



Profiling with Rice HPCToolkit

Mark W. Krentel

Department of Computer Science

Rice University

krentel@rice.edu



<http://hpctoolkit.org>



HPCToolkit Basic Features

- **Run application natively and every 100-200 times per second, interrupt program and record snapshot of call stack and then combine these into a calling context tree (CCT).**
- **Combine sampling data with a static analysis of the program structure: loops, inline functions, etc.**
- **Present top-down, bottom-up and flat views of calling context tree (CCT) and time-sequence trace view. Metrics are displayed per source line in the context of their call path.**
- **Can sample on Wallclock (itimer), POSIX timers and Hardware Performance Counter Events (PAPI preset and native events): cycles, flops, cache misses, etc.**

HPCToolkit Advanced Features

- **Finely-tuned unwinder to handle multi-lingual, fully-optimized code, no frame pointers, broken return pointers, stack trolling, etc.**
- **Derived metrics -- compute flops per cycle, or flops per memory reads, etc. and attribute to lines in source code.**
- **Compute strong and weak scaling loss, for example:**
strong: $8 * (\text{time at 8K cores}) - (\text{time at 1K cores})$
weak: $(\text{time at 8K cores and 8x size}) - (\text{time at 1K cores})$
- **Load imbalance -- display distribution and variance in metrics across processes and threads.**
- **Blame shifting -- when thread is idle or waiting on a lock, blame the working threads or holder of lock.**

The Good and the Not Quite So Good

- **What we do well**
 - core events tied to specific process/thread: cycles, flops, cache misses via hardware performance counters
 - attribute events to source code, function calls and loops
- **What we do not so well**
 - system-wide events: network and I/O latency, throughput, congestion, shared cache (L2 on BG/Q)
 - scaling issues to 1,000,000 cores

Where to find HPCToolkit

- Home site: user's manual, build instructions, links to source code, download viewers:

<http://hpctoolkit.org/>

- On mira, cetus, tukey:

</projects/Tools/hpctoolkit/pkgs-mira/hpctoolkit/bin>

</projects/Tools/hpctoolkit/pkgs-tukey/hpctoolkit/bin>

- Source code via anonymous svn checkout:

<http://code.google.com/p/hpctoolkit/>

- Send questions to:

[hpctoolkit-forum at mailman.rice.edu](mailto:hpctoolkit-forum@mailman.rice.edu)

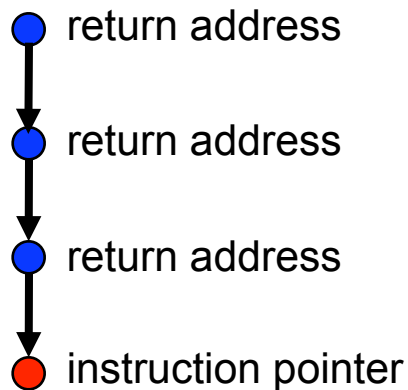
Call Path Profiling

Measure and attribute costs in context

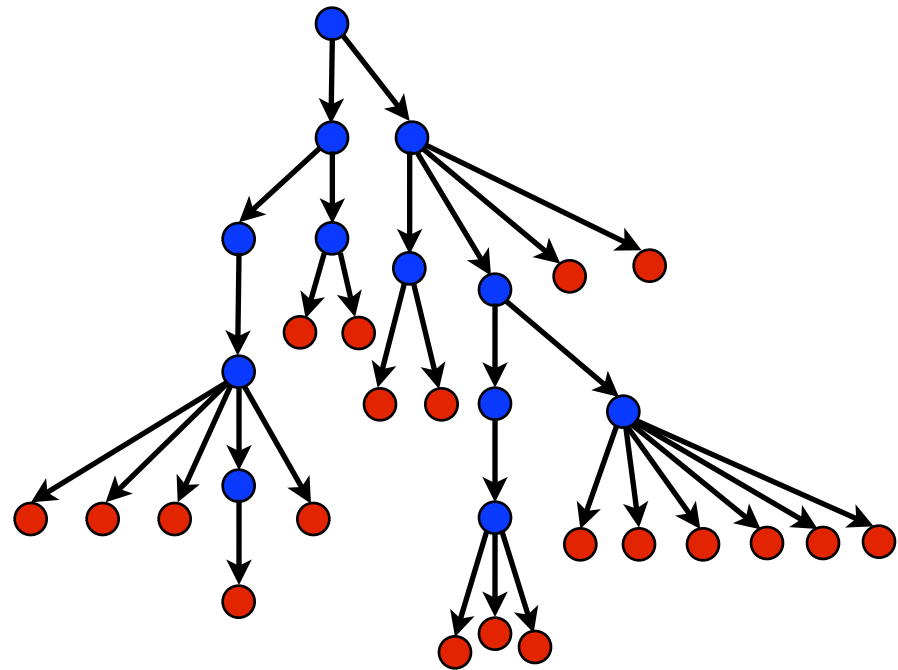
sample timer or hardware counter overflows

gather calling context using stack unwinding

Call path sample



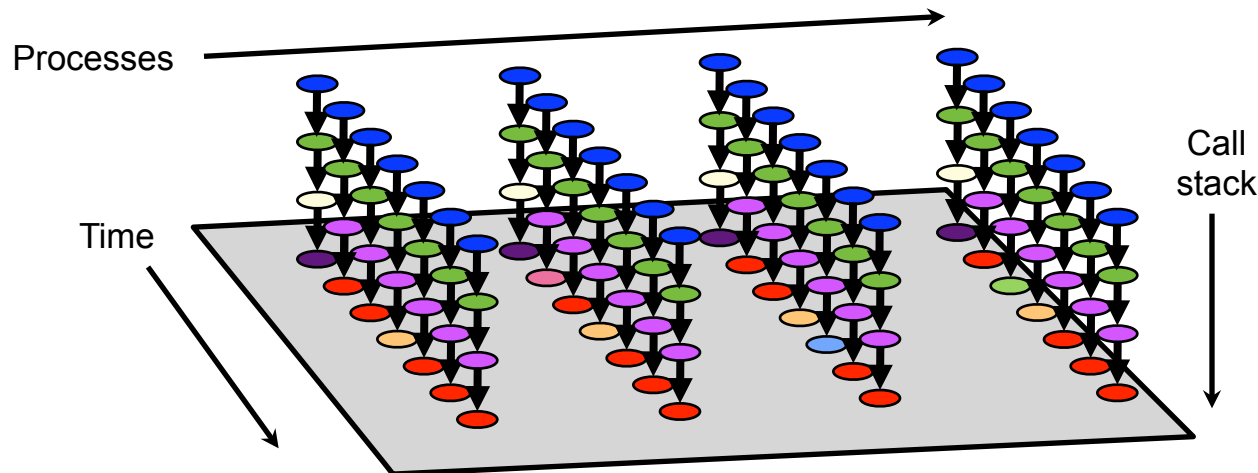
Calling context tree



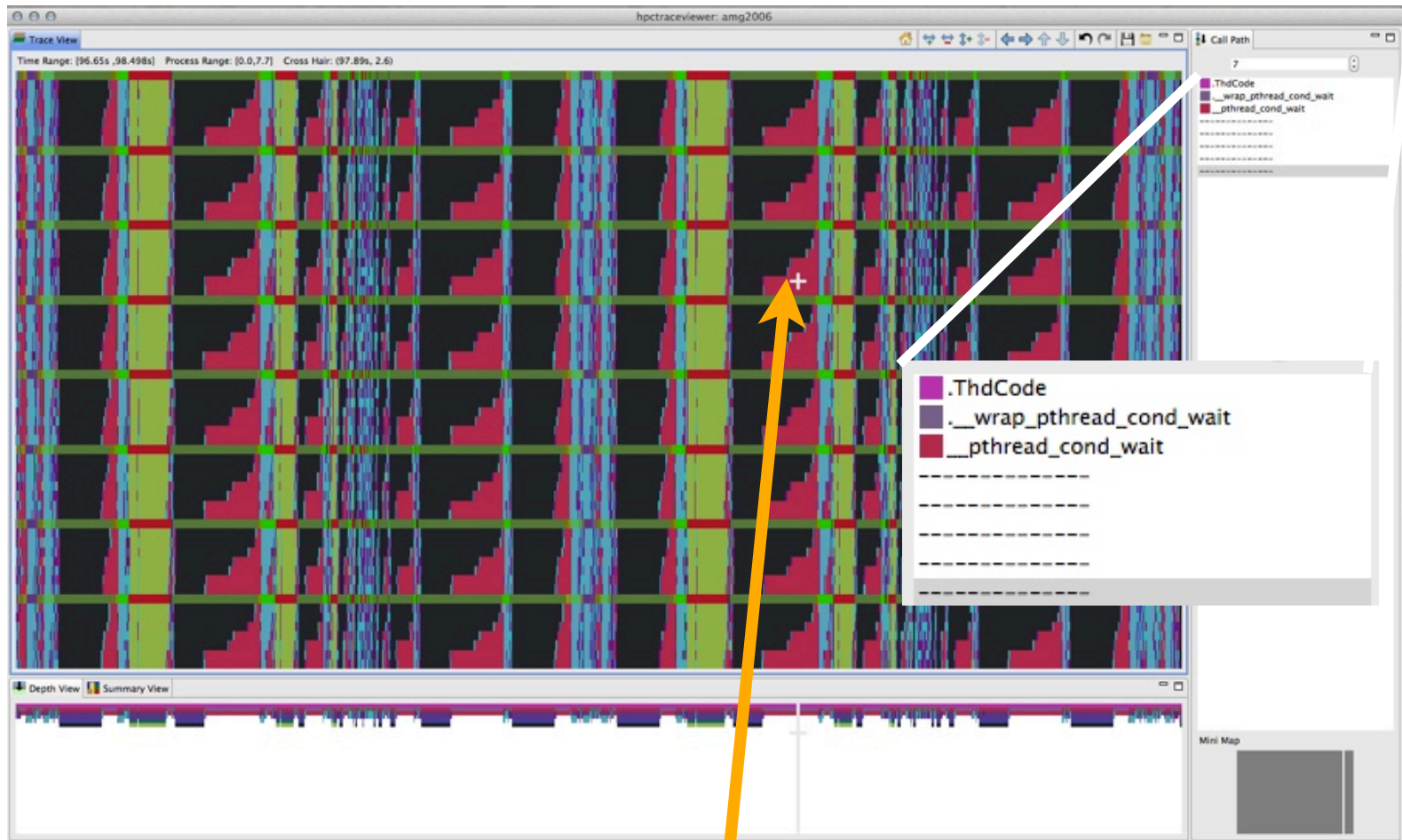
Overhead proportional to sampling frequency...
...not call frequency

Understanding Temporal Behavior

- 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



AMG2006: 8PE x 8 OMP Threads



OpenMP loop in hypre_BoomerAMGRelax using static scheduling has load imbalance; threads idle for a significant fraction of their time

Code-centric view: hypre_BoomerAMGRelax

The screenshot shows the hpcviewer interface with the file 'par_relax.c' open. The code is a C file defining HYPRE_SMP_PRIVATE and including utilities. A specific OpenMP loop is highlighted in blue. A note explains that this loop accounts for only 4.6% of the execution time. A red box highlights that across all instances of this loop, 19.7% of the time is spent idle. Below the code, the 'Calling Context View' is shown in 'Flat View' mode, displaying a table of performance metrics for various scopes.

Note: The highlighted OpenMP loop in hypre_BoomerAMGRelax accounts for only 4.6% of the execution time for this benchmark run. In real runs, solves using this loop are a dominant cost

across all instances of this OpenMP loop in hypre_BoomerAMGRelax
19.7% of time in this loop is spent idle w.r.t. total effort in this loop

Scope	WALLCLOCK (us):Sum (I)	WALLCLOCK (us):Sum (E)	idleness %	work %
▶ hypre PCGSetup	6.81e+08 11.1%		7.97e+01	2.03e+01
▶ HYPRE BoomerAMGSetup	6.81e+08 11.1%		7.97e+01	2.03e+01
▶ hypre BoomerAMGSetup	6.81e+08 11.1%		7.97e+01	2.03e+01
▶ . xlsmpParallelDoSetup TPO	3.77e+08 6.1%	3.20e+04 0.0%	2.35e+01	7.65e+01
▶ hypre BoomerAMGBuildCoarseOperator	3.16e+08 5.2%	1.44e+06 0.0%	4.80e+01	5.20e+01
▶ hypre BoomerAMGCoarsenFalgout	3.01e+08 4.9%	1.00e+03 0.0%	8.75e+01	1.25e+01
▼ hypre BoomerAMGRelax\$SOL\$24	2.81e+08 4.6%	2.81e+08 4.6%	1.97e+01	8.03e+01
▶ inlined from par_relax.c: 1638	2.81e+08 4.6%	2.00e+03 0.0%	1.97e+01	8.03e+01
▶ hypre BoomerAMGCoarsen	2.46e+08 4.0%	1.75e+08 2.9%	8.75e+01	1.25e+01
▶ hypre BoomerAMGBuildIterate\$SOL\$22	1.27e+08 2.1%	1.27e+08 2.1%	4.15e+01	5.85e+01

Serial Code in AMG2006 8 PE, 8 Threads

The screenshot displays the hpcviewer interface for the 'amg2006' application. The top pane shows the source code for 'par_relax.c', with lines 1632 through 1651 visible. The code defines a loop for worker threads (j = 0 to num_threads - 1). A red box with an arrow points to the performance table below, highlighting the 'loop at binsearch.c: 78' entry.

1632 #define HYPRE_SMP_PRIVATE i
1633 #include "../utilities/hypre_smp_forloop.h"
1634 for (i = 0; i < n; i++)
1635 tmp_data[i] = u_data[i];
1636 #define HYPRE_SMP_PRIVATE i,ii,j,jj,ns,ne,res,rest,size
1637 #include "../utilities/hypre_smp_forloop.h"
1638 for (j = 0; j < num_threads; j++)
1639 {
1640 size = n/num_threads;
1641 rest = n - size*num_threads;
1642 if (j < rest)
1643 {
1644 ns = j*size+j;
1645 ne = (j+1)*size+j+1;
1646 }
1647 else
1648 {
1649 ns = j*size+rest;
1650 ne = (j+1)*size+rest;
1651 }
1652 }

7 worker threads are idle in each process while its main MPI thread is working

Calling Context View Flat View

Scope	WALLCLOCK (us):Sum (I)	WALLCLOCK (us):Sum (E)	idleness %	work %
Experiment Aggregate Metrics	6.13e+09 100 %	6.13e+09 100 %	4.91e+01	5.09e+01
▶ loop at binsearch.c: 78	3.64e+07 0.6%	3.64e+07 0.6%	8.74e+01	1.26e+01
▶ loop at amg linklist.c: 78	8.47e+06 0.1%	8.47e+06 0.1%	8.75e+01	1.25e+01
▶ loop at amg linklist.c: 226	7.80e+06 0.1%	7.80e+06 0.1%	8.75e+01	1.25e+01
▶ inlined from RecChannel.h: 349	7.91e+06 0.1%	7.48e+06 0.1%	8.68e+01	1.32e+01
▶ inlined from InjGroup.h: 191	3.42e+06 0.1%	3.38e+06 0.1%	8.69e+01	1.31e+01
▶ inlined from Fifo.h: 195	2.89e+06 0.0%	2.89e+06 0.0%	8.69e+01	1.31e+01
▶ inlined from InjGroup.h: 161	2.78e+06 0.0%	2.78e+06 0.0%	8.69e+01	1.31e+01
▶ loop at par coarsen.c: 838	2.17e+06 0.0%	2.17e+06 0.0%	8.75e+01	1.25e+01
▶ loop at par coarsen.c: 1019	1.87e+06 0.0%	1.87e+06 0.0%	8.75e+01	1.25e+01

Pinpointing and Quantifying Scalability Bottlenecks

