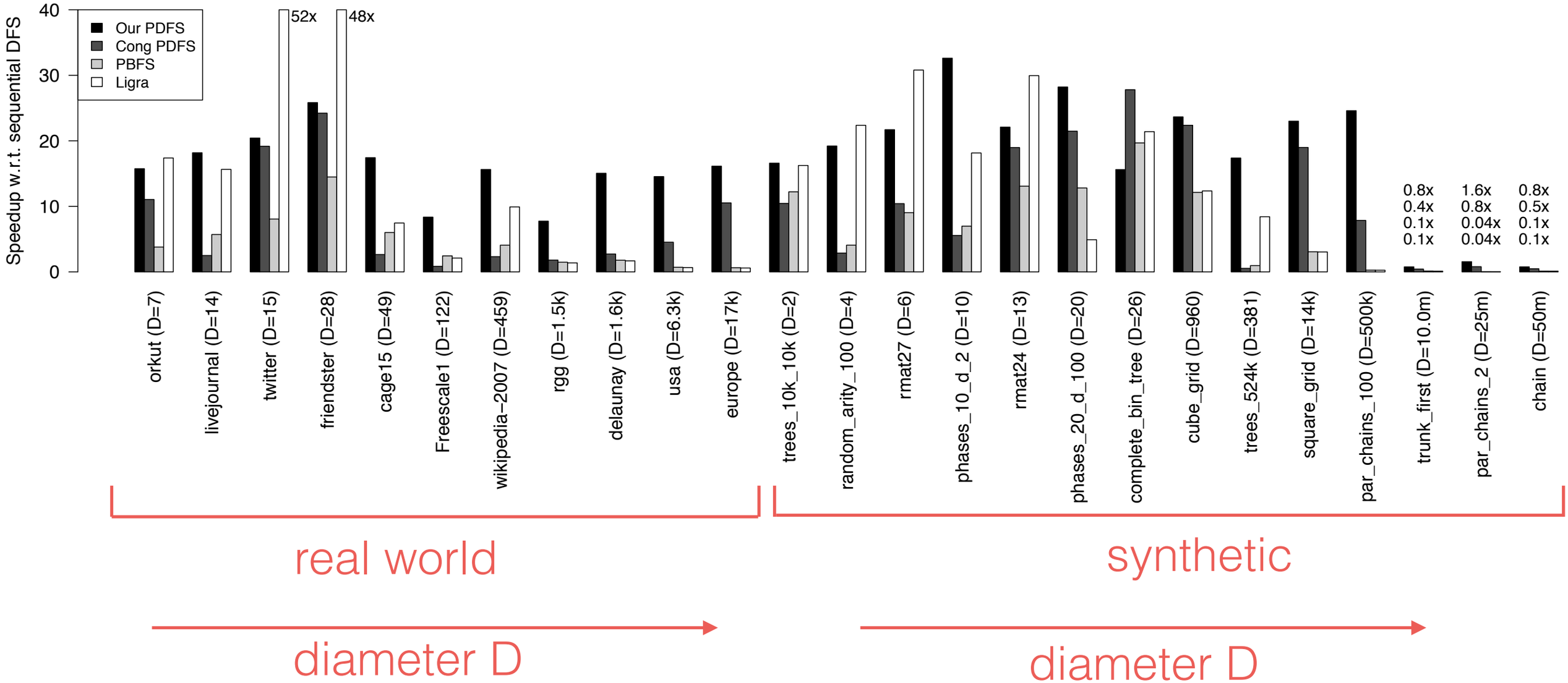
- It is based on our previous work on a chunked-tree data structure.
- It's a sequence data structure storing weighted items.
- It can
  - push/pop in constant time
  - split in half according to the weights of the items in logarithmic time.
- In the PDFS frontier, a weight represents the outdegree of a vertex.
- It enables:
  - rapidly migrating large chunks of frontier on the fly
  - efficiently parallelizing high-outdegree vertices



Experimental results

higher = better

- 40 Xeon cores @ 2.4Ghz
- 1 TB RAM



Related work

- PDFS
  - Batching PDFS (Cong et al 2008)
  - Parallel mark-sweep GC (Endo 1997 and Seibert 2010)
- PBFS
  - Work-efficient Parallel BFS (Leiserson & Schardl 2010)
  - Direction-optimizing BFS (Beamer et al 2012)
  - Ligra (Shun & Blelloch 2013)
- Hybrid PDFS/PBFS
  - KLA graph-processing framework (Harshvardhan et al 2014)

Summary

- We presented a new PDFS algorithm.
- Our results lift PDFS to a level of rigor similar to that of work-efficient PBFS.
- In our paper:
  - We show that PDFS exploits data locality as effectively as serial DFS.
- Our results show that PDFS performs well both in theory and practice.
- The results suggest that our PDFS may be useful as a component of other algorithms and graph-processing systems.

Oracle-guided versus PDFS-style granularity control

- Oracle-guided can do something PDFS cannot:
    - switch irrevocably to pure sequential code
        - it's desirable because compilers know how to optimize sequential code & because there is no polling overhead
- PDFS can do something oracle-guided cannot:
    - handle larger space of computations
        - oracle guided, just divide & conquer; PDFS-style, arbitrary DAGs,
        - enables parallel pipelining, for example
    - no need for complexity functions (or any such annotations)
- Characteristic difference:
    - oracle guided: amortize against future work
    - PDFS-style: amortize against past work