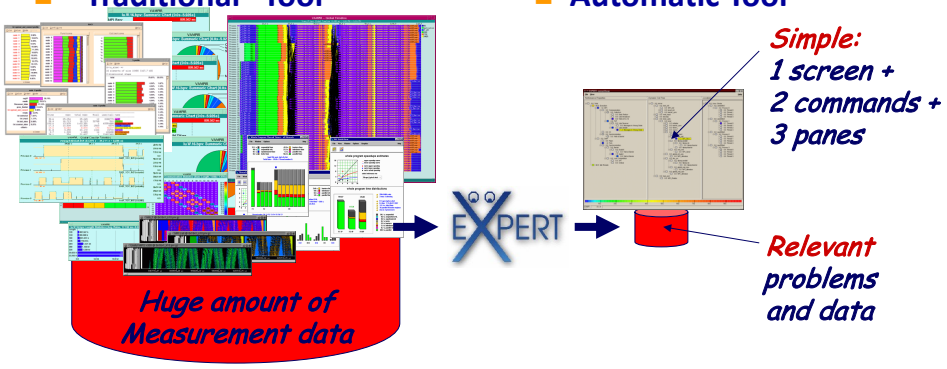
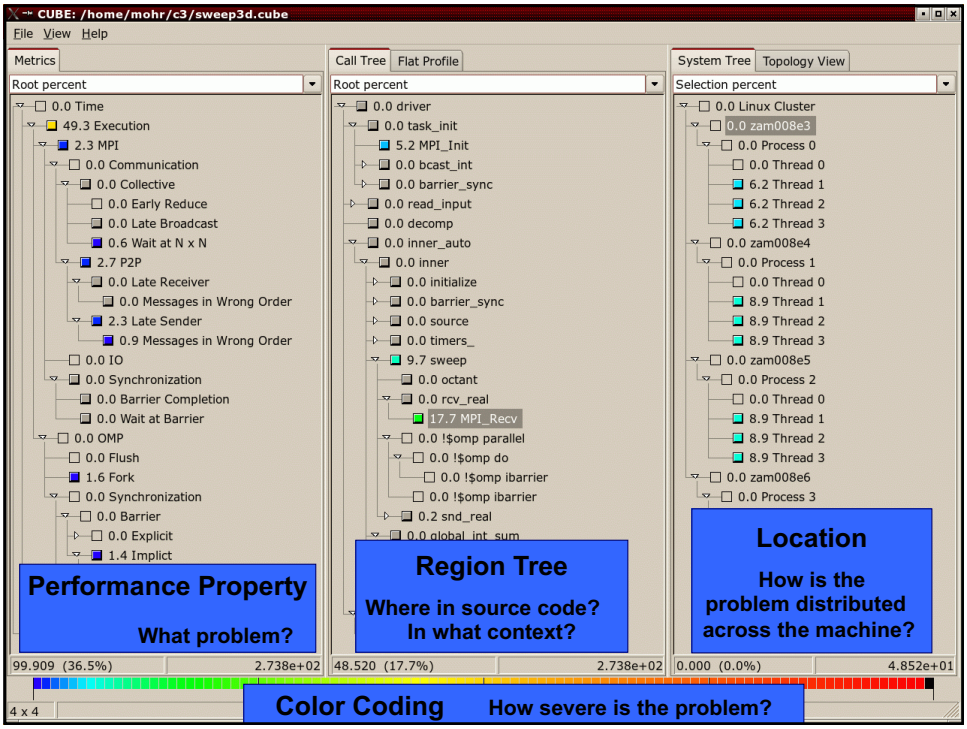
KOJAK / Scalasca – Basic Idea

- "Traditional" Tool
- Automatic Tool

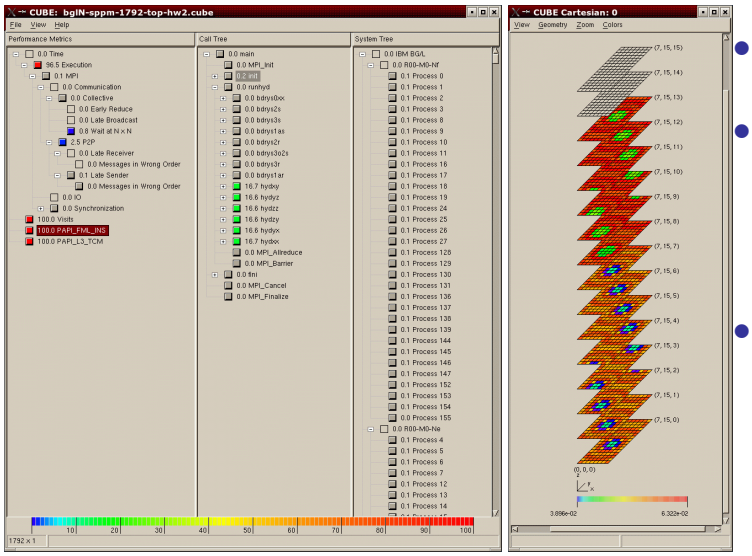


- For non-standard / tricky cases (10%)
- For expert users
  - ⇨ More productivity for performance analysis process!

- For standard cases (90% ?!)
- For "normal" users
- Starting point for experts



KOJAK: sPPM run on (8x16x14) 1792 PEs



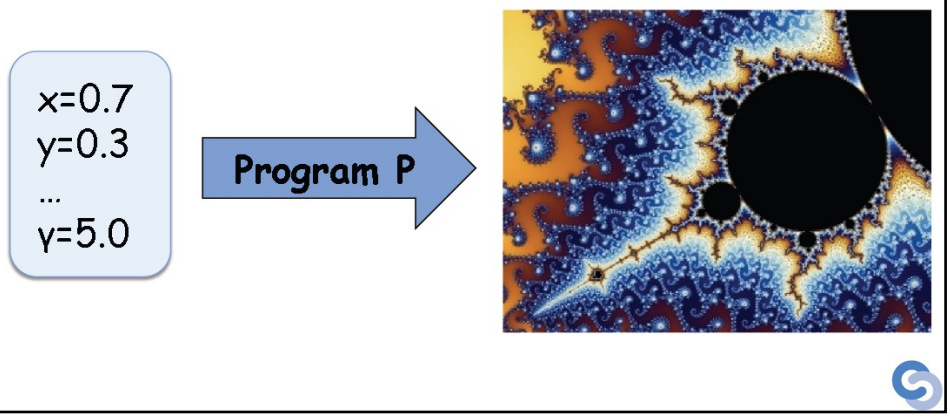
- Topology display
- Shows distribution of pattern over HW topology
- Easily scales to even larger systems

Parallel Correctness Challenges

- Parallel programming presents a number of new challenges to writing correct software
  - New kinds of bugs: data races, deadlocks, etc.
  - More difficult to test programs and find bugs
  - More difficult to reproduce errors
- Key Difficulty: Potential non-determinism
  - Order in which threads execute can change from run to run
  - Some runs are correct while others hit bugs

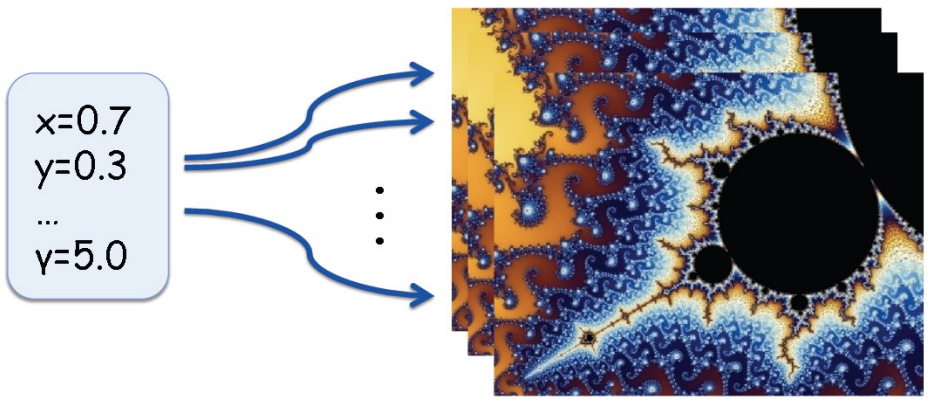
Parallel Correctness Challenges

- For sequential programs, we typically expect that same input → same output



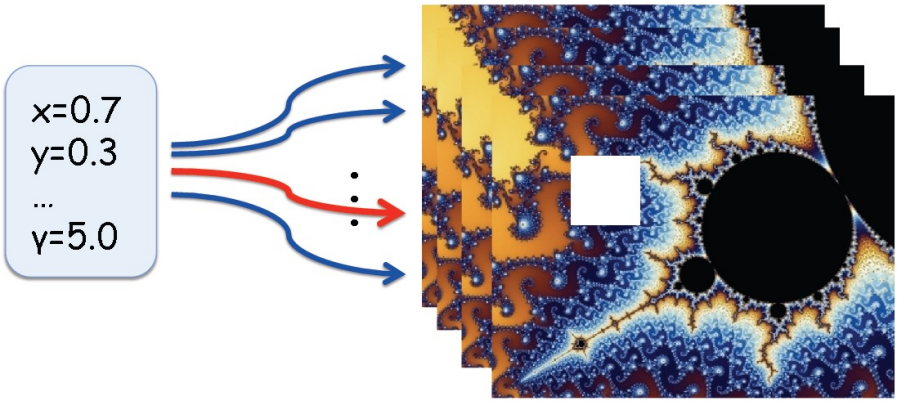
Parallel Correctness Challenges

- But for parallel programs, threads can be scheduled differently each run



Parallel Correctness Challenges

- But for parallel programs, threads can be scheduled differently each run



Parallel Correctness Challenges

- But for parallel programs, threads can be scheduled differently each run
- A bug may occur under only rare schedules.
  - In 1 run in 1000 or 10,000 or …
- May occur only under some configurations:
  - Particular OS scheduler
  - When machine is under heavy load
  - Only when debugging/logging is turned off!

Testing Parallel Programs

- For **sequential** programs:
  - Create several test inputs with known answers.
  - Run the code on each test input
  - If all tests give correct input, have some confidence in the program
  - Have intuition about which "edge cases" to test
- But for **parallel** programs:
  - Each run tests only a single schedule
  - How can we test many different schedules?
  - How confident can we be when our tests pass?

Testing Parallel Programs

- Possible Idea: Can we just run each test thousands of times?
- Problem: Often not much randomness in OS scheduling
  - May waste much effort, but test few different schedules
  - Recall: Some schedules tend to occur only under certain configurations – hardware, OS, etc
  - One easy parameter to change: load on machine

Stress Testing

- Idea: Test parallel program while oversubscribing the machine
  - On a 4-core system, run with 8 or 16 threads
  - Run several instances of the program at a time
  - Increase size to overflow cache/memory
  - Effect: Timing of threads will change, giving different thread schedules
- Pro: Very simple idea, easy to implement
  - And often works!

Noise Making / Random Scheduling

- Idea: Run with random thread schedules
  - E.g., insert code like:
    - if (rand() < 0.01) usleep(100)
    - if (rand() < 0.01) yield()
  - Can add to only “suspicious” or “tricky” code.
  - Or use tool to seize control of thread scheduling.
- Pros: Still fairly simple and often effective.
  - Explores different schedules than stress testing.
  - Many tools can perform this automatically

343

Noise Making / Random Scheduling

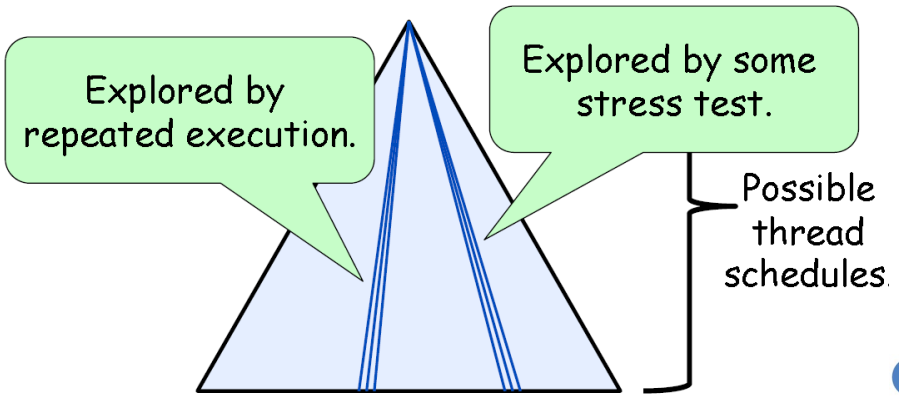
- IBM's ConTest: Noise-making for Java
  - Clever heuristics about where to insert delays
- Berkeley's Thrille (C + pthreads) and
- CalFuzzer (Java) do simple random scheduling
  - Extensible: Write testing scheduler for your app
- Microsoft Research's Cuzz (for .NET)
  - New random scheduling algorithm with probabilistic guarantees for finding bugs – available soon
- Many of these tools provide replay – same random number seed → same schedule

343

344

Limitations of Random Scheduling

- Parallel programs have huge number of schedules – exponential in length of a run

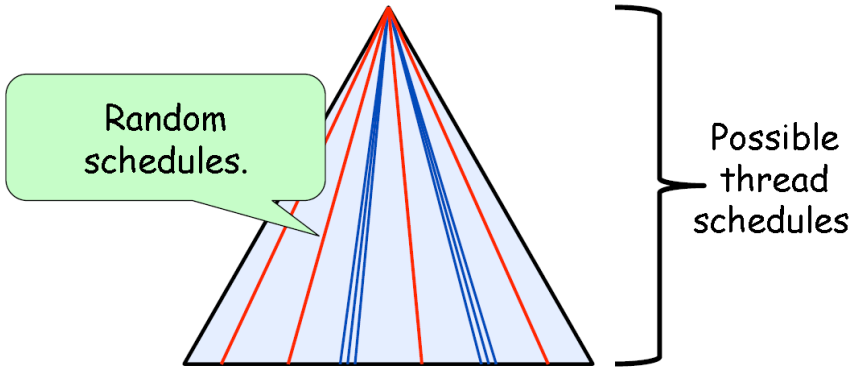


344

Limitations of Random Scheduling

- Parallel programs have huge number of schedules – exponential in length of a run

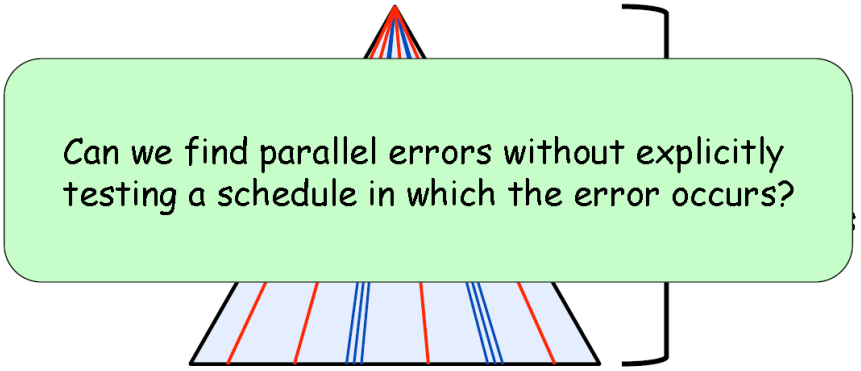
Vast majority of schedules will never be tested.



Limitations of Random Scheduling

- Parallel programs have huge number of schedules – exponential in length of a run

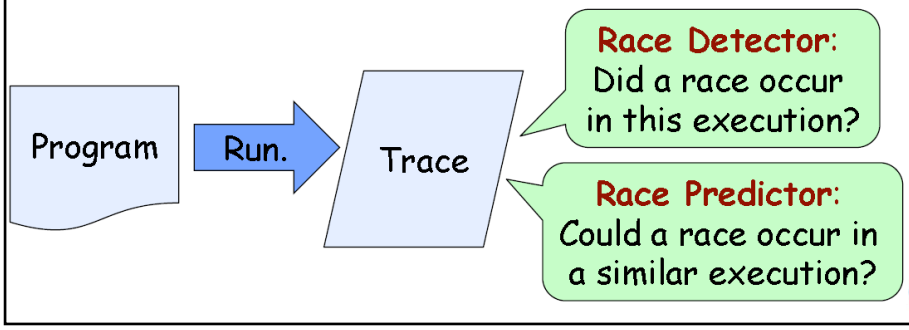
Vast majority of schedules will never be tested.



347

Detecting/Predicting Parallel Bugs

- Say we observe a test run of a parallel program that doesn't obviously fail
- **Key Question**: Can we find possible parallel bugs by examining the execution?

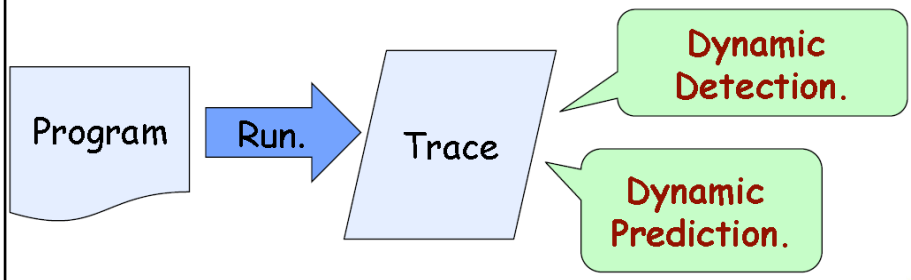


347

348

Detecting/Predicting Parallel Bugs

- Techniques/tools exist for:
  - Data races
  - Atomicity violations
  - Deadlocks
  - Memory consistency errors



348

Data Race Detection / Prediction

- 20+ years of research on race detection
- Happens-Before Race Detection [Schonberg '89]:
  - Do two accesses to a variable occur, at least one a write, with no intervening synchronization?
  - No false warnings
- Lockset Race Prediction [Savage, et al., '97]:
  - Does every access to a variable hold a common lock?
  - Efficient, but many false warnings
- Hybrid Race Prediction [O'Callahan, Choi, 03]:
  - Combines Lockset with Happens-Before for better performance and fewer false warnings vs. Lockset

Dynamic Data Race Tools

- Intel Thread Checker for C + pthreads
  - Happens-Before race detection
- Valgrind-based tools for C + pthreads
  - Helgrind and DRD (Happens-Before)
  - ThreadSanitizer (Hybrid)
- CHESS performs race detection for .NET
- CalFuzzer and Thrille: hybrid race detection for Java and C + pthreads

Static Analysis

- So far we've only discussed dynamic analyses
  - Examine a real run/trace of a program
- Static analyses predict data races, deadlocks, etc., **without** running a program
  - Only examine the source code
  - Area of active research for ~20 years
  - Potentially much better coverage than dynamic analysis – examines all possible runs
  - But typically also more false warnings
- CHORD: static race and deadlock prediction

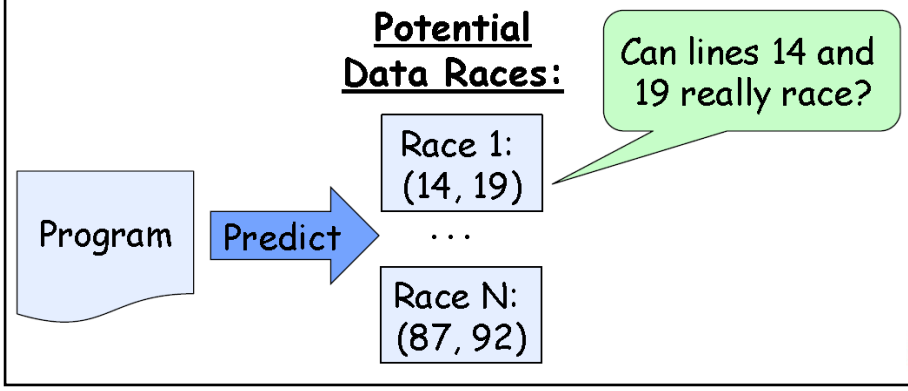
Active Random Testing Overview

- **Problem**: Random testing can be very effective for parallel programs, but can miss many potential bugs
- **Problem**: Predictive analyses find many bugs, but can have false warnings
  - Time consuming and difficult to examine reported bugs and determine whether or not they are real
- **Key Idea**: Combine them – use predictive analysis to find potential bugs, then **biased** random testing to actually create each bug

353

Active Random Testing

- **Key Idea**: Combine them – use predictive analysis to find potential bugs, then **biased** random testing to actually create each bug

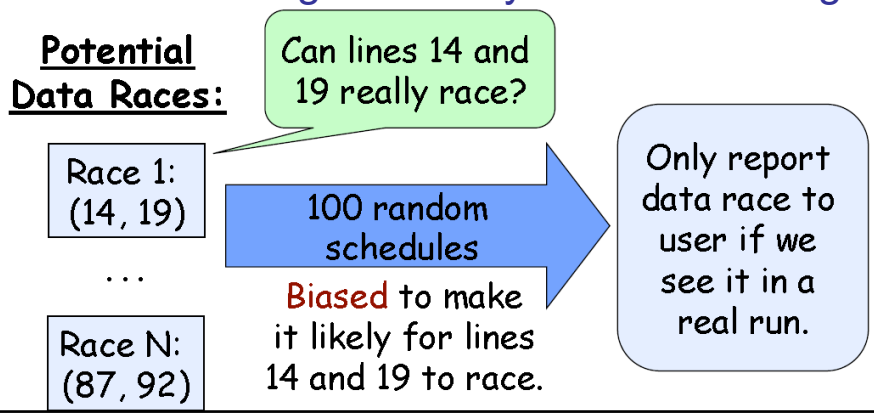


353

354

Active Random Testing

- **Key Idea**: Combine them – use predictive analysis to find potential bugs, then **biased** random testing to actually create each bug



354

Active Random Testing

- CalFuzzer is an extensible, open-source tool for active testing of Java programs
  - For data races, atomicity bugs, and deadlocks
  - RaceFuzzer is the active testing algorithm for data races
- Thrille for C + pthreads – Data Races
- And UPC-Thrille for Unified Parallel C
  - Part of the Berkeley UPC system

Active Testing Summary

- Combines benefits of random testing and predictive analysis
  - Random testing amazingly effective in practice
  - Even more so when biased with information about predicted bugs
  - Can replay executions for debugging
- Available now for Java (CalFuzzer) and Thrille (C + pthreads)
- UPC-Thrille for Unified Parallel C
  - Part of the Berkeley UPC system by year's end

Testing & Debugging Conclusions

- Many tools available right now to help find bugs in parallel software
  - Data races, atomicity violations, deadlocks
- **No silver bullet solution!**
  - Have to carefully design how an application threads will coordinate and share/protect data
  - Tools will help catch mistakes when the design is accidentally not followed
  - Ad hoc parallelization likely to never be correct, even with these tools

Tools References

- IBM's ConTest: Noise-making - Java
  - https://www.research.ibm.com/haifa/conferences/hvc2010/present/Testing_and_Debugging_Concurrent_Software.pdf
- Cuzz: Random scheduling for C++/.NET
  - http://research.microsoft.com/en-us/projects/cuzz/
- Intel Inspector - Locate and debug threading, memory errors in C/C++
  - https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/inspector.html?wapkw=Thread%20Checker%20and%20Parallel%20Inspector#gs.0pp194
- Helgrind, DRD, ThreadSanitizer Dynamic Data Race Detection/Prediction for C/C++
  - http://valgrind.org/docs/manual/hg-manual.html
  - http://code.google.com/p/data-race-test/
- CHORD - Static Race/Deadlock Detection for Java
  - https://www.cse.msu.edu/~cse914/Overheads/mmcgill-java-static-race-detector-6per.pdf

Tools References

- CalFuzzer - Java
  - https://github.com/ksen007/calfuzzer
  - https://www.seas.upenn.edu/~mhnaik/papers/cav09.pdf
- Thrille - C
  - http://github.com/nicholasjalbert/Thrille
- CHESS - C++/.NET Model Checking, Race Detection
  - http://research.microsoft.com/en-us/projects/chess/default.aspx
- Java Path Finder - Model Checking for Java
  - https://blog.devgenius.io/introduction-to-java-pathfinder-80828b53309a
- Tau Performance System - Fortran, C, C++, Java, Python
  - http://www.cs.uoregon.edu/research/tau/home.php
- Scalasca Performance Analysis - C/C++ and Fortran
  - https://apps.fz-juelich.de/scalasca/releases/scalasca/1.4/docs/UserGuide.pdf
- Performance Application Programming Interface - PAPI & Exa-PAPI
  - http://icl.cs.utk.edu/exa-papi/

Metode si Tehnici de Programare in High Performance Computing

AAC/IALA Master Modules

Thank you for your attention

Emil Slusanschi
emil.slusanschi@cs.pub.ro
University Politehnica of Bucharest