arm

Regional Application Profiling with Caliper

Florent.Lebeau@arm.com

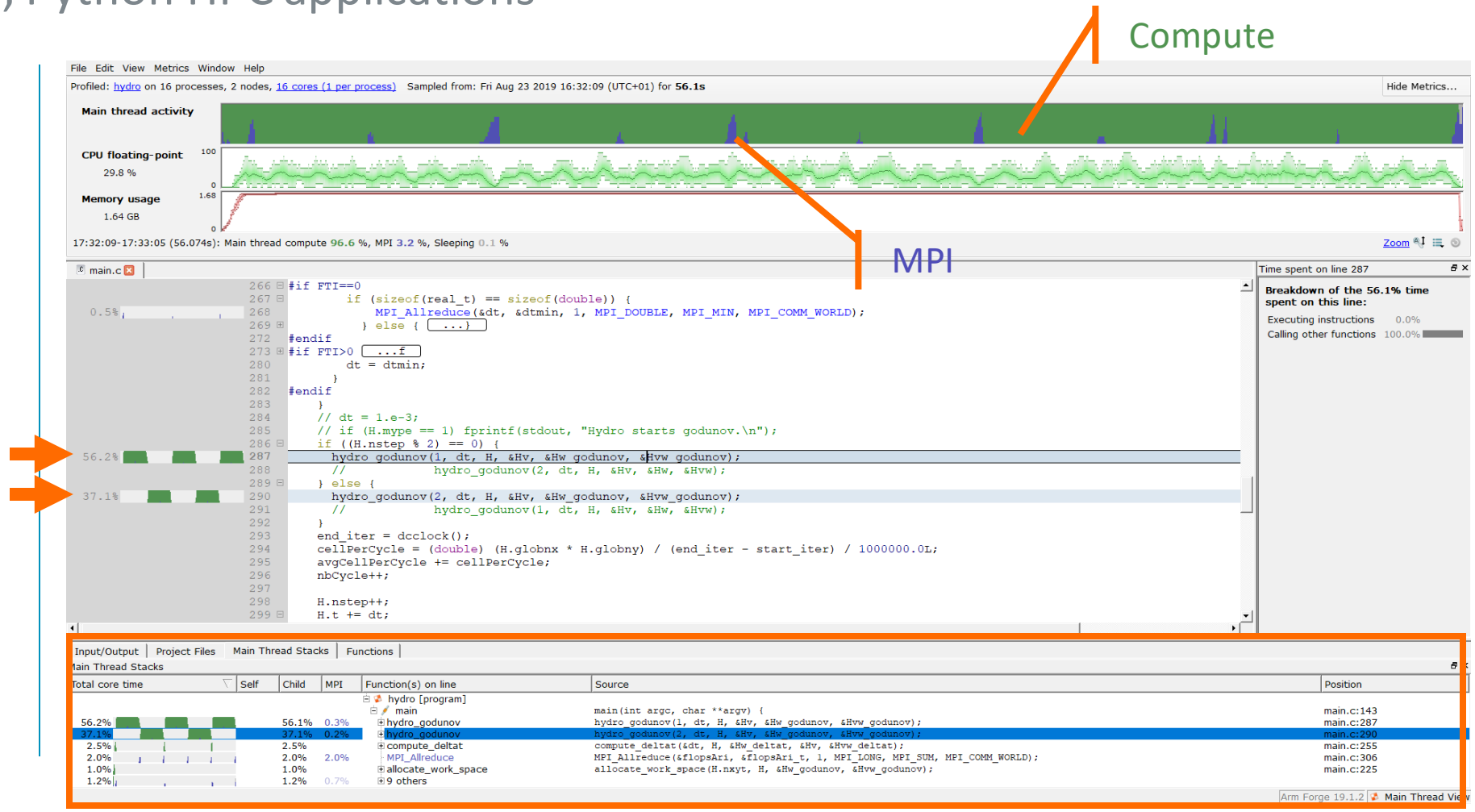
Outline

- Introduction to Arm MAP
- Regional application profiling with Caliper
- Profiling Caliper-instrumented application with Arm MAP
- Analyzing profiling results
- Extracting regional profiling data for code optimization

Performance Analysis with MAP, part of Arm Forge

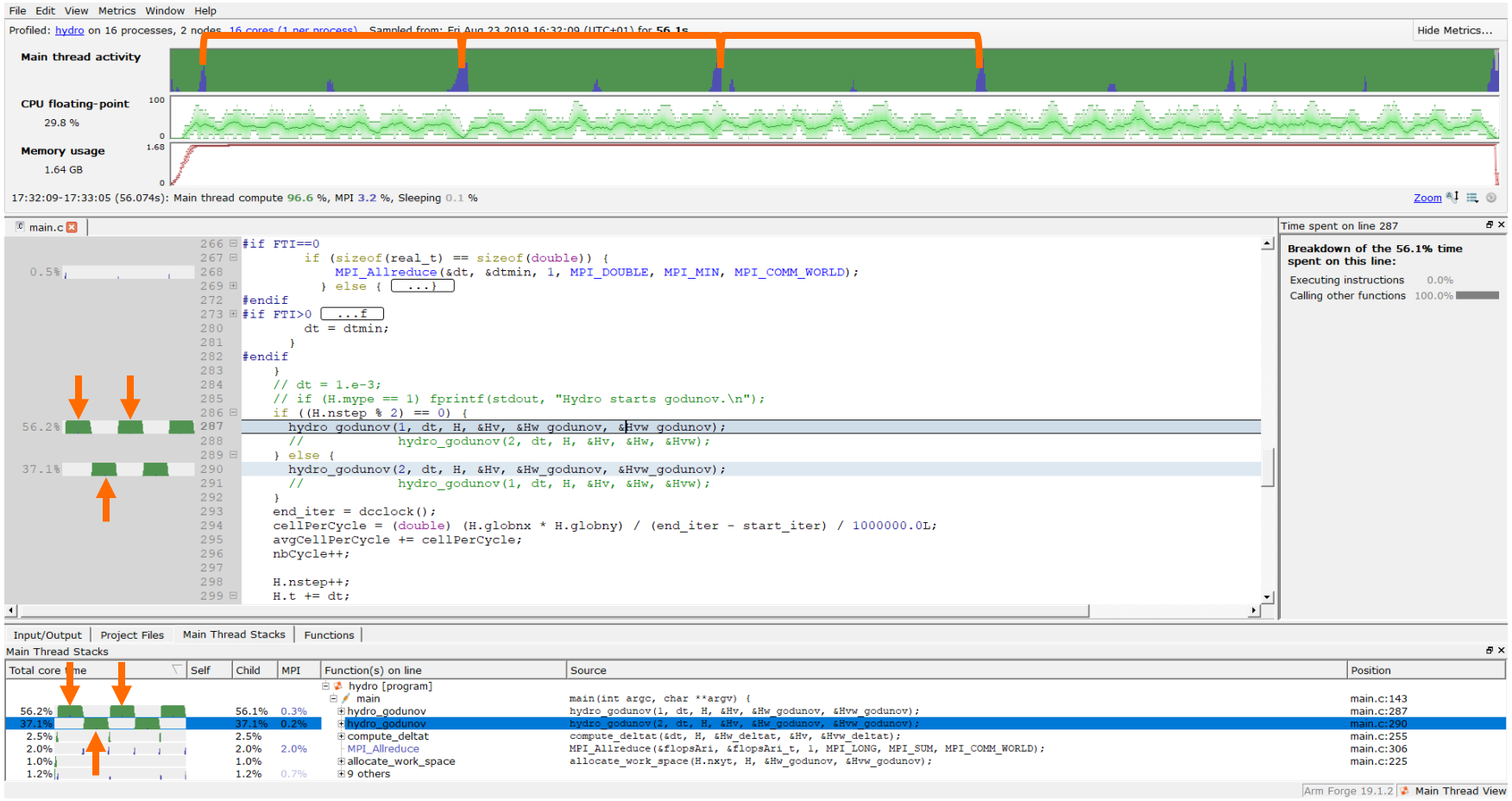
For C/C++, FORTRAN, Python HPC applications

- Easy to use
  - `$ map mpirun ./myapp`
- Lightweight
  - Results can be viewed offline
- Highly scalable
  - For GPGPU, OpenMP, MPI and hybrid applications
- Cross-platform
  - Arm v8-A, x86_64 and POWER



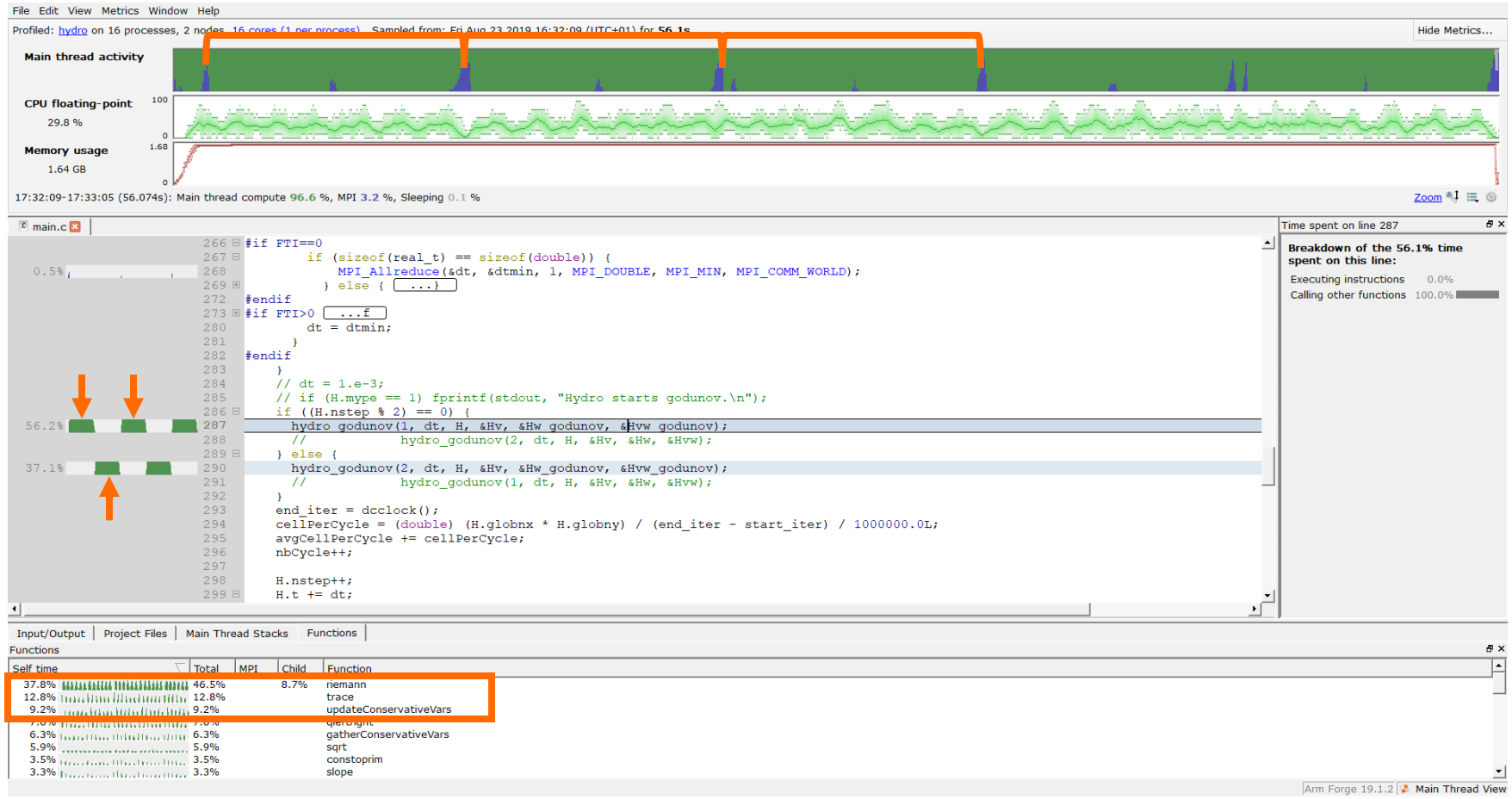
Analyzing profiling results

- Example based on HydroC
  - https://github.com/HydroBench/Hydro
- 2 nodes, 16 processes
- Runtime: 56s
- Iterative pattern



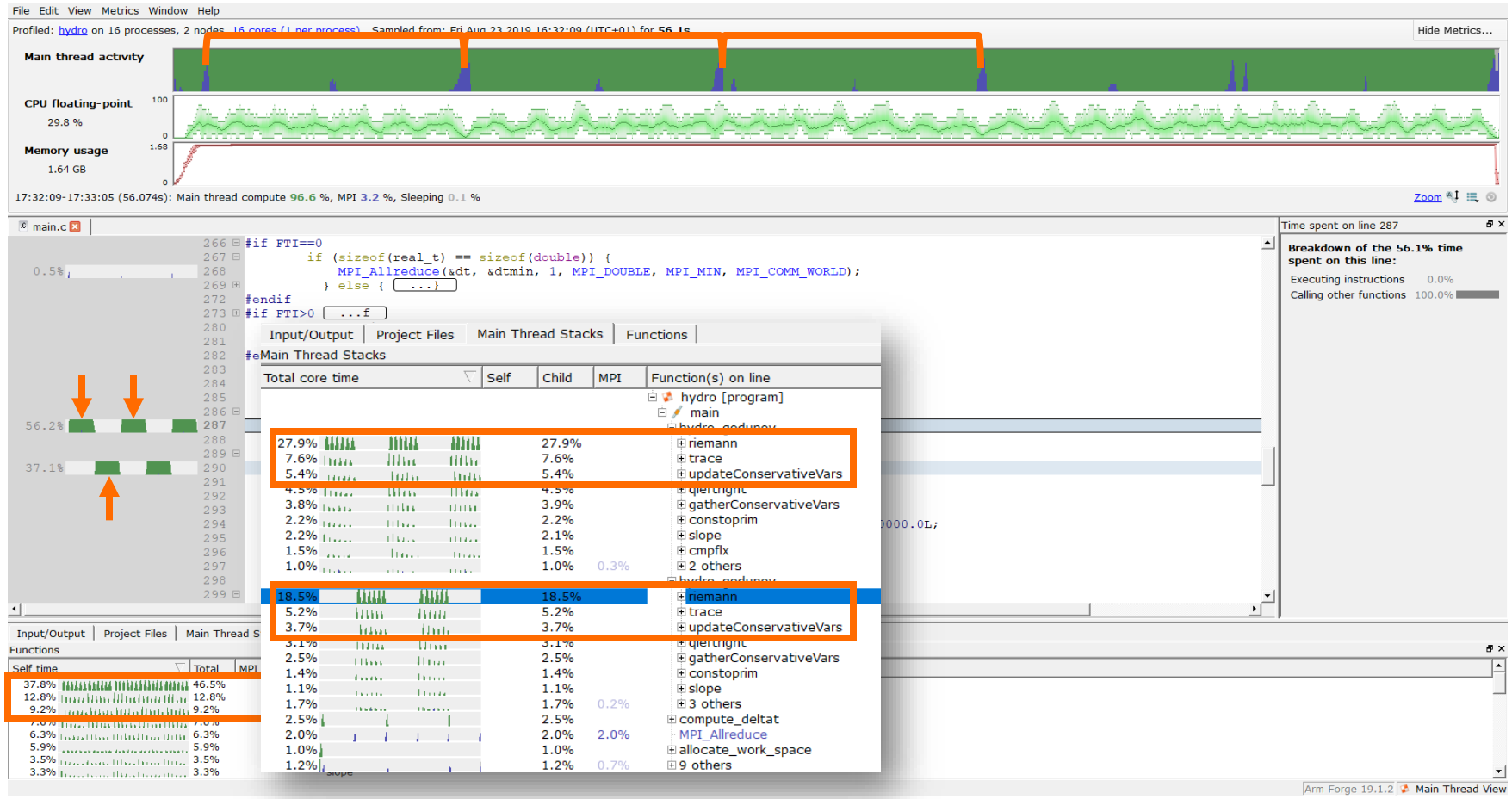
Analyzing profiling results

- 2 nodes, 16 processes
- Runtime: 55s
- Iterative pattern
- Bottlenecks:
  - ***riemann***
  - ***trace***
  - ***updateConservativeVars***



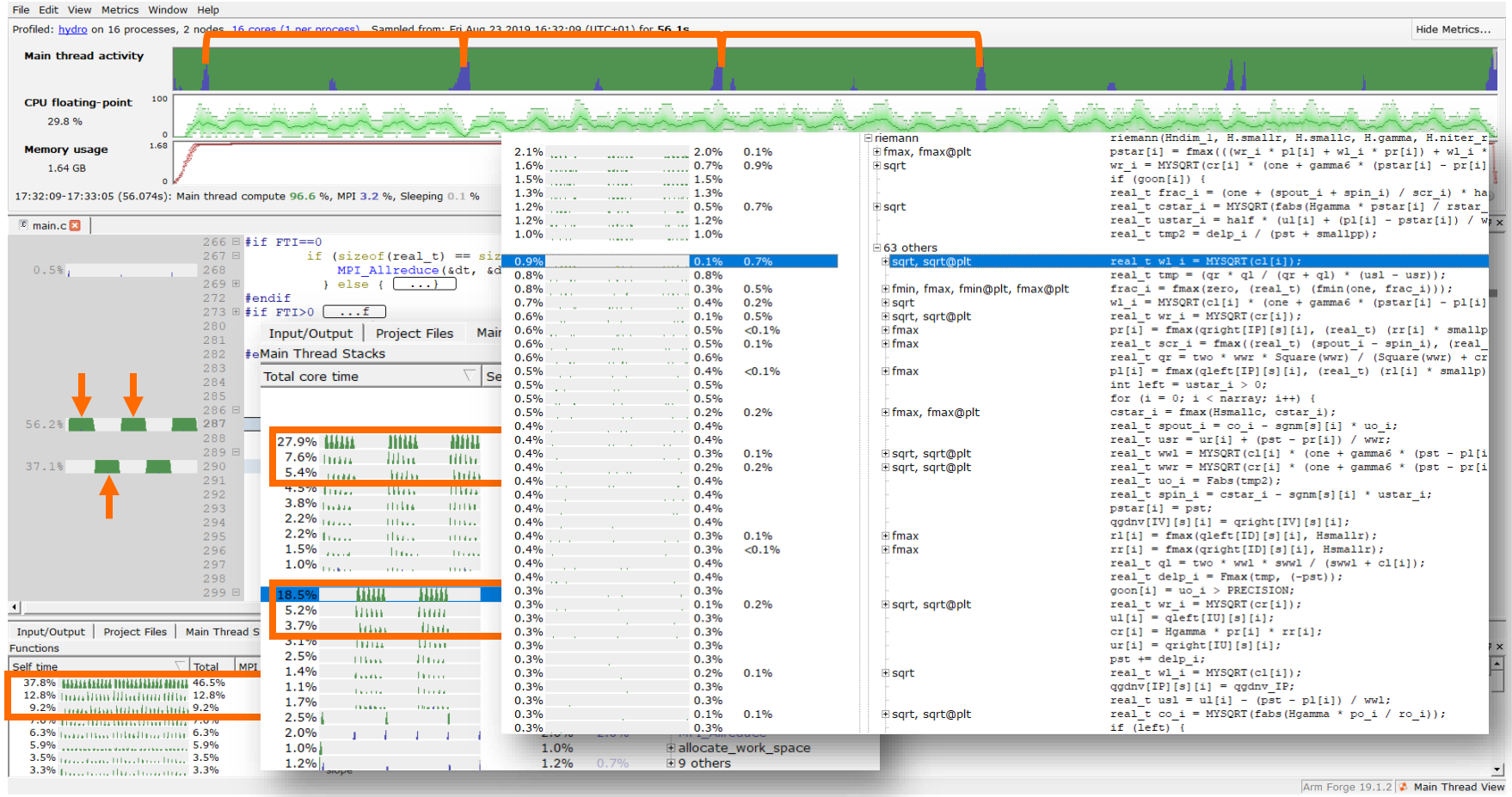
Analyzing profiling results

- 2 nodes, 16 processes
- Runtime: 55s
- Iterative pattern
- Bottlenecks:
  - ***riemann***
  - ***trace***
  - ***updateConservativeVars***
- Called in 2 code paths



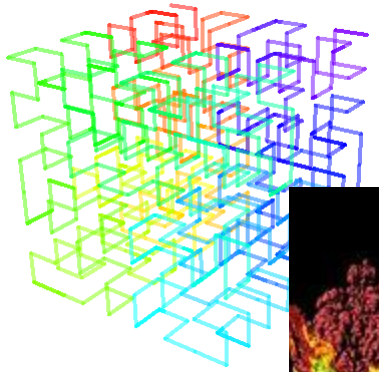
Analyzing profiling results

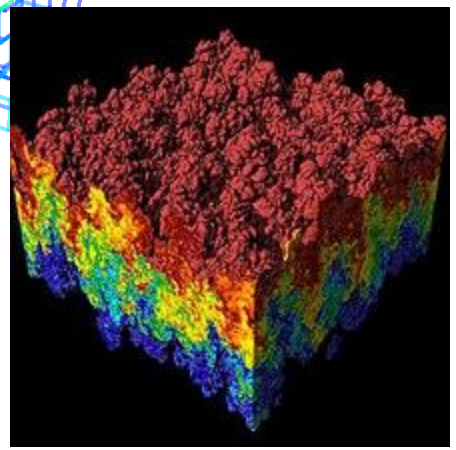
- 2 nodes, 16 processes
- Runtime: 55s
- Iterative pattern
- Bottlenecks:
  - ***riemann***
  - ***trace***
  - ***updateConservativeVars***
- Called in 2 code paths
- *updateConservativeVars*
  - 77% in 3 lines of code
  - Total of 70 lines of code
- *trace*
  - 50% in 11 lines of code
  - Total of 200 l.o.c.
- *riemann*: flat profile
  - 44% in 7 lines of code
  - Total of 340 l.o.c



Code region instrumentation with Caliper

- Performance data collection and analysis tool
  - https://computing.llnl.gov/projects/caliper
- Instrumentation-based for C, C++, FORTRAN codes
- Provides application and performance introspection
- Features:
  - Profiling, tracing of code regions,
  - MPI, PAPI stats
  - Annotations for third-party tools: TAU, Nvprof, now Arm MAP





Instrument functions with Caliper

- Let's instrument the following functions:
  - *riemann*
  - *trace*
  - *updateConvervativeVars*
  - And a few others: *qleftright, gatherConvervativeVar, constoprim, slope*

- Example: riemann.c

```
void riemann(int narray, const real_t Hsmallr, … )
{
  int i, s, ii, iimx;
  …

  CALI_MARK_FUNCTION_BEGIN;
  […]
  CALI_MARK_FUNCTION_END;
}
```

Instrument code sections with Caliper

- In *riemann*, the scientists left a few comments to describe the physics taking place
- Pseudo-code:

```
#compute pressure, density, velocity for each slice
for(s=0; s<slices; s++)
{
            #precompute values for this slice
            for(i=0; i<narray; i++)
            { […] }
            #solve the Riemann problem on the interfaces of this slice
            for(iter=0; iter<Hniter_riemann; iter++)
            { […] }
            #main Riemann solver on arrays of this slice
            for(i=0; i<narray; i++)
            { […] }
}
```

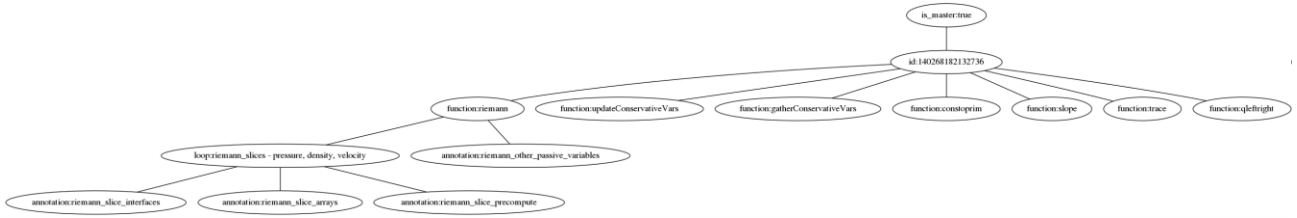
Caliper

```
CALI_MARK_LOOP_BEGIN(riemann_slice_id);
#compute pressure, density, velocity for each slice
for(s=0; s<slices; s++)
{
            CALI_MARK_ITERATION_BEGIN(riemann_slice_id, s);
            CALI_MARK_BEGIN("riemann_slice_precompute");
            for(i=0; i<narray; i++)
            { […] }
            CALI_MARK_END("riemann_slice_precompute");
            CALI_MARK_BEGIN("riemann_slice_interfaces");
            for(iter=0; iter<Hniter_riemann; iter++)
            { […] }
            CALI_MARK_END("riemann_slice_interfaces");
            CALI_MARK_BEGIN("riemann_slice_arrays");
            for(i=0; i<narray; i++)
            { […] }
            CALI_MARK_END("riemann_slice_arrays");
}
CALI_MARK_LOOP_END(riemann_slice_id);
```

Generating Caliper reports

- Modules or “services” provide measurement data
  - E.g. PAPI, CUPTI
- Can be configured through configuration files and environment variables
  - $ CALI_CONFIG_PROFILE=runtime-report ./app
- Output can be in stdout or in *.cali files for post-processing

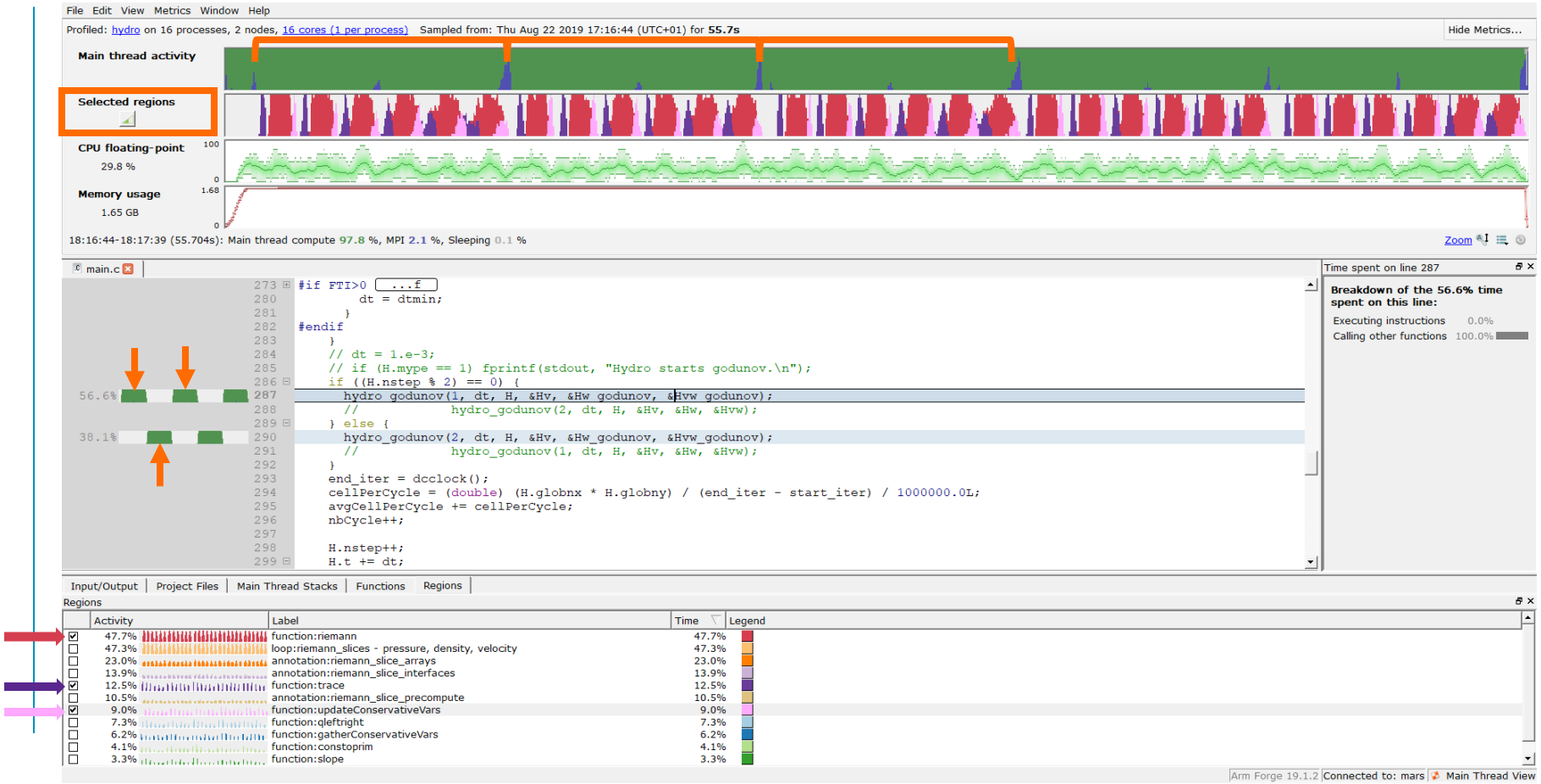
Path	Inclusive time (usec)	Exclusive time (usec)	Time (%)
updateConservativeVars	1637063.000000	1637063.000000	7.955088
riemann	8586175.000000	1317.000000	0.006400
riemann_slices - pressure	8584307.000000	1190957.000000	5.787295
riemann_slice_arrays	2907784.000000	2907784.000000	14.129986
riemann_slice_interfaces	2873562.000000	2873562.000000	13.963688
riemann_slice_precompute	1612004.000000	1612004.000000	7.833317
qleftright	1787885.000000	1787885.000000	8.687987
trace	2218274.000000	2218274.000000	10.779404
slope	1037166.000000	1037166.000000	5.039969
constoprim	1195203.000000	1195203.000000	5.807928
gatherConservativeVars	1559916.000000	1559916.000000	7.580202




```
$ cali-query -s "function=riemann" -a "sum(count)" -f "%[10]function% %[40]annotation% %[50]loop% %[10]count%" -o cali_selected.out
-T "FUNCTION ANNOTATIONS LOOP COUNT" hydro_0.cali

FUNCTION   ANNOTATIONS                          LOOP                                          COUNT

riemann    riemann_other_passive_variables                                                   30

riemann    riemann_slice_precompute             riemann_slices - pressure, density, velocity  30000

riemann    riemann_slice_interfaces             riemann_slices - pressure, density, velocity  30000

riemann    riemann_slice_arrays                 riemann_slices - pressure, density, velocity  30000
```

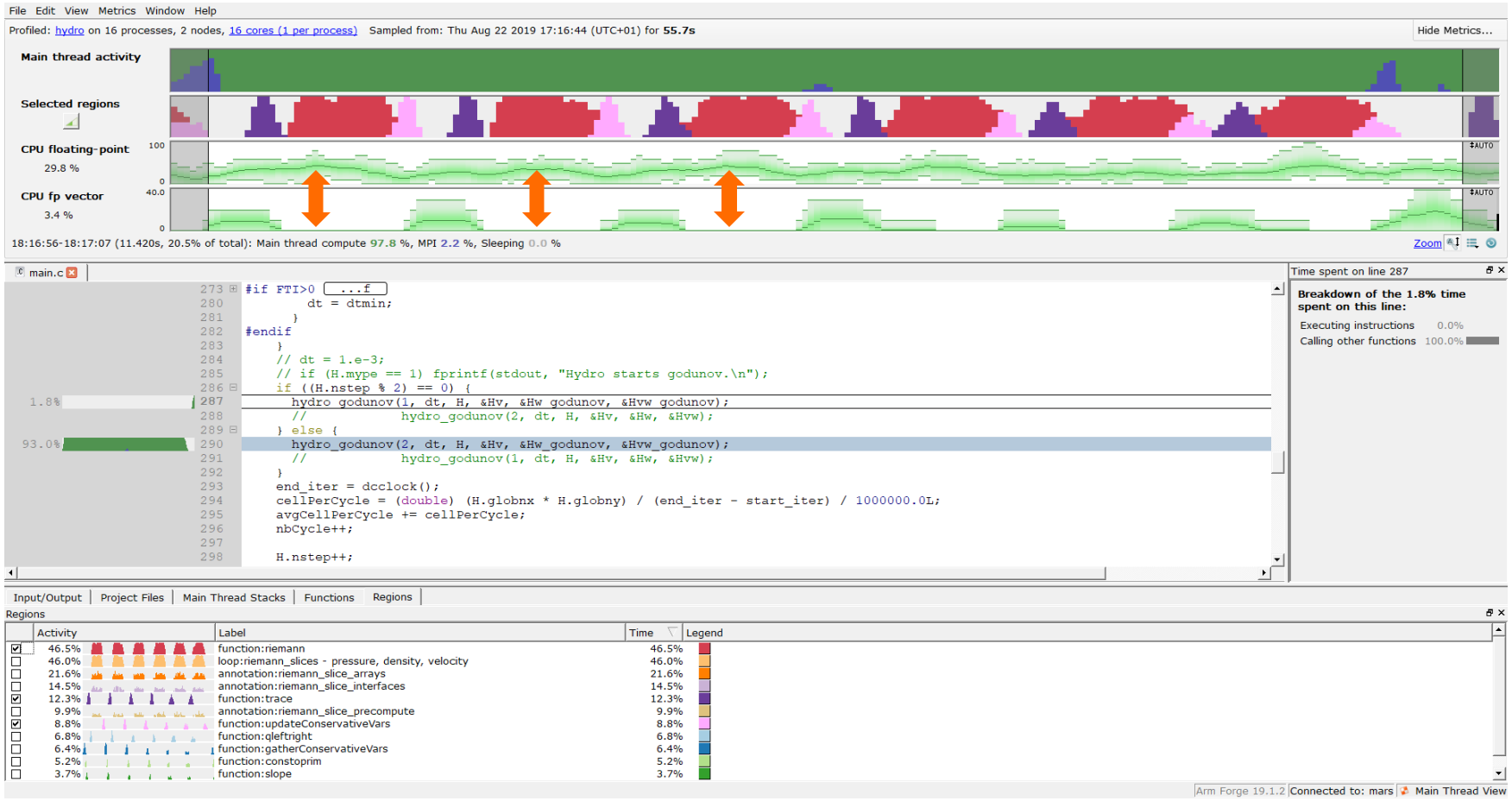
Visualize Caliper data in Arm MAP

- A “Selected region” is displayed
- Caliper functions can be selected
- Iterative patterns appear more clearly
- The timeline shows how bottleneck functions are called
  - *riemann*
  - *trace*
  - *updateConservativeVars*



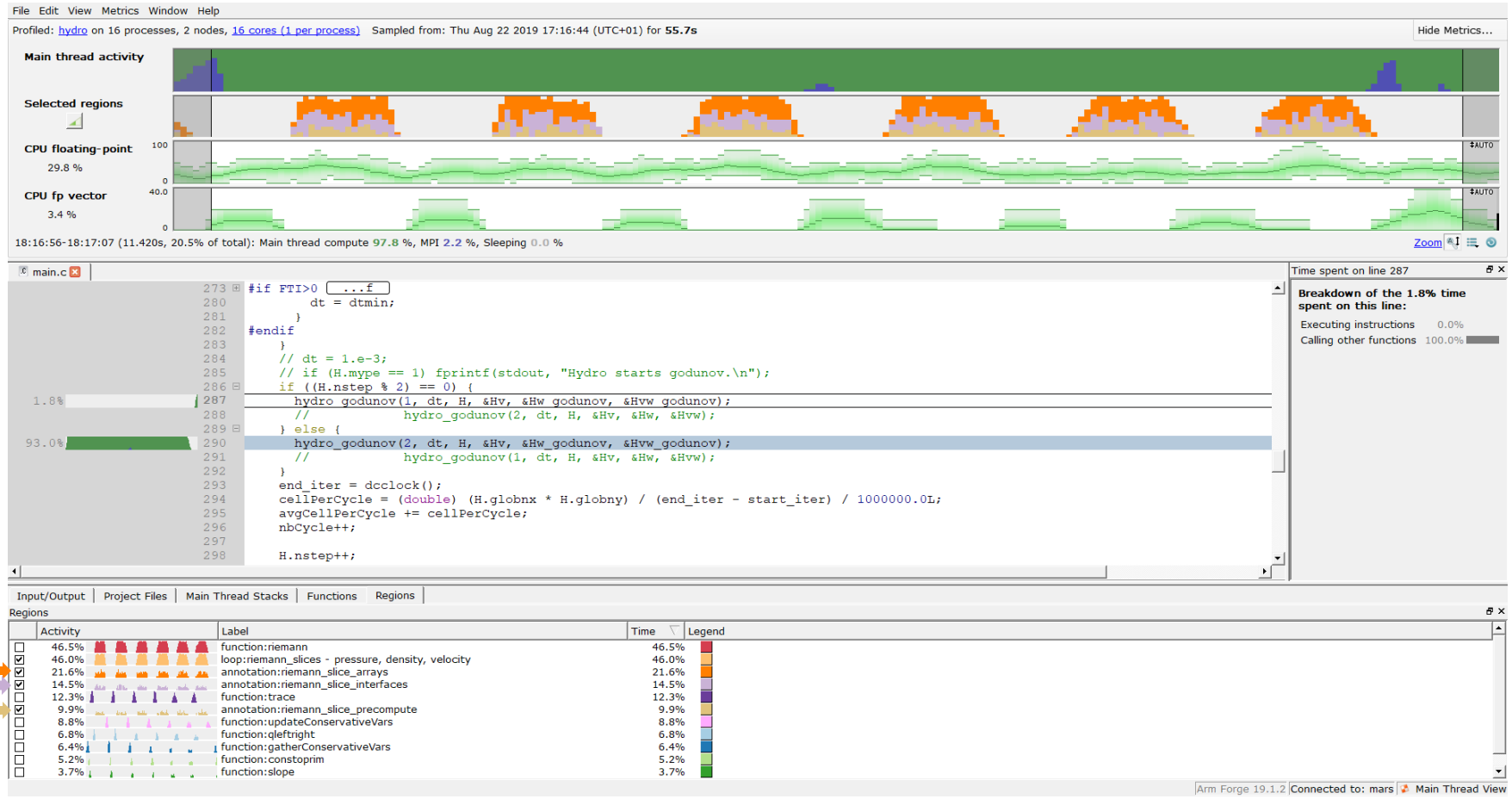
Analyze Caliper data in Arm MAP

- Zooming in enables to correlate function calls with metrics
- e.g. the example show peak CPU floating point when the *riemann* function is executed
- However, no vectorization is performed



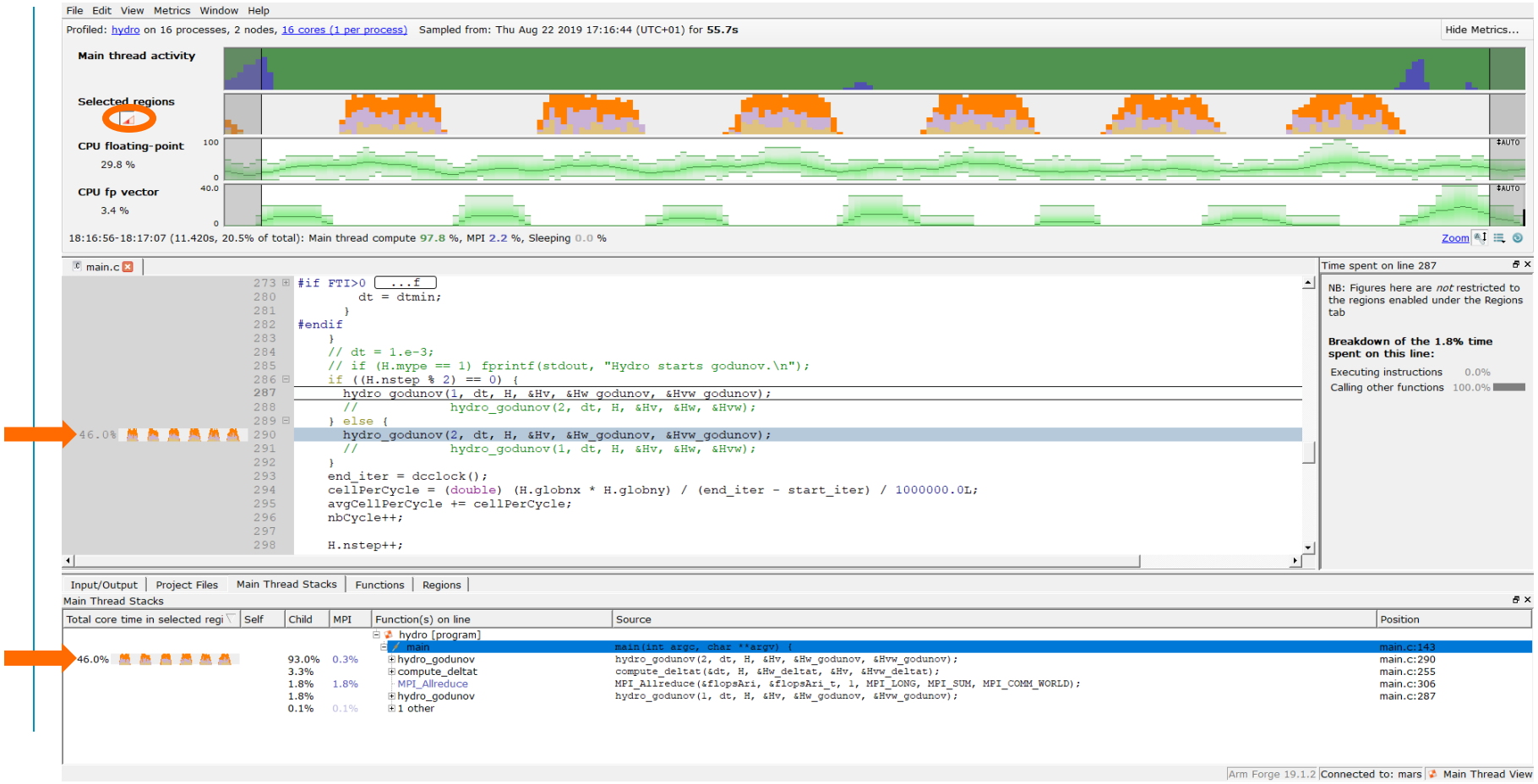
Analyze Caliper data in Arm MAP

- Enabling *riemann* code regions give insights about how the time is spent inside the function
  - **22%** in slice array solver
  - **15%** in slice interface solver
  - **10%** in slice precompute region



Analyze Caliper data in Arm MAP

- The “Activate regions focused view” is available
- It displays Caliper information in the source code annotations and in the “Main thread stack”



Ensure vectorization of specific code regions

- The “Main thread stack” gives code semantic information
- In 1 iteration
  - 5 lines (~10%) in slice array solver
  - 4 lines (~7%) in slice interface solver
  - 1 line (~3%) in slice precompute region
- When selecting the source code lines, the code appear not to be vectorized

Main Thread Stacks

Total core time in selected regi	Self	Child	MPI	Function(s) on line	Source
				riemann	riemann(Hndim_1, H.smallr, H.smallc, H.gamma, H.niter_
3.4%	3.2%	0.2%		fmax, fmax@plt	pstar[i] = fmax(((wr_i * pl[i] + wl_i * pr[i]) + wl_i
2.7%	1.4%	1.3%		sqrt	wr_i = MYSQRT(cr[i] * (one + gamma6 * (pstar[i] - pr[i
2.5%	2.5%				if (goon[i]) {
2.2%	2.2%				real_t ustar_i = half * (ul[i] + (pl[i] - pstar[i]) /
1.9%	1.9%				real_t frac_i = (one + (spout_i + spin_i) / scr_i) * h
1.9%	0.7%	1.2%		sqrt	real_t cstar_i = MYSQRT(fabs(Hgamma * pstar[i] / rstar
1.8%	1.8%				real_t tmp2 = delp_i / (pst + smallpp);
1.6%	1.6%				real_t tmp = (qr * ql / (qr + ql) * (usl - usr));
1.3%	1.3%				real_t qr = two * wwr * Square(wwr) / (Square(wwr) + c
1.2%	1.0%	0.2%		sqrt	wl_i = MYSQRT(cl[i] * (one + gamma6 * (pstar[i] - pl[i

/home/flebeau/Code/Demo_caliper/HydroC_mod/Src/riemann.c:170

0.15% of selected core time on the main thread was in **2** functions (*fmax, fmax@plt*) called from this line
3.23% of selected core time on the main thread in nonlfunction code on this line

3.38% of the selected core time was in the instrumented region annotation:riemann_slice_precompute

main.c | hydro_godunov.c | riemann.c

```
0.2%  155    for (i = 0; i < narray; i++) {
0.6%  156      rl[i] = fmax(qleft[ID][s][i], Hsmallr);
0.2%  157      ul[i] = qleft[IU][s][i];
0.9%  158      pl[i] = fmax(qleft[IP][s][i], (real_t) (rl[i] * smallp));
0.6%  159      rr[i] = fmax(qright[ID][s][i], Hsmallr);
0.5%  160      ur[i] = qright[IU][s][i];
0.8%  161      pr[i] = fmax(qright[IP][s][i], (real_t) (rr[i] * smallp));
      162
      163      // Lagrangian sound speed
0.4%  164      cl[i] = Hgamma * pl[i] * rl[i];
0.4%  165      cr[i] = Hgamma * pr[i] * rr[i];
      166      // First guess
      167
1.0%  168      real_t wl_i = MYSQRT(cl[i]);
0.5%  169      real_t wr_i = MYSQRT(cr[i]);
3.4%  170      pstar[i] = fmax(((wr_i * pl[i] + wl_i * pr[i]) + wl_i * wr_i * (ul[i] - ur[i])) / (wl_i + wr_i), 0.0);
0.2%  171      goon[i] = 1;
      172    }
      173  #ifdef CALIPER
0.2%  174    CALI_MARK_END("riemann_slice_precompute");
      175  #endif
```

Time spent on line 170

NB: Figures here are *not* restricted to the regions enabled under the Regions tab

Breakdown of the 3.4% time spent on this line:

Executing instructions	95.5%
Calling other functions	4.5%

Time in instructions executed:

Scalar floating-point	69.8%
Vector floating point	0.0%
Scalar integer	0.0%
Vector integer	0.0%
Memory access*	37.7%
Branch	0.0%
Other instructions	4.7%

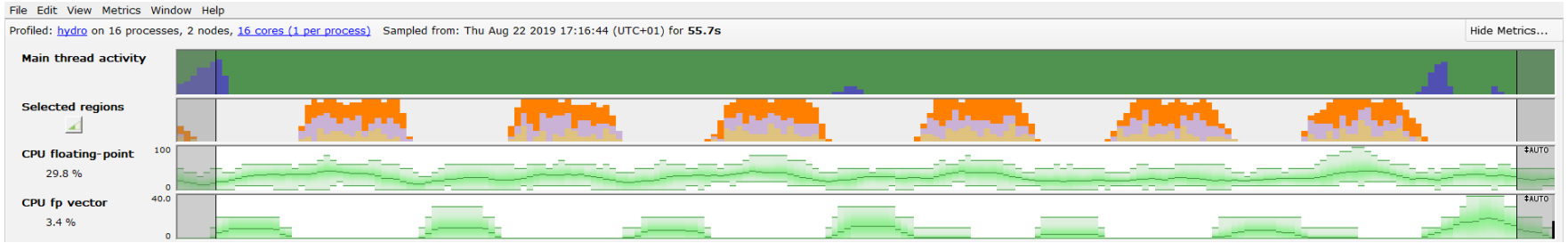
* 25.5% memory access instructions, 12.3% implicit memory accesses in other instructions, also counted in their categories

Ensure vectorization of specific code regions

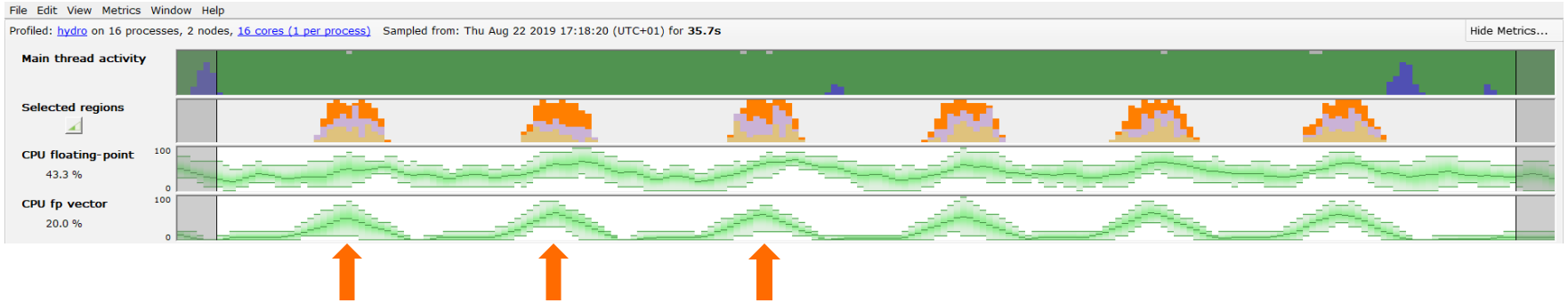
Use of #pragma omp simd

- 1.5 speedup
- Increased floating point performance
  - 43% vs. 30%
  - 20% vector vs. 3%

Without pragma



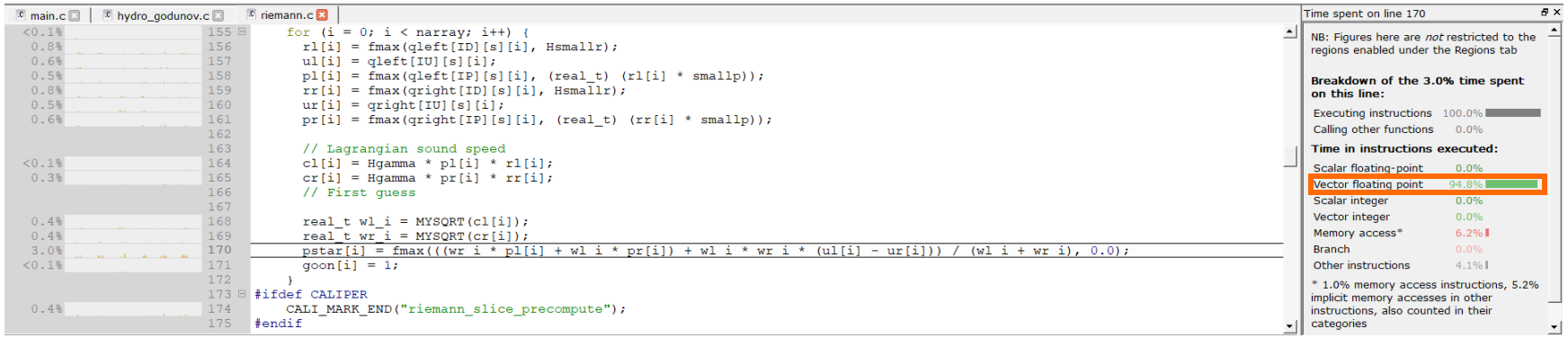
With pragma

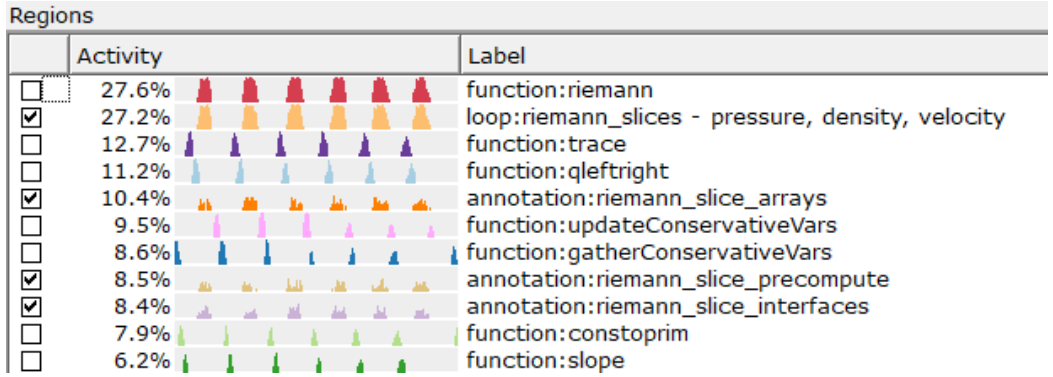


Ensure vectorization of specific code regions

Use of #pragma omp simd

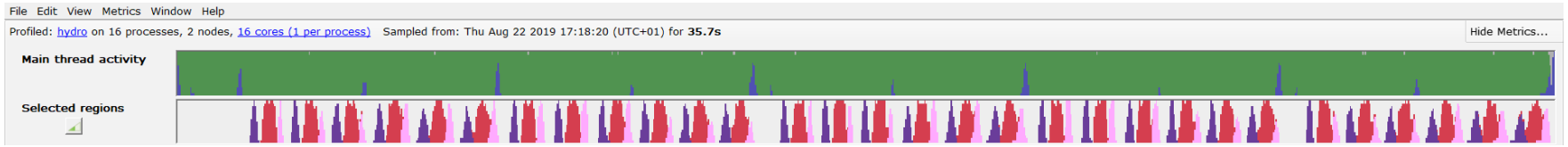
- 1.5 speedup
- Increased floating point performance
  - 43% vs. 30%
  - 20% vector vs. 3%
- Lines of code are well vectorized
- Riemann profile
  - **10%** in slice array solver
  - **8%** in slice interface solver
  - **9%** in slice precompute region





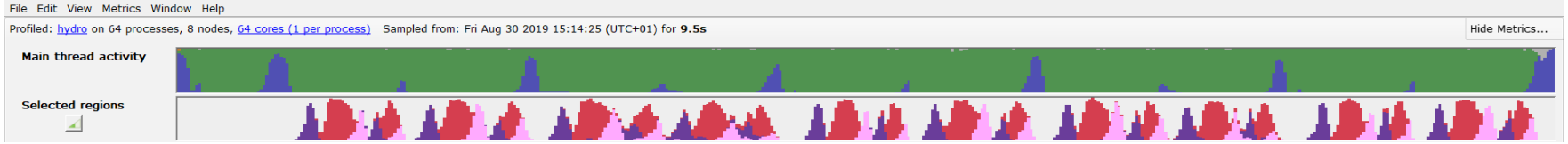
Scale up

- 2 nodes



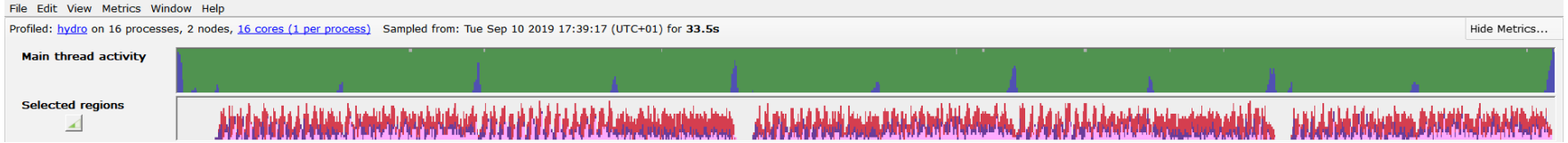
- 4 nodes

- 8 nodes

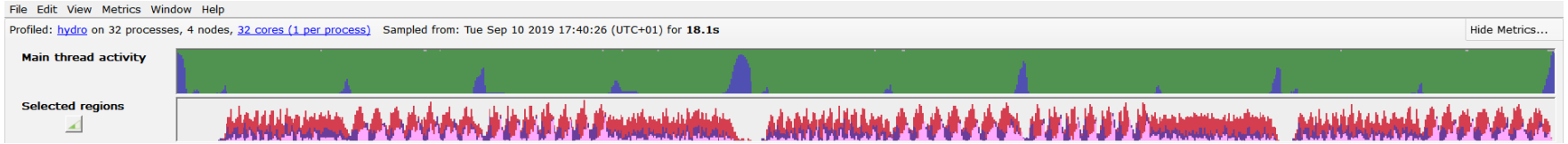


Scale up – with different input data

- 2 nodes



- 4 nodes



- 8 nodes

Conclusion: topics covered

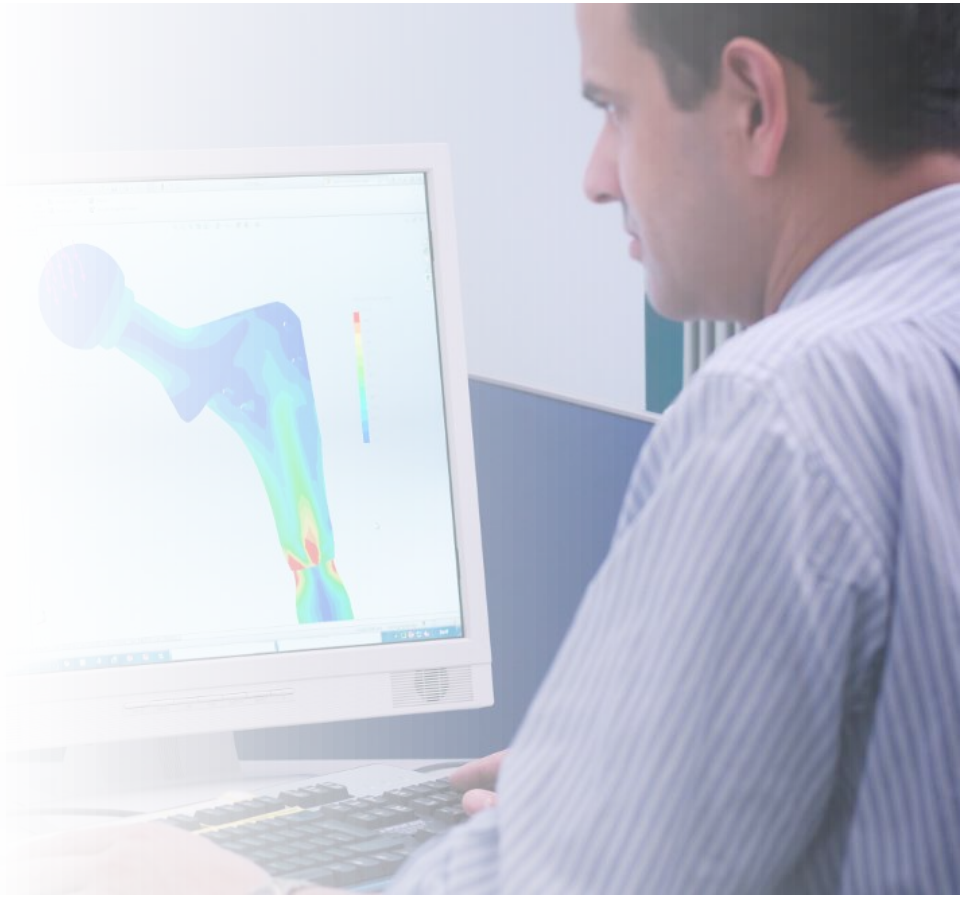
- Profiling at scale with Arm MAP
- Use Caliper annotations to define code regions
  - Supported by MAP version 19.1 onwards
  - Add application insight to profiling results
- Visualize results
  - Understand application behavior faster
  - Optimize code more efficiently

Request a trial license

- If you would like to profile your own Caliper-instrumented applications, visit our website

- A trial licence of Arm Forge enables:
  - Multi-node and multi-threaded profiling with Arm MAP
  - On all HPC architectures
  - On C/C++, FORTRAN and Python codes
  - Debugging with Arm DDT
  - Access to our support channel

- Plenty of resources are available online for your trial: developer.arm.com/hpc



arm

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
תודה

Questions

arm

arm