Lecture 20: Dynamic analysis & testing III

CS 5150, Spring 2022

Administrative announcements

- Report #4 due Friday
  - If you have deliverables to demonstrate or would benefit from client feedback, be sure to schedule a meeting
- In-class exam next Thursday
  - Sample questions will be shared this week
  - Multiple-choice, short-answer, diagraming

Lecture goals

- Leverage continuous integration to boost productivity by "shifting left"
- Leverage dynamic analysis tools to find bugs
- Evaluate application performance

Continuous integration ("CI")

- Build and test whole systems regularly
  - Discover issues earlier
  - Reduce integration pain through automation and isolation of issues
  - Test beyond single developer's resources
  - Eliminate reliance on developers' discipline
  - Continuously monitor readiness of code
- Applies to both development and release
  - Continuous build+test
  - Continuous delivery

CI decisions

- *How* to compose systems along release workflow
- *Which* tests to run *when* along release workflow
- Typical setup
  - Pre-submit test suite gates all merges
    - Compilation and fast tests relevant to affected code
  - Post-submit test suite verifies subset of commits on trunk
    - Contains larger, more integrated tests
    - Blesses commits that pass as "green"
  - Release promotion pipeline verifies candidates for release
    - Contains even larger tests, may require dedicated resources

Automation, speed, & infrastructure

- Builds, tests, and deployment must be automated and reliable
  - Ideally completely reproducible
- Most steps must be fast to avoid impeding productivity
  - Cache build products
  - Skip unaffected tests
  - Parallelize & invest in compute resources
- Benefits from tooling
  - Integration with version control and code review
    - Pre-merge and pre-release gates
    - "Last-known-good" branch (new work should branch from here, not trunk)
  - Bisect breakages
  - Log all results
  - Automatically rerun flaky tests

Multi-system CI

- Without monorepo, need to assemble system from several asynchronously-versioned repositories
- Large integration tests can't check every revision/combination
- Objective: identify "configurations" (revision combinations) suitable for promotion (larger-scale testing, release)

Dynamic analysis

Common dynamic analysis tools

- Coverage
- Debuggers
- Memory checkers
- Sanitizers
- Profilers

Debugging demo

1. Witness test failure
2. Understand testcase
3. No crash? Check for memory errors (`valgrind`)
4. Set breakpoint, run in debugger, explore stack
5. Already borked? Break earlier and try again, *or* use `rr` to run backwards!

- `bt`: Show stack trace
- `frame` *<n>*: Change stack frame
- `info locals`: Show local vars
  - `info args`: Show arguments
- `p` *<expr>*: Evaluate and print
- `b`: Set breakpoint
- `c`: Continue

- `reverse-cont`: Run in reverse

Fuzz testing

- Give program random input, look for crashes, assertion violations
- Increased in popularity in 2010s; very effective at finding security vulnerabilities
- Can be enhanced with coverage feedback
  - Use genetic algorithms, neural networks to construct input that exercises particular branches

What is a performance bug?

Avoid premature optimization!

- Does not meet deadlines / satisfy SLA
- Responsiveness, smoothness do not meet requirements
  - 100 ms: GUI
  - 15-30 ms: Animation (30-60 fps)
  - 10 ms: MIDI, VR
- Unexpected slowdown for certain inputs / DoS vulnerability
- Performance regression (gradual and acute degradation)
- Performance variability across platforms
- Sub-optimal throughput for HPC

Performance testing challenges

- How much room for improvement is there?
  - Amdahl's law: Limits to speedup from parallelization, local optimization
  - Roofline analysis: Do you expect to be limited by bandwidth or compute?
- Is slowdown localized, dispersed, or emergent?
- Getting reliable measurements is difficult
  - Inconsistency, load dependency, JIT compilation, non-representative datasets, intrusive tooling
  - Average case vs. worst case, tail metrics
  - Tension between latency and bandwidth

Latency vs. throughput

- Latency: Duration between a single trigger and the system's response
  - "Tail latency" (e.g. 95th percentile under a specified load) is more important than average

- Throughput: Time it takes to processes a fixed amount of work
  - Often a function of workload
    - Typically throughput increases with workload size up to a saturation point
  - Reduce overhead with batching
    - Typically at expense of latency

Poll: PollEv.com/cs5150

Consider adding new elements to a sorted list (initial size *N*) while maintaining sorted order.

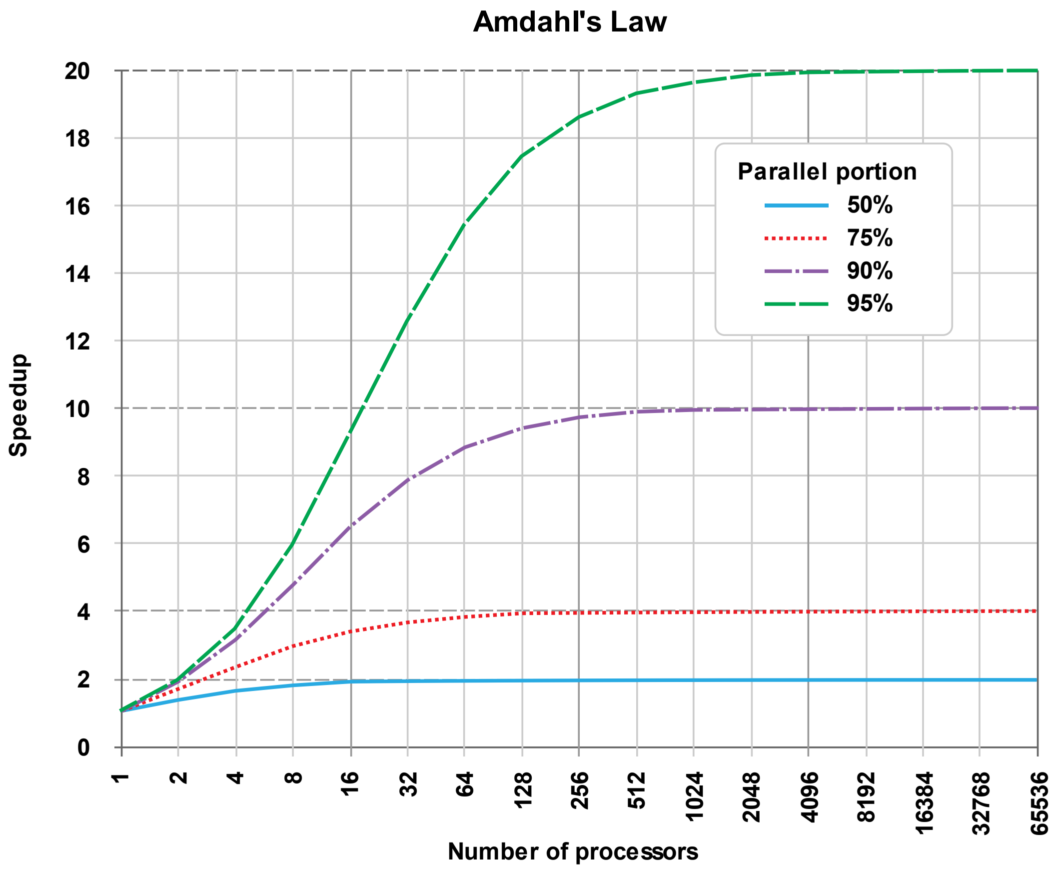
Scenario A: Elements are inserted into their proper position one at a time.

Scenario B: All elements are appended to the list, then the whole list is sorted (comparison sort).

Amdahl's Law

- Speedup: $S = T\_before / T\_after$
- Identify portion p of runtime cost amenable to optimization
  - $T\_before = p*T + (1 - p)*T$
- Let s be speedup of optimization on this portion
  - Example: $s = 10$ for parallelizing on a 10-core machine
  - Often interested in limit as $s \to \infty$
- $T\_after = p*T/s + (1 - p)*T$
- $S(s) = 1/(1 - p + p/s)$
- **$S \to 1/(1 - p)$**

Amdahl's Law implications



Poll: PollEv.com/cs5150

You use a text search application to look for all occurrences of a keyword in all the files of a large source code repository.

Using a single core, half of the time is spent reading files and looking for the keyword, and half the time is spent formatting and printing a sorted summary of the results to the console.

What is the maximum speedup that could be achieved by distributing the *embarrassingly parallel* work across multiple cores/nodes?