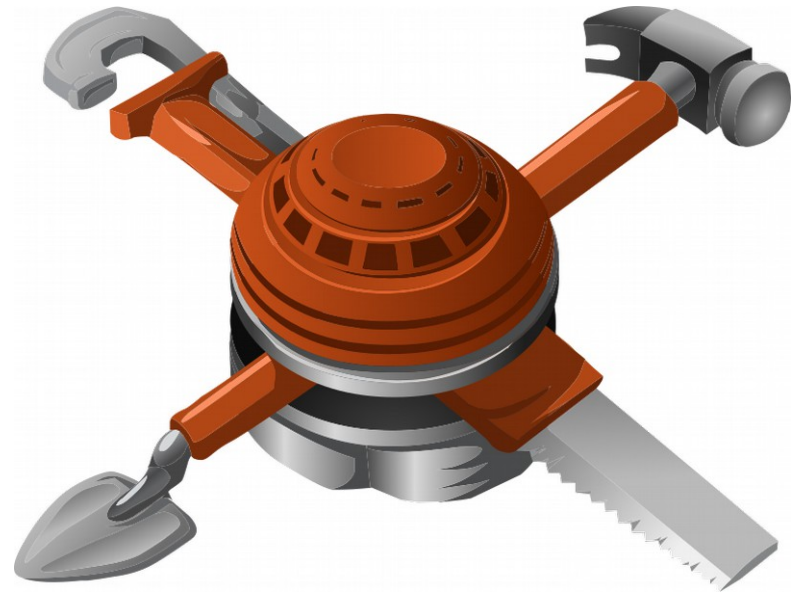


CS 470 Spring 2018

Mike Lam, Professor



Performance Tools

Software Tools

- **Software tool**: computer program used by developers to create, debug, maintain or support other programs

Traditional Software Tools

- Text editors
- Version control
- Debuggers
- Profilers
- Test automation frameworks
- Deployment frameworks
- Integrated development environments (IDEs)

Traditional Software Tools

- **Debuggers**
 - Purpose: finding and removing software defects
 - Often done via a process monitoring interface
- **Profilers**
 - Purpose: detecting performance characteristics and identifying **bottlenecks**
 - Often done via instrumentation (added code that tracks the program's execution)
- Both of these are difficult in parallel and distributed systems

Traditional Debugging

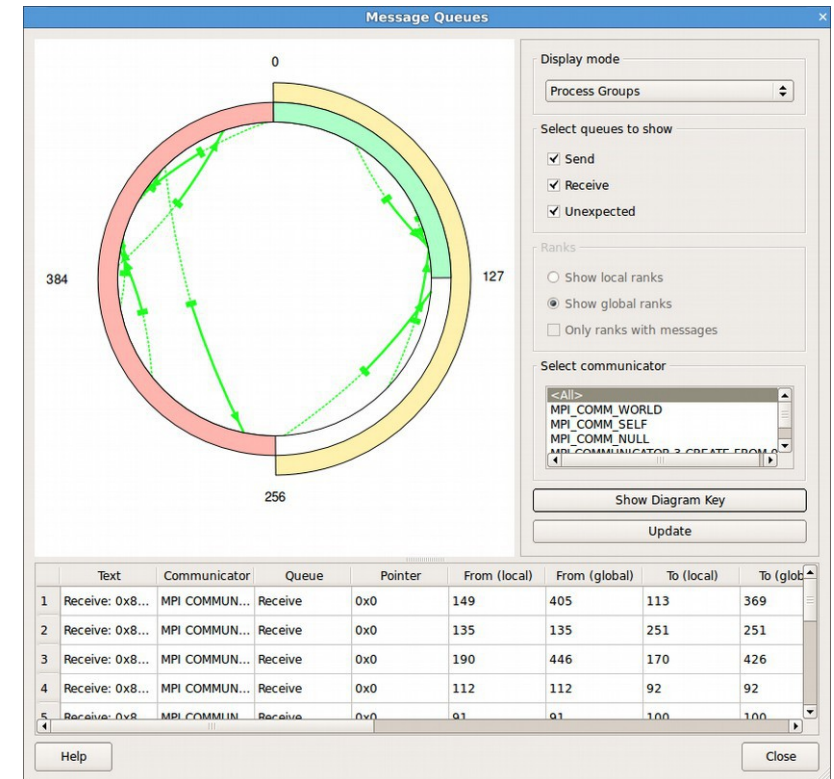
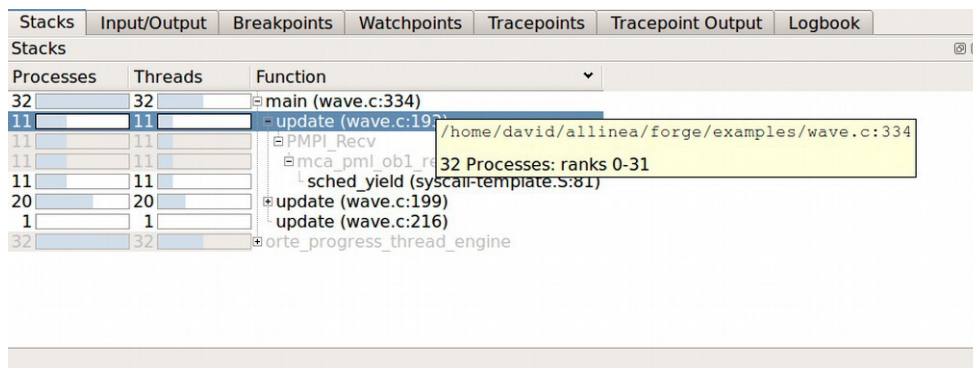
- Mechanisms
 - `ptrace`: system call that allows one process to control another
 - Simulation: slower, but safer
- Common features
 - Breakpoints and watchpoints
 - Single-stepping (by instruction or line of code)
 - Variable examination and modification
 - In newer debuggers: reverse-stepping
- Free debuggers: [gdb](#), [lldb](#), [Eclipse](#), [Valgrind \(Memcheck\)](#)

Parallel Debugging

- Multithreaded debugging can be difficult
 - Must attach to the correct thread
 - Must control other threads as well
 - Nondeterminism means unpredictability
 - GDB does include support for multithreading:
 - <http://sourceware.org/gdb/current/onlinedocs/gdb/Threads.html>
 - Valgrind also provides the [Helgrind](#) error detector
- Distributed debugging is even harder
 - Hundreds or thousands of nodes; millions of processes
 - Enormous launch overhead
 - Control and visualization issues

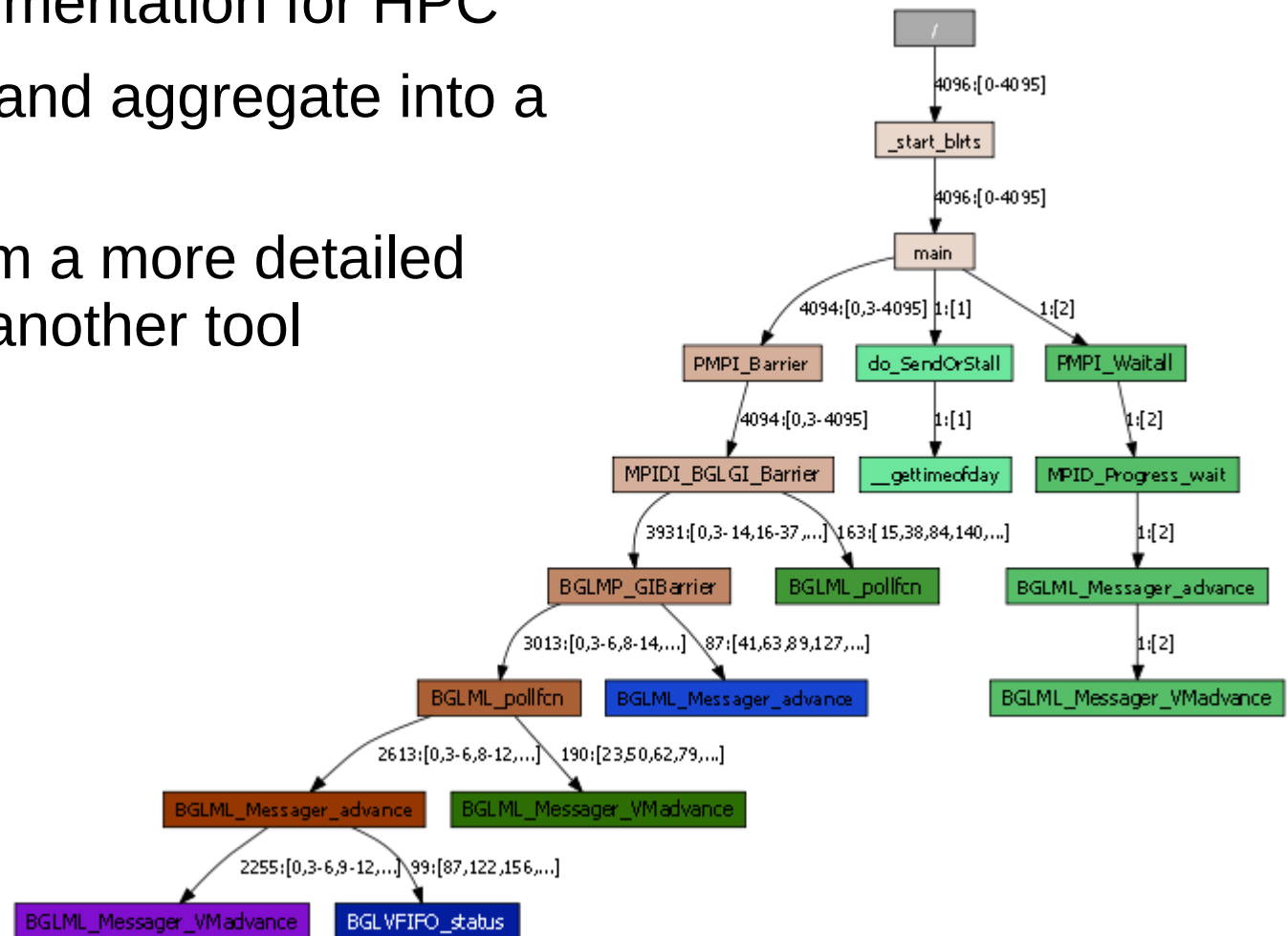
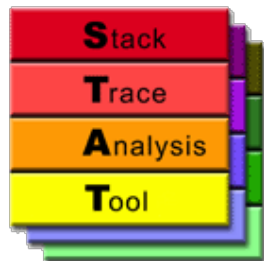
Commercial debuggers

- Microsoft Visual Studio
- Intel Debugger
- Rational Purify
- RogueWave TotalView
- Allinea DDT



Stack Trace Analysis Tool (STAT)

- Lightweight instrumentation for HPC
- Gather all traces and aggregate into a tree form
- Results can inform a more detailed debug run using another tool



Profiling

- Goal: gain insights concerning a program's performance characteristics
- Common **metrics**
 - Wall or CPU time
 - Memory use, page faults, and cache misses
 - Network traffic and saturation
 - Energy use
- Common **scopes**
 - Function
 - Basic block
 - Instruction
 - Source code line

Measurement

- **Instrumentation**: inserting analysis code
 - Binary vs. source
 - Static vs. dynamic
 - Best for event-based monitoring (e.g., function calls)
- **Sampling**: polling an analysis source
 - Hardware counters
 - Performance Application Programming Interface (**PAPI**)
 - Randomized vs. periodic
 - Averaging vs. min/max
 - Best for continuous monitoring (e.g., memory usage)

Measurement

- **Context**
 - Flat vs. call graph
 - Partial vs. full context
- **Profiling** vs. **tracing** (latter builds time-series)
- **Issues**
 - **Overhead**: added run time due to profiling software
 - **Perturbation**: skewing of behavior due to profiling software
 - **Skid**: execution may not stop immediately on sample
- **Tradeoff: better information vs. lower overhead**
 - Instrumentation: more instrumentation points
 - Sampling: higher frequency or less aggregation

GNU Profiler (gprof)

- Compile with “-pg” flag
- Run as usual; generates “gmon.out” file
- View results with “gprof” utility
 - “gprof <executable>”
- See <https://sourceware.org/binutils/docs/gprof/> for more documentation
- Google also has a multi-threaded profiler:
 - <https://github.com/gperftools/gperftools>

Callgrind/Cachegrind

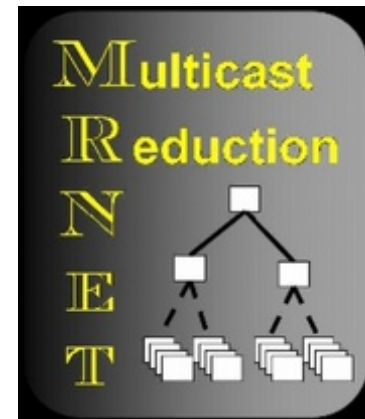
- Run with Valgrind
 - Callgrind: `valgrind --tool=callgrind <executable>`
 - Cachegrind: `valgrind --tool=cachegrind <executable>`
 - This will produce a `*.out.xxxx` file with raw results (could be large!)
 - Remember to call `mpirun` first if it's an MPI program
 - (And use `cg_merge` to merge multiple Cachegrind output files)
- Post-process results
 - Callgrind: `callgrind_annotate <output-files>`
 - GUI alternative: `kcachegrind` (or `qcachegrind` on Mac OS X)
 - Cachegrind: `cg_annotate <output-file>` (“- -auto=yes” for code)
 - Dx = data cache (level X) Ix = instruction cache (level X)
 - 1 = L1 cache L/LL = lowest level (on the cluster, this is L3)
 - r = read w = write m = miss Ir = Instructions read
- See <http://valgrind.org/docs/manual> for more documentation

Perf_events

- Sample-based performance profiler
 - Kernel module reads performance counters
 - More lightweight than Valgrind-based analysis
 - Can sample many different events
 - User space utility `perf` to interface with kernel
 - `perf record -F 49 <command>`
 - Generates `perf.data` file
 - `perf report -n [--stdio]`
 - `perf annotate [--stdio]`
 - Cheat sheet link on resources page

Distributed Analysis

- Lots of data!
 - Collect at each rank but only store compressed or aggregated data
 - Aggregate using a tree-based reduction structure to reduce communication overhead

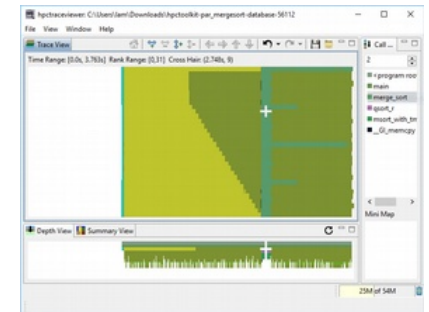


HPCToolkit (from Rice University)

- Integrated HPC program analysis tool chain
- Run program with `hpcrun`
 - On cluster, "`source /shared/bin/hpctoolkit_setup.sh`" first
 - Use “-t” for tracing information
 - With MPI, call `mpirun` first
 - (e.g., “`salloc -n 4 mpirun hpcrun -t ./my_program`”)
 - This generates a folder w/ measurement data
 - Make sure it will run for more than a few seconds!
 - However, remember that the instrumentation adds significant overhead
 - See also `/shared/bin/hpctoolkit_p2` for an example of how to run the analysis as a batch job

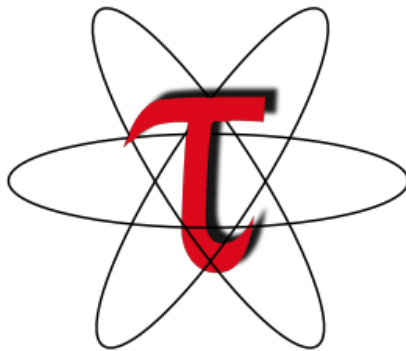
HPCToolkit (from Rice University)

- Post-process results using **hpcprof**
 - Pass it your measurement folder as a parameter
 - This generates a new folder w/ a results database
- View results using **hpcviewer** or **hpctraceviewer**
 - On cluster, make sure you forward X11 when you login
 - E.g., “`ssh -X <eid>@<host>`”
 - You may want to copy and view the results on your local computer
 - Viewers are available for Linux, Mac OS X, and Windows
- See <http://hpctoolkit.org/documentation.html> for more documentation



Other HPC analysis tools

- [Tuning and Analysis Utilities \(TAU\)](#) – University of Oregon
- [Open|SpeedShop](#) - Krell Institute
- [Scalasca](#)
- [Paraver](#)



Tool frameworks

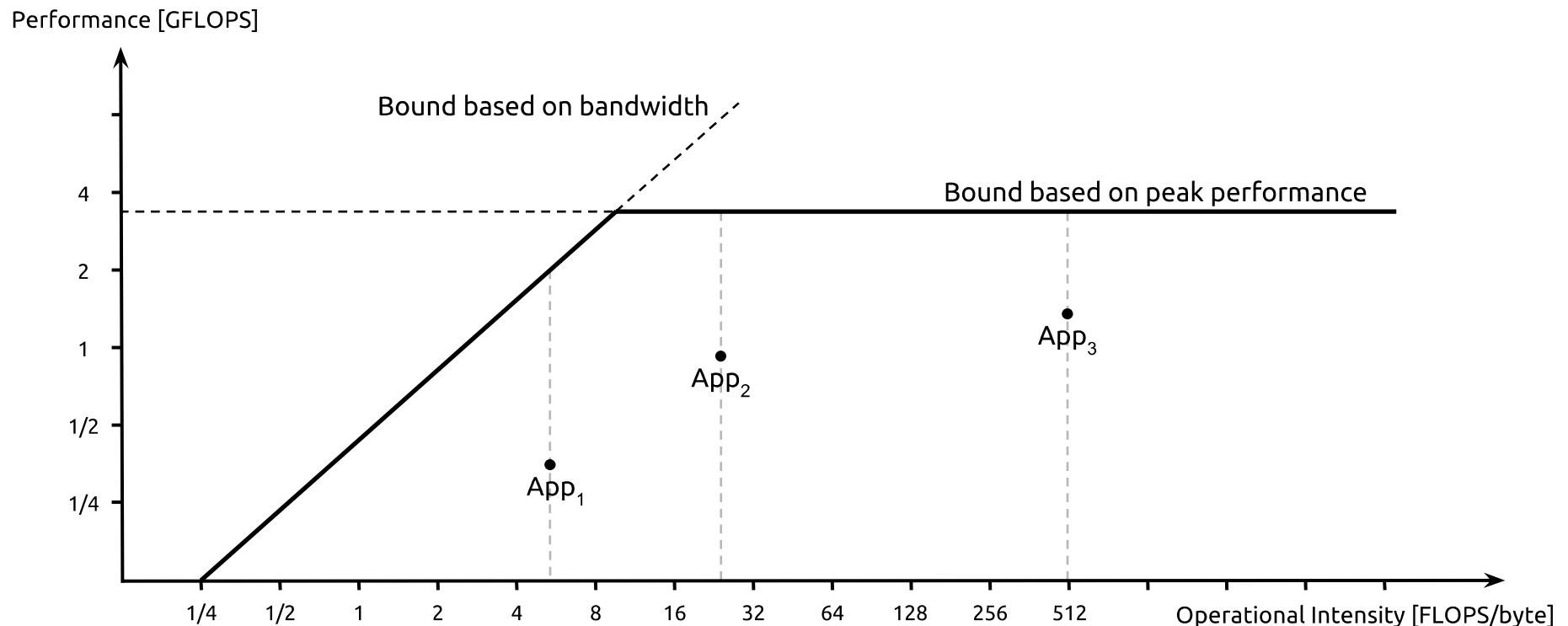
- Many analysis tools need similar functionality
 - Binary parsing
 - Instrumentation
 - Stack walking
- **Tool framework**: a library that provides common functionality upon which custom tools can be written
 - Intel Pin
 - Dyninst
 - libunwind
 - Valgrind
 - CRAFT

Modeling and autotuning

- Observation: modern systems have a lot of knobs
 - Message size, block size, # of threads, # of processes
 - Many of these factors influence each other
 - Different runs could require different “optimal” settings
- Idea #1: build a **model** of these interactions
 - Needs training data; could differ for every run
- Idea #2: **autotune** the system at runtime
 - Could be expensive or impossible to implement

Performance models

- One simple model: **roofline** model
 - Shows theoretical limits on performance
 - Based on computation and communication bounds



Performance models

