Bill Williams
IAK, ZIH

A Tracing-Oriented Approach to Parallel Performance Engineering

NLPE@HLRS
6 JUN 2025

What you don’t need to hear

Performance engineering matters

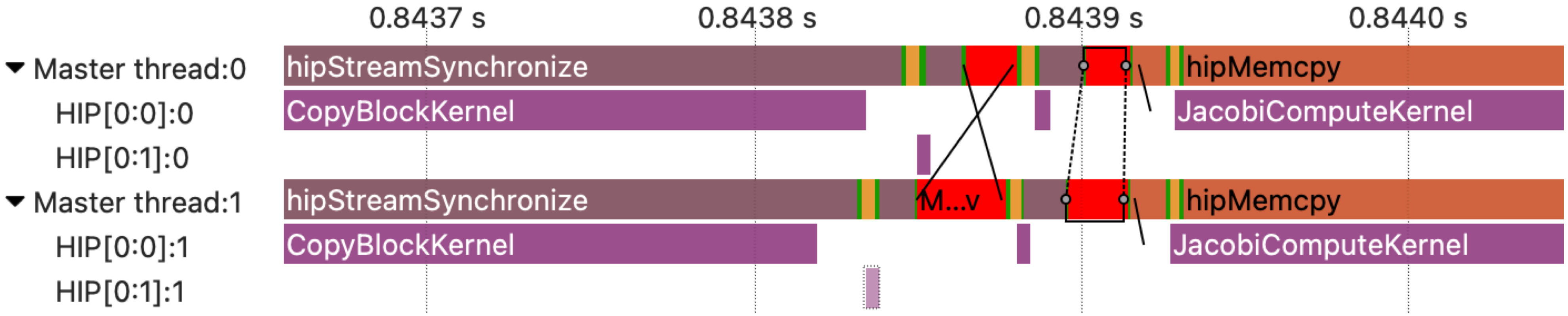
We live in a (nearly) post-Moore’s-law world, thus parallelism matters

Modeling and sane decisions about measurement are critical

What we offer

All-in-one measurement system: collection of profiling and tracing data with the same framework

The ability to collect unified, detailed data from a variety of sources:
- Process-level parallelism (MPI, OpenSHMEM)
- Thread-level parallelism (OpenMP, Pthreads)
- Accelerators (CUDA, ROCm, OpenACC, OpenCL)
- I/O operations
- Compiler instrumentation
- User instrumentation (NVTX, Score-P annotations, ROCTX)



An overview of measurement techniques

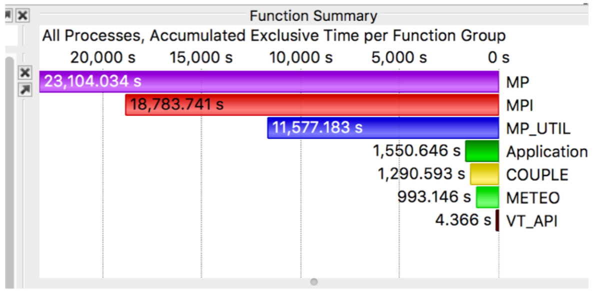
Sampling vs. **instrumentation**

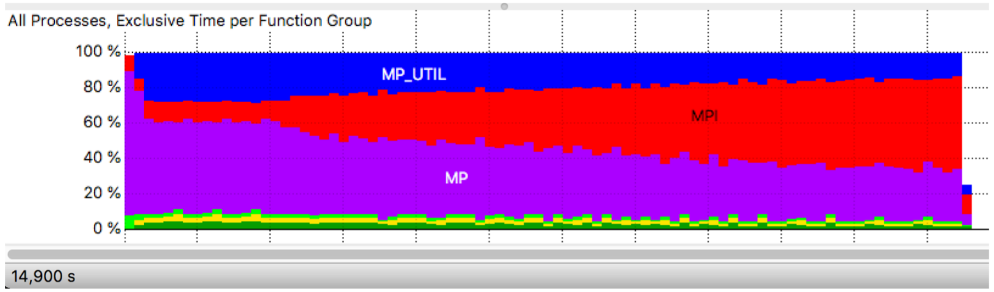
Profiling vs. **tracing**

Filtering approaches: paradigm vs. phase vs. region, compile-time vs. runtime

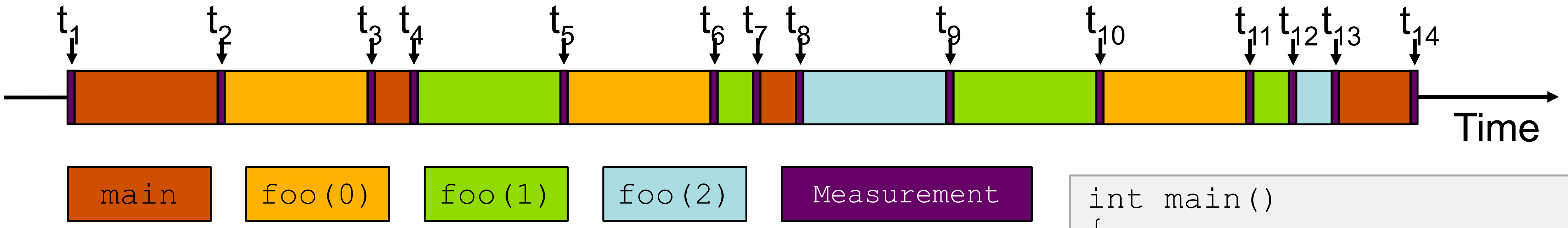
When to trace: the interesting behavior is *dynamic* with respect to place and time

— Summary: “we spent 30% of our time in MPI wait states.” Bad, but possibly necessary?
— Trace: “we went from 10% of our time in MPI wait states to 60% over the course of the run.” Clearly a load balancing problem!





Instrumentation



Measurement code is inserted such that every event of interest is captured directly

- Can be done in various ways

Advantage:

- Much more detailed information

Disadvantage:

- Processing of source-code / executable necessary
- Large relative overheads for small functions

```
int main()
{
  int i;
  Enter("main");
  for (i=0; i < 3; i++)
    foo(i);
  Leave("main");
  return 0;
}

void foo(int i)
{
  Enter("foo");
  if (i > 0)
    foo(i - 1);
  Leave("foo");
}
```

Tracing

- Recording detailed information about significant points (events) during execution of the program
  - Enter / leave of a region (function, loop, …)
  - Send / receive a message, …
- Save information in event record
  - Timestamp, location, event type
  - Plus event-specific information (e.g., communicator, sender / receiver, …)
- Abstract execution model on level of defined events

☞ *Event trace = Chronologically ordered sequence of event records*

Tracing Pros & Cons

- Tracing advantages
  - Event traces preserve the **temporal** and **spatial** relationships among individual events (☞ context)
  - Allows reconstruction of **dynamic** application behaviour on any required level of abstraction
  - Most general measurement technique
    - Profile data can be reconstructed from event traces
- Disadvantages
  - Traces can very quickly become extremely large
  - Writing events to file at runtime may causes perturbation

Dealing with trace sizes

One key insight: not every bit of performance data matters to every problem!

- Does it not affect your performance model?
- Have you already optimized that part of the code?
- Is it not part of the application's critical path?

Throw it out!

The key to manageable traces is good *filtering* of the data we collect

How do we make a filter?

We need to know which code regions:

- are *frequently executed*
- have *little importance* to the current problem
- are *short*

Short and frequently executed == high measurement overhead

Little importance == safe to remove from the measurement

What tells us all of these things? **A profile**

The key insight

The performance data you need to solve many performance problems *without* a trace is the same performance data you need to *collect* a trace with good targeting and efficiency!

Workflow:

- Instrument *once*
- Collect profile
- Possible initial pass of analysis and optimization
- Refine measurement to reduce overhead and allow tracing
- More detailed pass of trace-based analysis

The second key insight

When considering the parallel part of the problem, the serial details are often irrelevant; when considering the serial part, the parallel details are often irrelevant

If it doesn’t lead to a parallel paradigm, we can potentially filter it out!

A sneak preview of the Score-P workflow

1. Collect reference data from uninstrumented application
2. Build in/LD_PRELOAD your instrumentation of choice
3. Configure the environment to control data collection
4. Collect a profile
5. Evaluate, based on the profile data, what causes unneeded measurement overhead: *scoring*
6. Create a *filter* to collect usable measurement data
7. Evaluate program behavior based on this (now low-overhead and meaningful) profile
8. If needed, look more precisely at a trace file, generated only by changing the environment at runtime

Related tools

VIHPS partners

- TAU: general-purpose collection and visualization of profiling and trace data
- CUBE: visualization of Score-P profiles
- Scalasca: automatic analysis of traces produced by Score-P to produce annotated profiles for CUBE
- Paraver/Extrae: general-purpose collection and visualization of profiling and trace data

Other

- Nsight/NVTX: CUDA+NVTX source-level instrumentation
- ROCm tools: ROCm+ROCTX source-level instrumentation
- Dyninst: general-purpose binary instrumentation and modification
- Intel, TotalView: generally very good for understanding single-node behavior

Conclusion: when is Score-P right for you?

- When you want to collect *many* measurements from *one* instrumented build of your application
- When it is important to see *connections and interactions* between various types of parallelism
  - Does an OpenMP imbalance propagate to MPI wait states?
  - Why is my parallel file I/O slow?
  - How effectively do I use the CPU and GPU together?
- When behavior *changes over time*
- When you want to minimize or eliminate manual changes to your source code