Introduction to Correctness and Performance Tools for Parallel Programming

John Mellor-Crummey
Department of Computer Science
Rice University
johnmc@rice.edu
Correctness Tools Landscape

- Formal verification
- Static and dynamic analysis
- Interactive debugging
- Lightweight debugging

Fine grained on left
Coarse grained on right
Non Scalable on top
Scalable on bottom
Formal Verification: Model Checking

Example tool: Spin (http://spinroot.com)

- Formal verification of parallel software
  - performs on-the-fly exploration of execution state spaces
- Used to identify logical design errors in parallel programs
  - e.g., communication and synchronization protocols, data structures
- Supports multiple communication models
  - message passing: both rendezvous and buffered
  - communication through shared memory
- Checks logical consistency of a specification
  - reports deadlocks, race conditions, incompleteness
  - identifies assumptions about relative speeds of processes
- Verifies properties specified with linear temporal logic
- Specify system descriptions in PROMELA modeling language
Dynamic Analysis

• Valgrind: framework for dynamic analysis tools
http://valgrind.org


— two useful valgrind tools
  – memcheck: detects memory-management problems
    accesses memory it shouldn't
    areas not yet allocated, areas that have been freed, areas past the end of heap blocks, inaccessible areas of the stack
    reads uninitialized values
    leaks memory
    performs double or mismatched frees of heap blocks
  – helgrind: finds data races in multithreaded programs
    memory locations accessed by >1 thread, unprotected by a lock

• A notable race detector: cilkscreen

A Sophisticated Shared-Memory Synchronization Algorithm

- High performance locking protocol for NUMA machines
  - lock-holder typically passes lock to a “nearby” lock-waiter
    - improves locality of shared data
  - limit lock passing to nearby threads to avoid starvation
- Accesses shared variables with RMW operations, loads, stores

A Concurrency Bug!

- **Symptom:** multiple threads simultaneously in the critical section

- **Investigation**
  - no obvious flaw in the algorithm
  - no obvious issue in the implementation
  - code+algorithm reviewed by multiple synchronization experts

- **Traditional debugging efforts failed**
  - debugging with assertions
    - assertions trigger long after the bug occurs
    - can’t backtrack to see thread interleaving that leads to assertion failure
  - single stepping in the debugger (gdb)
    - bug sporadically appears only with > 9 threads
    - does not appear when closely observed (a “Heisenbug”)
PinPlay: Deterministic Replay of Parallel Programs

1. Record a log of “buggy” execution
   (Leave overnight with multiple runs to hit assertion / segfault)
   records access order (RAW, WAR, and WAW)
   to shared memory locations

2. Replay the execution with “buggy” thread interleaving

g++ Code Generation Bug Corrupts Algorithm!

- Source code

```c
// Expect atomic 64-bit write
cacheline_aligned_64_bit_var = 0xfffffffffffffffffd;
```

- GNU g++ 4.4.5 generated machine code

```
movl $0xfffffffd,(%rax) // write low 32-bits
movl $0x0xffffffff,0x4(%rax) // write high 32-bits
```

- Splitting this 64-bit write into two parts creates a small window of inconsistent state

- Bug was not noticeable at source (the point of write)

- Bug was not noticeable at sync (read a clobbered, yet valid, 64-bit value)

- Required a record/replay tool to step through the execution under debugger to identify the inconsistent state
Interactive Debugging

- Popular tools: TotalView (Rogue Wave), DDT (Allinea)

- What can be debugged?
  - MPI applications
  - multithreaded processes
  - accelerated codes

- Laptops to supercomputers
  - debug over 100K processes

- Integrated GUI for controlling entire application
  - variable value inspection in different processes
  - data visualization

- Reverse debugging with Totalview’s Replay Engine
  - records orderings and state changes as program executes
  - recovers prior states on demand with “backward stepping”

Figure credit: http://www.allinea.com/sites/default/files/uploads/products/sparkline.png
Lightweight Debugging

See poster in SC14 technical poster session

“DySectAPI: Scalable Prescriptive Debugging”

TU Denmark, Mesosphere Inc., LLNL
Performance Tools Landscape

My research focus on tools
Instruction-level Analysis

- Analyze dynamic characteristics of applications using a binary instrumentation framework, e.g., Valgrind, Pin

- Sample analyses
  - instruction mix (e.g. floating point, integer, branch, memory)
  - computational intensity (flops per memory access)
  - percentage of floating point ops that are vectorized (SIMD)
  - reuse distance
    - unique cache lines touched between two accesses to same cache line
    - gathered using detailed simulation of caches in your CPU
    - pinpoint sources of cache misses in your code
MPI Tracing

- Collect MPI traces: comm type, participants
- Visualize traces with various tools, e.g. Jumpshot (ANL)

Jumpshot: Asynchronous Dynamic Load Balancing
Sample Tracing and Profiling

Rice University’s HPCToolkit Performance Tools

- Source code
- Optimized binary
- Profile execution
- Call path profile
- Binary analysis
- Program structure
- Interpret profile
- Correlate with source
- Presentation
- Database
Call Path Profiling

Measure and attribute costs in context
- sample timer or hardware counter overflows
- gather calling context using stack unwinding

Call path sample:
- return address
- return address
- return address
- instruction pointer

Calling context tree

Overhead proportional to sampling frequency... 
...not call frequency
Analyzing Chombo@1024PE with hpcviewer

- Costs for:
  - inlined procedures
  - loops
  - function calls in full context

Source pane

View control

Metric display

Navigation pane

Metric pane
Profiling compresses out the temporal dimension —temporal patterns, e.g. serialization, are invisible in profiles.

What can we do? Trace call path samples

—sketch:
  - N times per second, take a call path sample of each thread
  - organize the samples for each thread along a time line
  - view how the execution evolves left to right
  - what do we view?
    assign each procedure a color; view a depth slice of an execution
Load imbalance among threads appears as different lengths of colored bands along the x axis.
Summary

• Lots of tools available

• Existing tools address diverse needs
  — proving parallel code correct
  — detecting data races
  — repeating intricate thread interleavings for debugging
  — interactive debugging of huge process counts
  — understanding MPI communication by tracing and visualization
  — profiling to understand where an application spends its time
  — visualizing sample traces to understand behavior over time

• Tool frameworks for building custom tools

• Challenges
  — better tools for accelerated computing
  — tools for correctness checking, debugging, and performance analysis of a billion dynamic tasks!
    – measurement, analysis, attribution, presentation
References

- Spin: http://spinroot.com
- valgrind: http://valgrind.org
- Pin: http://pintool.org
- Pinplay: http://pinplay.org
- cilkscreen: http://www.cilkplus.org
- totalview: http://www.roguewave.com
- ddt: http://www.allinea.com
- HPCToolkit: http://hpctoolkit.org
Acknowledgments

• The inverted pyramid diagrams were inspired by a diagram on the poster cited below:

• The Jumpshot figure is from Rusty Lusk (ANL, retired)

• The slides about using PinPlay to debug the implementation of a sophisticated concurrent data structure is from Milind Chabbi (Rice University)