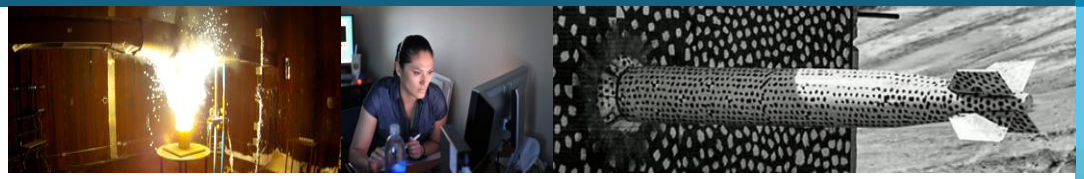
Case Study: Debugging Other People's Libraries via PRELOAD



Chris Siefert and James Elliott

Sandia National Laboratories

Special thanks to Christian Trott

SAND2020-5956 C

Motivation: What in the World is that Library doing?

When using libraries it is often hard to understand what they are doing

- MPI makes this more complicated!
- CUDA even more so!

Profiler output (X happened at time T) are insufficient

- Linear graph is impossible to follow for long enough calculation.
- No stack information (e.g. the code called MPI_Barrier, but who did that??).

Goal: Stack-based output with a special attention to MPI and Cuda.

Method: Use of LD_PRELOAD

Case Study: Understanding how the Kokkos::deep_copy() calls handle device synchronization.

Case Study: Kokkos::deep_copy() on NVIDIA GPUs

For this discussion, Kokkos::deep_copy() does one of two things

- GPU/CPU: Copy data between GPU and CPU memory.
- GPU/GPU: Copies data between two GPU buffers.

These two tasks imply different synch semantics

- GPU-to-CPU: We need to wait until Cuda streams are done working before copying to CPU memory (e.g. call cudaDeviceSynchronize()).
- GPU-to-GPU: For single stream operation, this should just queue up as a regular kernel launch. No sync needed.

Question: Does Kokkos actually do that correctly? How can we tell?

Hand inspection won't cut it (Kokkos is too complicated). We could add lots of printf's… but there's a better way.

Method: LD_PRELOAD

We use a PRELOAD mechanism to intercept MPI and Cuda library calls and dlsym to call the "real" function.

Use a Teuchos::TimeMonitor to wrap the calls.

This integrates with Teuchos::StackedTimers which give us stack-based output, across MPI ranks.

Caveats: Requires Shared builds.

Goal: Release this tool as part of Trilinos.

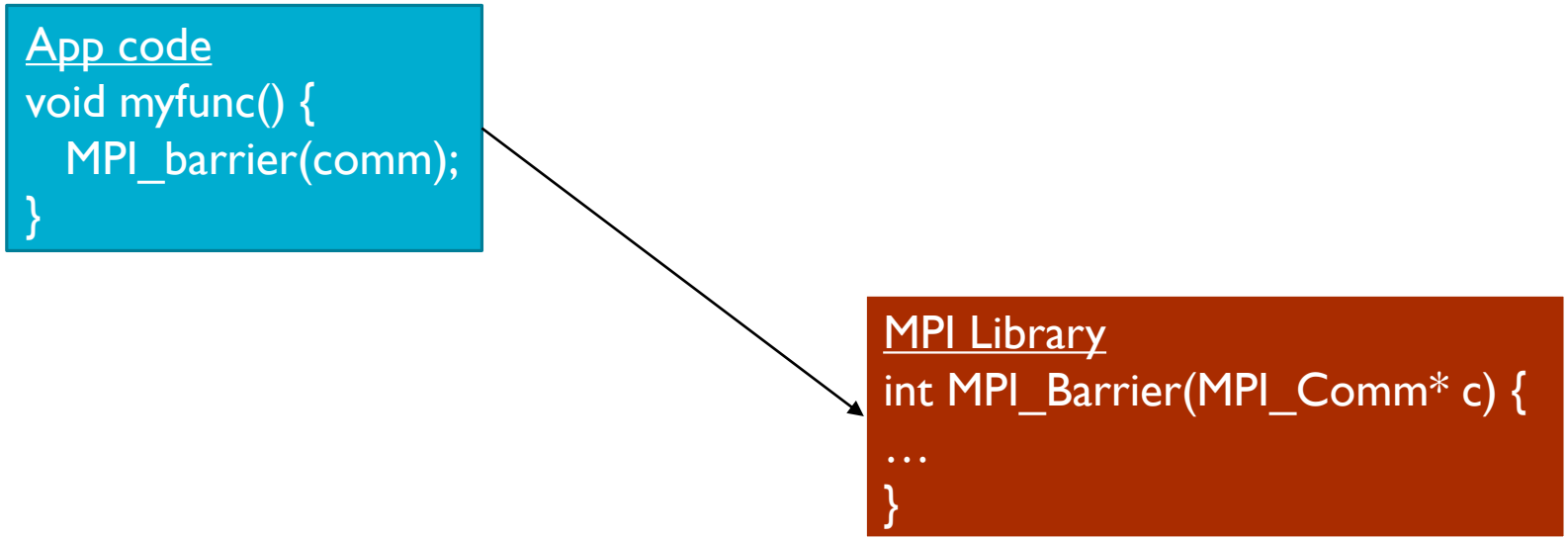
Method: LD_PRELOAD … How does it work?

A very common R&D problem is having a binary and wanting to understand what that binary is doing

- Having access to the original source is not guaranteed, and even if you do, you may not know what hacks/edits/adulterations went into it.
- If you do have the source, build times can be prohibitive, and you run the risk of building in a manner different from the original developer.

Solution:

- Use shared libraries and inject.



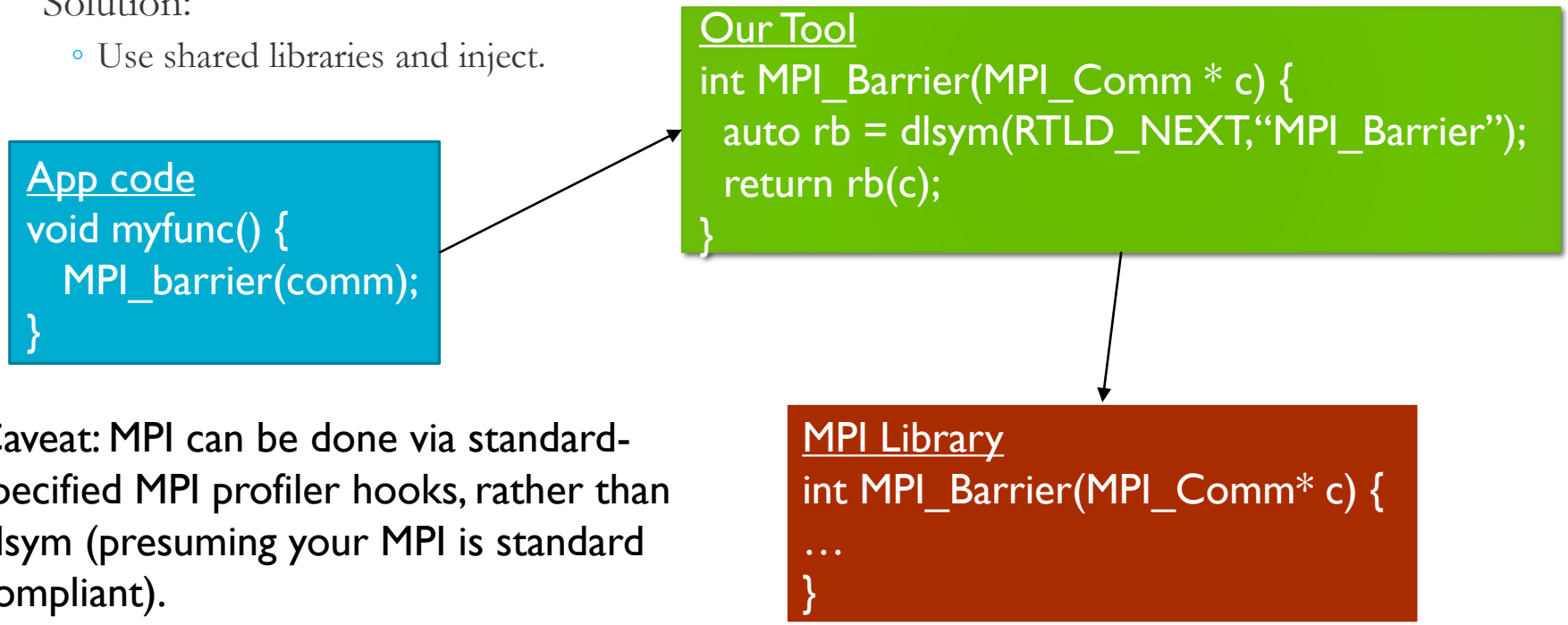
Method: LD_PRELOAD … How does it work?

A very common R&D problem is having a binary and wanting to understand what that binary is doing

- Having access to the original source is not guaranteed, and even if you do, you may not know what hacks/edits/adulterations went into it.
- If you do have the source, build times can be prohibitive, and you run the risk of building in a manner different from the original developer.

Solution:

- Use shared libraries and inject.



Caveat: MPI can be done via standard-specified MPI profiler hooks, rather than dlsym (presuming your MPI is standard compliant).

Kokkos::deep_copy() Test Code

```
{ // This uses the GPU-to-CPU style semantic
  Teuchos::TimeMonitor timer2(*Teuchos::TimeMonitor::getNewTimer("deep_copy(v2,v1) x3"));
  Kokkos::deep_copy(v2,v1);
  Kokkos::deep_copy(v2,v1);
  Kokkos::deep_copy(v2,v1);
}
{ // This uses the GPU-to-GPU style semantic
  Teuchos::TimeMonitor timer2(*Teuchos::TimeMonitor::getNewTimer("deep_copy(space,v3,v1) x3"));
  Kokkos::deep_copy(MySpace,v3,v1);
  Kokkos::deep_copy(MySpace,v3,v1);
  Kokkos::deep_copy(MySpace,v3,v1);
}
```

Test Code Output

```
{ // This uses the GPU-to-CPU style semantic
 // deep_copy(v2,v1) timer
  Kokkos::deep_copy(v2,v1);
   Kokkos::deep_copy(v2,v1);
   Kokkos::deep_copy(v2,v1);
 }
 { // This uses the GPU-to-GPU style semantic
//    deep_copy(space,v3,v1) timer
   Kokkos::deep_copy(MySpace,v3,v1);
   Kokkos::deep_copy(MySpace,v3,v1);
   Kokkos::deep_copy(MySpace,v3,v1);
 }
```

```
Driver: 0 [1]
|   deep_copy(v2,v1) x3: 0.000599708 - 7.44129% [1]
|   |   cudaDeviceSynchronize: 6.5433e-05 - 10.9108% [6]
|   |   cudaMemcpy: 6.8193e-05 - 11.371% [3]
|   |   Remainder: 0.000466082 - 77.7182%
|   deep_copy(space,v3,v1) x3: 3.9182e-05 - 0.486178% [1]
|   |   cudaMemcpyAsync: 2.9618e-05 - 75.5908% [3]
|   |   Remainder: 9.564e-06 - 24.4092%
|   Remainder: -0.00805919
```

Results for Kokkos 3.1. Kokkos 3.0 didn't do this right *and this tool helped us expose the issue.*

Conclusions

Used existing profiler API (Teuchos Timers)

- Developers familiar usage and standard output.

Intercept tools developed independent of app profiled

- Went from Trilinos test to Kokkos mini-app (previous slide) flawlessly.
- Have used with ATDM apps as well.

Tool provides unadulterated report of API usage

- We discovered API calls we did not expect to find!
- Profiling technique avoids risk of only finding what you intentionally search for.
- Output format is natural for Trilinos users and required no code modifications.