Checkpointing the Un-checkpointable: the Split-Process Approach for MPI and Formal Verification

Gene Cooperman*
gene@ccs.neu.edu

mailto:gene@ccs.neu.edu

Khoury College of Computer Sciences
Northeastern University, Boston, USA

November 15, 2019

*Partially supported by NSF Grants ACI-1440788 and OAC-1740218, and by grants from Intel Corporation, Raytheon, and Mentor Graphics (a division of Siemens).

Table of Contents

Outline

DMTCP History

The DMTCP project and antecedents began almost 15 years ago, and is widely used today:
`http://dmtcp.sourceforge.net/publications.html`

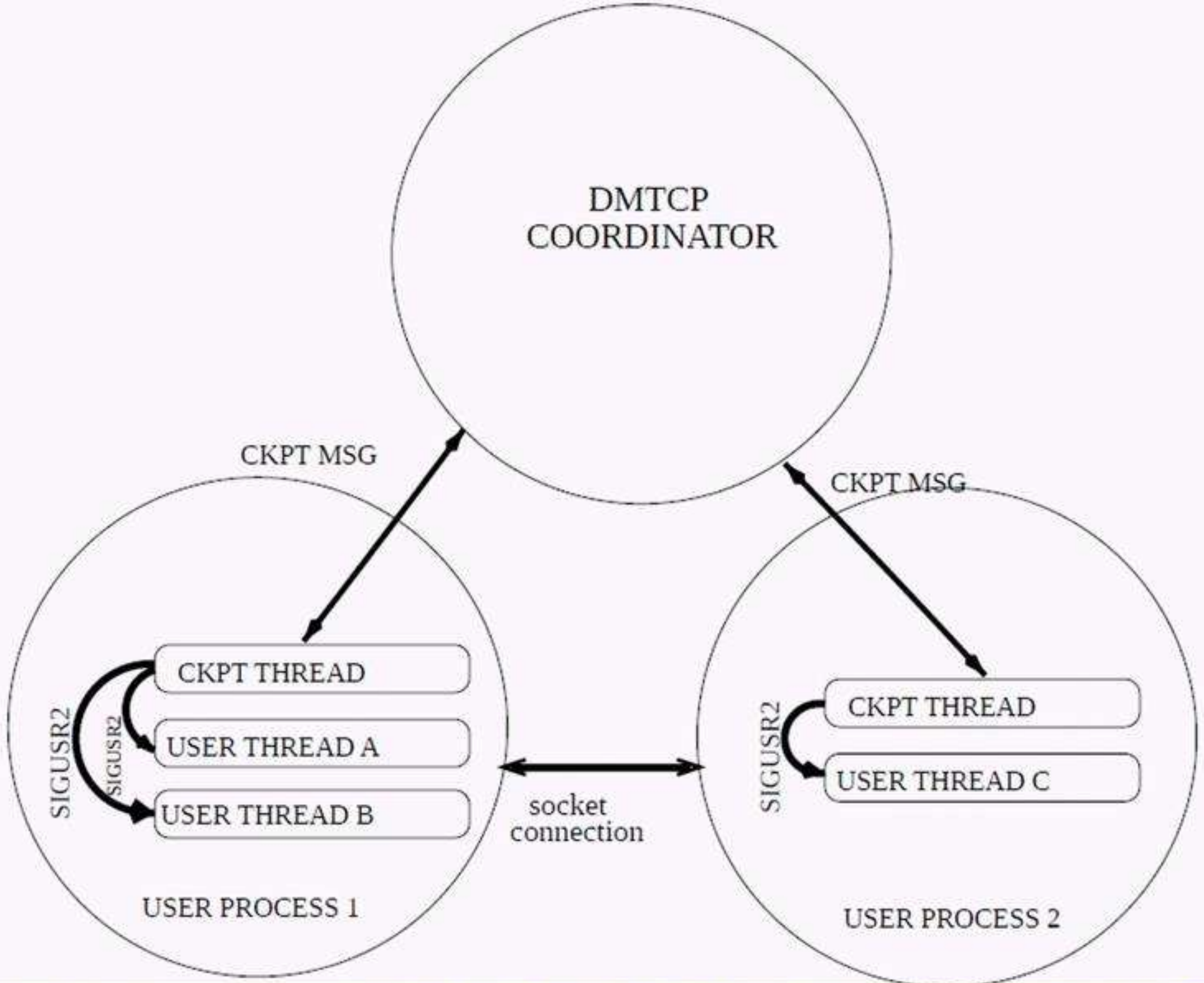
Typical use case for HPC: 12-hour batch time slot, an application is expected to finish in 18 hours.
(NOTE: A resource manager such as SLURM can send a signal one hour before the end of the time slot, giving the application time to checkpoint; DMTCP can then transparently checkpoint the state.)

Ease of use (unprivileged and transparent):

```
 dmtcp_launch a.out arg1 arg2 ...
 dmtcp_command --checkpoint # from other terminal or window
 dmtcp_restart ckpt_a.out_*.dmtcp
```

(DMTCP also works for programs that are multi-threaded, distributed, MPI-based, ...)

DMTCP Architecture: Coordinated Checkpointing



Fundamental Research Question for the DMTCP Team

"What are the limits of checkpointing applications in the real-world?"

PROBLEM: An application may store a session id, tty, network peer address, TMPDIR for temporary files, process id, thread id, etc. On restart, some or all of these are likely to change.

SOLUTION: Process virtualization: interpose on all calls to such ids; replace actual id by DMTCP-defined virtual id.
PRINCIPLE: "Never let the application see a real id!"

Scalability:

Checkpointing HPCG: 32,752 CPU cores), and (NAMD: 16,368 CPU cores) for checkpointing MPI applications natively over InfiniBand at TACC:
"System-level Scalable Checkpoint-Restart for Petascale Computing", Jiajun Cao et al., *Int. Conf. on Parallel and Dist. Sys.* (ICPADS'16), 2016
(To the best of our knowledge, this is ***100 times larger*** *than the previously largest transparent checkpointing study in the literature.)*

Reinvent HPC (Pthread or MPI): Ckpt + Split Processes + SimGrid

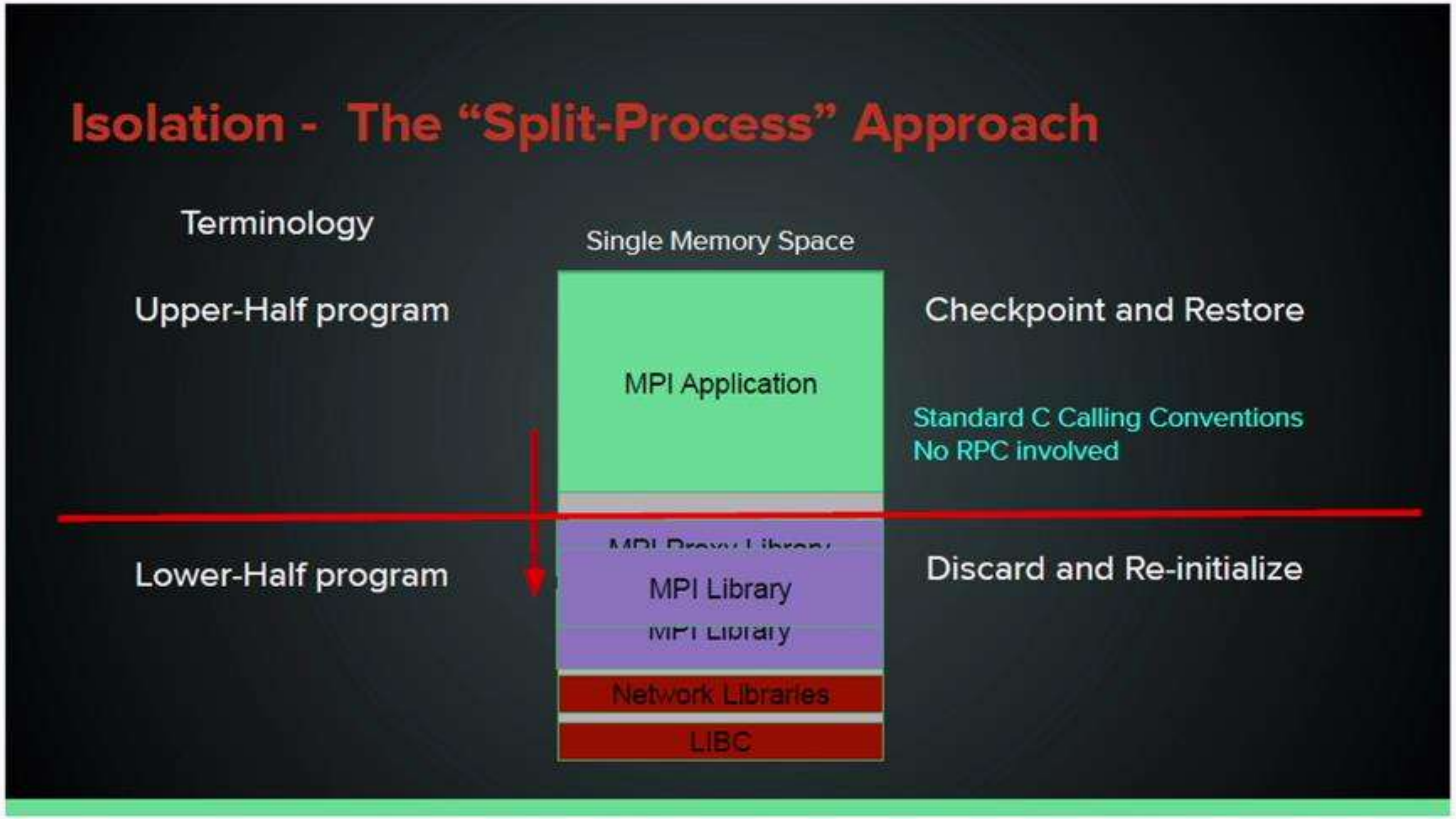
Goal: Efficient, flexible MPI programming (and Pthread and CUDA and ...)

****

Ckpt + Split Processes + SimGrid = **A NEW WORLD**

****

Advantages: Software migration, cross-cluster migration, fault tolerance, model checking with no intermediate model, bug diagnosis even after a long computation, ...



Isolation - The "Split-Process" Approach

Terminology

Upper-Half program

Lower-Half program

Checkpoint and Restore

Discard and Re-initialize

Standard C Calling Conventions
No RPC involved

Outline

DMTCP Plugins

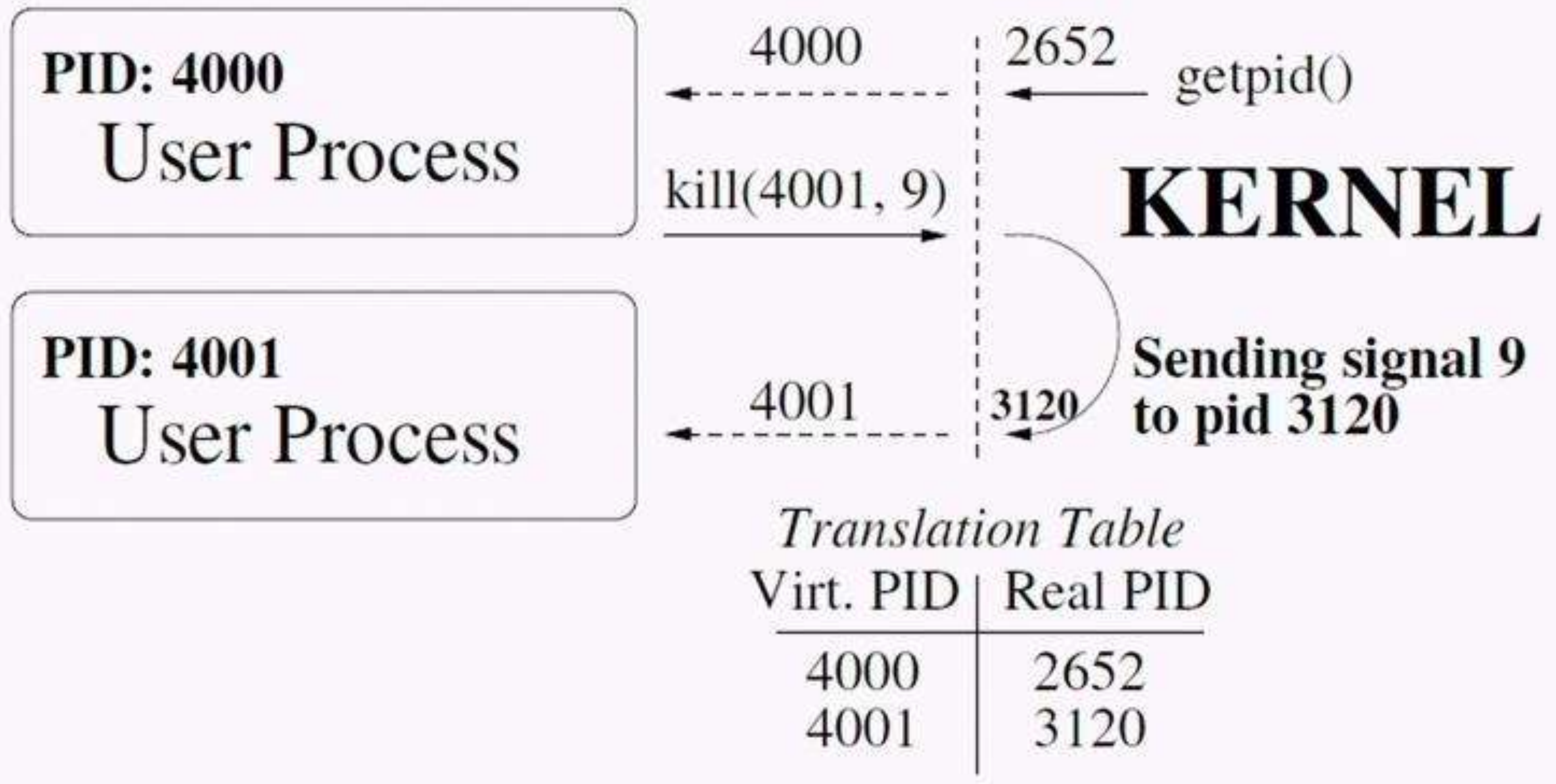
WHY PLUGINS?

- Processes must talk with the rest of the world!
- *Process virtualization:* virtualize the connections to the rest of the world

In short, a plugin is responsible for modeling an external subsystem, and then creating a semantically equivalent construct at the time of restart.

A Simple DMTCP Plugin: Virtualizing the Process Id

- **PRINCIPLE:**
  The user sees only virtual pids; The kernel sees only real pids



Translation Table

Virt. PID	Real PID
4000	2652
4001	3120

Plugins for EDA: A Real-world Example

EDA is "Electronic Design Automation" (circuit design for chips).
Part of a four-year collaboration between DMTCP team and Intel:

> "Be Kind, Rewind — Checkpoint & Restore Capability for Improving Reliability of Large-scale Semiconductor Design", I. Ljubuncic, R. Giri, A. Rozenfeld, and A. Goldis, IEEE HPEC-14, Sept., 2014.
> *(published solely by Intel co-authors)*

Fictional scenario with ball-park numbers (no particular vendor):

- Software circuit simulation: about 1 million times slowdown
- Hardware emulation at back-end: about 1 thousand times slowdown
- Cost of back-end hardware emulator: about $800,000
- **Use case A** (for a new CPU design): Boot Microsoft Windows overnight with emulator, and then test Microsoft Office.
- **Use case B**: Boot Microsoft Windows overnight with emulator, and checkpoint. In later iterations, restart, and then test Microsoft Office.

The above fictional scenario requires a DMTCP plugin to model the back-end emulator. *See publications with emulator vendors for details.*

Outline

MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing

Full paper with details at:

"MANA for MPI: *MPI-Agnostic Network-Agnostic* Transparent Checkpointing", by R. Garg, G. Price, and G. Cooperman, *High Performance Distributed Computing* (HPDC'19)

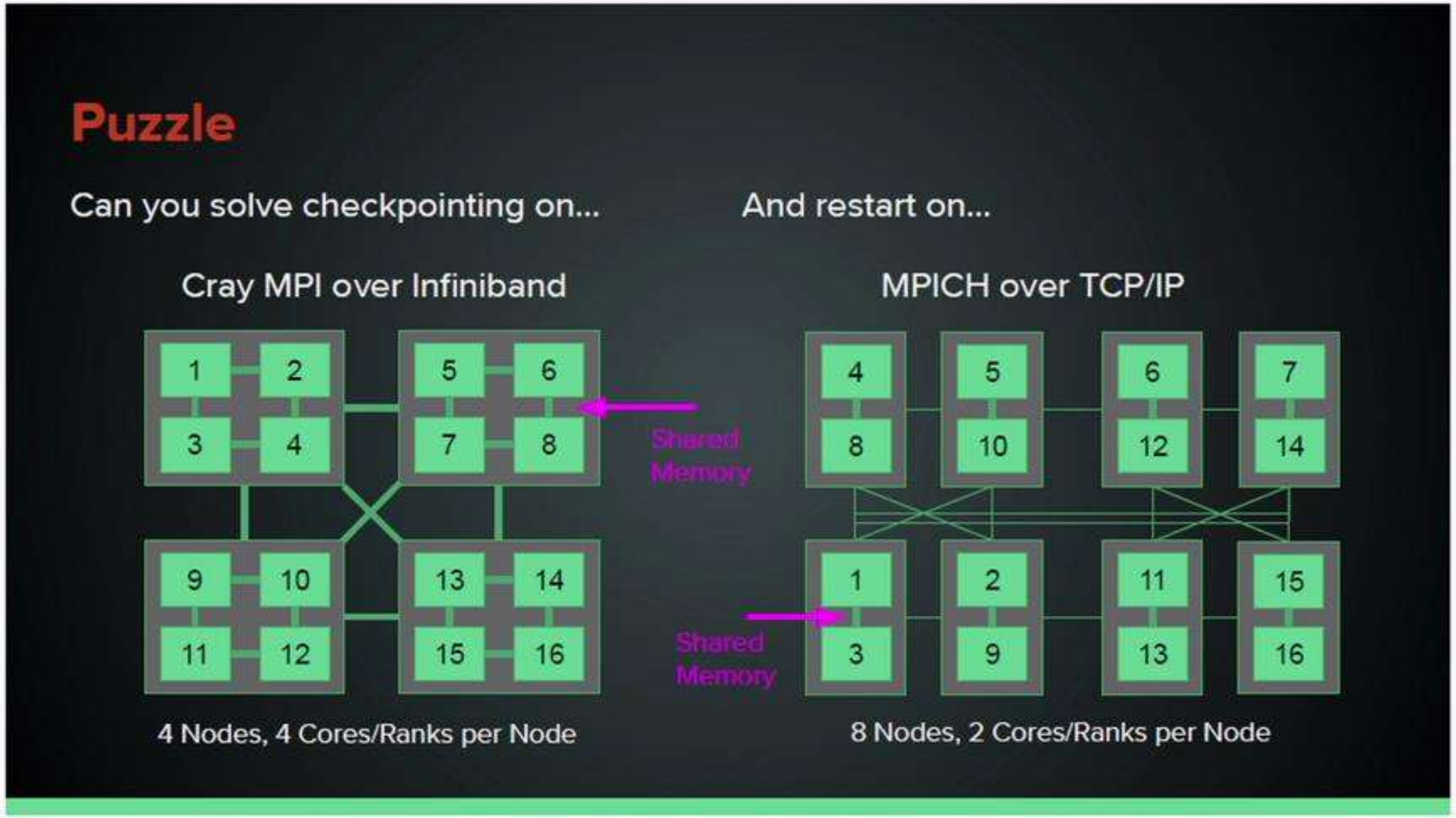
Not just an implementation, but a flexible principle with many applications:

- **IDEA:** Load two independent programs into a *single* process, sharing a single address space. (For a different approach, see "Process-in-process", HPDC'18 from RIKEN et al., employing multiple link maps/dlmopen.)
- **LOW OVERHEAD!** This completely eliminates the overhead of the proxy approach. (No need for shared memory, cross-memory-attach (cma), XPMEM. One program can directly pass an internal pointer to the other program.)
- **Isolate the MPI/network libraries into their own program; completely separate from the program running the MPI application.**

MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing (cont.)

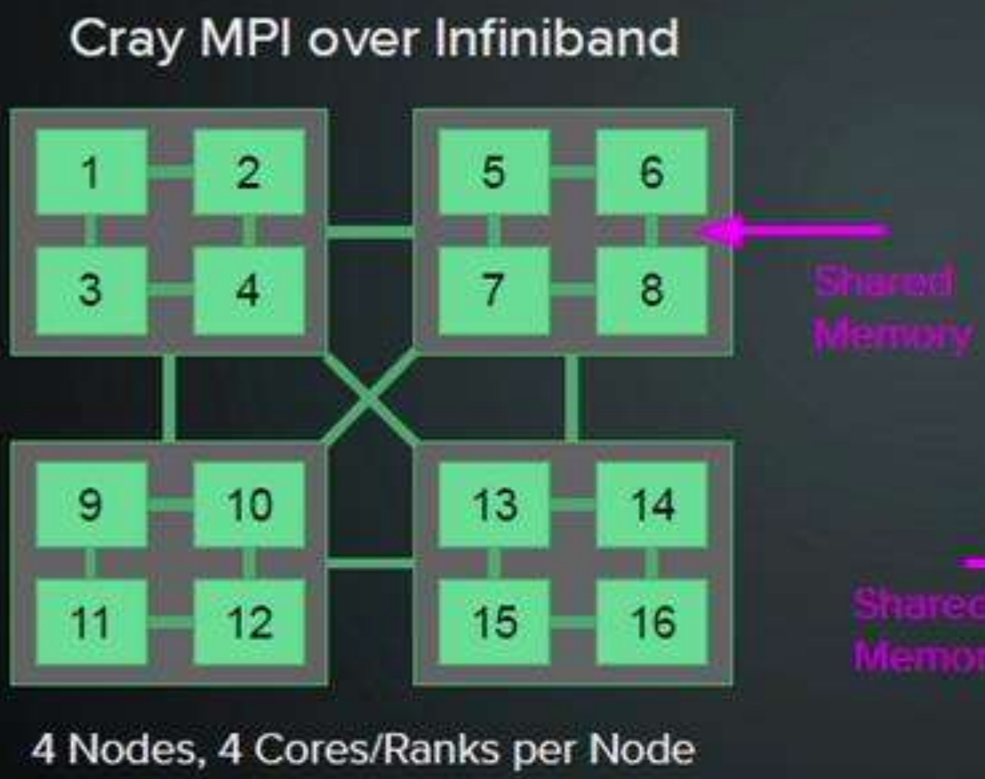
FEATURES OF THE SPLIT-PROCESS APPROACH:

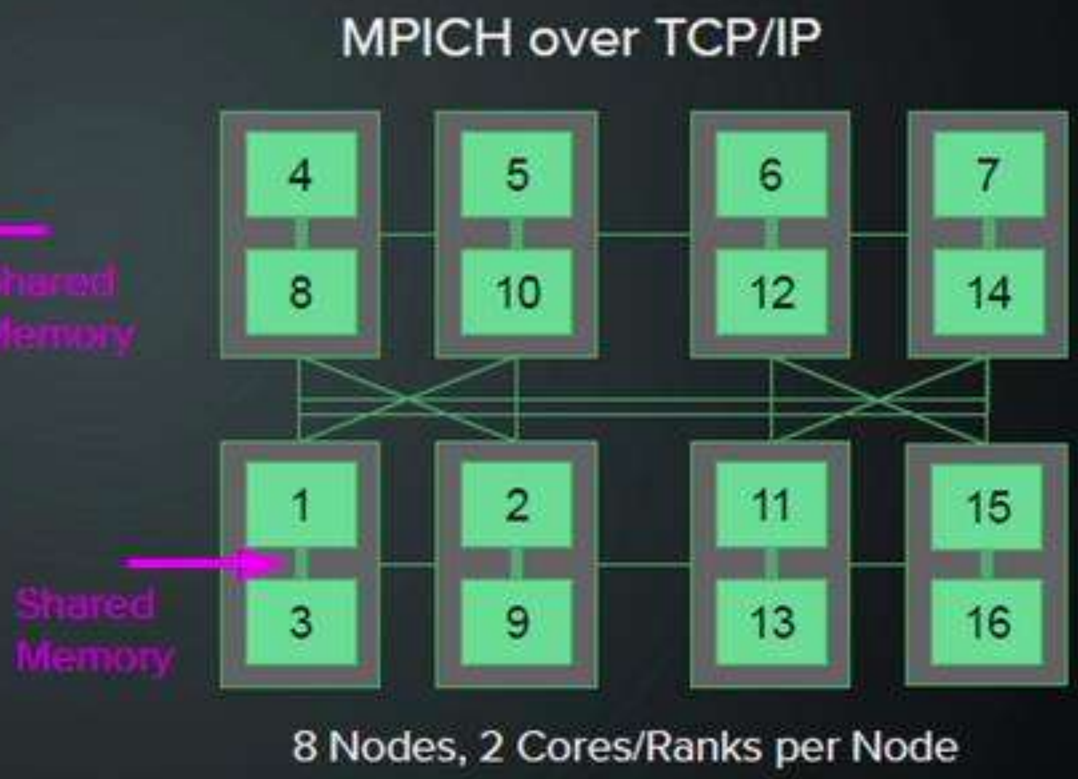
- Can dynamically change configuration of underlying MPI at runtime. *(Why not? The MPI libraries run in a separate program, unrelated to the MPI application program.)*
- Can even change the choice of the underlying MPI and network (e.g., InfiniBand vs. Cray GNI) at runtime! *(Why not? The MPI and network libraries run in a separate program, unrelated to the MPI application program.)*
- Can even migrate to a new cluster at runtime, in which the number of CPU cores per nodes is different! *(Why not? The binding of the MPI libraries to the CPU cores is part of a separate program, unrelated to the MPI application program.)*



Puzzle

Can you solve checkpointing on... And restart on...



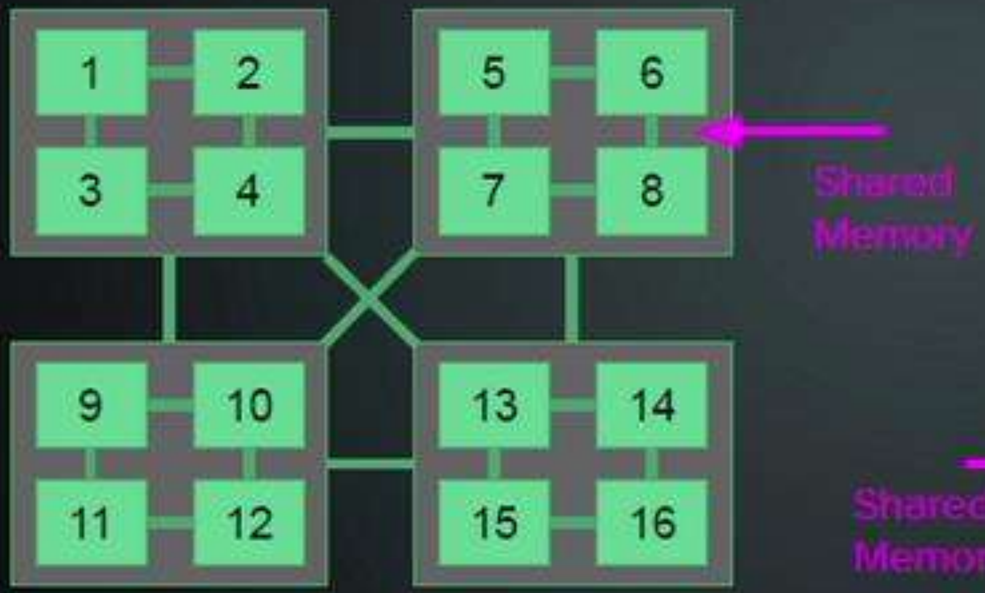


Cross-Cluster Migration

It is now possible to checkpoint on

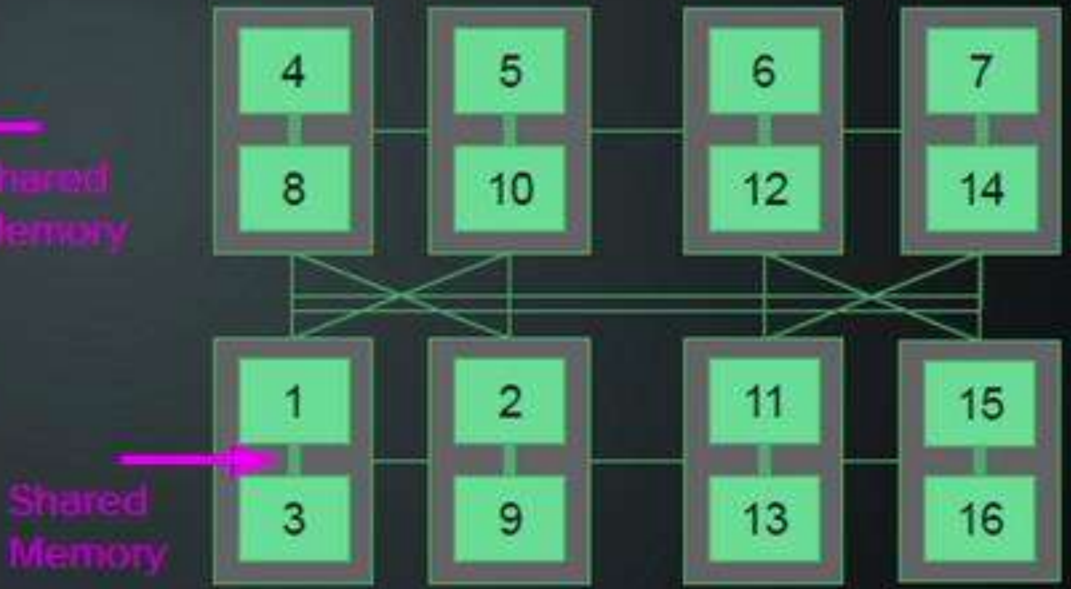
And restart on...

Cray MPI over Infiniband

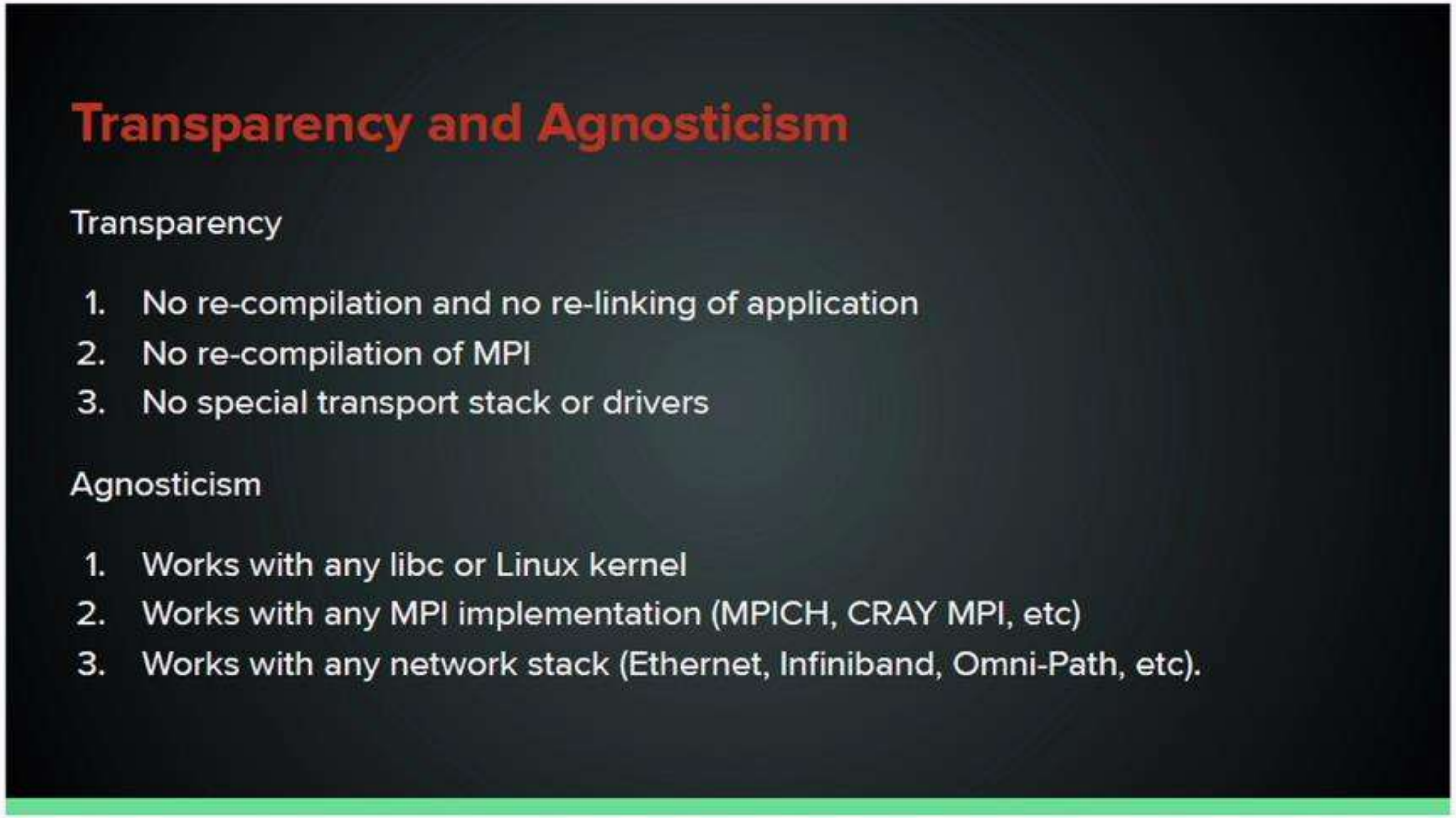


4 Nodes, 4 Cores/Ranks per Node

MPICH over TCP/IP



8 Nodes, 2 Cores/Ranks per Node



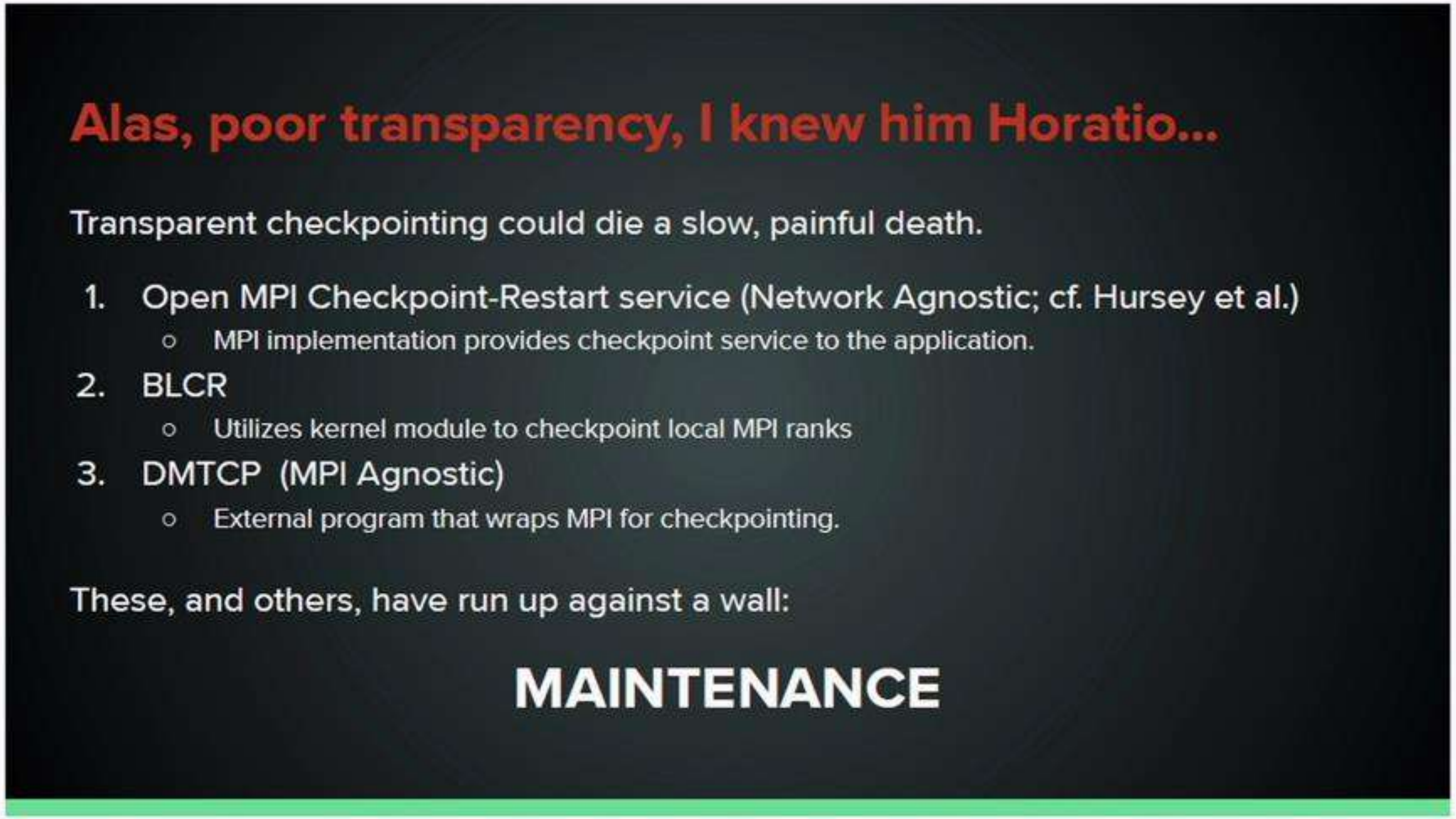
Transparency and Agnosticism

Transparency

1. No re-compilation and no re-linking of application
2. No re-compilation of MPI
3. No special transport stack or drivers

Agnosticism

1. Works with any libc or Linux kernel
2. Works with any MPI implementation (MPICH, CRAY MPI, etc)
3. Works with any network stack (Ethernet, Infiniband, Omni-Path, etc).



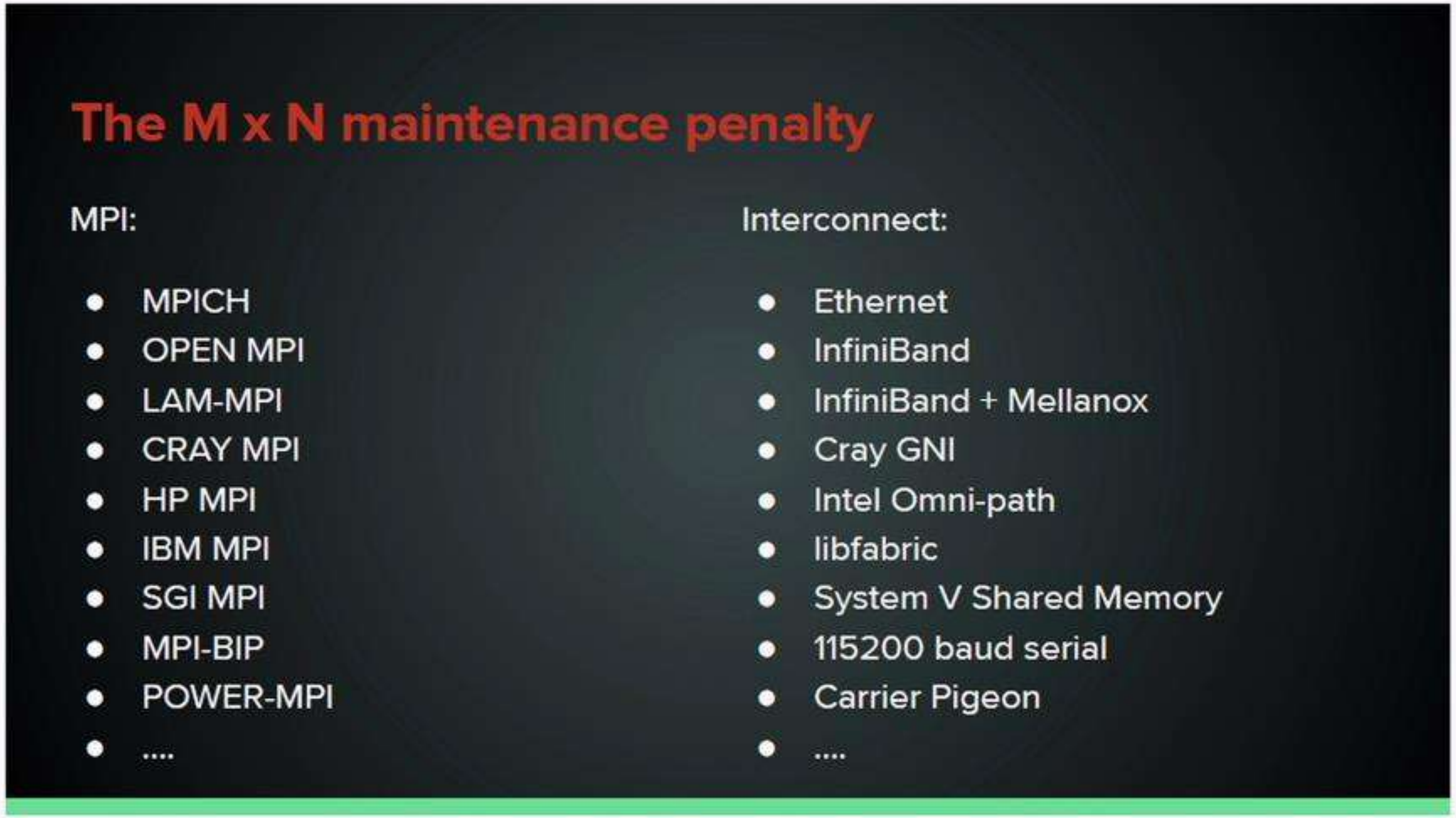
Alas, poor transparency, I knew him Horatio...

Transparent checkpointing could die a slow, painful death.

1. Open MPI Checkpoint-Restart service (Network Agnostic; cf. Hursey et al.)
   - MPI implementation provides checkpoint service to the application.
2. BLCR
   - Utilizes kernel module to checkpoint local MPI ranks
3. DMTCP (MPI Agnostic)
   - External program that wraps MPI for checkpointing.

These, and others, have run up against a wall:

MAINTENANCE



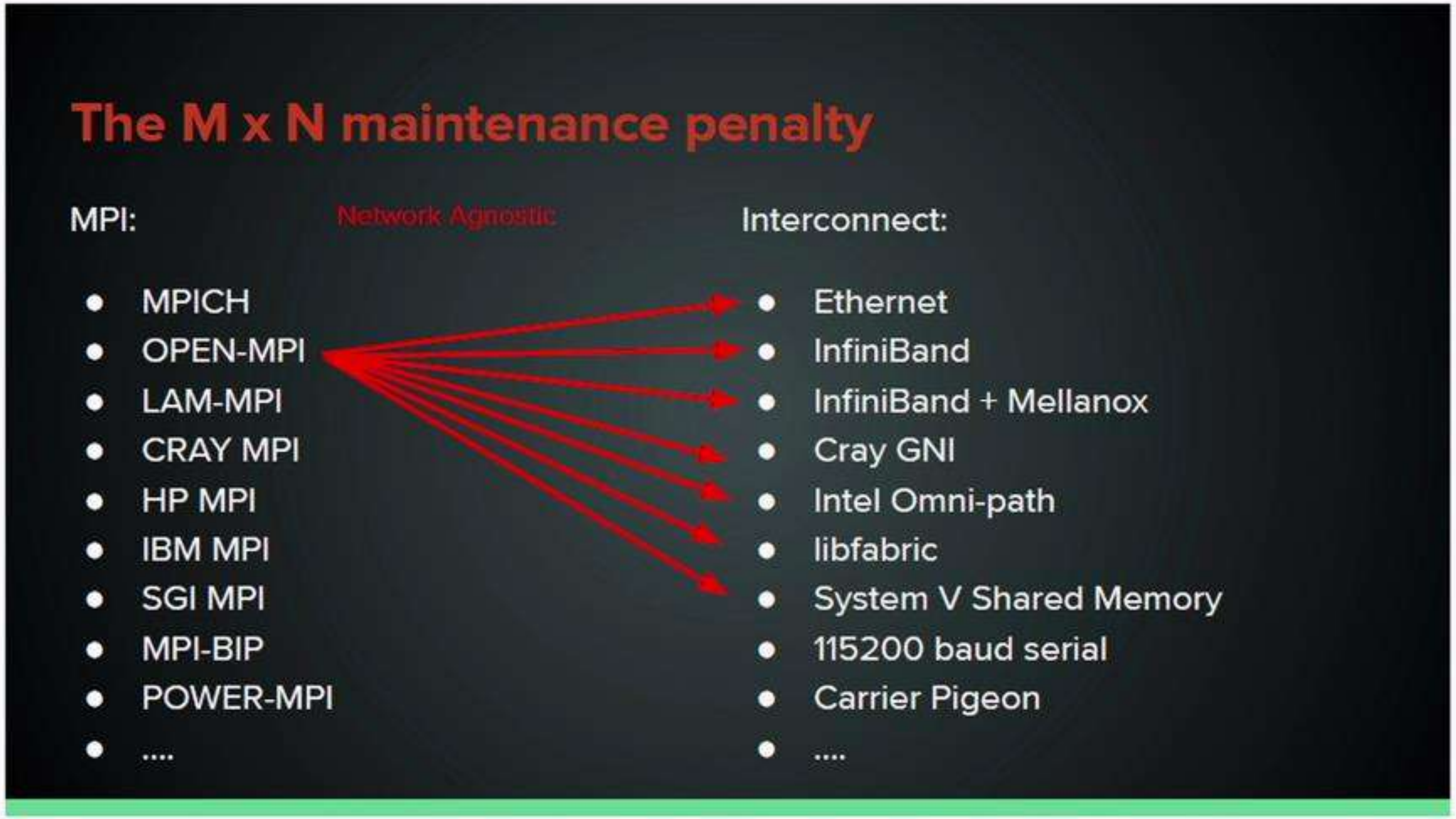
The M x N maintenance penalty

MPI:

- MPICH
- OPEN MPI
- LAM-MPI
- CRAY MPI
- HP MPI
- IBM MPI
- SGI MPI
- MPI-BIP
- POWER-MPI
- ....

Interconnect:

- Ethernet
- InfiniBand
- InfiniBand + Mellanox
- Cray GNI
- Intel Omni-path
- libfabric
- System V Shared Memory
- 115200 baud serial
- Carrier Pigeon
- ....



The M x N maintenance penalty

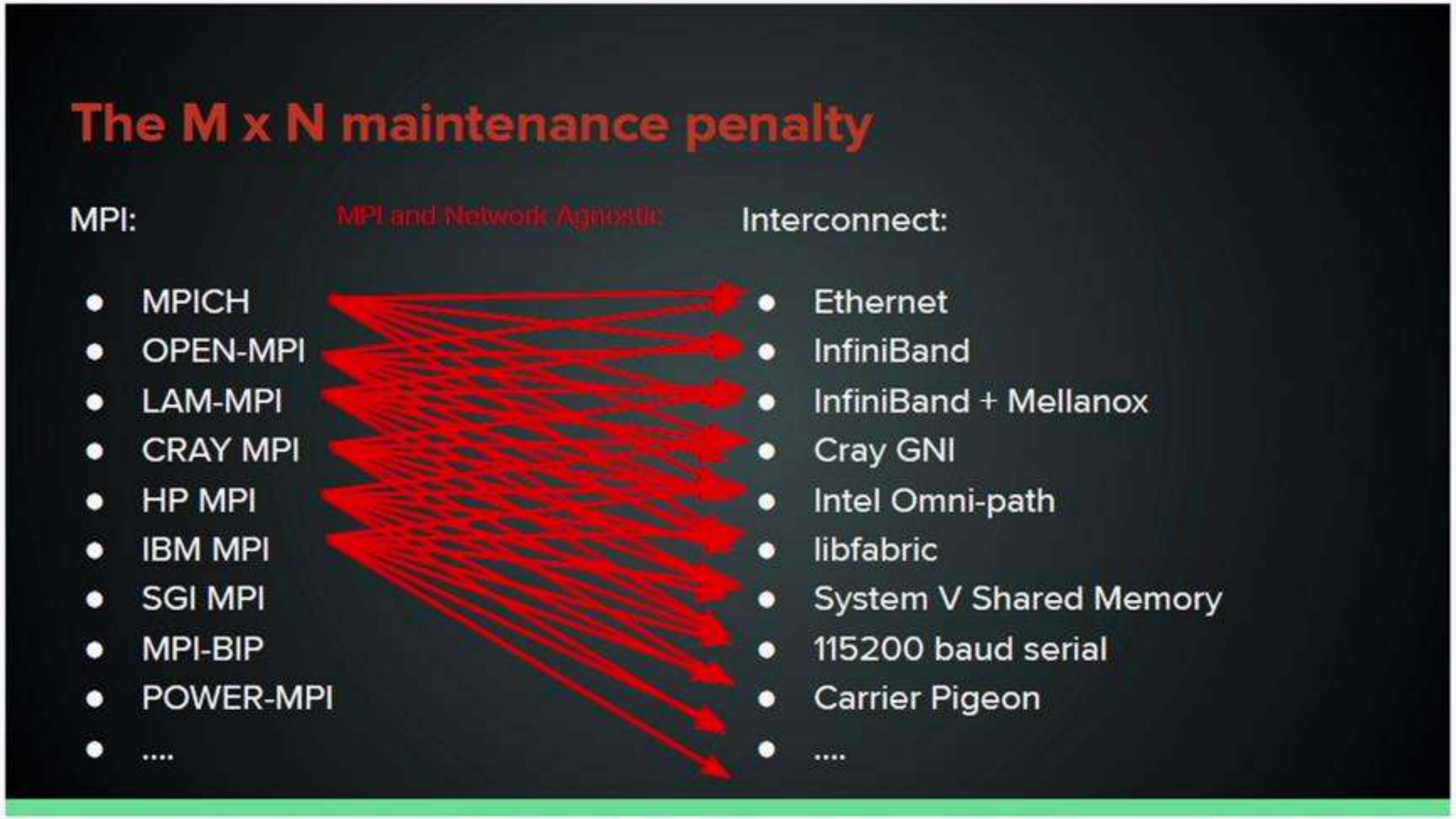
Network Agnostic

MPI:

- MPICH
- OPEN-MPI
- LAM-MPI
- CRAY MPI
- HP MPI
- IBM MPI
- SGI MPI
- MPI-BIP
- POWER-MPI
- ....

Interconnect:

- Ethernet
- InfiniBand
- InfiniBand + Mellanox
- Cray GNI
- Intel Omni-path
- libfabric
- System V Shared Memory
- 115200 baud serial
- Carrier Pigeon
- ....



The M x N maintenance penalty

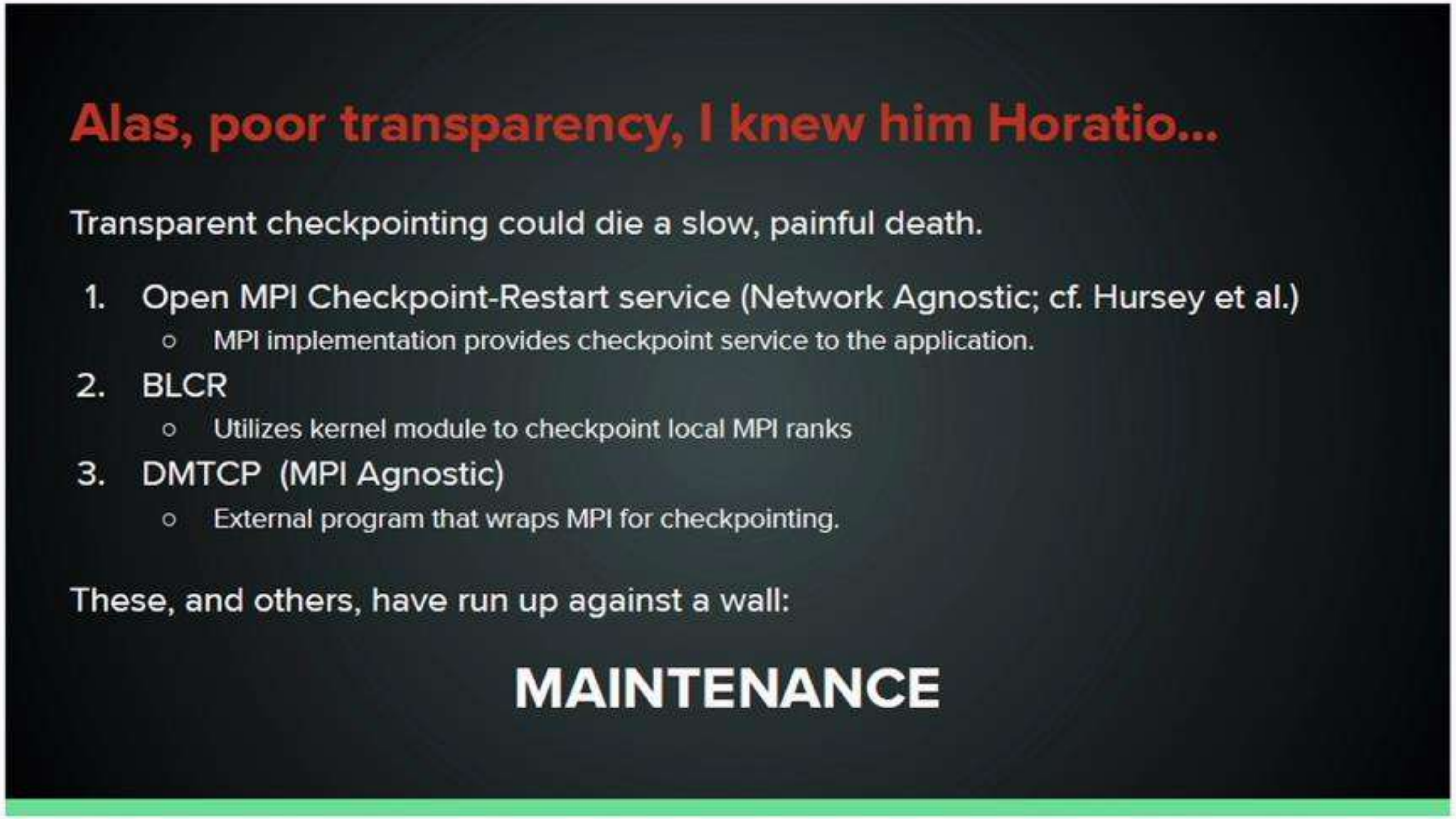
MPI and Network Agnostic:

MPI:

- MPICH
- OPEN-MPI
- LAM-MPI
- CRAY MPI
- HP MPI
- IBM MPI
- SGI MPI
- MPI-BIP
- POWER-MPI
- ....

Interconnect:

- Ethernet
- InfiniBand
- InfiniBand + Mellanox
- Cray GNI
- Intel Omni-path
- libfabric
- System V Shared Memory
- 115200 baud serial
- Carrier Pigeon
- ....



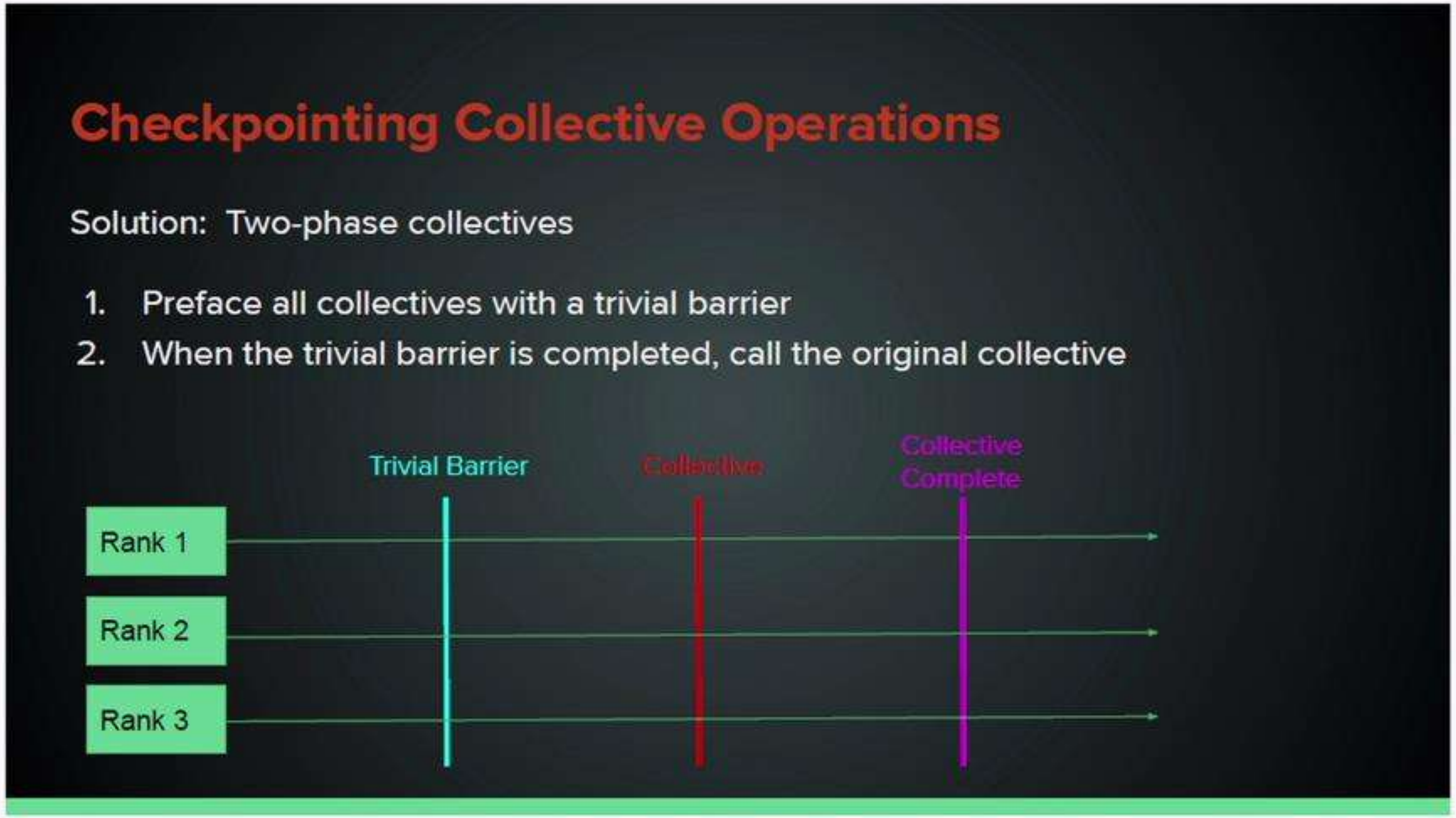
Alas, poor transparency, I knew him Horatio...

Transparent checkpointing could die a slow, painful death.

1. Open MPI Checkpoint-Restart service (Network Agnostic; cf. Hursey et al.)
    - MPI implementation provides checkpoint service to the application.
2. BLCR
    - Utilizes kernel module to checkpoint local MPI ranks
3. DMTCP (MPI Agnostic)
    - External program that wraps MPI for checkpointing.

These, and others, have run up against a wall:

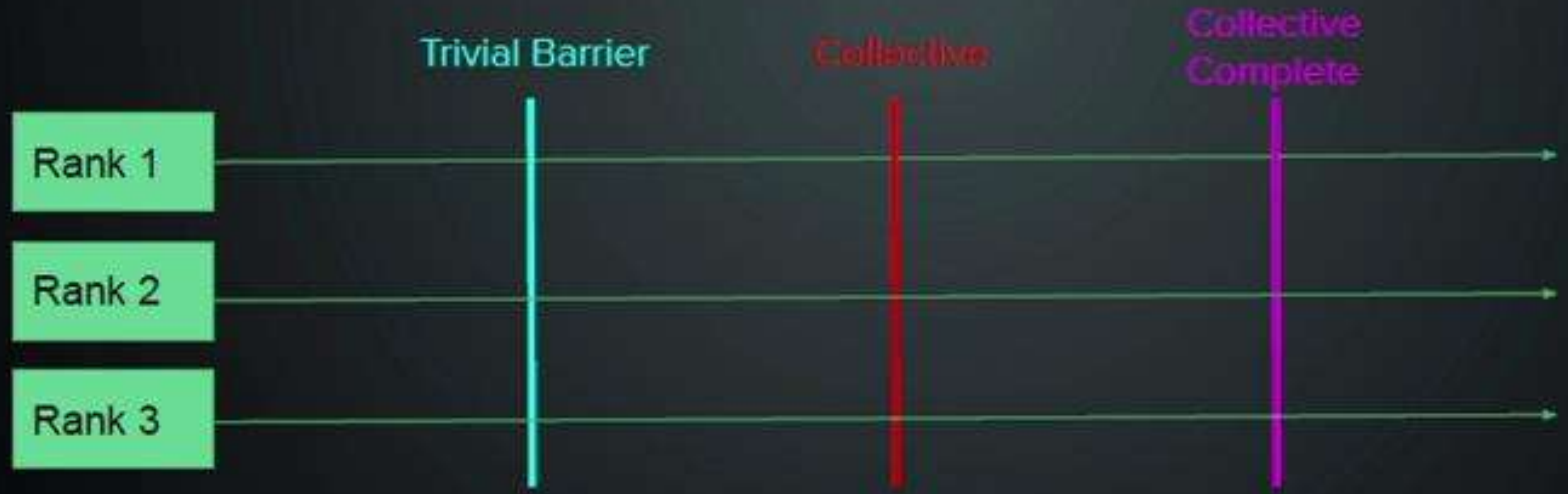
MAINTENANCE

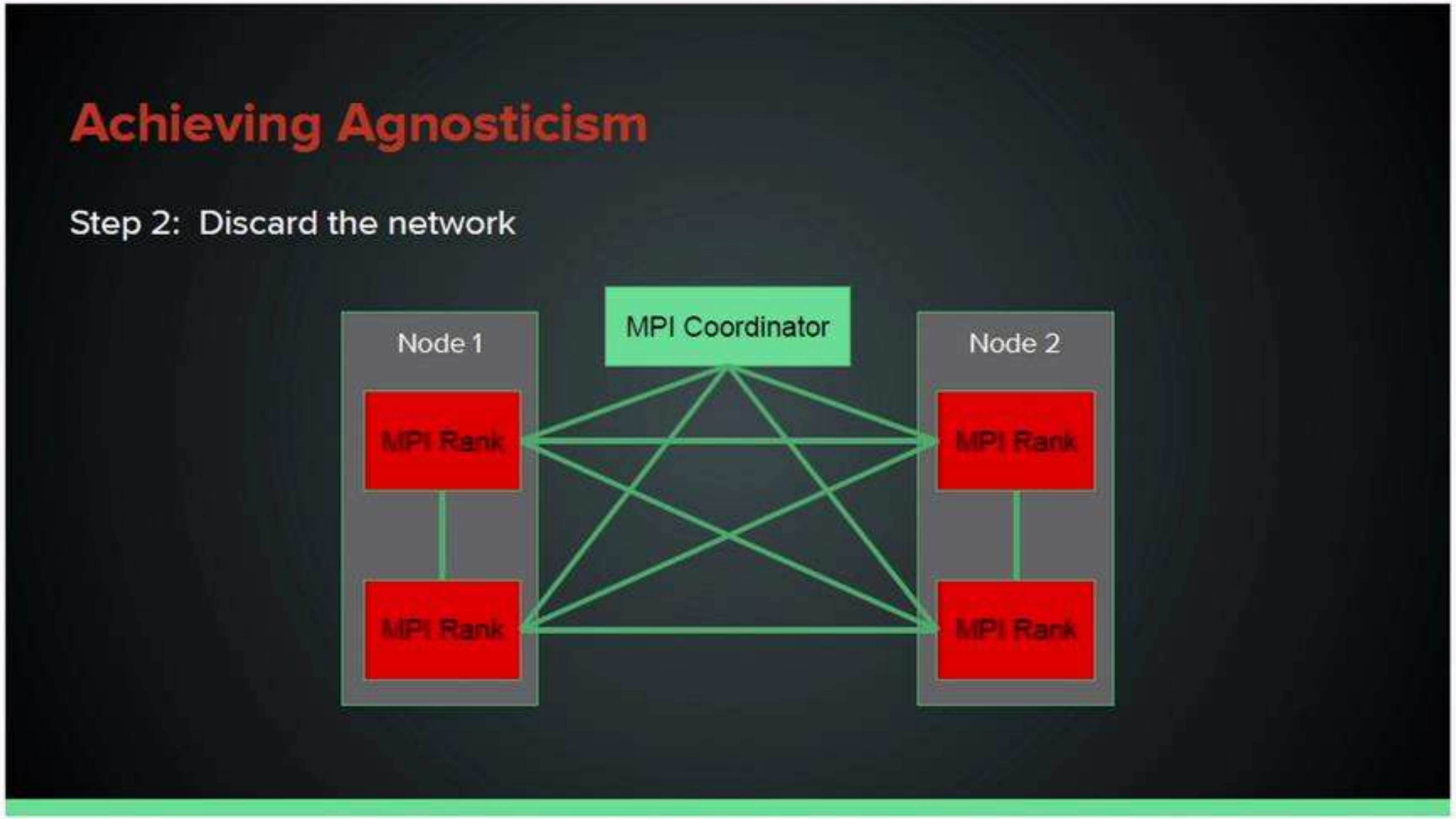


Checkpointing Collective Operations

Solution: Two-phase collectives

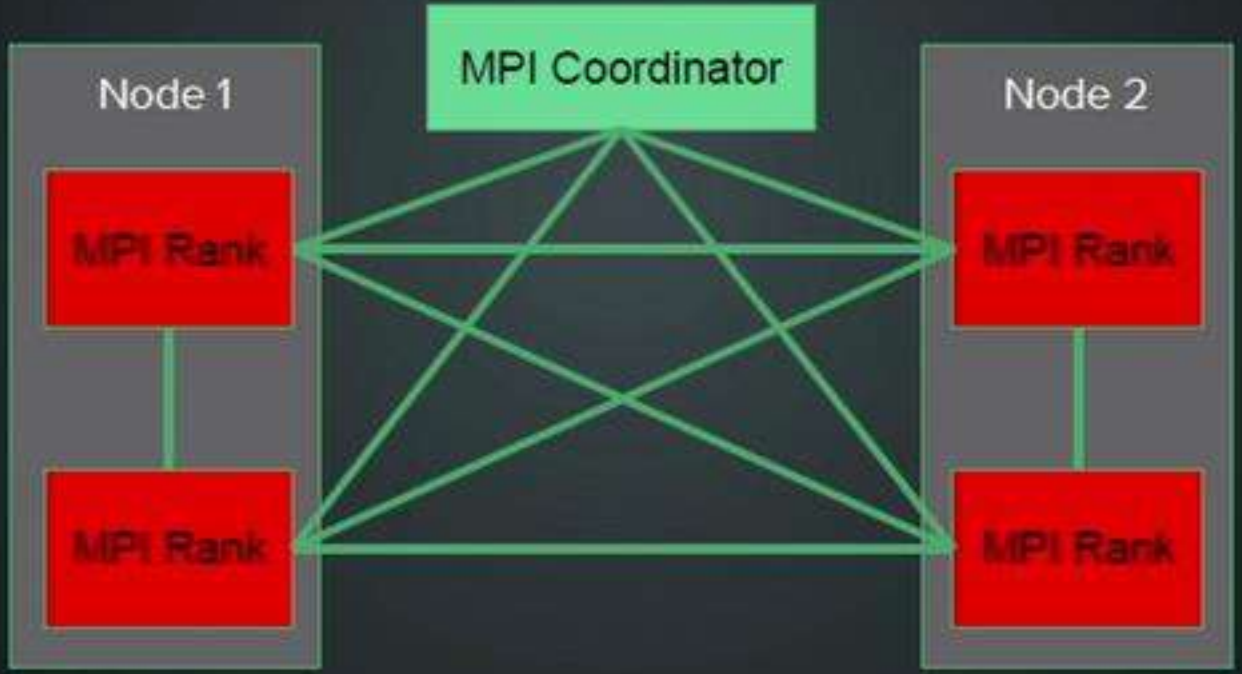
1. Preface all collectives with a trivial barrier
2. When the trivial barrier is completed, call the original collective

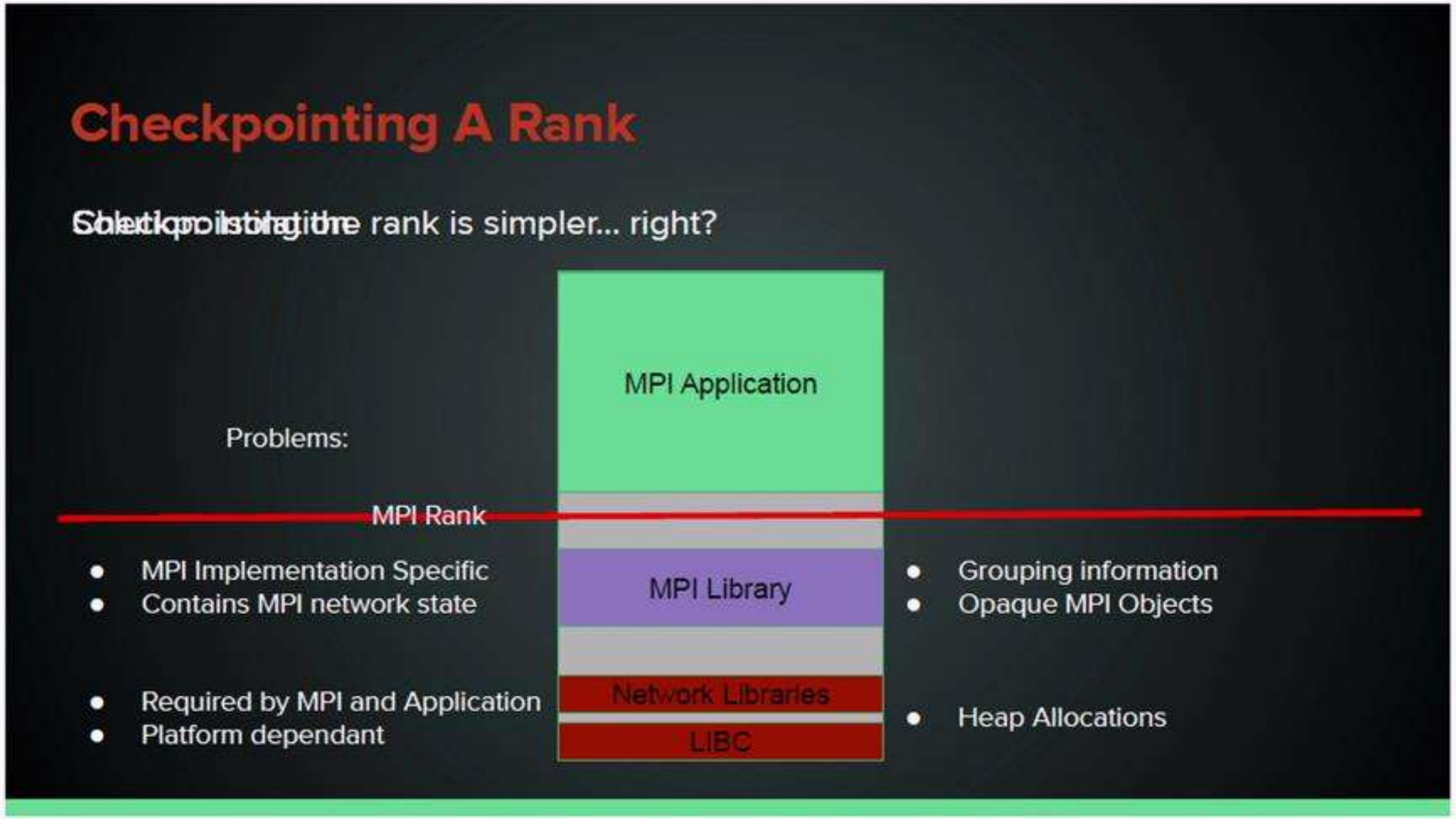




Achieving Agnosticism

Step 2: Discard the network





Checkpointing A Rank

Checkpointing the rank is simpler... right?

Problems:

MPI Rank

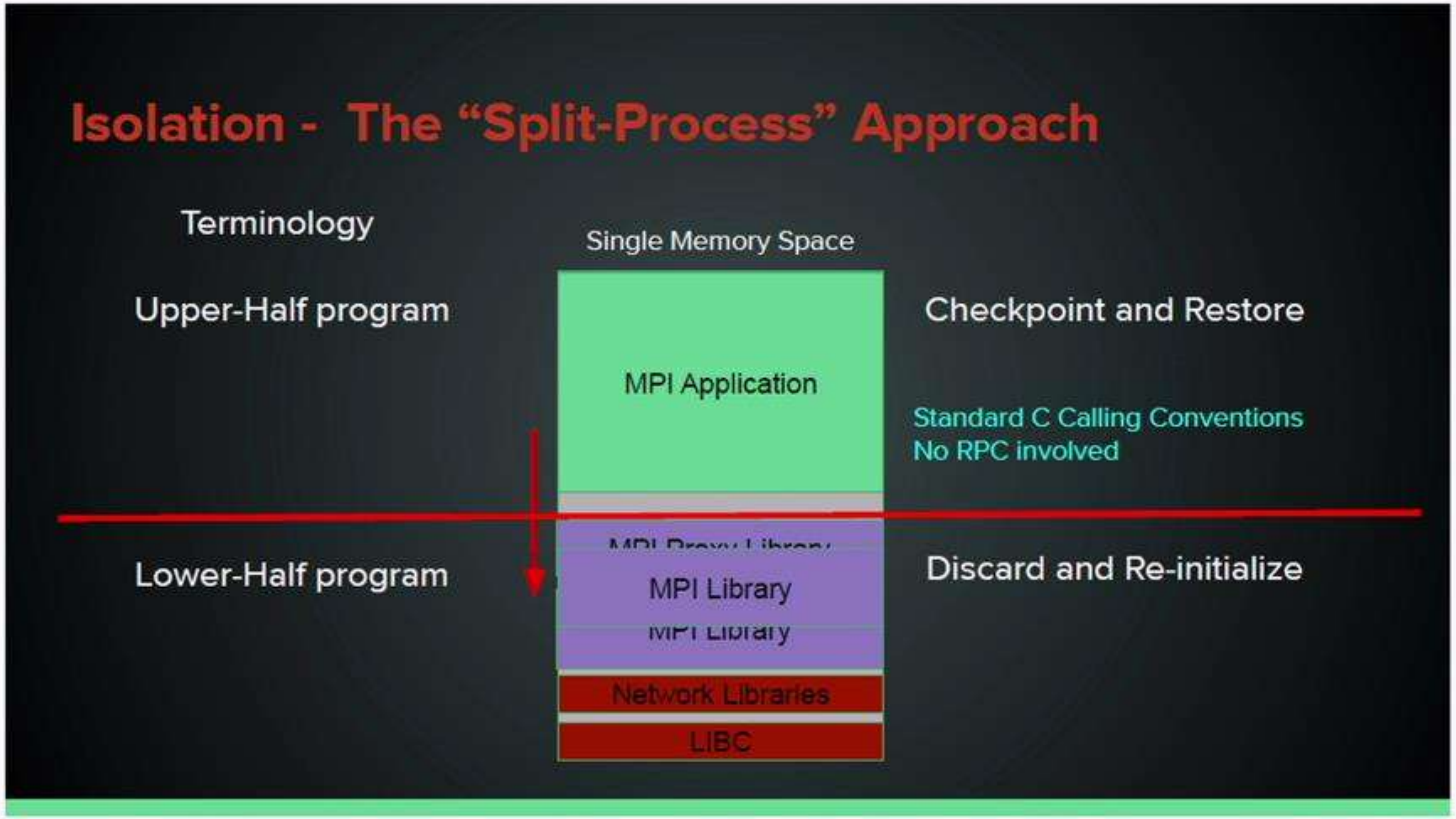
- MPI Implementation Specific
- Contains MPI network state

- Required by MPI and Application
- Platform dependant

- Grouping information
- Opaque MPI Objects

- Heap Allocations

MPI Application
MPI Library
Network Libraries
LIBC



Isolation - The “Split-Process” Approach

Terminology

Upper-Half program

Lower-Half program

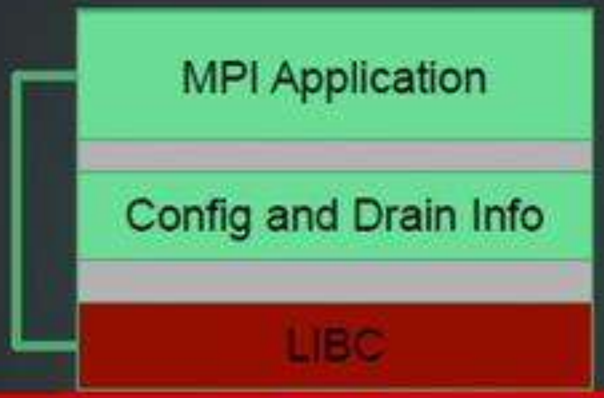
Checkpoint and Restore

Standard C Calling Conventions
No RPC involved

Discard and Re-initialize

MPI Agnosticism Achieved

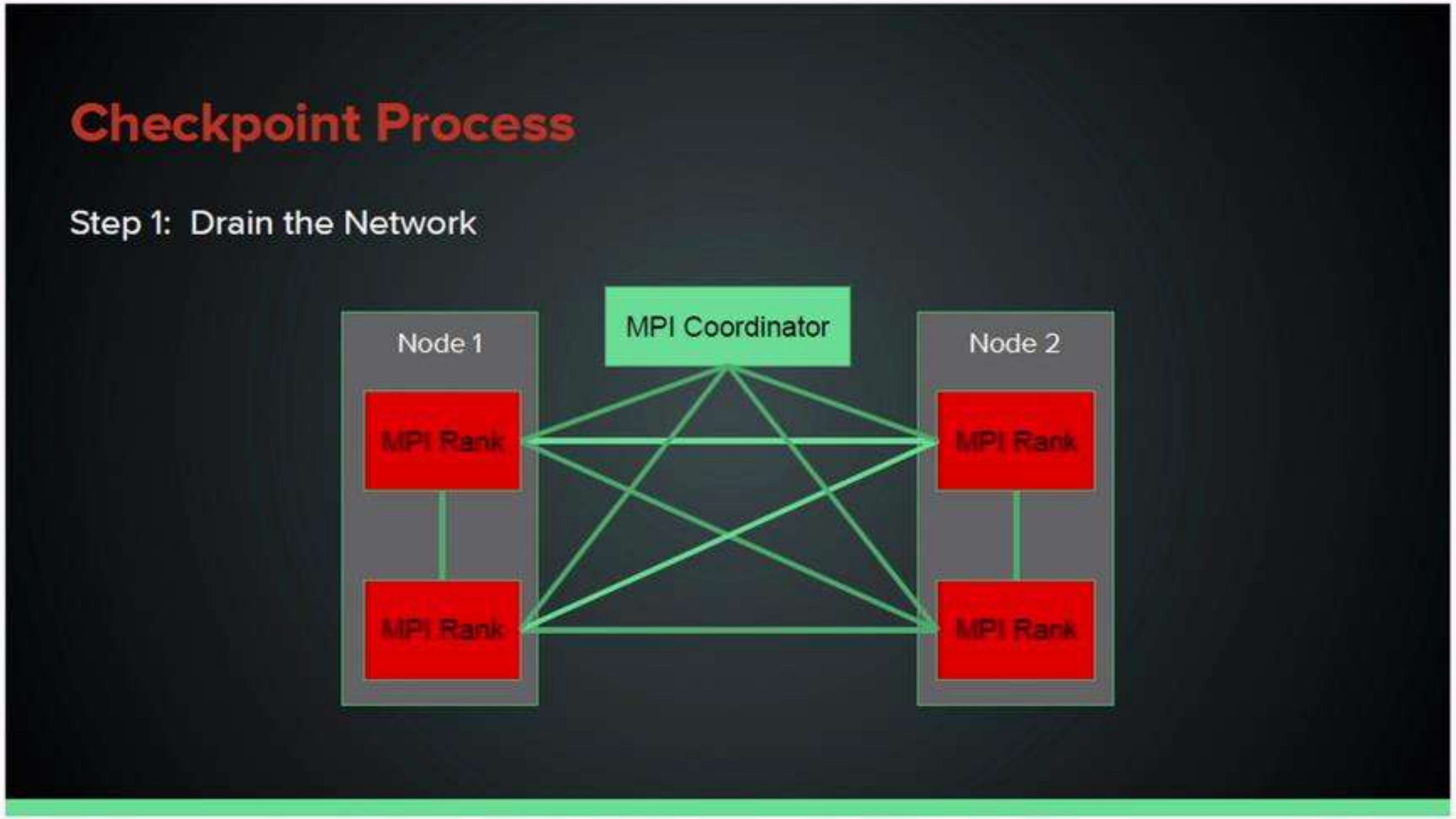
Upper Half:
Persistent Data



*Special care must be taken when replacing upper half libraries

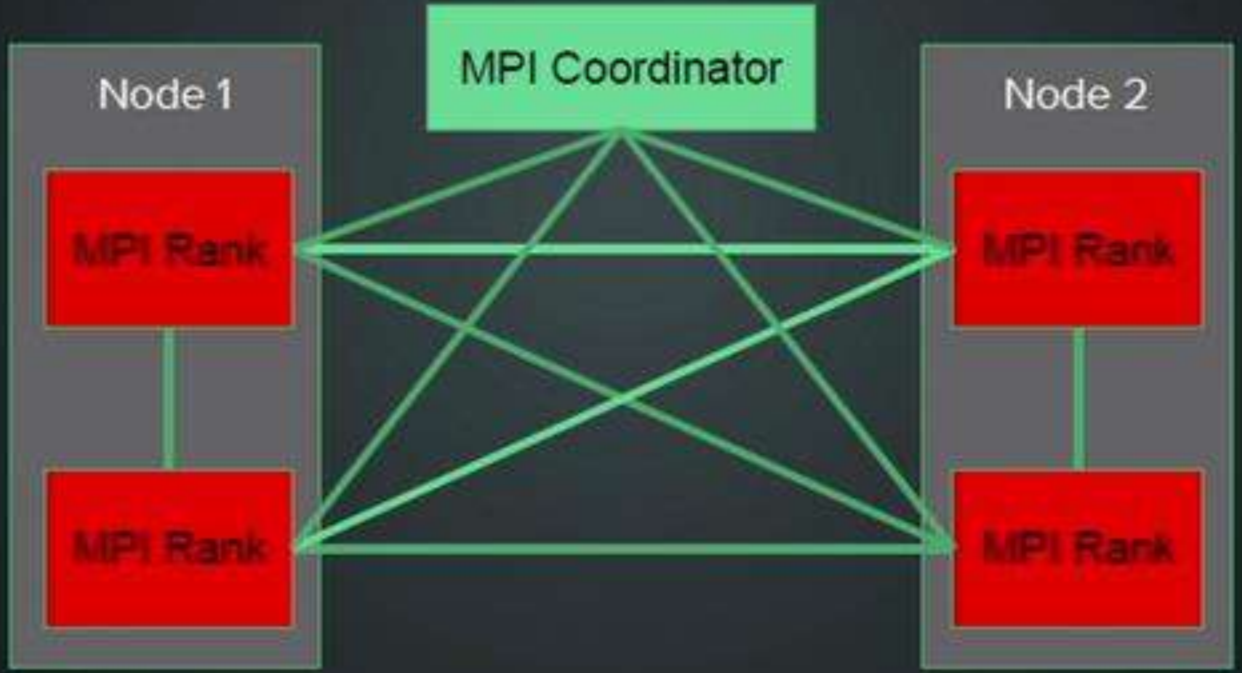
Lower Half
Ephemeral Data

Lower half data can be replaced by new and different implementations of MPI and related libraries.



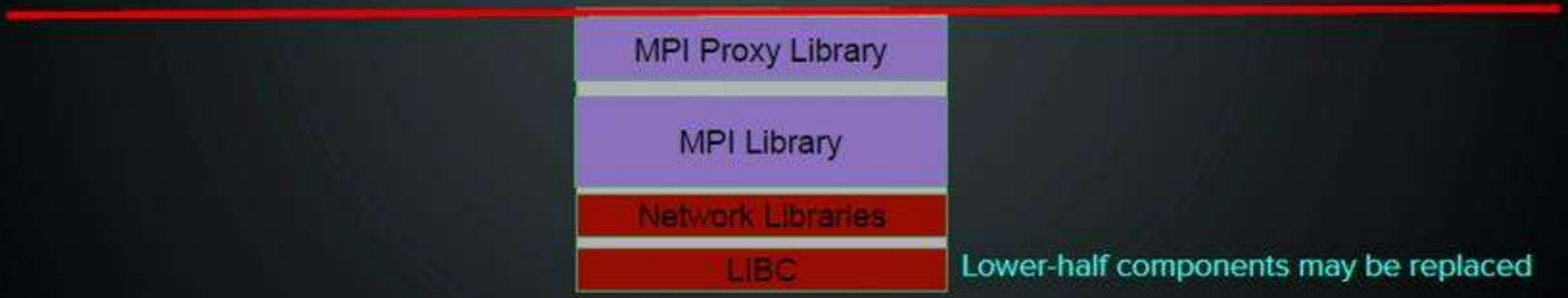
Checkpoint Process

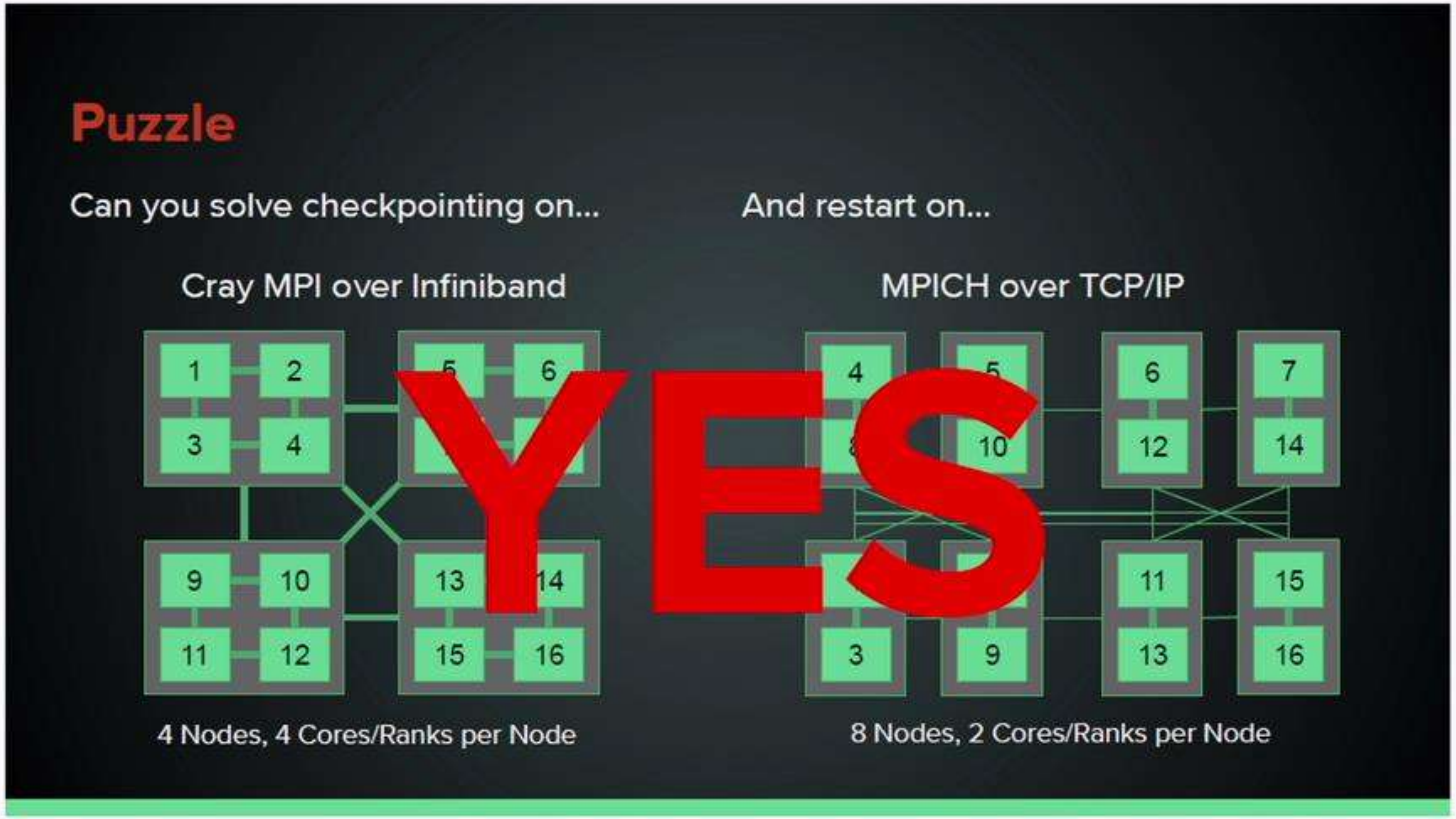
Step 1: Drain the Network



Restart Process

Step 1: Restore Lower-Half





Puzzle

Can you solve checkpointing on...

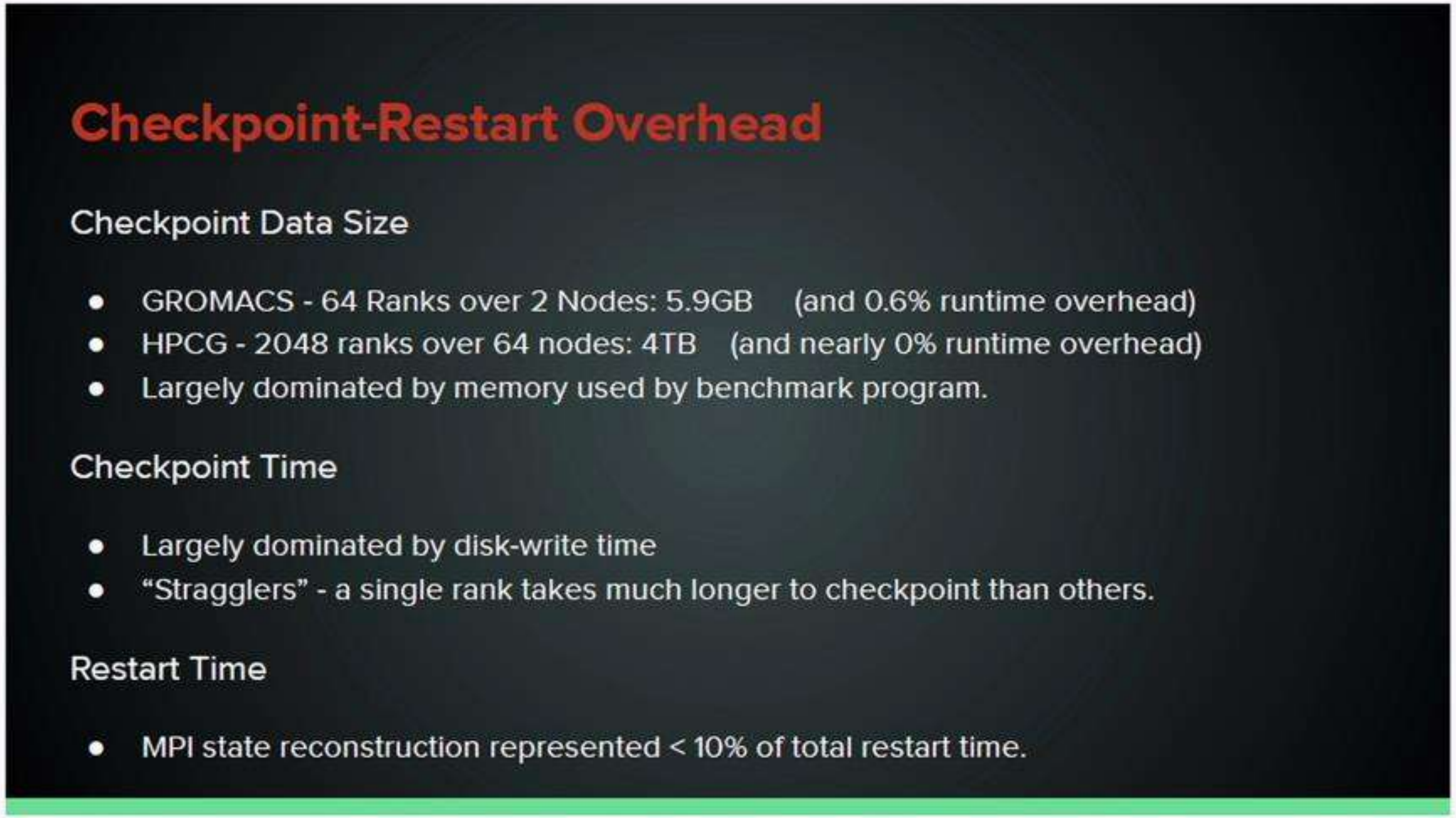
Cray MPI over Infiniband

4 Nodes, 4 Cores/Ranks per Node

And restart on...

MPICH over TCP/IP

8 Nodes, 2 Cores/Ranks per Node



Checkpoint-Restart Overhead

Checkpoint Data Size

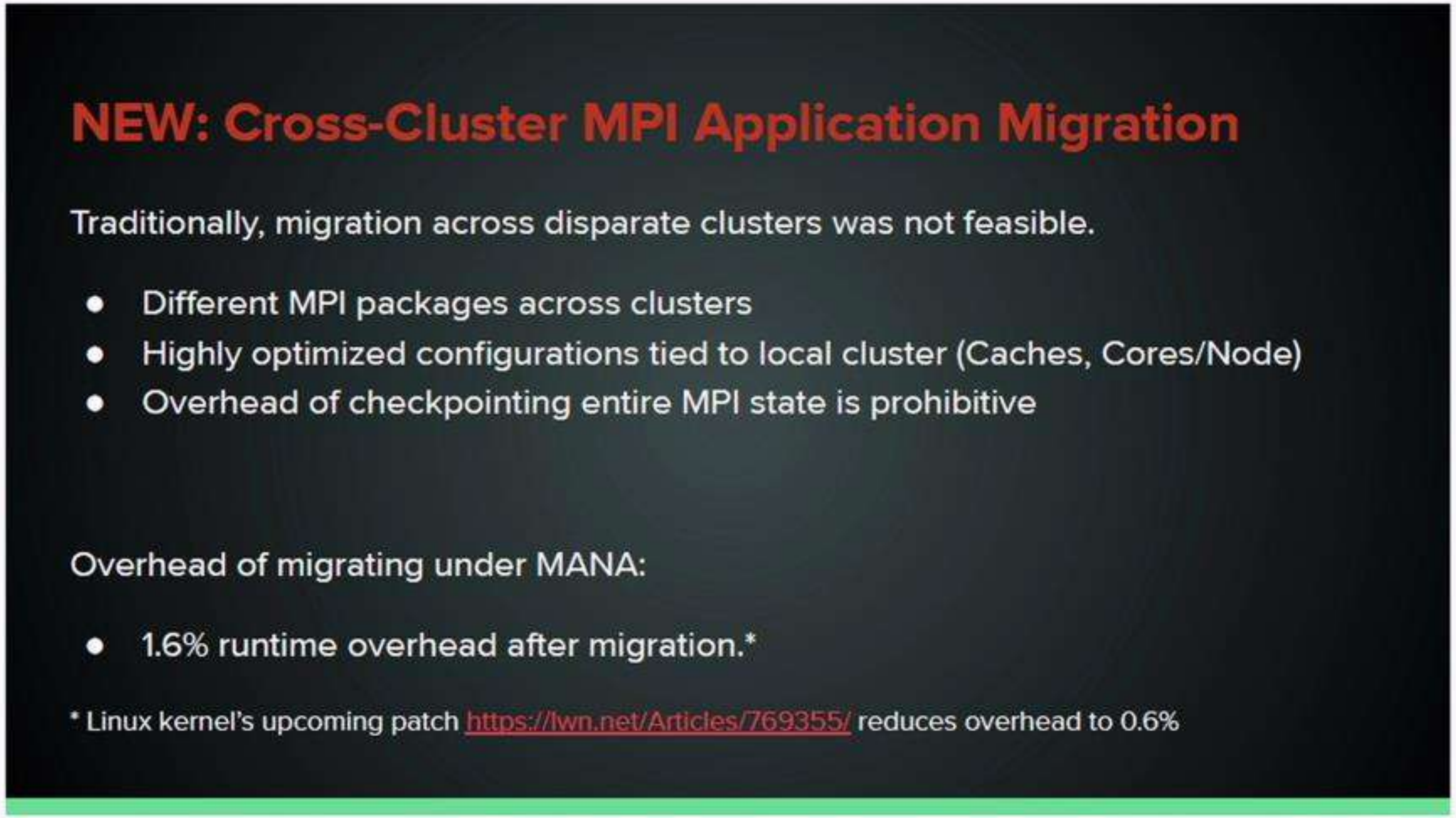
- GROMACS - 64 Ranks over 2 Nodes: 5.9GB (and 0.6% runtime overhead)
- HPCG - 2048 ranks over 64 nodes: 4TB (and nearly 0% runtime overhead)
- Largely dominated by memory used by benchmark program.

Checkpoint Time

- Largely dominated by disk-write time
- “Stragglers” - a single rank takes much longer to checkpoint than others.

Restart Time

- MPI state reconstruction represented < 10% of total restart time.



NEW: Cross-Cluster MPI Application Migration

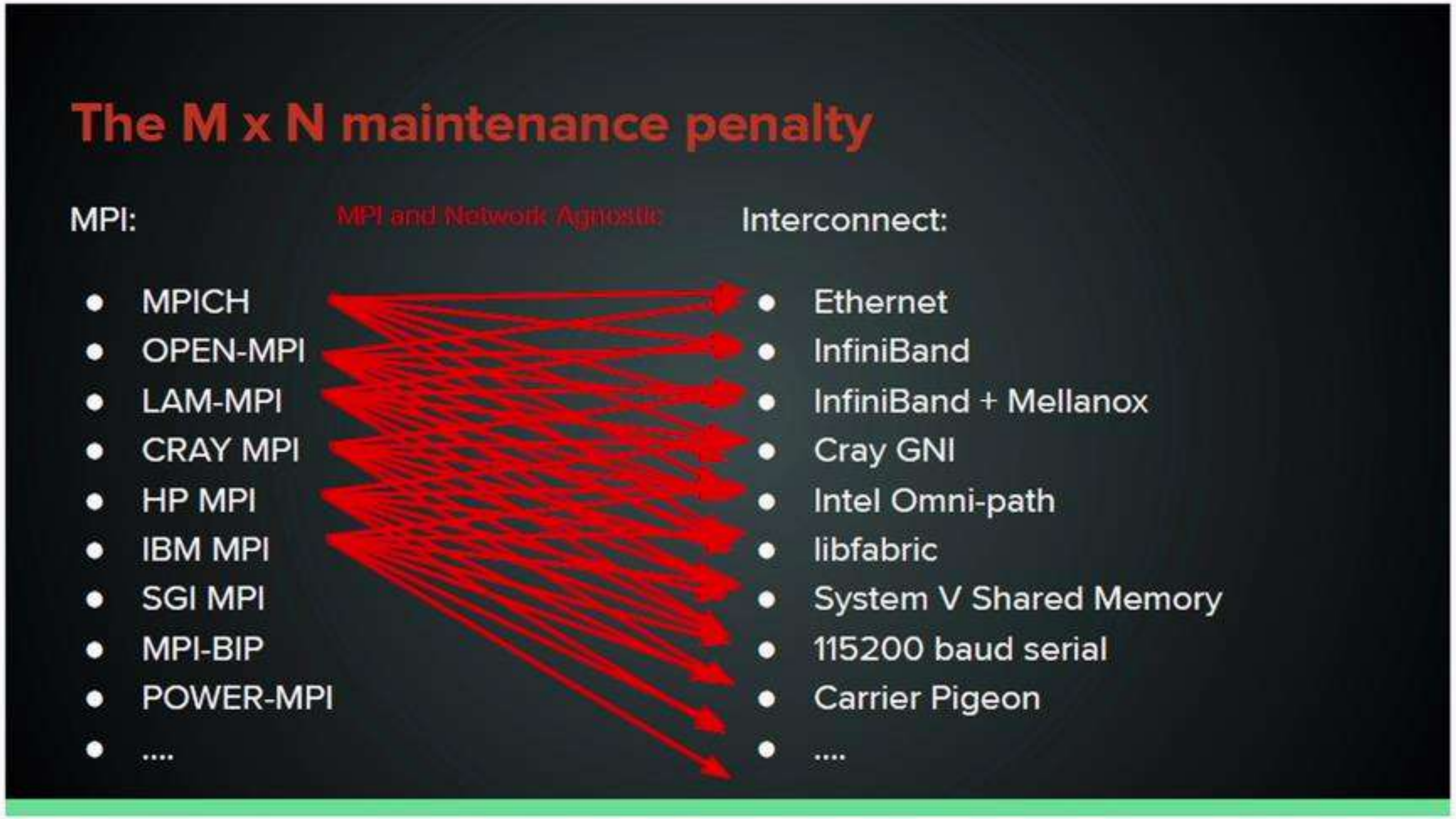
Traditionally, migration across disparate clusters was not feasible.

- Different MPI packages across clusters
- Highly optimized configurations tied to local cluster (Caches, Cores/Node)
- Overhead of checkpointing entire MPI state is prohibitive

Overhead of migrating under MANA:

- 1.6% runtime overhead after migration.*

* Linux kernel's upcoming patch https://lwn.net/Articles/769355/ reduces overhead to 0.6%



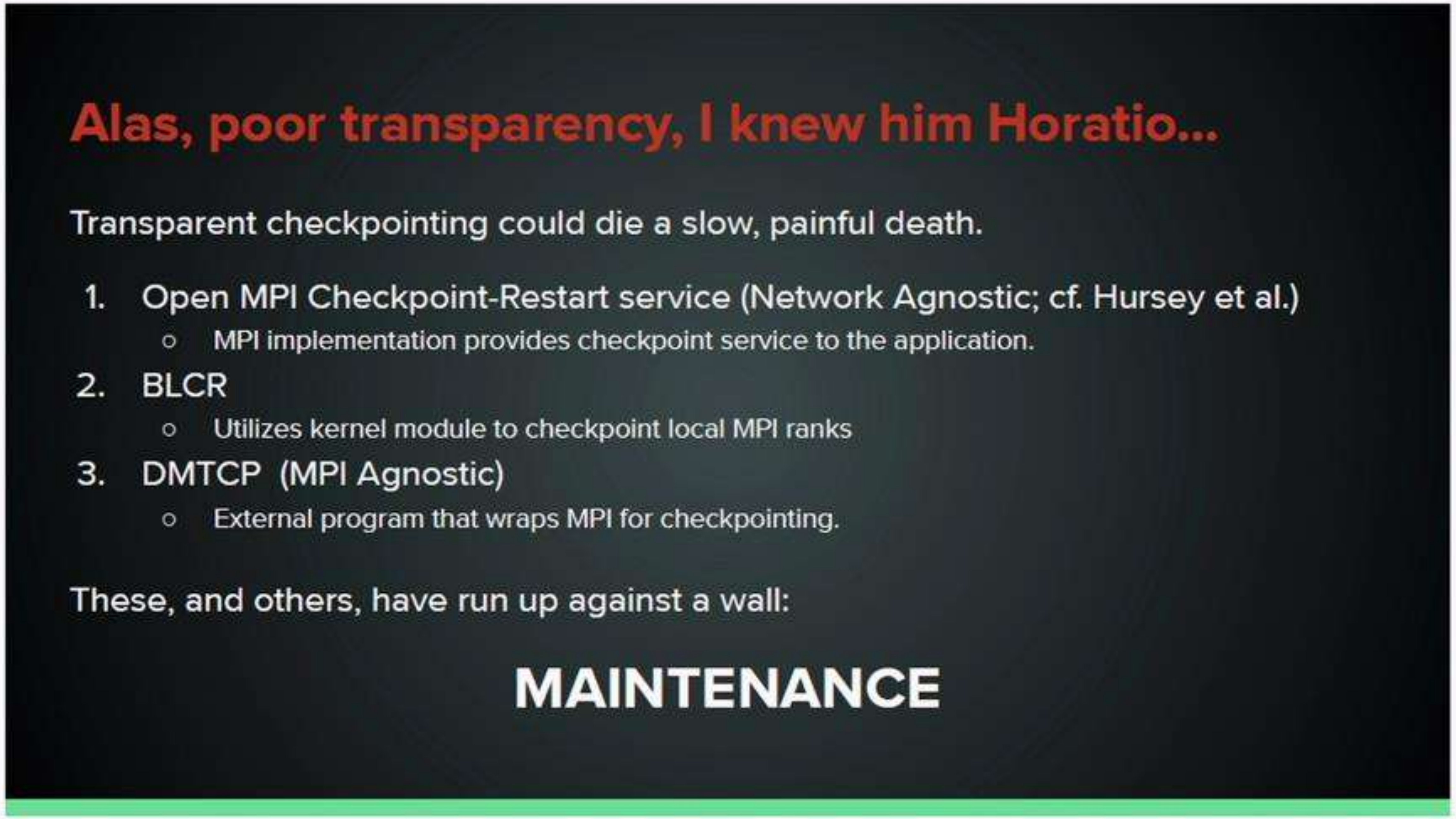
The M x N maintenance penalty

MPI:

MPI and Network Agnostic

Interconnect:

MPI:	Interconnect:
MPICH	Ethernet
OPEN-MPI	InfiniBand
LAM-MPI	InfiniBand + Mellanox
CRAY MPI	Cray GNI
HP MPI	Intel Omni-path
IBM MPI	libfabric
SGI MPI	System V Shared Memory
MPI-BIP	115200 baud serial
POWER-MPI	Carrier Pigeon
....	....



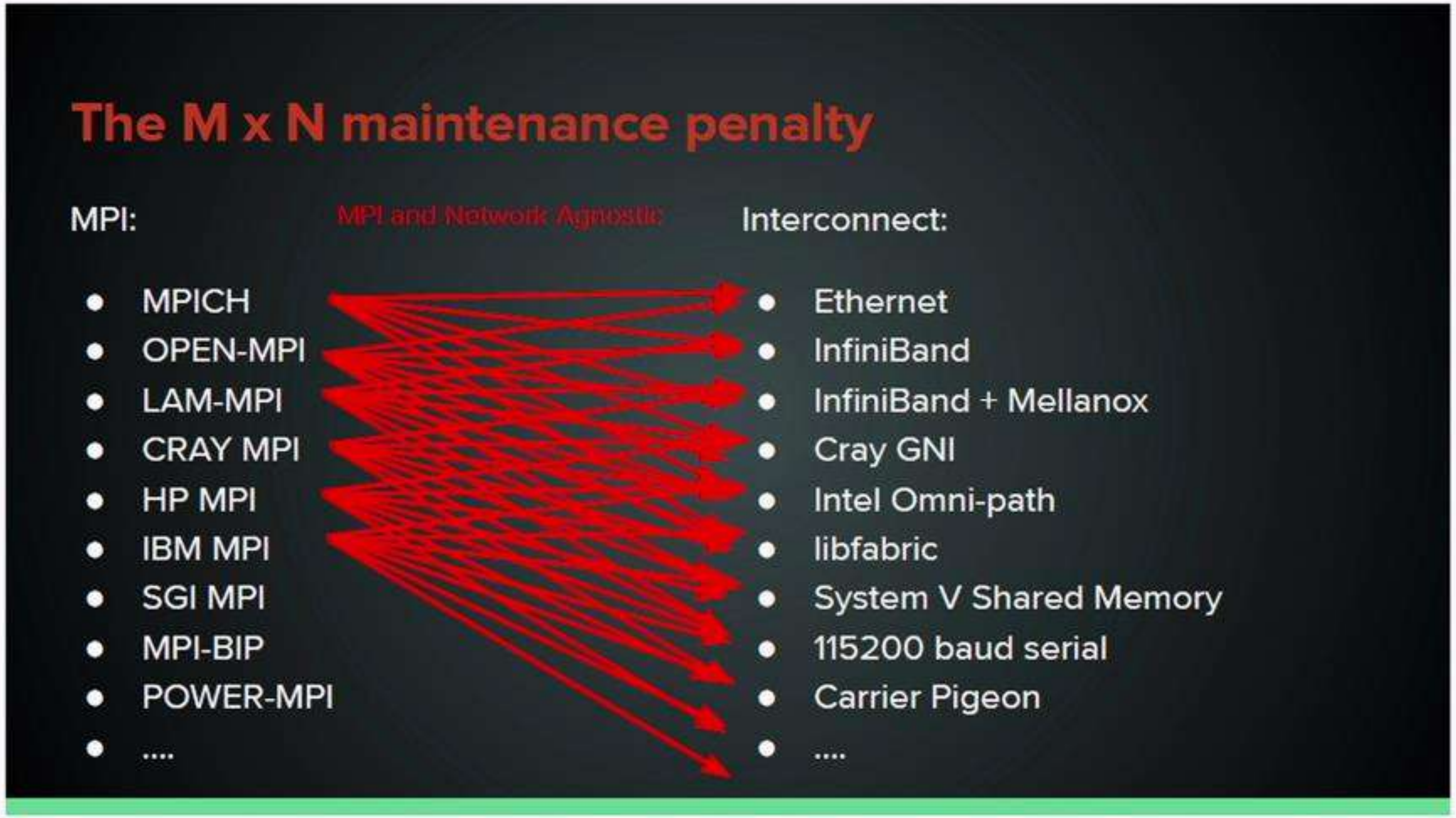
Alas, poor transparency, I knew him Horatio...

Transparent checkpointing could die a slow, painful death.

1. Open MPI Checkpoint-Restart service (Network Agnostic; cf. Hursey et al.)
    - MPI implementation provides checkpoint service to the application.
2. BLCR
    - Utilizes kernel module to checkpoint local MPI ranks
3. DMTCP (MPI Agnostic)
    - External program that wraps MPI for checkpointing.

These, and others, have run up against a wall:

MAINTENANCE



The M x N maintenance penalty

MPI: MPI and Network Agnostic Interconnect:

MPI:	Interconnect:
MPICH	Ethernet
OPEN-MPI	InfiniBand
LAM-MPI	InfiniBand + Mellanox
CRAY MPI	Cray GNI
HP MPI	Intel Omni-path
IBM MPI	libfabric
SGI MPI	System V Shared Memory
MPI-BIP	115200 baud serial
POWER-MPI	Carrier Pigeon
....	....

Why is Transparent Checkpoint Difficult in Windows?

"To the best of our knowledge, there is still no package for transparent checkpointing in Windows. WHY?

1. There is a long history of transparent checkpointing in Linux: DMTCP, CRIU, BLCR, and many others in history.
2. The core algorithm is:

Checkpoint:	(a) Record information about current kernel (e.g., open files and file offsets);
	(b) Save all of user-accessible memory to a ckpt file.
Restart:	(a) Start a new process (usually in a new kernel);
	(b) overwrite the user-space memory by the original memory in the ckpt file;
	(c) use system calls to restore information about current kernel (e.g, re-open files, seek to previous file offset).

PROBLEM: Some of Windows kernel state is in the user's process memory, in a hidden format. Restoring it into a new Windows kernel doesn't work.

So, What's Different in Windows?

The Windows Process Control Block PCB) includes a *Process Environment Block (PEB)*, and even *Thread Environment Blocks (TEBs)*, in user space, but in an *internal, undocumented format*.

From Wikipedia:
`https://en.wikipedia.org/wiki/Process_Environment_Block`

> *"The PEB is closely associated with the kernel mode EPROCESS data structure, as well as with per-process data structures managed within the address space of the Client-Server Runtime Sub-System process. However, … the PEB is not a kernel mode data structure itself. It resides in the application mode address space of the process that it relates to. This is because it is designed to be used by … the operating system libraries … that execute outside of kernel mode,"*

Do Split Processes Help? (HINT: “Yes”)

RECALL: The essence of split processes is to:

(a) Combine two programs into a single address space.

(b) Tag each memory block as either upper half or lower half memory.

(c) On checkpoint, save upper-half memory; a new lower-half process can later restore this upper-half memory from a checkpoint file.

SOLUTION for Windows (work in progress):: In part (b) above, tag the user application memory as upper-half, but allow the Process Environment Block (PEB) to be implicitly tagged a lower-half.

The essence is to isolate the application code from the systems libraries. This should seem familiar. Recall the history with: Drawbridge (and Windows Subsystem for Linux (WSL); Library operating systems, Proxy processes (extensively used in many domains; WINE, Unikernels, etc.

The difficult part is a *clean* isolation paradigm. The goal of our project is *transparent*, *user-space* isolation.

Outline

Sthread, using SimGrid

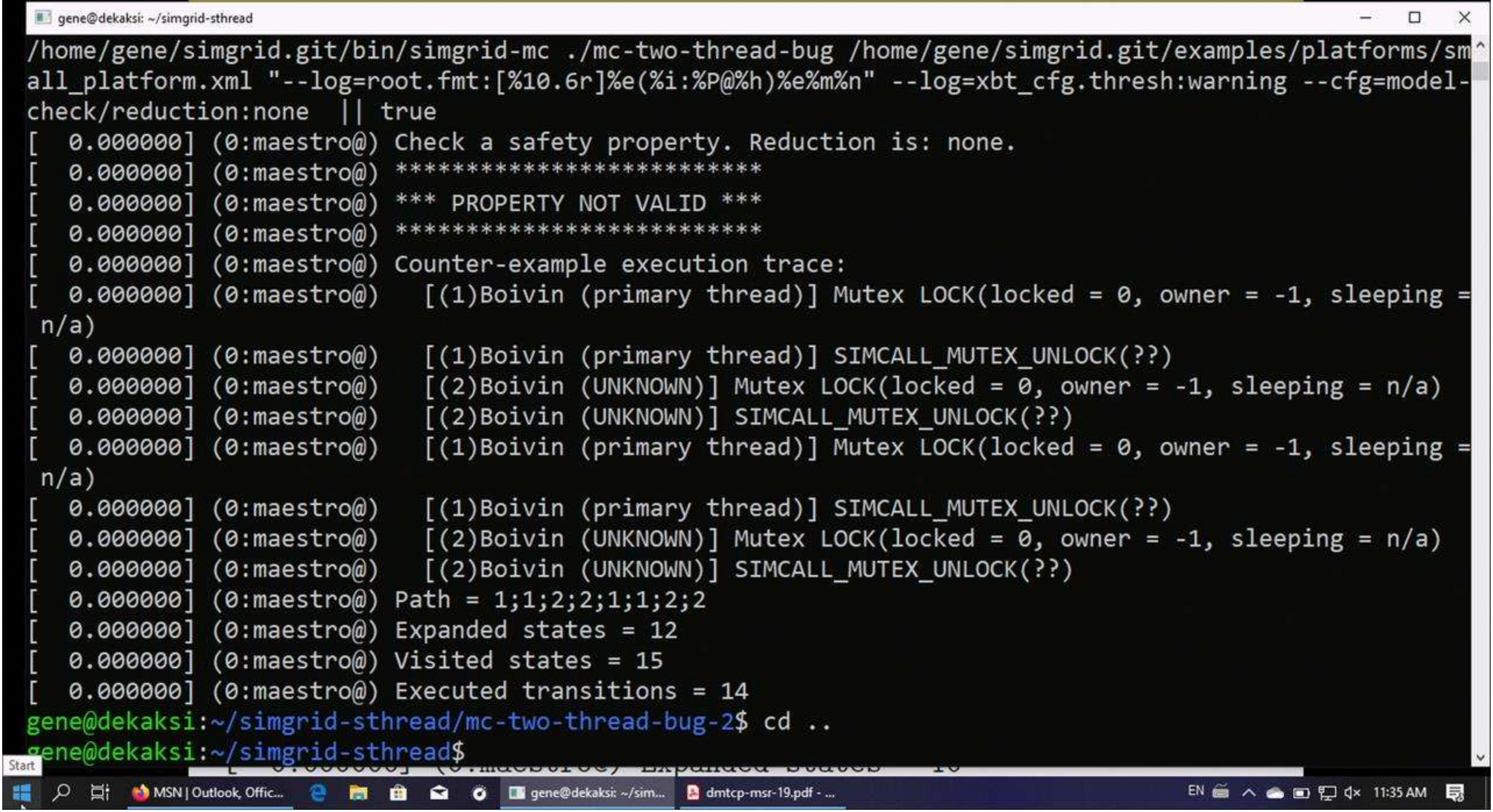
```
\#\#\#\#include <sthread.h>
bool val1=false;
bool val2=false;
static void* thread2_start() {
  // yield before shared 'val1'
  sched_yield();
  if (not val1) {
    // Yield before shared 'val2'
    sched_yield();
    val2 = true;
  }
  assert(not (val1 and val2));
 RETURN NULL; }

```

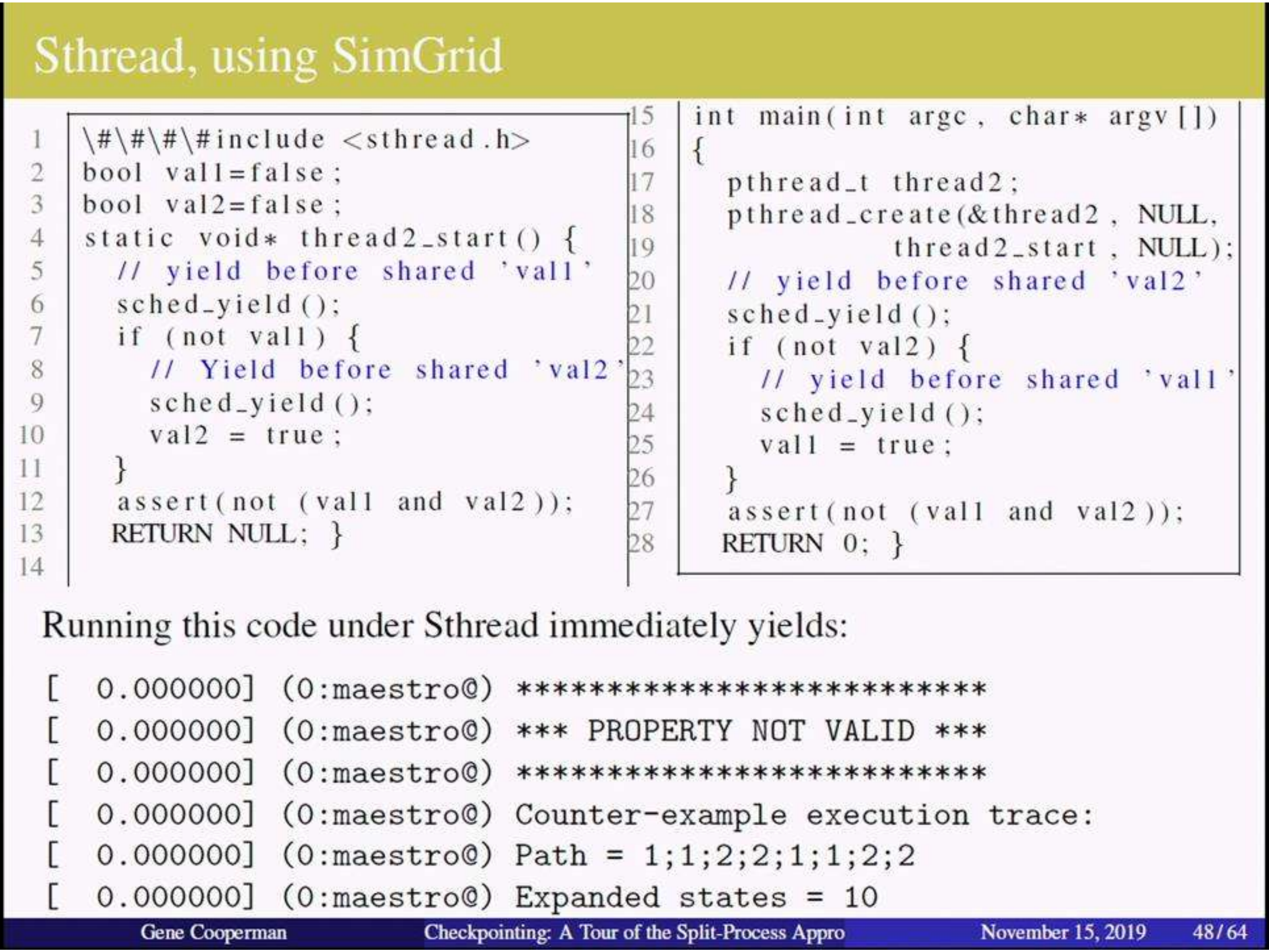
```
int main(int argc, char* argv[])
{
  pthread_t thread2;
  pthread_create(&thread2, NULL,
             thread2_start, NULL);
  // yield before shared 'val2'
  sched_yield();
  if (not val2) {
    // yield before shared 'val1'
    sched_yield();
    val1 = true;
  }
  assert(not (val1 and val2));
 RETURN 0; }
```

Running this code under Sthread immediately yields:

```
[  0.000000] (0:maestro@) ***************************
[  0.000000] (0:maestro@) *** PROPERTY NOT VALID ***
[  0.000000] (0:maestro@) ***************************
[  0.000000] (0:maestro@) Counter-example execution trace:
[  0.000000] (0:maestro@) Path = 1;1;2;2;1;1;2;2
[  0.000000] (0:maestro@) Expanded states = 10
```



```
/home/gene/simgrid.git/bin/simgrid-mc ./mc-two-thread-bug /home/gene/simgrid.git/examples/platforms/sm
all_platform.xml "--log=root.fmt:[%10.6r]%e(%i:%P@%h)%e%m%n" --log=xbt_cfg.thresh:warning --cfg=model-
check/reduction:none  || true
[  0.000000] (0:maestro@) Check a safety property. Reduction is: none.
[  0.000000] (0:maestro@) **************************
[  0.000000] (0:maestro@) *** PROPERTY NOT VALID ***
[  0.000000] (0:maestro@) **************************
[  0.000000] (0:maestro@) Counter-example execution trace:
[  0.000000] (0:maestro@)   [(1)Boivin (primary thread)] Mutex LOCK(locked = 0, owner = -1, sleeping =
 n/a)
[  0.000000] (0:maestro@)   [(1)Boivin (primary thread)] SIMCALL_MUTEX_UNLOCK(??)
[  0.000000] (0:maestro@)   [(2)Boivin (UNKNOWN)] Mutex LOCK(locked = 0, owner = -1, sleeping = n/a)
[  0.000000] (0:maestro@)   [(2)Boivin (UNKNOWN)] SIMCALL_MUTEX_UNLOCK(??)
[  0.000000] (0:maestro@)   [(1)Boivin (primary thread)] Mutex LOCK(locked = 0, owner = -1, sleeping =
 n/a)
[  0.000000] (0:maestro@)   [(1)Boivin (primary thread)] SIMCALL_MUTEX_UNLOCK(??)
[  0.000000] (0:maestro@)   [(2)Boivin (UNKNOWN)] Mutex LOCK(locked = 0, owner = -1, sleeping = n/a)
[  0.000000] (0:maestro@)   [(2)Boivin (UNKNOWN)] SIMCALL_MUTEX_UNLOCK(??)
[  0.000000] (0:maestro@) Path = 1;1;2;2;1;1;2;2
[  0.000000] (0:maestro@) Expanded states = 12
[  0.000000] (0:maestro@) Visited states = 15
[  0.000000] (0:maestro@) Executed transitions = 14
gene@dekaksi:~/simgrid-sthread/mc-two-thread-bug-2$ cd ..
gene@dekaksi:~/simgrid-sthread$
```



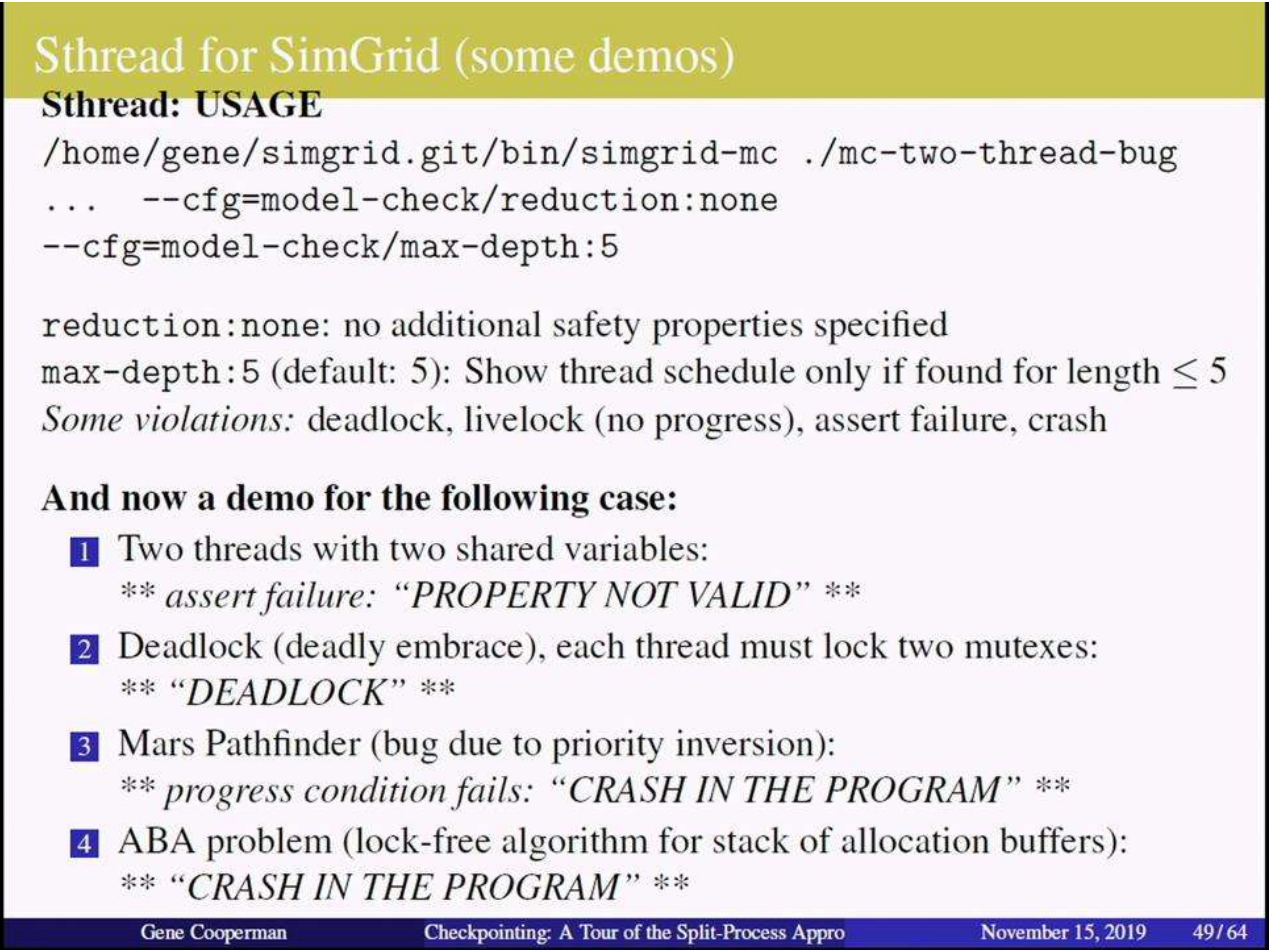
Sthread, using SimGrid

```
\#\#\#\#include <sthread.h>
bool val1=false;
bool val2=false;
static void* thread2_start() {
  // yield before shared 'val1'
  sched_yield();
  if (not val1) {
    // Yield before shared 'val2'
    sched_yield();
    val2 = true;
  }
  assert(not (val1 and val2));
 RETURN NULL; }

int main(int argc, char* argv[])
{
  pthread_t thread2;
  pthread_create(&thread2, NULL,
                 thread2_start, NULL);
  // yield before shared 'val2'
  sched_yield();
  if (not val2) {
    // yield before shared 'val1'
    sched_yield();
    val1 = true;
  }
  assert(not (val1 and val2));
 RETURN 0; }
```

Running this code under Sthread immediately yields:

```
[  0.000000] (0:maestro@) *************************
[  0.000000] (0:maestro@) *** PROPERTY NOT VALID ***
[  0.000000] (0:maestro@) *************************
[  0.000000] (0:maestro@) Counter-example execution trace:
[  0.000000] (0:maestro@) Path = 1;1;2;2;1;1;2;2
[  0.000000] (0:maestro@) Expanded states = 10
```



Sthread for SimGrid (some demos)

Sthread: USAGE

```
/home/gene/simgrid.git/bin/simgrid-mc ./mc-two-thread-bug
...  --cfg=model-check/reduction:none
--cfg=model-check/max-depth:5
```

`reduction:none`: no additional safety properties specified
`max-depth:5` (default: 5): Show thread schedule only if found for length ≤ 5
Some violations: deadlock, livelock (no progress), assert failure, crash

And now a demo for the following case:

1. Two threads with two shared variables:
   ** *assert failure: "PROPERTY NOT VALID"* **
2. Deadlock (deadly embrace), each thread must lock two mutexes:
   ** *"DEADLOCK"* **
3. Mars Pathfinder (bug due to priority inversion):
   ** *progress condition fails: "CRASH IN THE PROGRAM"* **
4. ABA problem (lock-free algorithm for stack of allocation buffers):
   ** *"CRASH IN THE PROGRAM"* **

Smpi: using SimGrid for MPI

Like Sthread for POSIX threads, there is also Smpi for MPI. Both are based on SimGrid.

For further information, see:

- `https://simgrid.org/tutorials/simgrid-smpi-101.pdf` (Especially, see pages 5 and 6, out of 22 pages.)

Outline

JOKE: In high-end HPC, we don't debug!!!

Why do practitioners repeat this joke?
ANSWER: High-end clusters are expensive, and should be used only for production — especially for "**deep executions**", where bugs might occur only after hours. So, development and debugging are done on small clusters, where the bug does not appear.

Why do new bugs appear during deep execution?

Some race conditions are rare. If the bug appears once in a billion times, we may have to execute a lot to see the bug.

Why is it expensive to debug on a high-end cluster?

a. Race conditions are difficult to debug. If only three nodes out of a hundred are involved in the bug, how do we trace those three nodes in isolation? (How do we even know which three nodes to trace?)
b. We could use GDB and breakpoints on a deep execution, but if we're using a hundred nodes and we debug for hours, this becomes expensive.
c. We could add extra print and assert statements and then run another production job. But if we do this five or ten more times in order to explore a deep execution, this also becomes expensive.

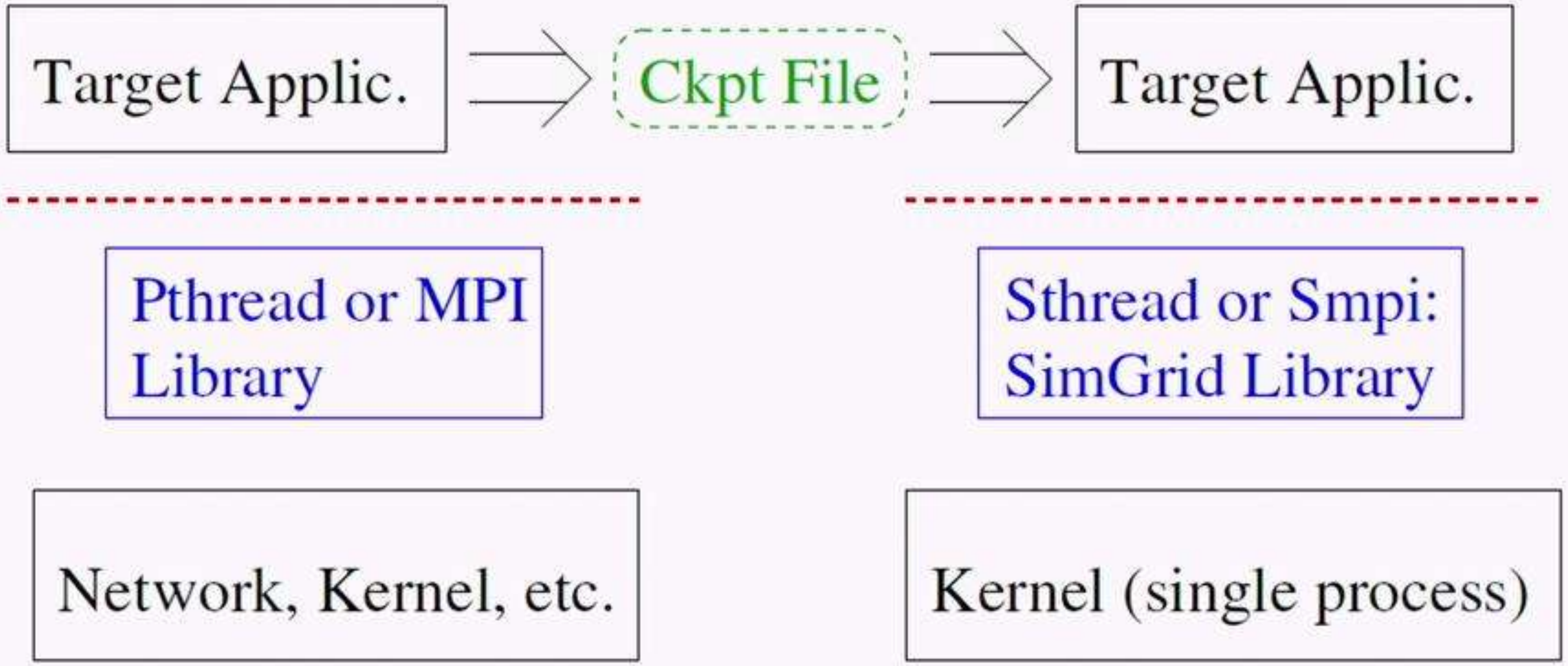
Deep SimGrid requires Checkpointing (work in progress)

This approach extends both Sthread (POSIX threads) and Smpi (MPI). It does *not* discover new bugs. But it can be used to diagnose new bugs.

1. Set DMTCP flags for periodic checkpointing (perhaps, every 5 minutes).
2. Run target (MPI or POSIX threads) application under DMTCP.
3. After program crash or assert failure:
   Restart from the last checkpoint, but under SimGrid (Sthread or Smpi).
4. Sthread/Smpi will produce a schedule for threads (Sthread) or MPI ranks (Smpi). This is the bug diagnosis.
5. Replay from the checkpoint, but in a “replay mode” under SimGrid or GDB.

Deep SimGrid requires Split Processes (and Checkpointing)

The Roadmap forward (in pictures):



Outline

Proxies for CUDA: the old approach

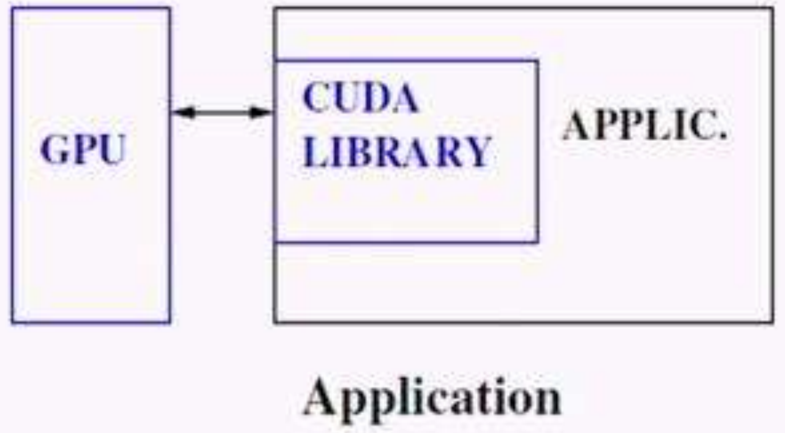
GOAL: ***To convince you of a general proxy paradigm for handling the "hard" checkpointing challenges that remain.***

We take as testbeds two such "hard" examples:

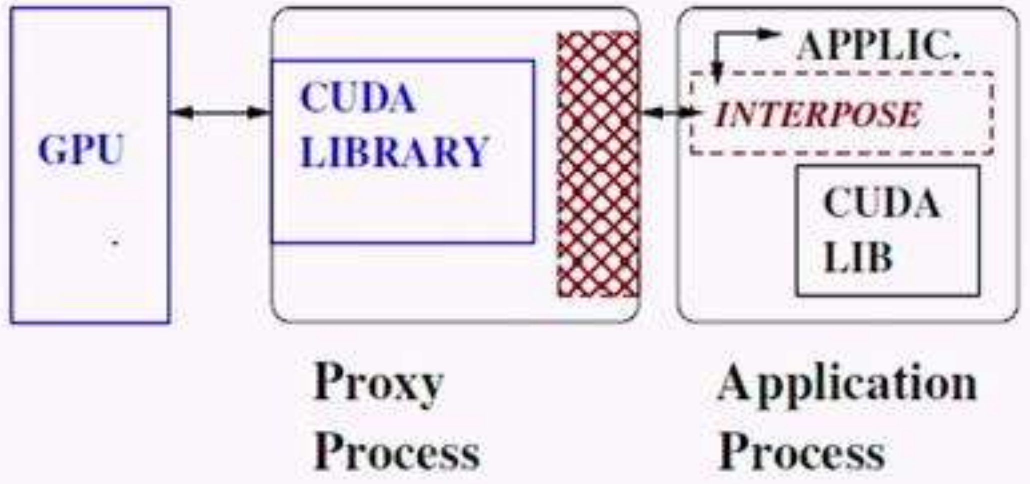
1. Checkpoint a CUDA application running on a GPU (Problem: how to save and restore the state of GPU hardware)
   (joint with Rohan Garg, Apoorve Mohan, Michael Sullivan)
   To appear, IEEE Cluster'18; see, also, technical report:
   `https://arxiv.org/abs/1808.00117`
2. Checkpoint MPI on NERSC/Cori supercomputer (with GNI network; no checkpoint-restart service to support it)

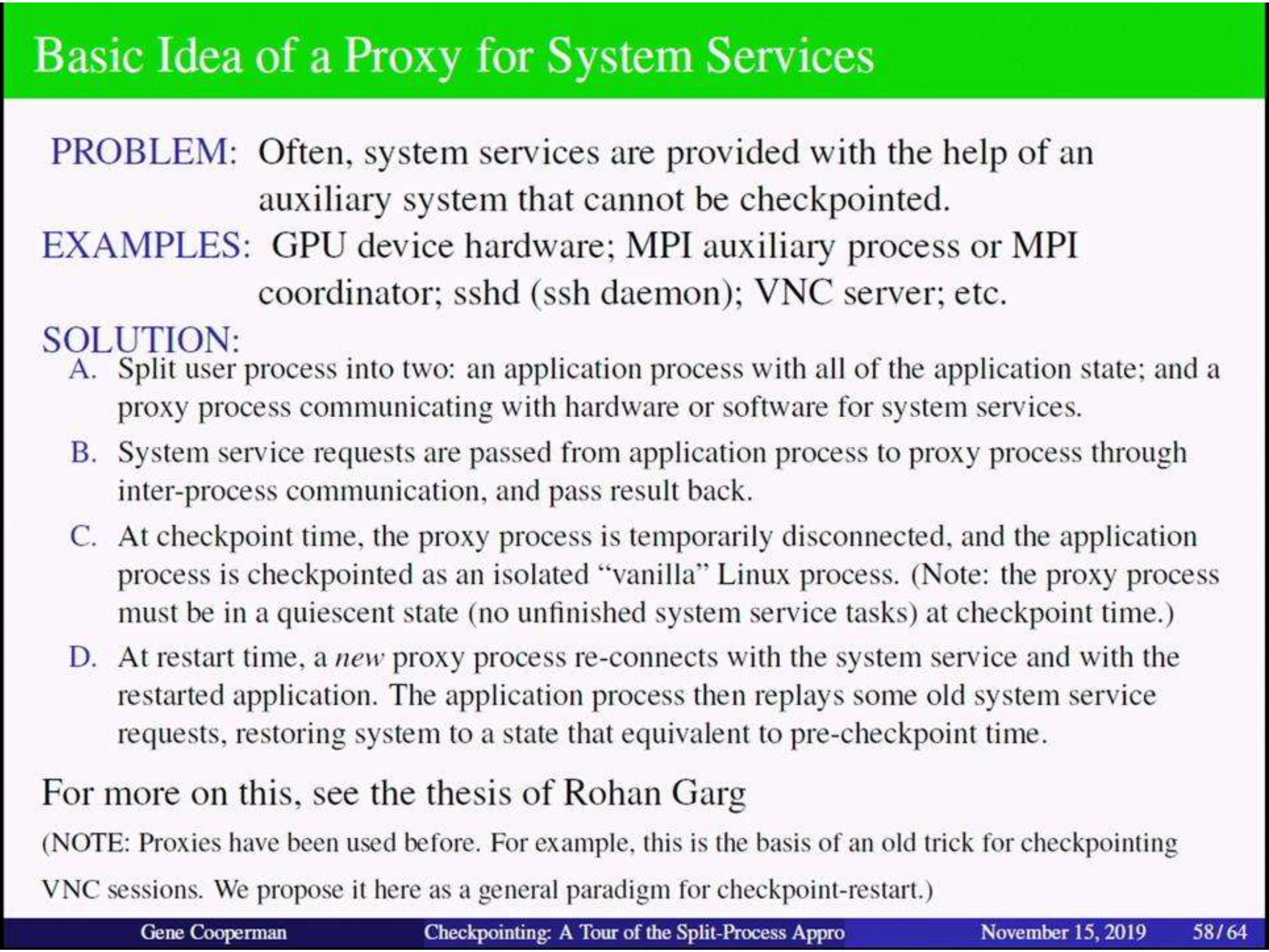
Proxies: basic idea

BEFORE:



AFTER:





Basic Idea of a Proxy for System Services

PROBLEM: Often, system services are provided with the help of an auxiliary system that cannot be checkpointed.

EXAMPLES: GPU device hardware; MPI auxiliary process or MPI coordinator; sshd (ssh daemon); VNC server; etc.

SOLUTION:

A. Split user process into two: an application process with all of the application state; and a proxy process communicating with hardware or software for system services.

B. System service requests are passed from application process to proxy process through inter-process communication, and pass result back.

C. At checkpoint time, the proxy process is temporarily disconnected, and the application process is checkpointed as an isolated "vanilla" Linux process. (Note: the proxy process must be in a quiescent state (no unfinished system service tasks) at checkpoint time.)

D. At restart time, a *new* proxy process re-connects with the system service and with the restarted application. The application process then replays some old system service requests, restoring system to a state that equivalent to pre-checkpoint time.

For more on this, see the thesis of Rohan Garg

(NOTE: Proxies have been used before. For example, this is the basis of an old trick for checkpointing VNC sessions. We propose it here as a general paradigm for checkpoint-restart.)

CRUM: "Checkpoint-Restart for Unified Memory" for CUDA

Proxy-based shadow paging for CUDA UVM (Unified Virtual Memory)

- *Shadow UVM page synchronization:* Catches memory transfers between proxy and application through memory permissions and segfault detection. The difficulty for transparent checkpointing with CUDA-managed memory: **How to make this efficient?** See the CRUM paper (Algorithm 1, shadow-page synchronization) for details.
- Enables fast *forked checkpointing model* for UVM memory that overlaps writing a checkpoint image to stable storage, while the application continues. *Almost free benefit of our approach!* (This was difficult in the past due to the need to share memory among the GPU device, and the UVM-based host that was split among parent and forked child processes.)

A Question on Split Processes from a Previous Audience

This all seems pretty obvious, to have two programs in the same address space in memory. Why hasn't somebody done this before?

ANSWER: Well, we think it's new. Besides, we still haven't heard of somebody else who has a kernel loader.

A Better Answer to the Previous Question

1. There are experts on the Linux kernel, and they know everything about launching a classic process (text/data/stack).
2. There are experts on the user-space side of operating systems: the linker, the loader, the compiler, the fork/exec process model.
   - (And there's even a cool result using shared memory and dlmopen (see 'man dlmopen') to force the linker to load two independent programs: see "Process-in-Process", Hori et al., HPDC'18)
     But that method uses multiple threads, and the cost of context-switching can be expensive: recall the problem with the 'fs' register
3. But there aren't enough experts on the interface between the two. By working on transparent checkpointing, we get to know the interface deeply: link maps, fs register, thread-local storage, thread-control block, auxv (auxiliary vector), /proc/*/maps, memory guard pages, user-space page fault handling, vdso, vvar, etc.
4. **CONCLUSION:** Come work with us on other cool things. We're always looking for collaborators on our open-source software, on research, and *of course for potential new Ph.D. students.*

Questions?

THANKS TO THE MANY STUDENTS AND OTHERS WHO HAVE CONTRIBUTED TO DMTCP OVER THE YEARS:

Jason Ansel, Kapil Arya, Alex Brick, Jiajun Cao, Tyler Denniston, Xin Dong, William Enright, Rohan Garg, Paul Grosu, Twinkle Jain, Samaneh Kazemi, Jay Kim, Gregory Kerr, Apoorve Mohan, Mark Mossberg, Gregory Price, Manuel Rodríguez Pascual, Artem Y. Polyakov, Michael Rieker, Praveen S. Solanki, Ana-Maria Visan

OTHER QUESTIONS?

Checkpointing the Un-checkpointable: the Split-Process Approach for MPI and Formal Verification

Gene Cooperman[*]
gene@ccs.neu.edu

Khoury College of Computer Sciences
Northeastern University, Boston, USA

November 15, 2019

[*]Partially supported by NSF Grants ACI-1440788 and OAC-1740218, and by grants from Intel Corporation, Raytheon, and Mentor Graphics (a division of Siemens).

DMTCP History

The DMTCP project and antecedents began almost 15 years ago, and is widely used today:
`http://dmtcp.sourceforge.net/publications.html`

Typical use case for HPC: 12-hour batch time slot, an application is expected to finish in 18 hours.
(NOTE: A resource manager such as SLURM can send a signal one hour before the end of the time slot, giving the application time to checkpoint; DMTCP can then transparently checkpoint the state.)

Ease of use (unprivileged and transparent):

```
dmtcp_launch a.out arg1 arg2 ...
dmtcp_command --checkpoint # from other terminal or window
dmtcp_restart ckpt_a.out_*.dmtcp
```

(DMTCP also works for programs that are multi-threaded, distributed, MPI-based, ...)