Technische
Universität
Braunschweig

# Algorithm Engineering 2026: Sheet 01

Rouven Kniep, April 23, 2026

# Consumer Producer

- Scenario: 60 Producers, 1 Consumer
- Production rate: 1 per 2 weeks
- Consumption rate: 1 per 20-30 minutes
- $\implies$ Consumer Workload: 60 hours/month
- $\implies$ Consumer Overload!

# Consumer Producer (alternative)

- Scenario: 60 Producers, 1 Consumer
- **Optimization: pair producers**
- Production rate: 1 per 2 weeks
- Consumption rate: 1 per 20-30 minutes
- $\Longrightarrow$ Consumer Workload: **30 hours/month**
- $\Longrightarrow$ **Consumer happy!**
- $\Longrightarrow$ **Producers practice collaboration**

# In other news......

**GROUP UP!**

**or i’ll do it**

**Email me with your partner before sunday night.**

**Otherwise: Skill-based matchmaking.**

# Branch Prediction: correct prediction

| PC | instruction | Cycle 1, 2, ... | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0x20 | add r3 = r2+r1 | IF | ID | EX | M | WB | | | | |
| 0x24 | bge r3 >= r4, 0x8 | | IF | ID | EX | M | WB | | | |
| 0x32 | load r5[r3] | | | IF | ID | EX | M | WB | | |
| 0x36 | load r6[r1] | | | | IF | ID | EX | M | WB | |
| 0x40 | call turn() | | | | | IF | ID | EX | M | WB |

branch correctly predicted, no penalty

# Branch Prediction: misprediction penalty

| PC | instruction | Cycle 1, 2, ... | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x20 | add r3 = r2+r1 | IF | ID | EX | M | WB | | | | | | |
| 0x24 | blt r3 < r4, 0x8 | | IF | ID | EX | M | WB | | | | | |
| 0x28 | sub r3 = r3 - r4 | | | IF | ID | - | | | | | | |
| 0x32 | load r5[r3] | | | | IF | - | | | | | | |
| 0x32 | load r5[r3] | | | | | IF | ID | EX | M | WB | | |
| 0x36 | load r6[r1] | | | | | | IF | ID | EX | M | WB | |
| 0x40 | call turn() | | | | | | | IF | ID | EX | M | WB |

misprediction! first wrong instruction in Cycle 3, correct in Cycle 5 $\Longrightarrow$ 2 cycles penalty

Real CPUs: AMD Zen4 $\approx$13; Intel Lion Cove $\approx$14; ARM Oryon $\approx$13; . . .

# CMOV in assembly

`cmovae r3, r4` = overwrite `r3` with `r4` if "above or equal" flag is set

without CMOV: branch!

```
0x20: add r3 = r2+r1
0x24: blt r3 < r4, +0x8
0x28: sub r3 = r3-r4
0x32: load r5[r3]
```

with CMOV: branchless

```
0x20: add r3 = r2+r1
0x24: sub r4 = r3-r4
0x28: cmovae r3, r4,
0x32: load r5[r3]
```

# 2a: Branch Misses

```
std::size_t ax = i + j;
if(i + j >= n) {
  ax -= n;
}
std::int64_t a_i = a[ax];
std::int64_t b_j = b[j];
auto t1 = turn({x, y}, {a_i, b_j}, {b_j, a_i});
if(t1 >= 0) {
  auto t2 = turn({x, y}, {-b_j, -a_i}, {-a_i, -b_j});
  if(t2 >= 0) {
    result += c[(a_i + b_j) % m] * (t1 + t2 + 1);
  }
}
```

# 2a: State (dirty data)

```
sudo perf stat ./Release/problem_program_a 16384

 8.944.560.992      cpu_atom/instructions/          #    1,52  insn per cycl
25.285.677.974      cpu_core/instructions/          #    1,64  insn per cycl
 1.165.887.659      cpu_atom/branches/              #  285,789 M/sec
 2.868.382.421      cpu_core/branches/              #  703,113 M/sec
    69.752.046      cpu_atom/branch-misses/         #    5,98% of all branch
   272.508.200      cpu_core/branch-misses/         #    9,50% of all branch
```

...but inconsistent! why? - multiple core types and switching between them

# 2a: State

```
taskset -c 1 sudo perf stat ./Release/problem_program_a 16384

25.215.879.289      cpu_core/instructions/          #    1,61  insn per cycl
 2.859.518.614      cpu_core/branches/              #  838,477 M/sec
   271.903.112      cpu_core/branch-misses/         #    9,51% of all branch
```

# 2a: first if is a CMOV

```
64                    std::size_t ax = i + j;
65                    if(i + j >= n) {
   0x15f0 <+80>:  4c 39 ef              cmp    %r13,%rdi
   0x15f3 <+83>:  48 8d 04 1e           lea    (%rsi,%rbx,1),%rax

69                    std::int64_t b_j = b[j];
   0x15f7 <+87>:  4d 8b 14 f4           mov    (%r12,%rsi,8),%r10

66                        ax -= n;
   0x15fb <+91>:  48 0f 42 c7           cmovb  %rdi,%rax
```

# 2a: nested if.. [1]

```
auto t1 = turn({x, y}, {a_i, b_j}, {b_j, a_i});
if(t1 >= 0) {
  auto t2 = turn({x, y}, {-b_j, -a_i}, {-a_i, -b_j});
  if(t2 >= 0) {
    result += c[(a_i + b_j) % m] * (t1 + t2 + 1);
}}
```

| | |
|---|---|
| 1.25 | seconds |
| 2.86B | branches |
| 271M | misses |
| 9.50% | |

# 2a: nested if.. [2]

```
auto t1 = turn({x, y}, {a_i, b_j}, {b_j, a_i});
auto t2 = turn({x, y}, {-b_j, -a_i}, {-a_i, -b_j});
if(t1 >= 0 && t2 >= 0) {
  result += c[(a_i + b_j) % m] * (t1 + t2 + 1);
}
```

| | |
|---|---|
| 0.35 | seconds |
| 2.72B | branches |
| 137M | misses |
| 5.04% | misses |

- idea: second turn "cheap" (4 Sub + 2 Mul + no loads)
- compiler cannot safely move calls

# 2a: nested if..? [3]

```
auto t1 = turn({x, y}, {a_i, b_j}, {b_j, a_i});
auto t2 = turn({x, y}, {-b_j, -a_i}, {-a_i, -b_j});
if((t1 | t2) & std::numeric_limits<std::int64_t>::min()) {
  continue;
}
result += c[(a_i + b_j) % m] * (t1 + t2 + 1);
```

| | |
|---|---|
| 0.35 | seconds |
| 2.72B | branches |
| 137M | misses |
| 5.04% | |

- help compiler with if-merge => not required
- last branch: predictable (take-rate 2.86e-06)

# 2a: function calls? [3]

```
inline std::int64_t turn(std::array<std::int64_t, 2> p1,
                         std::array<std::int64_t, 2> p2,
                         std::array<std::int64_t, 2> p3) {
  return (p2[0] - p1[0]) * (p3[1] - p1[1]) -
  (p2[1] - p1[1]) * (p3[0] - p1[0]);
}
```

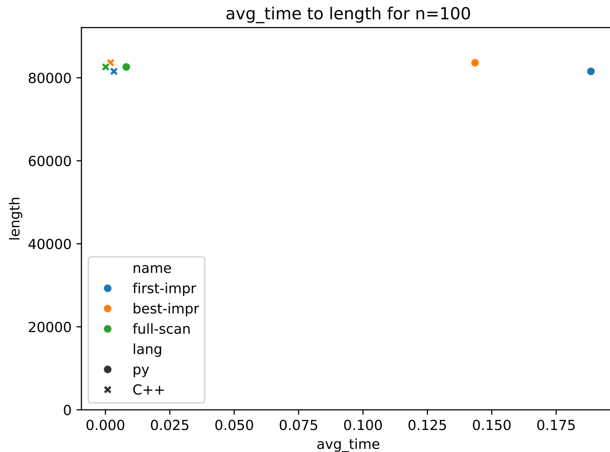
| | |
|---|---|
| 0.35 | seconds |
| 2.72B | branches |
| 137M | misses |
| 5.04% | misses |

- "force-inline" function
- annoying to debug, slower to compile, larger binary
- $\Longrightarrow$ dirty "fix"
- . . . but effective

# 2b: data structures [1] - extensions

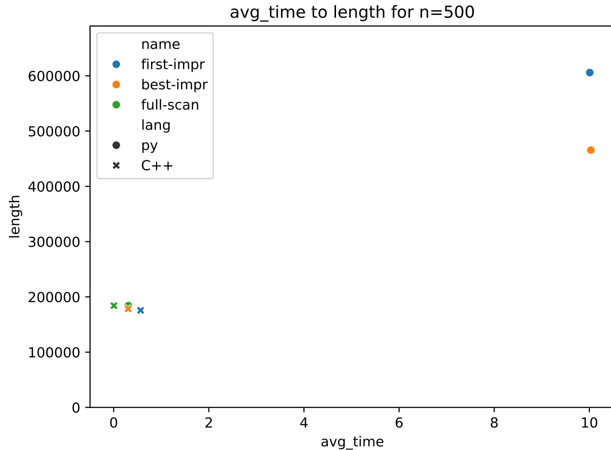
| candidates<br>graph | set<br>adj. list | vector<br>adj. list | vector<bool><br>adj. matrix | vector<br>adj.matrix |
|---|---|---|---|---|
| graph1 | 0.15 s | 0.09 s | 0.13 s | $\leqslant$0.01 s |
| graph2 | 0.17 s | 0.10 s | 0.14 s | $\leqslant$0.01 s |
| graph3 | 0.28 s | 0.13 s | 0.12 s | $\leqslant$0.01 s |
| graph4 | 0.44 s | 0.19 s | 0.11 s | 0.01 s |
| graph5 | 0.82 s | 0.15 s | 0.12 s | 0.04 s |
| graph6 | 1.76 s | 0.84 s | 0.12 s | 0.12 s |
| graph7 | 2.43 s | 1.25 s | 0.12 s | 0.20 s |
| graph8 | 5.30 s | 2.90 s | 0.12 s | 0.35 s |
| graph9 | 4.62 s | 3.02 s | 1.40 s | 0.11 s |

# 3: twoopt + analysis



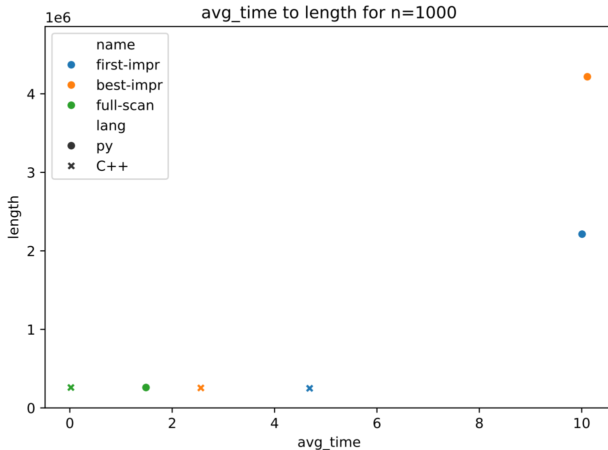

- python MUCH slower
- full-scan: fastest
- first-impr: best
  BUT slowest!

# 3: twoopt + analysis



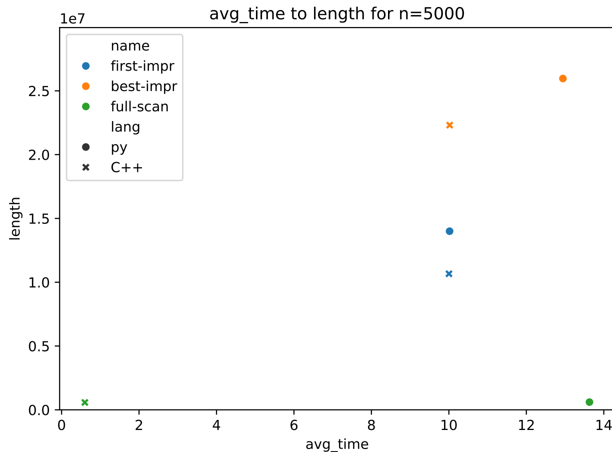

- python MUCH slower
- full-scan: fastest
- first-impr: best
  BUT slowest!
- timeout = horrible

# 3: twoopt + analysis



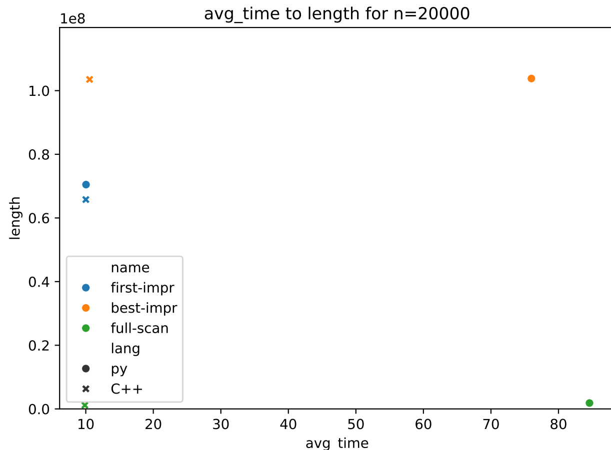

- python MUCH slower
- full-scan: fastest by a LOT
- first-impr: best BUT slowest!
- timeout = horrible

# 3: twoopt + analysis



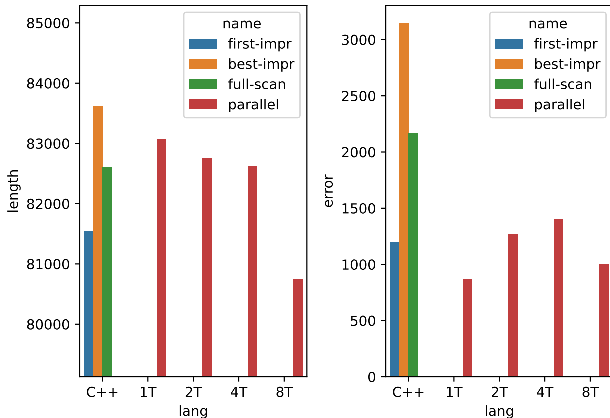

- python MUCH slower
- full-scan: fastest by a LOT
- first-impr: best BUT slowest!
- timeout = horrible

# 3: twoopt + analysis



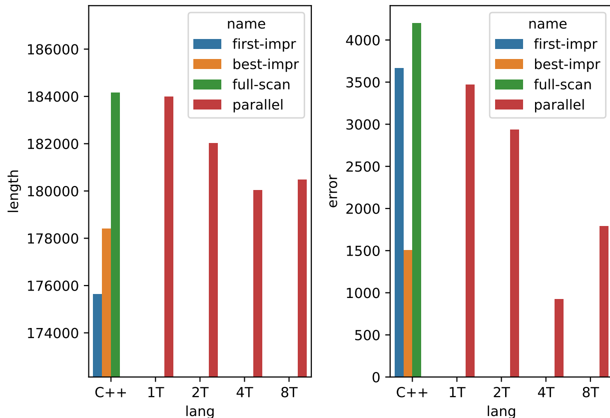

- python MUCH slower
- full-scan: fastest by a LOT
- first-impr: best BUT slowest!
- timeout = horrible
- insufficient timeout on python fullscan + python best-impr

# 3: twoopt + analysis [2]



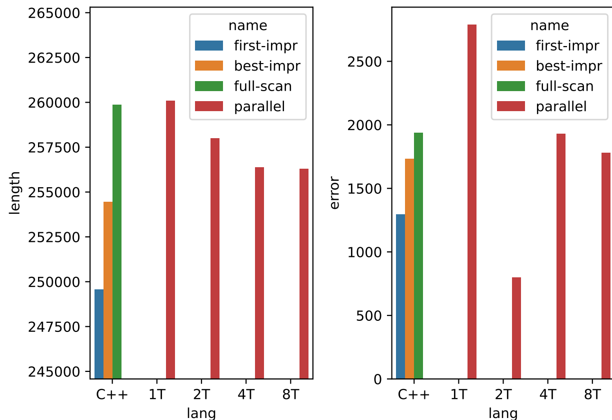

- idea: run fastest algo multiple times
- use randomness for (usually) good solutions

# 3: twoopt + analysis [2]



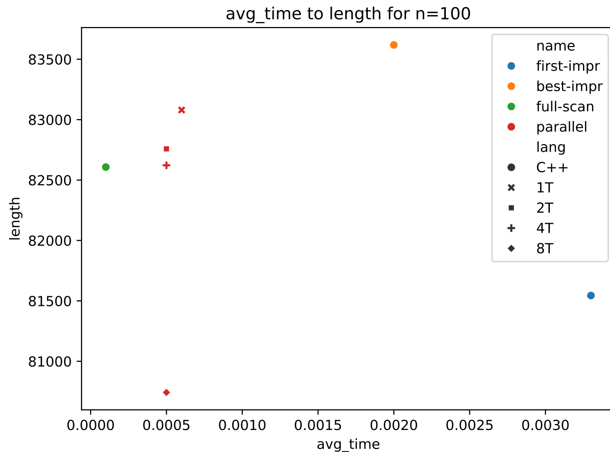

- idea: run fastest algo multiple times
- use randomness for (usually) good solutions

# 3: twoopt + analysis [2]



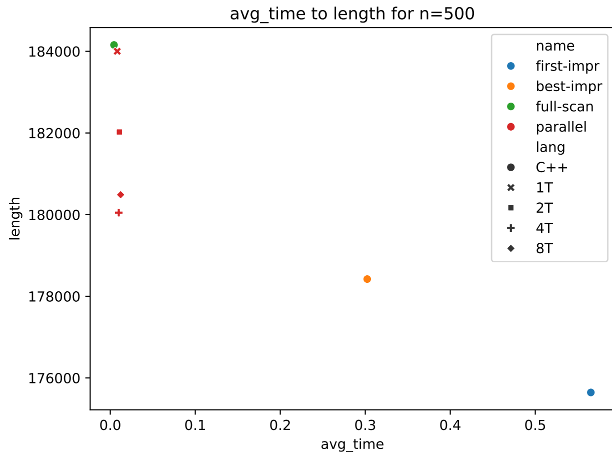

- idea: run fastest algo multiple times
- use randomness for (usually) good solutions
- more threads = better solutions (usually)
- more threads = less deviation (usually)

# 3: twoopt + analysis [3]



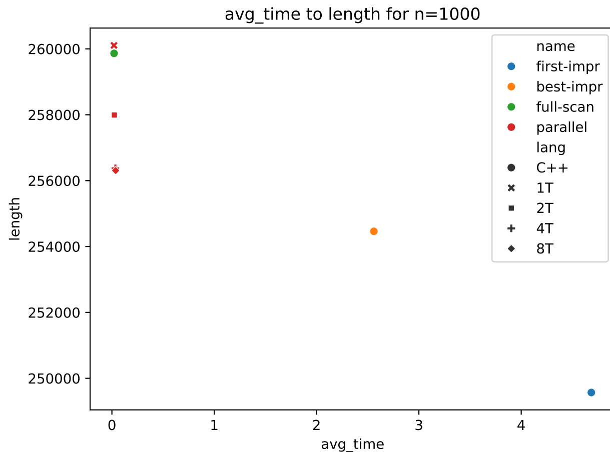

- slower than full-scan
  $\Longrightarrow$ threading overhead
- faster than others

# 3: twoopt + analysis [3]



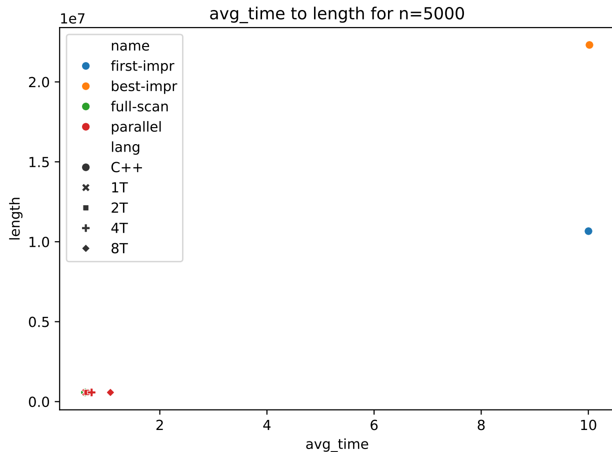

- slower than full-scan
  $\Longrightarrow$ threading overhead
- faster than others
- threading overhead fades with higher $n$

# 3: twoopt + analysis [3]



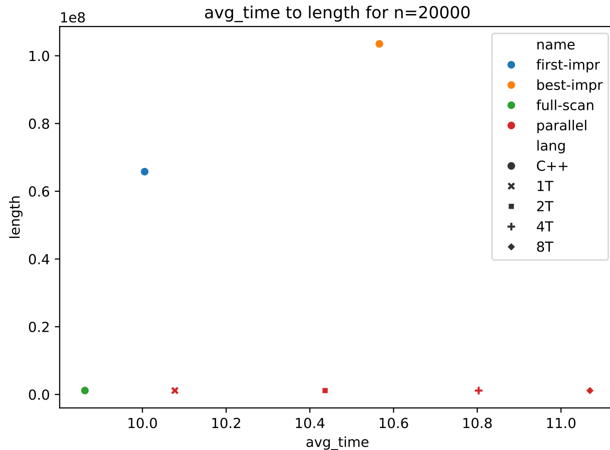

- slower than full-scan
  $\implies$ threading overhead
- faster than others
- threading overhead fades with higher $n$
- solution quality improved

# 3: twoopt + analysis [3]




- slower than full-scan
  $\Longrightarrow$ threading overhead
- faster than others
- threading overhead fades with higher $n$
- solution quality improved

# 3: twoopt + analysis [3]




- slower than full-scan
  $\Longrightarrow$ threading overhead
- faster than others
- threading overhead fades with higher $n$
- solution quality improved...
- ... unless in timeout