Scalability! But at what COST?

Frank McSherry, Michael Isard, Derek G. Murray

Presented by: Patrick Insinger

“You can have a second computer once you’ve shown you know how to use the first one.”

–Paul Barham

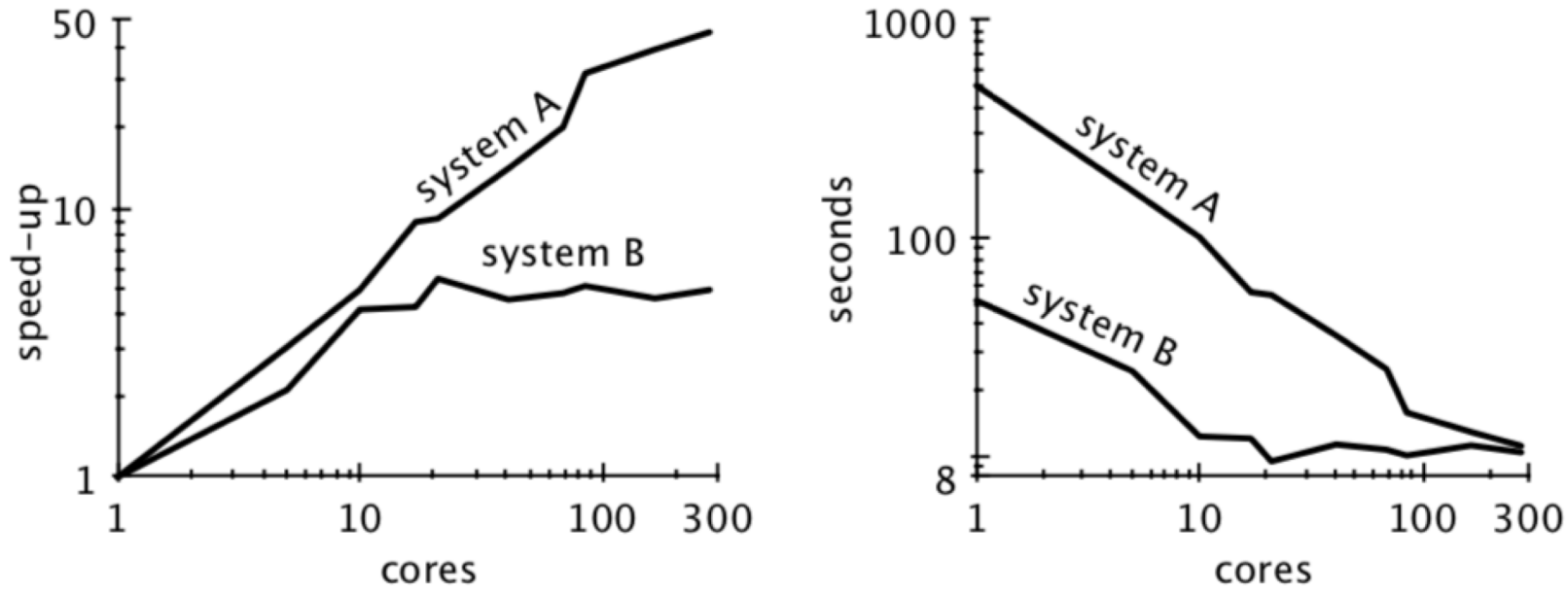


Figure 1: Scaling and performance measurements for a data-parallel algorithm, before (system A) and after (system B) a simple performance optimization. The unoptimized implementation "scales" far better, despite (or rather, because of) its poor performance.

While nearly all such publications detail their system's **impressive scalability**, few directly evaluate their absolute performance against reasonable benchmarks.

To what degree are these systems truly improving performance, as **opposed to parallelizing overheads that they themselves introduce**?

Idea: Evaluate COST

- COST = Configuration that outperforms the best single threaded implementation
  - Problem & data specific (e.g. the cost for solving PageRank on twitter_rv)
  - In this paper, configuration = # of cores
  - Somewhat formally: COST = core_count for which runtime of parallel system = runtime of single-threaded implementation
- Disclaimer
  - The comparisons are neither perfect nor always fair, but the conclusions are sufficiently dramatic that some concern must be raised

Graph Computation of Interest: PageRank

scalable system	cores	twitter	uk-2007-05
GraphChi [12]	2	3160s	6972s
Stratosphere [8]	16	2250s	-
X-Stream [21]	16	1488s	-
Spark [10]	128	857s	1759s
Giraph [10]	128	596s	1235s
GraphLab [10]	128	249s	833s
GraphX [10]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

Table 2: Reported elapsed times for 20 PageRank iterations, compared with measured times for single-threaded implementations from SSD and from RAM. GraphChi and X-Stream report times for 5 PageRank iterations, which we multiplied by four.

name	twitter_rv [13]	uk-2007-05 [5, 6]
nodes	41,652,230	105,896,555
edges	1,468,365,182	3,738,733,648
size	5.76GB	14.72GB

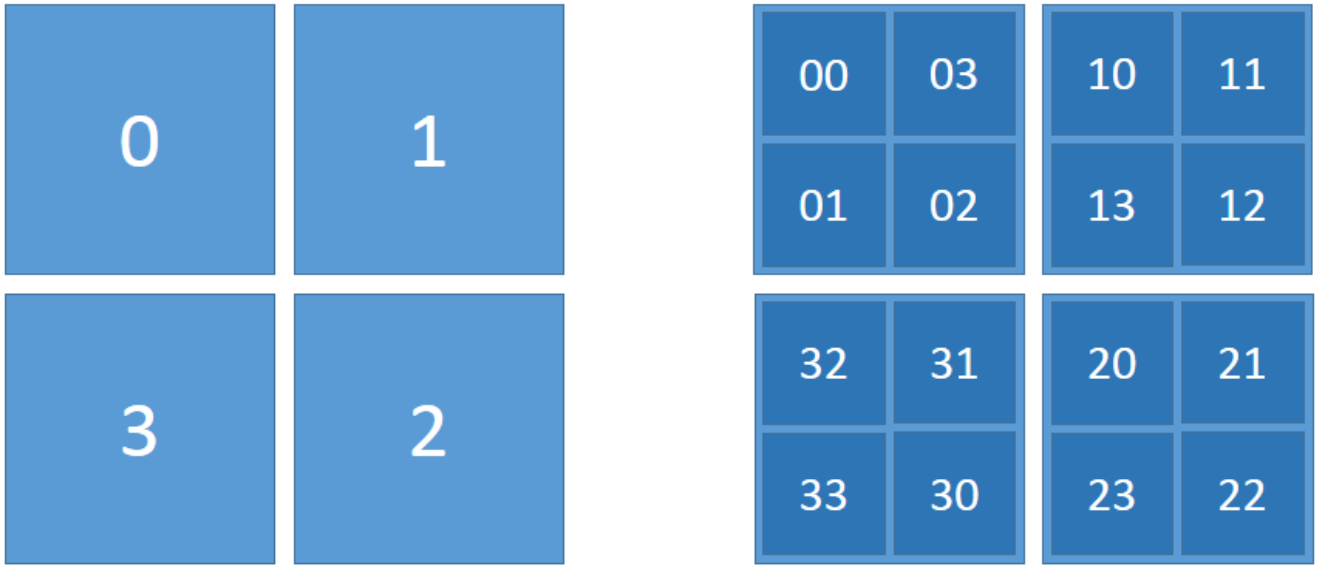
Table 1: The "twitter_rv" and "uk-2007-05" graphs.

```
fn PageRank20(graph: GraphIterator, alpha: f32) {
  let mut a = vec![0f32; graph.nodes()];
  let mut b = vec![0f32; graph.nodes()];
  let mut d = vec![0f32; graph.nodes()];

  graph.map_edges(|x, y| { d[x] += 1; });

  for iter in 0..20 {
    for i in 0..graph.nodes() {
      b[i] = alpha * a[i] / d[i];
      a[i] = 1f32 - alpha;
    }

    graph.map_edges(|x, y| { a[y] += b[x]; });
  }
}
```

Figure 2: Twenty PageRank iterations.

PageRank: Improved Single Threaded Performance

- Significant performance improvements by using Hilbert curve to order edges, taking advantage of cache locality



PageRank: Improved Single Threaded Performance

- Significant performance improvements by using Hilbert curve to order edges, taking advantage of cache locality
- Transformation takes 179 s for twitter, authors say this can be a performance win even if pre-processing is counted against runtime
  - 110 + 179 = 289 > 249

scalable system	cores	twitter	uk-2007-05
GraphLab	128	249s	833s
GraphX	128	419s	462s
Vertex order (SSD)	1	300s	651s
Vertex order (RAM)	1	275s	-
Hilbert order (SSD)	1	242s	256s
Hilbert order (RAM)	1	110s	-

Table 4: Reported elapsed times for 20 PageRank iterations, compared with measured times for single-threaded implementations from SSD and from RAM. The single-threaded times use identical algorithms, but with different edge orders.

PageRank: The **COST**

- Naiad has a COST of 16 cores on PageRank twitter
- GraphLab, not presented to the right, has a COST of 512 cores for PageRank twitter
- GraphX does not intersect the single-threaded measurement so it has unbounded COST

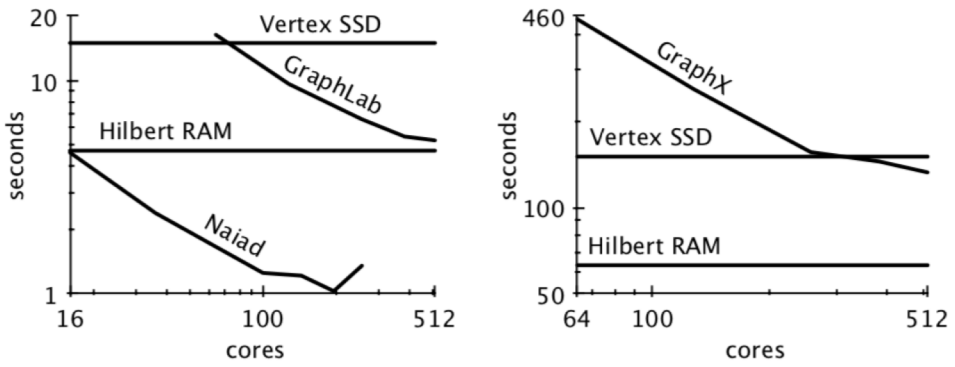


Figure 5: Published scaling measurements for PageRank on twitter_rv. The first plot is the time per warm iteration. The second plot is the time for ten iterations from a cold start. Horizontal lines are single-threaded measurements.

Graph Computation of Interest: Connected Components

scalable system	cores	twitter	uk-2007-05
Stratosphere [8]	16	950s	-
X-Stream [21]	16	1159s	-
Spark [10]	128	1784s	$\geq$ 8000s
Giraph [10]	128	200s	$\geq$ 8000s
GraphLab [10]	128	242s	714s
GraphX [10]	128	251s	800s
Single thread (SSD)	1	153s	417s

Table 3: Reported elapsed times for label propagation, compared with measured times for single-threaded label propagation from SSD.

```
fn LabelPropagation(graph: GraphIterator) {
  let mut label = (0..graph.nodes()).to_vec();
  let mut done = false;

  while !done {
    done = true;
    graph.map_edges(|x, y| {
      if label[x] != label[y] {
        done = false;
        label[x] = min(label[x], label[y]);
        label[y] = min(label[x], label[y]);
      }
    });
  }
}
```

Figure 3: Label propagation.

Connected Components: Single Thread Improvements

- The label propagation algorithm is used for graph connectivity not because it is a good algorithm, but because it fits within the "think like a vertex" computational model.

```
fn UnionFind(graph: GraphIterator) {
  let mut root = (0..graph.nodes()).to_vec();
  let mut rank = [0u8; graph.nodes()];

  graph.map_edges(|mut x, mut y| {
    while (x != root[x]) { x = root[x]; }
    while (y != root[y]) { y = root[y]; }
    if x != y {
      match rank[x].cmp(&rank[y]) {
        Less    => { root[x] = y; },
        Greater => { root[y] = x; },
        Equal   => { root[y] = x; rank[x] += 1; },
      }
    }
  });
}
```

Figure 4: Union-Find with weighted union.

Connected Components: The COST

- Naiad UF Slow: uses hash tables
  COST = 10 cores
- Naiad UF: uses arrays
  COST = 16 cores
- Tradeoff: hash tables don't require node IDs falling in a compact set of integers

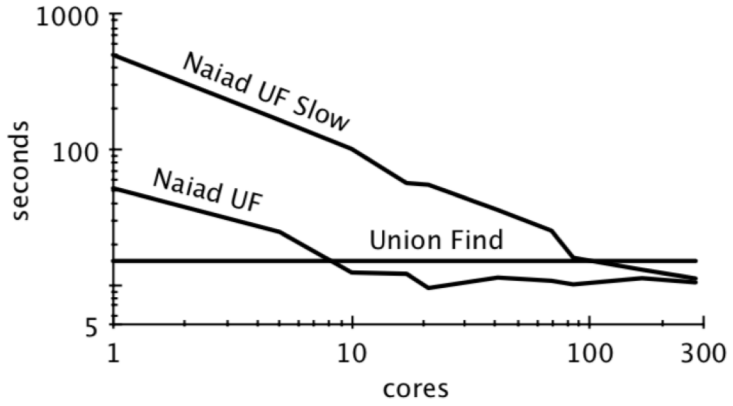


Figure 6: Two Naiad implementations of union find.

Conclusion: What drives **COST**

- Computation model restricts programs that can be expressed
- Target hardware reflects different tradeoffs
- Implementation may add overheads that a single thread doesn’t require

Conclusion: Some legitimate reasons for high COST

- Targeting a different set of problems
- Suited for a different deployment
- Prototype designed to assess components of a full system
- Integration with existing ecosystem
- High availability

Conclusion

- We stress that these problems lie **not** necessarily with **the systems** themselves, which may be improved with time, but **rather** with **the measurements** that the authors provide and the standard that reviewers and readers demand.