CMSC 714
Lecture 8
Profiling – gprof and HPCToolkit

Alan Sussman

Notes

- MPI project due Monday, 6PM
  - Questions on project?
  - I will try to do grading within a week
- OpenMP project will be posted on the same day, and due 2 weeks later
- Readings posted for next week
  - Don’t forget to send questions when you are assigned

Performance analysis

- Parallel performance of a program might not be what the developer expects
- How do we find performance bottlenecks?
- Two parts to performance analysis: measurement and analysis/visualization
- Simplest tool: timers in the code and printf

Using timers

```
double start, end;
double phase1, phase2, phase3;

start = MPI_Wtime();
 ... phase1 code ...
end = MPI_Wtime();
phase1 = end - start;

start = MPI_Wtime();
 ... phase2 ...
end = MPI_Wtime();
phase2 = end - start;

start = MPI_Wtime();
 ... phase3 ...
end = MPI_Wtime();
phase3 = end - start;
```

Phase 1 took 2.45 s

Phase 2 took 11.79 s

Phase 3 took 4.37 s

Performance Tools

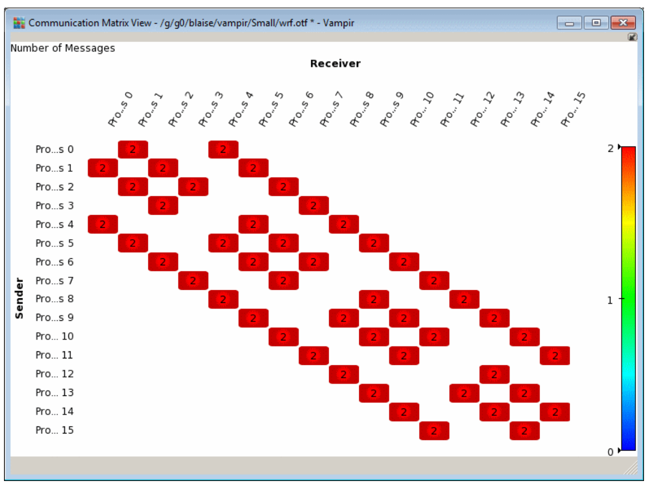
- Tracing tools
  - Capture entire execution trace
  - Vampir, Score-P
- Profiling tools
  - Provide aggregated information
  - Typically use statistical sampling
  - Gprof, pyinstrument, cprofile
- Many tools can do both
  - TAU, HPCToolkit, Projections

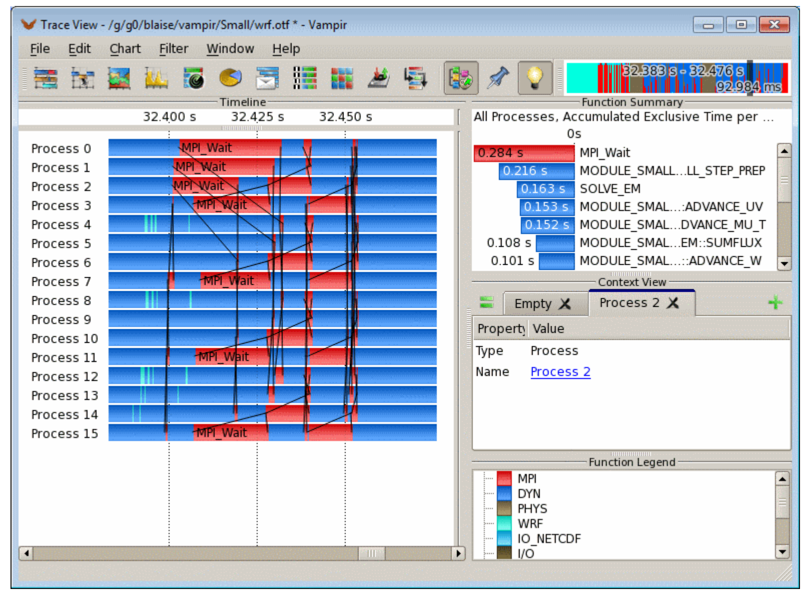
Metrics recorded

- Counts of function invocations
- Time spent in code
- Number of bytes sent
- Hardware counters
- To fix performance problems — we need to connect metrics to source code

Tracing tools

- Record all the events in the program with timestamps
- Events: function calls, MPI events, etc.

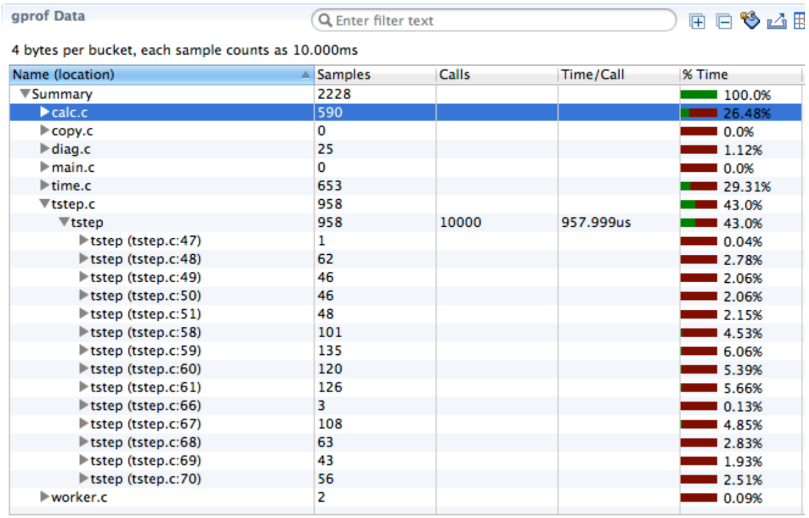




Vampir visualization: https://hpc.llnl.gov/software/development-environment-software/vampir-vampir-server

Profiling tools

- Ignore the specific times at which events occurred
- Provide aggregate information about different parts of the code
- Examples:
  - gprof, perf
  - mpiP
  - HPCToolkit, caliper
- Python tools: cprofile, pyinstrument, scalene



gprof Data

Enter filter text

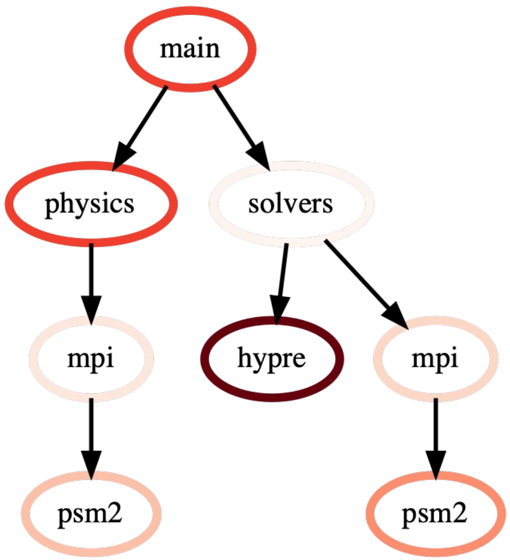
4 bytes per bucket, each sample counts as 10.000ms

Name (location)	Samples	Calls	Time/Call	% Time
Summary	2228			100.0%
calc.c	590			26.48%
copy.c	0			0.0%
diag.c	25			1.12%
main.c	0			0.0%
time.c	653			29.31%
tstep.c	958			43.0%
tstep	958	10000	957.999us	43.0%
tstep (tstep.c:47)	1			0.04%
tstep (tstep.c:48)	62			2.78%
tstep (tstep.c:49)	46			2.06%
tstep (tstep.c:50)	46			2.06%
tstep (tstep.c:51)	48			2.15%
tstep (tstep.c:58)	101			4.53%
tstep (tstep.c:59)	135			6.06%
tstep (tstep.c:60)	120			5.39%
tstep (tstep.c:61)	126			5.66%
tstep (tstep.c:66)	3			0.13%
tstep (tstep.c:67)	108			4.85%
tstep (tstep.c:68)	63			2.83%
tstep (tstep.c:69)	43			1.93%
tstep (tstep.c:70)	56			2.51%
worker.c	2			0.09%

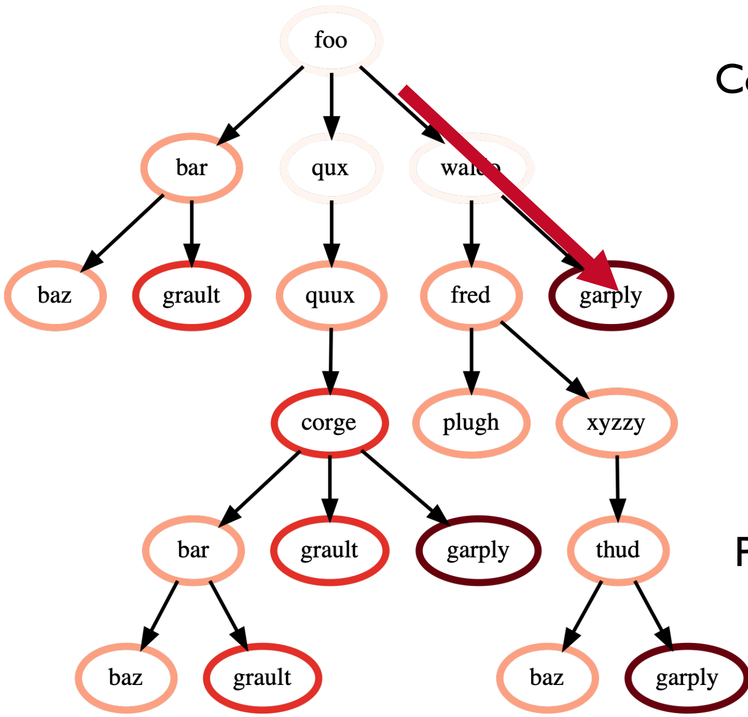
gprof data in hpctView

Calling contexts, trees, and graphs

- Calling context or call path: Sequence of function invocations leading to the current sample
- Calling context tree (CCT): dynamic prefix tree of all call paths in an execution
- Call graph: merge nodes in a CCT with the same name into a single node but keep caller-callee relationships as arcs



Calling context trees, call graphs, …



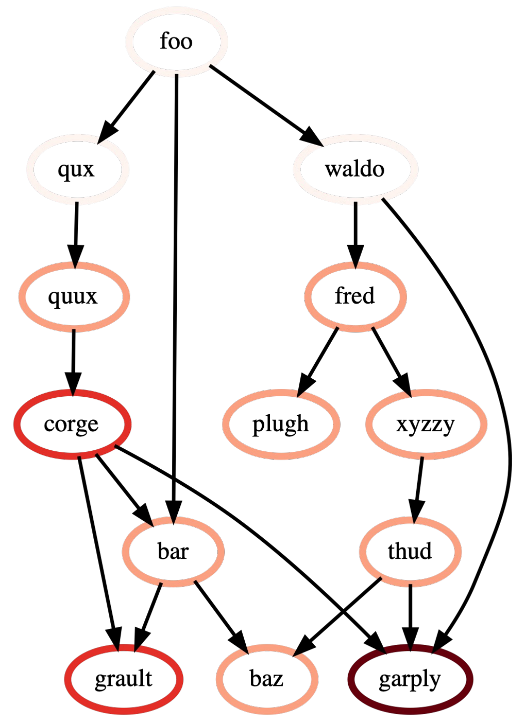
Calling context tree (CCT)

Contextual information

```
File
Line number
Function name
Callpath
Load module
Process ID
Thread ID
```

Performance Metrics

```
Time
Flops
Cache misses
```



Call graph

gprof

- Goal is to collect profiling information
  - Static and dynamic call graphs
  - How many times each function is called
  - How much time is spent in each function, and in the functions that a function calls
- Process is to first compile with a flag (-pg for C compilers typically)
  - To insert calls to monitoring code at entry (and/or exit) from a function
- Then the program will generate monitoring output in a file (by default **gmon.out**) that can be post-processed by the **gprof** program to produce profiling information

gprof (cont.)

- Since the profiling info is collected during a run, can combine info collected over multiple runs (presumably with different input, to exercise different program paths)
- Execution time info is not collected via timing routines, but via *sampling*
  - To minimize profiling overhead
  - Basically sample periodically which function is currently executing and assign the time for that interval to the function currently running – only requires interval timer from the OS
  - The time interval for each sample needs to be short enough to not miss too many function calls (so depends on processor speed/performance)

gprof (cont.)

- Output includes number of times each function is called, the time spent in each function, and the time spent in a given function and all the functions it calls
- One difficulty is with mutually recursive functions
  - Problem is that call graph then has a cycle
  - So how to assign time for a function and everything it calls
- The overall goal is to use the profiling info to optimize the program
  - And the most important thing to know to do that is where your program is spending its time!
- So use gprof iteratively to optimize parts of your program
- Available in Linux and other Unix systems

HPCToolkit

- Set of tools for measurement, analysis, attribution, and presentation of application performance for sequential and parallel (multi-threaded and message-passing) programs
- Capabilities/goals include:
  - collecting performance measurements of fully optimized executables *without* adding instrumentation
  - analyzing application *binaries* to understand the structure of optimized code
  - correlating measurements with program structure
  - presenting the resulting performance data in a top-down way to facilitate rapid (human) analysis
- Available at http://hpctoolkit.org/
  - As part of DOE Exascale Computing Project (ECP) tools

HPCToolkit features

- Language independent
  - Works on binaries, so works with C, C++, Fortran, …
- No code instrumentation
  - So no instrumentation overhead
  - Uses statistical sampling to measure performance
- Avoids ”blind spots”
  - Works on optimized and stripped binaries, including (dynamically and statically linked) libraries, so requires binary analysis
- Keeps track of context to help understand behavior of modern OO software designs
  - Uses call path profiling to assign costs to specific execution paths
- Presents measurement data in a hierarchical way
  - To support a top-down analysis methodology that helps users to quickly locate bottlenecks

Features (cont.)

- Hierarchical attribution and presentation of measurement data
  - From function, to loop, to statement, etc., to make data easier for users to understand and take action on
- Measurement and analysis is scalable
  - Sampling-based measurement limits the size of the performance data to be collected and analyzed, even on large parallel systems

HPCToolkit workflow

- Collect performance measurements while application executes via sampling – *hpcrun*
  - Can user hardware performance counters if available
  - Can deal with threads and MPI calls
- Analyze program binaries to recover program structure, and map to source code (if available) – *hpcstruct*
- Produce performance database by combining application structure with performance measurements – *hpcprof*
- Explore performance database to find bottlenecks – *hpcviewer*
- Prototype visualization in space and time of a parallel program – *hpctraceview* – now part of *hpcviewer*