Fastware

DotNext, Saint Petersburg, Russia

Andrei Alexandrescu, Ph.D.
andrei@erdani.com

2018-04-22

The Art of Benchmarking

Mind Amdahl's Law

- Improvement from a component in a system is limited by the component's participation to the system

- Collect app profile data before optimizing
- Focus on optimizing the top time-spenders

Mind Amdahl's Law

Choose hotspots from whole application

Mind Lhadma's Law

Optimize hotspots outside whole application; profile again within application

Discussion

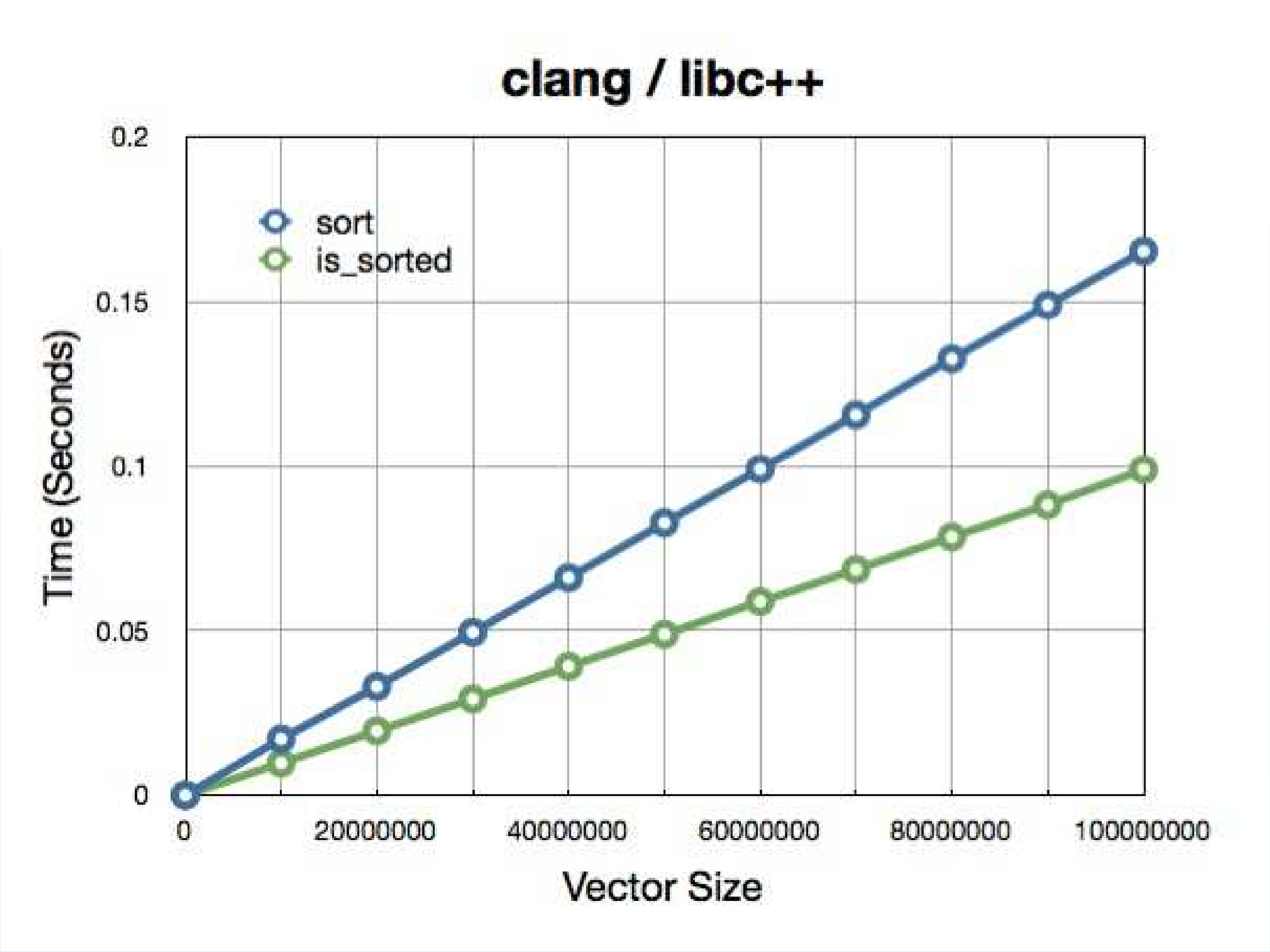
- Often one 20% speedup comes as nine 2% speedups
  - They'd be lost in the noise
- Edit/benchmark cycle slow on whole app

- Don't forget optimizations have global effects
  - Cache effects
  - Memory allocation/use
  - Branch predictor hogging
  - Lock contention

This Slide Does Not Exist

- Common benchmarking pitfalls (coworkers):
- Measuring speed of debug builds
- Different setup for baseline and measured
  - Sequencing: heap allocator
  - Warmth of cache, files, databases, DNS
- Including ancillary work in measurement
  - common: allocation, printing
- Procedural: change more than 1 thing at a time
- Optimize rare cases, pessimize others

Optimizing Rare Cases



Today’s Computing Architectures

- Extremely complex
- Trade reproducible performance for average speed
- Interrupts, multiprocessing are the norm
- Dynamic frequency control is becoming common
- Virtually impossible to get identical timings for experiments

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- "Fewer instructions = faster code"

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”
- “Computation is faster than data”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”
- “Computation is faster than data”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- "Fewer instructions = faster code"
- "Data is faster than computation"
- "Computation is faster than data"

- The only good intuition: *"I should time this."*

Deep Thought

Measuring gives you a leg up on experts who don't need to measure

Reliable Heuristics

- Math Is Good
  - Your single most important tool
  - Informs all algorithmic choices
  - Informs all implementation “tricks”
- Computers like boring/hate surprises
  - Branching
  - Dependencies
  - Indirect calls
- Indirect writes
- Strength hurts
- Small is beautiful
  - (Though that may change)

But before that. . .

Benchmarking Speed

- Goal: estimate the speed of some specific algorithm
- Usual procedure:
  - Choose experimental conditions
  - Run experiment multiple times
  - Take the average

Average Time

- Almost *never* a good idea!
  - "Bill Gates hops on a bus in Seattle. . . "
- Noise in computers:
  - Always additive
  - Small at human scale, eons for μbenchmarks
- Averaging captures noise

If Not Average, then what?

- Deterministic algorithms:
  - Take the distribution's *mode*
  - For most, mode is near the minimum!
- Randomized algorithms (including networking):
  - Throw away largest 5% times
  - Average over the rest
  - Or take thresholds à la p95

Replace Branches with Arithmetic

Replace Branches with Arithmetic

- Instead of a branch, integrate its result as a 0/1 value
- Less pressure on branch predictor
- Less failed speculation (power consumption)
- Fewer stalls

Replace Branches with Arithmetic

Task: swap minimums of 4 to the first quartile of an array

Baseline

```
static void min4(double[] p) {
  int n = p.Length;
  int i = 0, j = n / 4, k = n / 2, l = 3 * n / 4;
  for (; i < n / 4; ++i, ++j, ++k, ++l) {
    int m = p[i] <= p[j] ? i : j;
    if (p[k] < p[m]) m = k;
    if (p[l] < p[m]) m = l;
    Swap(ref p[i], ref p[m]);
  }
}
```

- Test on organpipe, random, random01, sorted, realdata

Pass 1: Sense & Sensibility

```
static void min4(double[] p) {
  int q = p.Length / 4;
  int i = 0, j = n / 4, k = n / 2, l = 3 * n / 4;
  for (; i < q; ++i, ++j, ++k, ++l) {
    int m = p[i] <= p[j] ? i : j;
    if (p[k] < p[m]) m = k;
    if (p[l] < p[m]) m = l;
    Swap(ref p[i], ref p[m]);
  }
}
```

- Same speed, smaller space → win

Pass 2: Reduce Dependencies

```
static void min4(double[] p) {
  int q = p.Length / 4;
  int i = 0, j = q, k = 2 * q, l = 3 * q;
  for (; i < q; ++i, ++j, ++k, ++l) {
    int m0 = p[i] <= p[j] ? i : j;
    int m1 = p[k] <= p[l] ? k : l;
    Swap(ref p[i], ref p[p[m0] <= p[m1] ? m0 : m1]);
  }
}
```

- Actually generates slightly larger code
- Same speed

Pass 3: One Induction Variable

```
static void min4(double[] p) {
  int q = p.Length / 4;
  for (int i = 0; i < q; ++i) {
    int m0 = p[i] <= p[i + q] ? i : i + q;
    int m1 = p[i + 2 * q] <= p[i + 3 * q] ?
      i + 2 * q : i + 3 * q;
    Swap(ref p[i], ref p[p[m0] <= p[m1] ? m0 : m1]);
  }
}
```

- Same speed

Pass 4: Get rid of multiplication by 3

```
static void min4(double[] p) {
  int q = p.Length / 4, q2 = q + q;
  for (int i = q; i < q2; ++i) {
    int m0 = p[i - q] < p[i] ? i - q : i;
    int m1 = p[i + q2] < p[i + q] ? i + q2 ? i + q;
    Swap(ref p[i - q], ref p[p[m0] <= p[m1] ? m0 : m1]);
  }
}
```

- Marginally (3%) faster on organpipe and sorted

Enter optional

- If only there were a way to optionally add a value. . .

```
// Returns: value if flag is true, 0 otherwise
static int optional(bool flag, int value) {
  return -Convert.ToInt32(flag) & value;
}
```

Pass 5: Replace branches with optional

```
static void min4(double[] p) {
  int q = p.Length / 4, q2 = q + q;
  for (int i = q; i < q2; ++i) {
    int m0 = i - optional(p[i - q] <= p[i], q);
    int m1 = i + q + optional(p[i + q2] < p[i + q], q);
    Swap(ref p[i - q], ref p[p[m0] <= p[m1] ? m0 : m1]);
  }
}
```

- Let’s measure this

Bingo!

– 16% slower on organpipe
– 18% slower on sorted
+ 23% faster on random01
+ 2.2x faster on random
+ 2.1x faster on real data

- Small loss on low-entropy data (why?) for huge wins on general data

Too much of a good thing (1/3)

```
static void min4(double[] p) {
  int q = p.Length / 4, q2 = q + q;
  for (int i = 0; i < q; ++i) {
    int m = i + optional(p[i + q] < p[i], q);
    m += optional(p[i + q2] < p[m], q);
    m += optional(p[i + q2 + q] < p[m], q);
    Swap(ref p[i], ref p[m]);
  }
}
```

- Branchless
- Yet smaller gains, larger loss
- Why?

Too much of a good thing (2/3)

```
// Returns: v1 if flag is true, v2 otherwise
static int ifelse(bool flag, int v1, int v2) {
  return (-Convert.ToInt32(flag) & v1) |
    ((Convert.ToInt32(flag) - 1) & v2);
}
```

- Branchless ternary operator

Too much of a good thing (3/3)

```
static void min4(double[] p) {
  int q = p.Length / 4, q2 = q + q;
  for (int i = q; i < q2; ++i) {
    int m0 = i - optional(p[i - q] < p[i], q);
    int m1 = i + q + optional(p[i + q2] < p[i + q], q);
    Swap(ref p[i], ref p[ifelse(p[m0] <= p[m1], m0, m1)]);
  }
}
```

- Slightly slower than our best
- Slightly larger code
- `ifelse` still useful elsewhere

Large Set Intersection

Motivation

- Fundamental CS algo present as building block in:
  - Dot (scalar) product
  - Inverted index lookup: Given a few words, what are the most relevant documents?
  - Common friends of several people
  - Database queries (all join operations)
- Prediction: this specialized research area will become as common knowledge as e.g. sort

Basics

- `Intersect(a1, a2, target)` completes in $O(l_1 + l_2)$ time
- Assumes both inputs sorted
- Approach: compare `a1[i1]` and `a2[i2]`
  - If <, `++i1`
  - If >, `++i2`
  - Else, output and increment both

- How do we make it faster?

Basic implementation

```
int Intersect(double[] a1, double[] a2, double[] t) {
  if (a1.Length == 0 || a2.Length == 0) return 0;
  int i1 = 0, i2 = 0, i = 0;
  for (;;)
    if (a1[i1] < a2[i2]) {
      if (++i1 == a1.Length) break;
    } else if (a2[i2] < a1[i1]) {
      if (++i2 == a2.Length) break;
    } else {
      t[i++] = a1[i1];
      if (++i1 == a1.Length || ++i2 == a2.Length)
        break;
    }
  return i;
}
```

Discussion

- + Universal pattern applicable to search, scalar product etc.
- + Works with forward iteration (streaming)
- + Works well for identical/almost identical sets
- − Works badly for highly different sets
  - Sets of very different sizes
  - Sets of very different distributions

Attempt at improvement

```
int Intersect(double[] a1, double[] a2, double[] t) {
  int i1 = 0, i = 0;
  for (; i1 != a1.Length; ++i1) {
    auto m = Bsearch(a2, a1[i1]);
    if (m == a2.Length) continue;
    --m;
    if (!(a2[m] < a1[i1]))
      t[i++] = a1[i1];
  }
  return i;
}
```

Cpt. Obvious: Reduce size searched

```
int Intersect(double[] a1, double[] a2, double[] t) {
  int i1 = 0, i2 = 0, i = 0;
  for (; i1 != a1.Length; ++i1) {
    auto m = Bsearch(a2, i2, a2.length, a1[i1]);
    if (m == i2) continue;
    if (!(a2[m - 1] < a1[i1]))
      t[i++] = a1[i1];
    i2 = m + 1;
  }
  return i;
}
```

Improvement?

- $\pm$ Complexity is $O(l_1 \log l_2)$
- $-$ Works well only with random access
- $+$ Works great for large differences in length/stats
- $-$ Terrible worst cases
  - Identical inputs
  - `a1` is a short affix of `a2`
  - These are common!

Solution: Galloping Search

```
int GBsearch(double[] a, int i, int j, double v) {
  for (int step = 1;; step *= 2) {
    if (i >= j) break;
    if (a[i] > v) break;
    if (i + step >= j)
      return Bsearch(a, i + 1, j, v);
    if (a[i + step] > v)
      return Bsearch(a, i + 1, i + step, v);
    i += step + 1;
  }
  return i;
}
```

Best of both worlds

- Replace `BSearch` with `GBSearch` in `Intersect`
- Start search with left side, great if inputs almost identical
- Continue with exponentially-increasing steps
- When overshooting finish with classic binary search
- Supports assumption that searched value is most likely left-prone in each search
- Balances cache friendliness with fast search
- Complexity same as binary search!

Scaling Up

- In practice we have n, not 2, sets to compare
- How do we intersect all simultaneously?
- (Naïvely: intersect the running result with each)

- SvS
- Small Adaptive
- Baeza-Yates
- Sequential, Random Sequential
- Overview: `www.cs.utoronto.ca/~tl/papers/fiats.pdf`

Destroy!