

Profiling with HPCToolkit

Mark W. Krentel

Department of Computer Science

Rice University

krentel@rice.edu

<http://hpctoolkit.org>



HPCToolkit Basic Features

- **Run application natively (optimized) and every 100-200 times per second, interrupt program, unwind back to main(), record call stack, and combine these into a calling context tree (CCT).**
- **Combine sampling data with a static analysis of the program structure for loops and inline functions (hpcstruct).**
- **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 POSIX timers and Hardware Performance Counters (Perfmon or PAPI events): cycles, flops, cache misses, etc.**
- **Note: always include -g in compile flags (plus optimization) for attribution to source lines.**

HPCToolkit Advanced Features

- **Finely-tuned unwinder to handle multi-lingual, 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.**
- **Inline sequences — show full inline sequence for C++ templates.**

New Features

- **Spack** — now build hpctoolkit and prereqs with spack and install with spack modules.
- **Simplified use case for hpcstruct and hpcprof.**
- **Kernel Blocktime** — use Perf Events to count time spent blocked inside kernel, eg, I/O, barriers, locks, etc. (requires kernel perf events paranoid level 0 or 1).
 - **hpcrun -e CYCLES -e BLOCKTIME ...**
- **GPU** — full support for Nvidia and AMD, in progress for Intel.
- **Support for OpenMP parallel regions** — splice thread call paths onto master thread and identify work and idle (requires libomp replacement library), part of OpenMP 5.

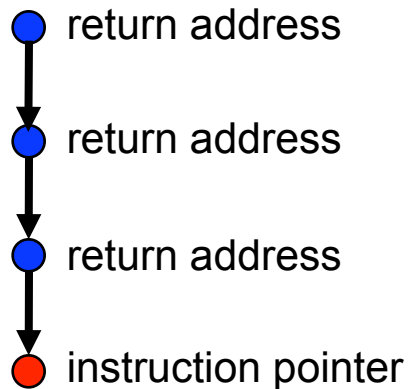
Call Path Profiling

Measure and attribute costs in context

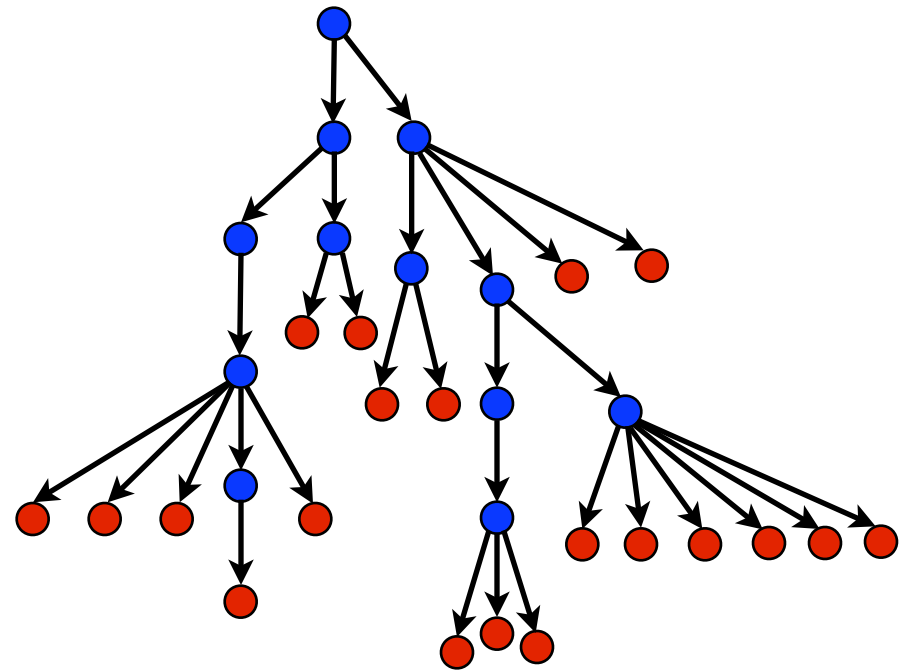
sample timer or hardware counter overflows

gather calling context using stack unwinding

Call path sample



Calling context tree



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

Where to find HPCToolkit

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

<http://hpctoolkit.org/>

- On theta, available as module hpctoolkit (includes hpcviewer on theta login nodes).

`module load hpctoolkit/2022.05.15` (theta)

`module load hpctoolkit/2022.05.15-gpu` (theta-gpu)

See: [/soft/perftools/hpctoolkit/workshop-2022](#) for build/run notes, example databases, etc.

- Source code on GitHub

<https://github.com/hpctoolkit>

`git clone https://github.com/hpctoolkit/hpctoolkit`

spack build instructions: [README.Install](#)

- Send questions to:

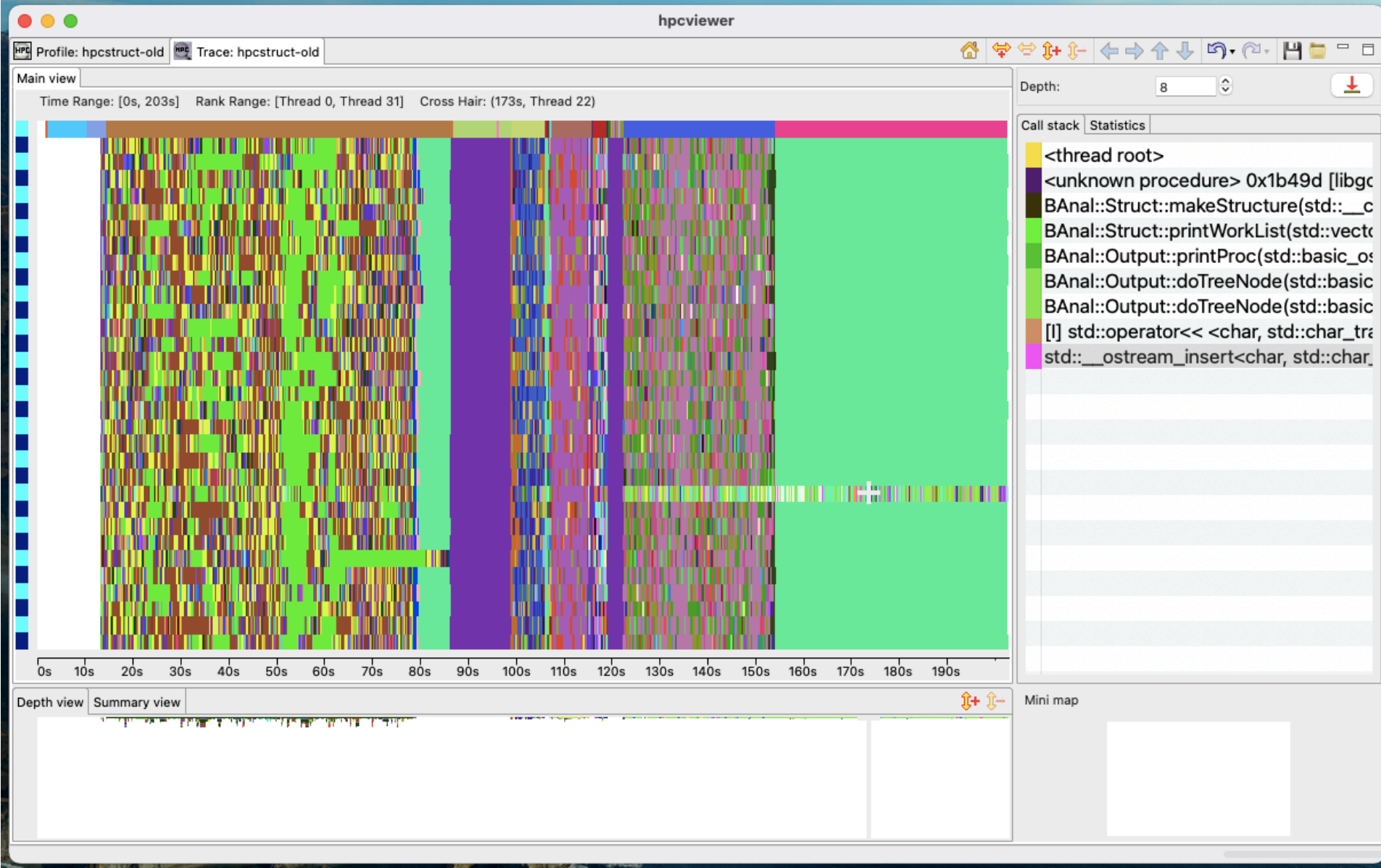
[hpctoolkit-forum at mailman.rice.edu](mailto:hpctoolkit-forum@mailman.rice.edu)

Quickstart for theta-gpu

- On theta-gpu,
 - `module load hpctoolkit/2022.05.15-gpu`
- Run application as follows. The first example is low-overhead. The 'pc' option displays the internals of the gpu kernels but can cause high overhead (1.5x to 4x).
 - `hpcrun [-t] -e REALTIME -e gpu=nvidia app ...`
 - `hpcrun [-t] -e REALTIME -e gpu=nvidia,pc app ...`
- Post-run analysis.
 - `hpcstruct hpctoolkit-measurements-directory`
 - `hpcprof hpctoolkit-measurements-directory`
- Finally, run hpcviewer and select database directory in the File menu chooser.

Using OpenMP Tools Library

- Use hpctoolkit ompt module.
`module load hpctoolkit/2020.04.ompt`
- Compile with `-fopenmp`, but on hpclink link line, replace `-fopenmp` with `libomp.a` from LLVM runtime. Supports GNU, Intel and Clang. On theta,
`/projects/Tools/hpctoolkit/pkgs-theta/llvm-openmp/lib/libomp.a`
- Add event `OMP_IDLE` (no number) plus time-based event: `REALTIME`, `PAPI_TOT_CYC` or `CYCLES`.
- Workarounds on theta to turn off thread affinity.
`aprun --cc none ...`
`export KMP_AFFINITY=none`



Trace of hpcstruct analyzing 8 Gig .so file.

Metric properties | main.cc | main.cc | CycleTracking.cc | CollisionEvent.cc | MacroscopicCrossSection.cc

```

66
67   for (int isoIndex = 0; isoIndex < numIsos && currentCrossSection >= 0; isoIndex++)
68   {
69       int uniqueNumber = monteCarlo->_materialDatabase->_mat[globalMatIndex]._iso[isoIndex]._gid;
70       int numReacts = monteCarlo->_nuclearData->getNumberReactions(uniqueNumber);
71       for (int reactIndex = 0; reactIndex < numReacts; reactIndex++)
72       {
73           currentCrossSection -= macroscopicCrossSection(monteCarlo, reactIndex, mc_particle.domain,
74               isoIndex, mc_particle.energy_group);
75           if (currentCrossSection < 0)
76           {
77               selectedIso = isoIndex;
78               selectedUniqueNumber = uniqueNumber;

```

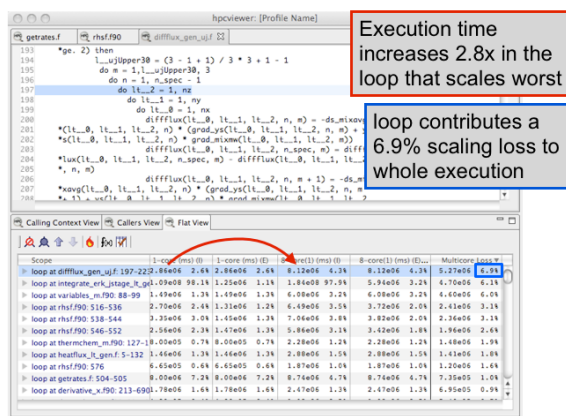
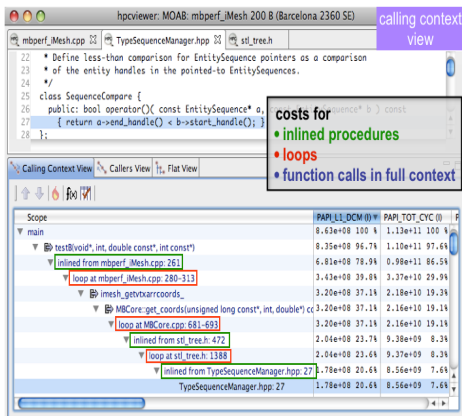
Top-down view | Bottom-up view | Flat view

↑ ↓ 🔥 f(x) 📄 📌 A+ A- || 🔍

Scope	GINs:Sum (I)	REAL
↳ 127 ⇒ __device_stub__Z19CycleTrackingKernelP10MonteCarloiP13ParticleVaultS2_(MonteCarlo*, int, P...	9.37e+10 100.0%	
↳ 14 ⇒ [I] cudaLaunchKernel<char>	9.37e+10 100.0%	
↳ 211 ⇒ <gpu kernel>	9.37e+10 100.0%	
↳ ⇒ CycleTrackingKernel(MonteCarlo*, int, ParticleVault*, ParticleVault*) [54b67c8fdf4def7a08c8d1fe1...	9.37e+10 100.0%	
↳ 132 ⇒ CycleTrackingGuts(MonteCarlo*, int, ParticleVault*, ParticleVault*) [54b67c8fdf4def7a08c8d...	9.36e+10 100.0%	
↳ 26 ⇒ [I] CycleTrackingFunction(MonteCarlo*, MC_Particle&, int, ParticleVault*, ParticleVault*)	7.00e+10 74.7%	
↳ loop at CycleTracking.cc: 118	7.00e+10 74.7%	
↳ 63 ⇒ CollisionEvent(MonteCarlo*, MC_Particle&, unsigned int) [54b67c8fdf4def7a08c8d1fe1b0...	4.41e+10 47.0%	
↳ loop at CollisionEvent.cc: 67	3.40e+10 36.3%	
↳ loop at CollisionEvent.cc: 71	3.21e+10 34.2%	
↳ 73 ⇒ macroscopicCrossSection(MonteCarlo*, int, int, int, int, int) [54b67c8fdf4def7a08c8d1f...	2.98e+10 31.8%	
↳ 41 ⇒ NuclearData::getReactionCrossSection(unsigned int, unsigned int, unsigned int) [54b6...	1.74e+10 18.6%	
↳ 253 ⇒ [I] NuclearDataReaction::getCrossSection(unsigned int)	5.85e+09 6.7%	

Code inside GPU kernel in quicksilver proxy app.

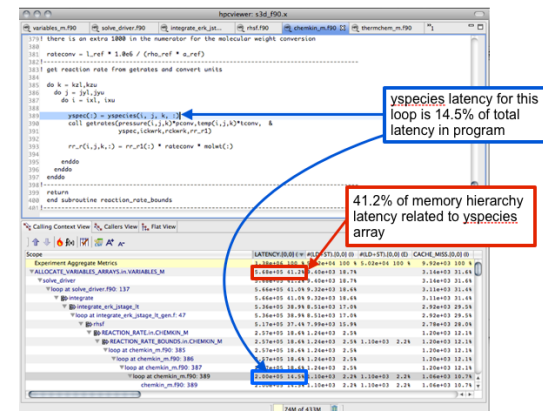
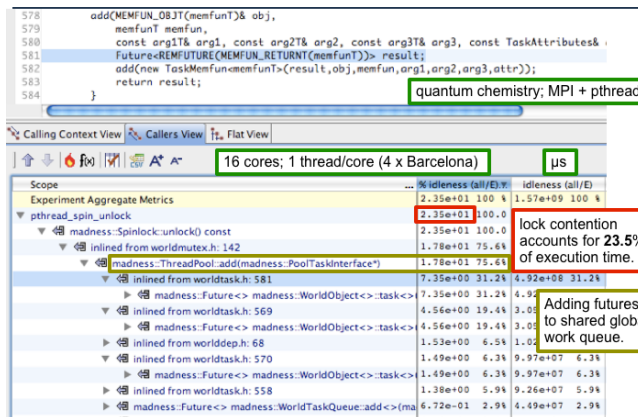
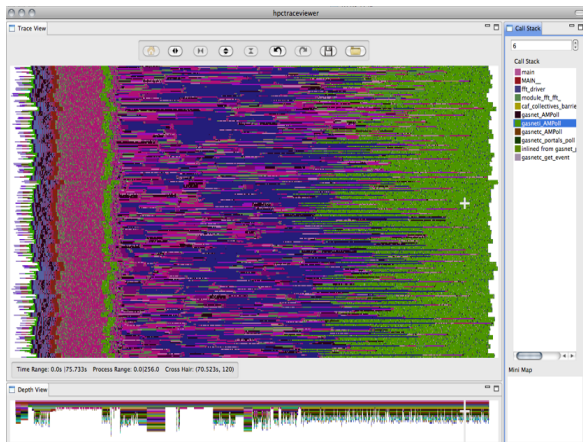
HPCToolkit Capabilities at a Glance



Attribute Costs to Code

Pinpoint & Quantify Scaling Bottlenecks

Assess Imbalance and Variability



Analyze Behavior over Time

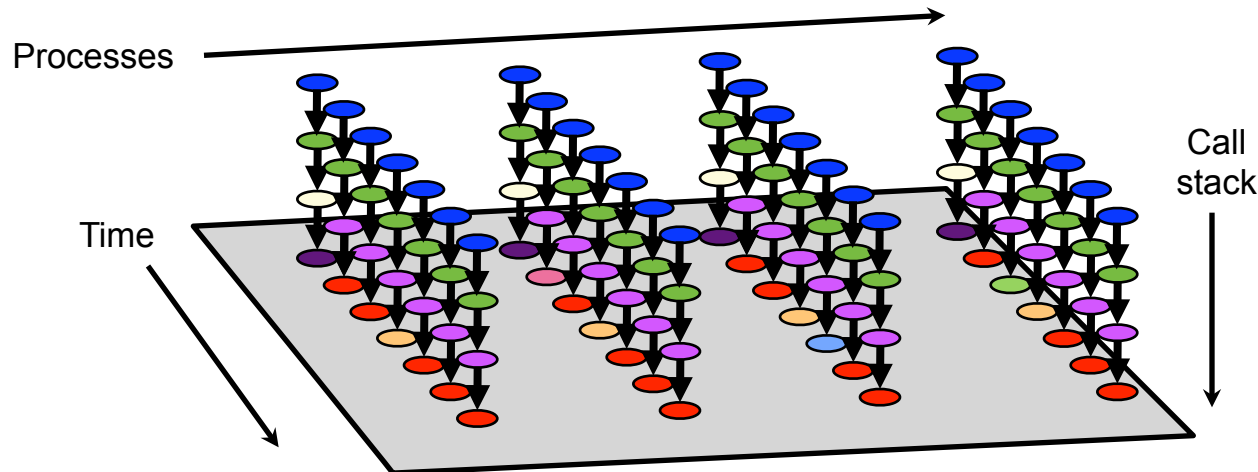
Shift Blame from Symptoms to Causes

Associate Costs with Data

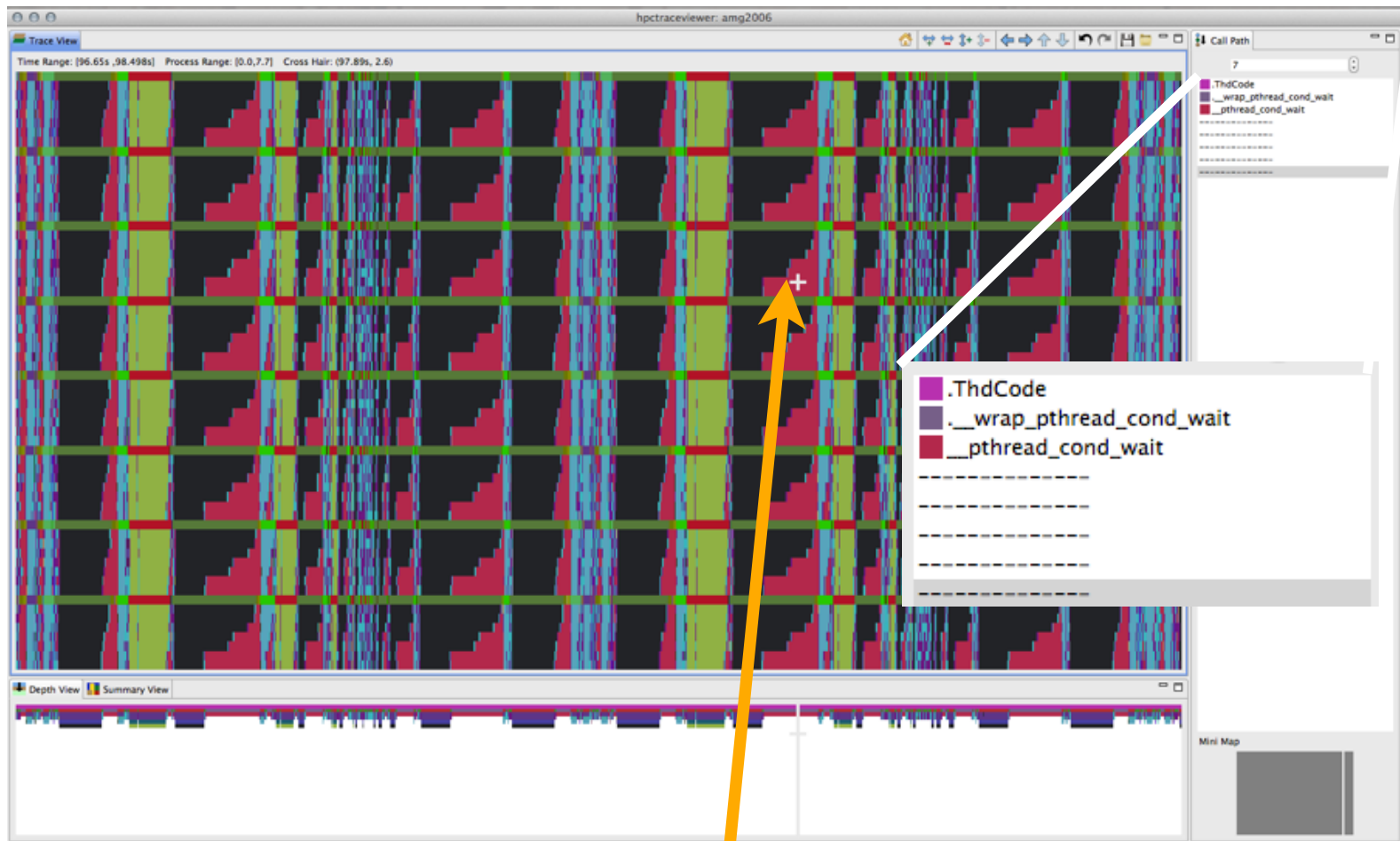
hpctoolkit.org

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 for a benchmark run named 'amg2006'. The top pane displays the source code for 'par_relax.c', with an OpenMP loop highlighted in blue. The bottom pane shows a performance table with columns for Scope, WALLCLOCK (us):Sum (I), WALLCLOCK (us):Sum (E), idleness %, and work %.

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 BoomerAMGCoarsenFalqout	3.01e+08 4.9%	1.00e+03 0.0%	8.75e+01	1.25e+01
▼ hypre BoomerAMGRelax\$SOLS\$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\$SOLS\$24	1.27e+08 2.1%	1.27e+08 2.1%	4.15e+01	5.85e+01

Serial Code in AMG2006 8 PE, 8 Threads

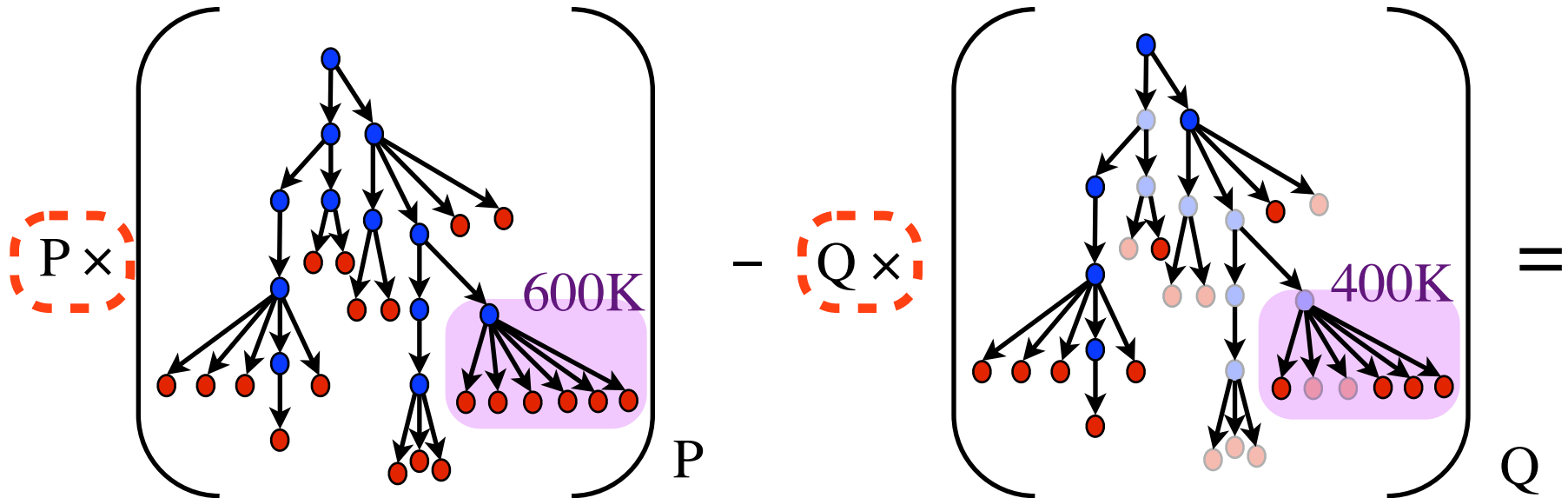
The screenshot displays the hpcviewer interface for the 'amg2006' application. The top pane shows the source code for 'par_relax.c', with lines 1632-1661 visible. A red box highlights a loop over 'num_threads' (lines 1638-1661). The bottom pane shows the 'Flat View' of performance metrics. A red arrow points from the text box to the 'loop at binsearch.c: 78' entry in the table.

```
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

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



coefficients for analysis
of strong scaling

