Runtime Techniques for Automatic Process Virtualization

Evan Ramos, Sam White, Aditya Bhosale, and Laxmikant V. Kale

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

P2S2 Workshop

August 29th, 2022

Motivation

- **MPI** is the dominant parallel programming system used for high performance scientific computing

- MPI-1.0 standard published in 1994
  - Supercomputers at the time:
    - Composed of uniprocessor nodes
    - Offered predictable performance
    - Generally reliable

Motivation

- **MPI** is the dominant parallel programming system used for high performance scientific computing

- MPI-1.0 standard published in 1994
  - Supercomputers ~~at the time~~ today:
    - Composed of wide multiprocessor nodes
    - Performance subject to runtime variation
    - Faults increasingly frequent at large scales

Motivation

- Programming models for modern HPC systems:
  - **MPI-everywhere** still quite popular
    - Still performs well on CPU systems
    - Legacy code: no porting required to new models
  - **MPI+X** is becoming increasingly popular
    - X is a shared memory parallel prog. model, i.e. OpenMP
    - Useful for heterogeneous systems, reduced memory footprint
  - **Task-based prog. models**: Legion, Charm++, Chapel, etc.
    - Separate logical expression of parallelism from scheduling, mapping of work to the system
    - Either new codes or pervasive rewrites of existing code

Motivation

- **Idea**: virtualize MPI ranks as threads rather than operating system processes
  - Potential benefits:
    - Co-schedule ranks on each core to overlap communication
    - Migrate ranks between cores/nodes for load balancing
    - Share memory and resources across each node
    - Optimize for communication locality within the shared address space on each node
    - . . . without application code changes! (Ideally)

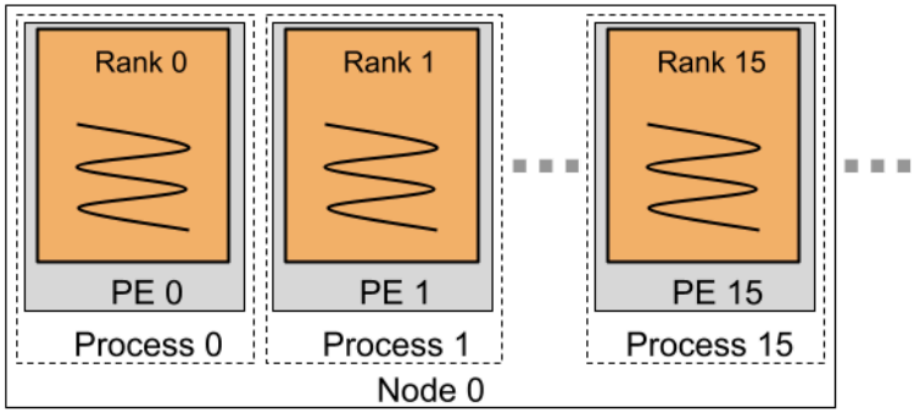
Motivation

- **Adaptive MPI (AMPI)** is an implementation of MPI on top of Charm++
  - We base our work on AMPI and its runtime system
  - MPC, FG-MPI, TMPI are other threaded MPI implementations
  - Also similar to MPI endpoints proposal

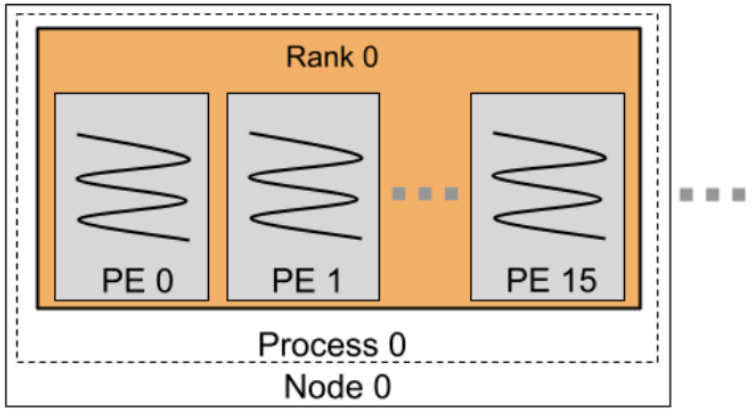
  - **Goal**: make running legacy MPI codes on AMPI trivial

Motivation

MPI-everywhere

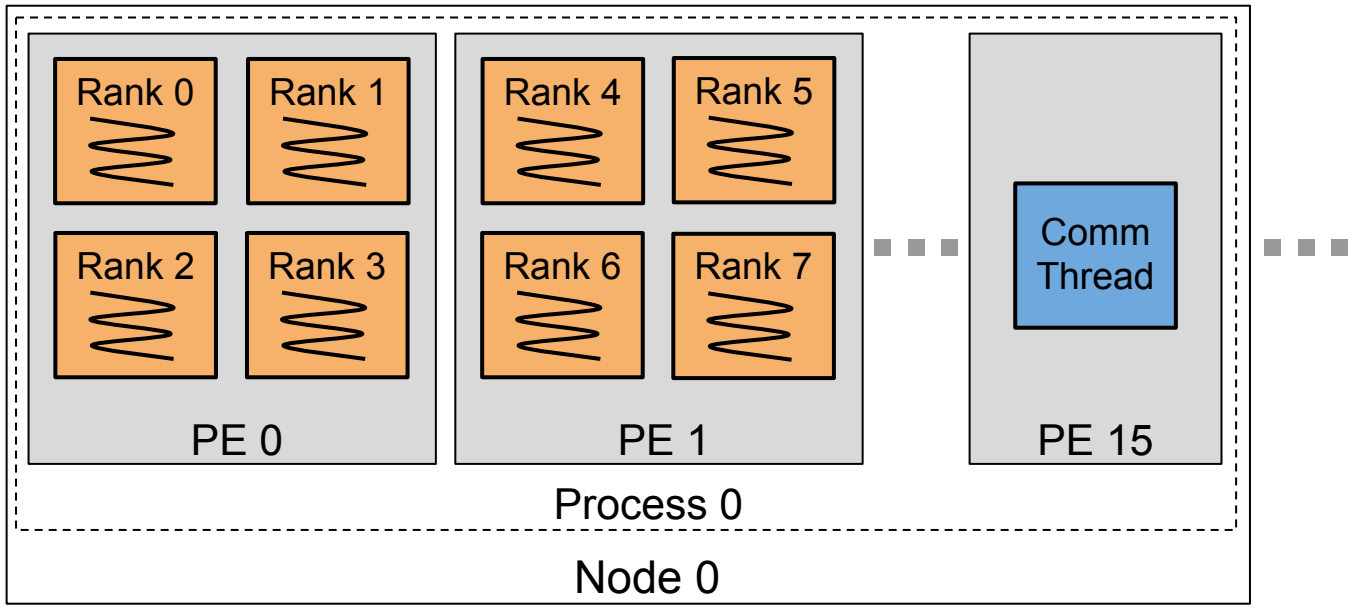


MPI+X



Motivation

Thread-based MPI



Overview

- Background
- Proposed Solutions
- Performance Evaluation
- Conclusion

Background

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Background

- **Problem**: implementing MPI ranks as threads can break user code that assumes ranks as processes

Code:

```
int my_rank;
int num_ranks;

int main(int argc, char** argv)
{
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);

  MPI_Barrier(MPI_COMM_WORLD);
  printf("rank: %d\n", my_rank);

  MPI_Finalize();
  return 0;
}
```

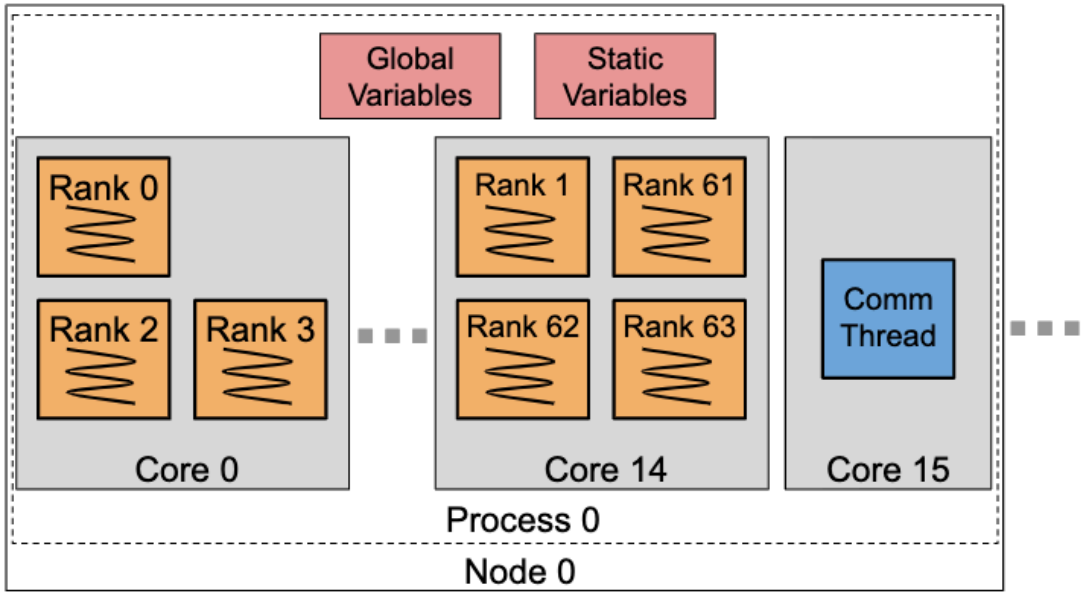
Mutable global variables

Possible output:

```
\$ ./hello_world +vp 2
rank: 1
rank: 1
```

Background

- **Problem**: mutable global & static variables become unexpectedly shared among ranks in the same process when virtualized



Background

- Existing techniques for code privatization lack either:
  - Portability: OS, compiler, linker, etc.
  - Automation: requiring developer effort
  - Support for dynamic runtime capabilities: SMP or migration

Method	Automation	Portability	SMP Mode Support	Migration Support
Manual refactoring	Poor	Good	Yes	Yes
Photran	Fortran-specific	Good	Yes	Yes
Swapglobals	No static vars	Linker-specific	No	Yes
TLSglobals	Mediocre	Compiler-specific	Yes	Yes
-fmpc-privatize	Good	Compiler/linker-specific	Yes	Not implemented, but possible
Process-in-Process	Good	Requires GNU libc extension	Limited w/o patched glibc	Unknown

Background

- Goals:
  - **Automation**: no developer effort required
  - **Portability**: across compilers, linkers, operating systems, etc.
  - **Performance**: minimal runtime overheads
  - **Runtime capabilities**: arbitrary degrees of virtualization, support for shared memory execution & rank migration

Proposed Solution

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Proposed Solution

- Based on our objectives, we developed support for 3 novel runtime techniques for process virtualization:
  - All inspired by PIP*'s idea of compiling code as a Position-Independent Executable (PIE) and providing each rank a view of code segments
  - Whereas PIP has typically been used for sharing code between processes, we use the idea to privatize code for each virtual process

** Atsushi Hori et al. "Process-in-process: techniques for practical address-space sharing," HPDC 2018.*

Solution

- All three of our approaches share the same basic methodology
  - Compile the application code as a PIE
    - This results in a global/static variables being defined relative to the instruction pointer
  - Link with our runtime as a function pointer shim library
  - Create unique views of the PIE binary code per virtual process
- **Benefits**: fully automatic privatization, no compiler/linker dependency, no special context switch handling

Solution

- The difference in our three methods comes from how the unique views of the PIE code are created:
  - **PIPglobals** uses *dlmopen* (a GNU libc extension) with unique namespace indices per rank
  - **FSglobals** copies the code segments separately onto a shared file system per rank
  - **PIEglobals** allocates separate copies of the code in memory per rank via Isomalloc, our migratable memory allocator

Limitations

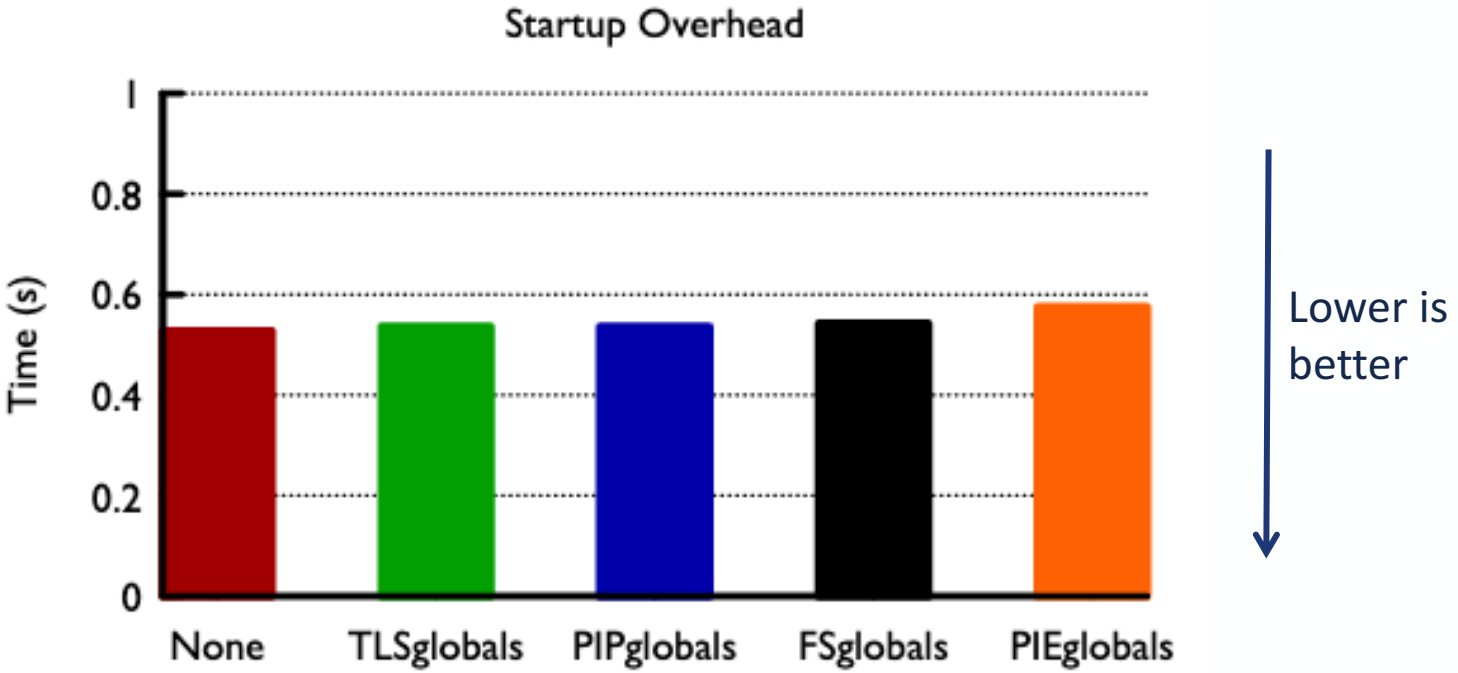
- **PIPglobals**
  - Only supports 10x virtual ranks per process without a patched version of glibc
  - No support for migrating virtualized code segments
  - Portability: *dlmopen*
- **FSglobals**
  - Portability: requires a shared file system
  - Unscalable file I/O usage: file per virtual rank
  - No support for rank migration

PIEglobals

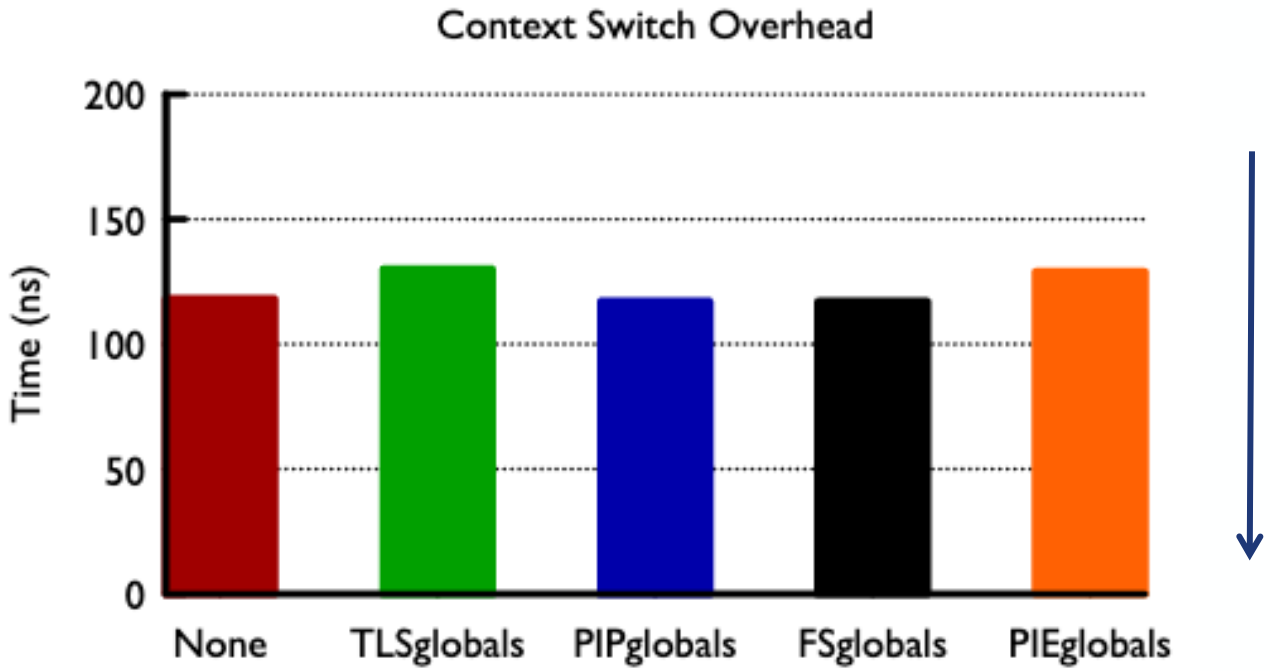
- **Benefits**:
  - Fully automated privatization
  - Support for arbitrary degrees of virtualization / SMP mode
  - Support for user-level runtime rank migration
  - Portable: in practice, more so than PIPglobals
- **Limitations**:
  - Memory overhead: static footprint & migration overhead
  - Debuggability: relocated code segments

Performance Evaluation

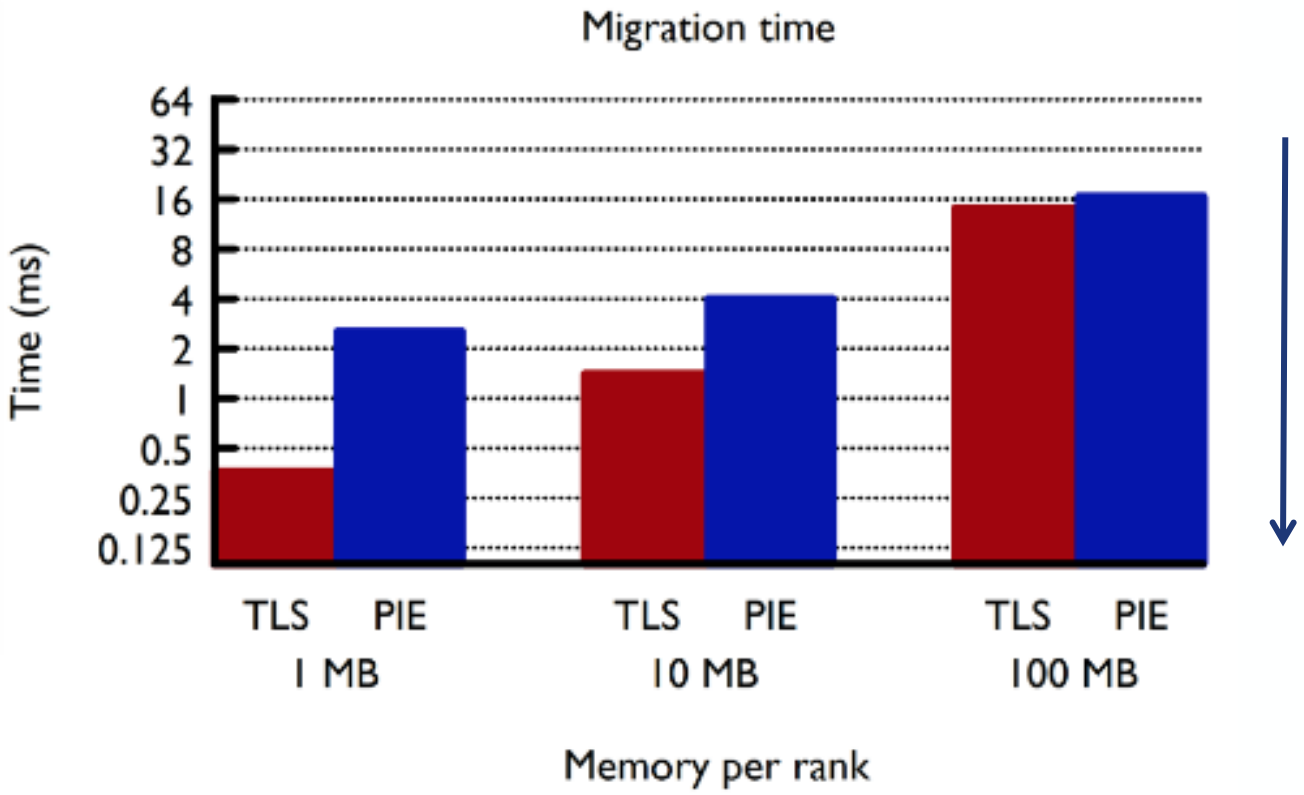
Performance



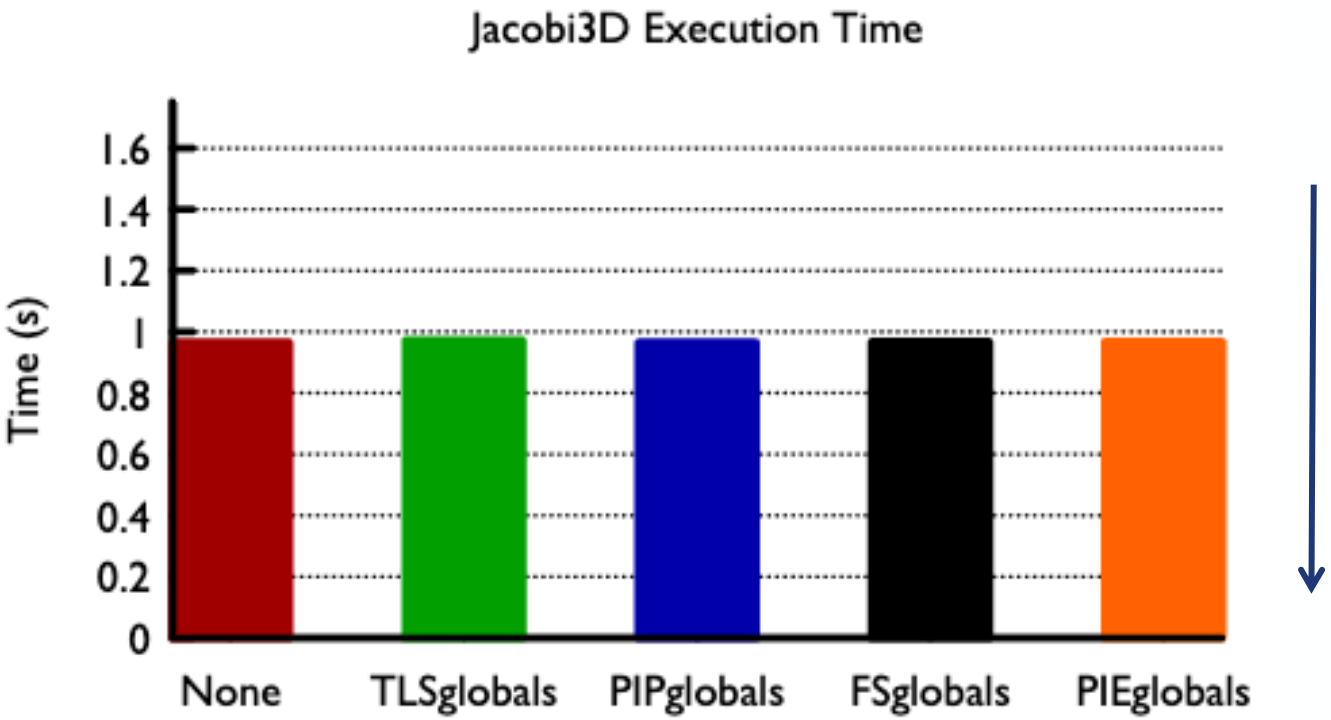
Performance



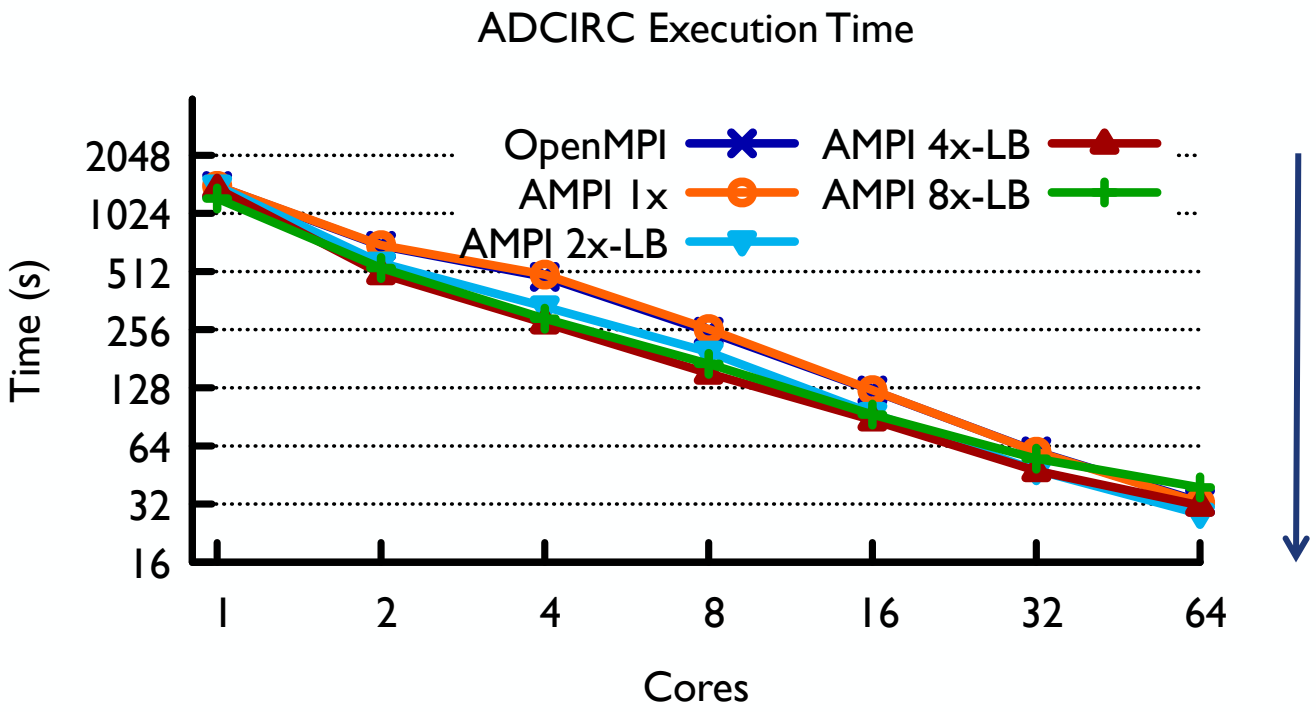
Performance



Performance



Performance: ADCIRC



Cores	1	2	4	8	16	32	64
Speedup %	12	58	78	70	43	24	18

Conclusion

- Process virtualization for legacy codes is challenging
- Our runtime techniques provide:
  - Full automation: no developer effort required
  - Improved portability: no compiler/linker dependency
  - Support for SMP mode and rank migration (PIEglobals only)
- Future work:
  - Improve memory usage & debuggability of PIEglobals
  - Prove PIEglobals on more production applications
  - Compiler automation remains appealing, if portable and migration supported

Questions?

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN