Privatization Techniques for Process Virtualization:
AMPI (+ PiP)

Laxmikant (Sanjay) Kale
Sam White, Evan Ramos
Background

- AMPI virtualizes the ranks of MPI_COMM_WORLD
  - AMPI ranks are user-level threads (ULTs), not OS processes
  - Benefits:
    - **Overdecomposition**: run with more ranks than cores
    - **Asynchrony**: overlap one rank’s communication with another rank’s computation
    - **Migratability**: ULTs are migratable at runtime across address spaces
      - Enables dynamic load balancing and fault tolerance
AMPI Benefits

- Communication Optimizations
  - Overlap of computation and communication
  - Communication locality of virtual ranks in shared address space
- Dynamic Load Balancing
  - Many strategies available, balance by migrating AMPI virtual ranks
  - Isomalloc memory allocator serializes all of a rank’s state
- Fault Tolerance
  - Can restart online, within the same job
Migratability

- Isomalloc memory allocator *reserves* a globally unique slice of the virtual memory address space in each process for each virtual rank
- **Benefit:** no application-specific serialization code
  - Handles the user-level thread stack and all user heap allocations
  - Works everywhere except BGQ and Windows
  - Enables dynamic load balancing and fault tolerance
- **Hurdle:** Address-Space Layout Randomization
  - Can synchronize at startup and determine the lowest address space common among all processes
Privatization

Illustration of unsafe global/static variable accesses:

```c
int rank_global;

void func(void)
{
    MPI_Comm_rank(MPI_COMM_WORLD, &rank_global);

    MPI_Barrier(MPI_COMM_WORLD);

    printf("rank: %d\n", rank_global);
}
```
Privatization Goals

- Fully automatic privatization, or at least semi-automated
- Portable across OSes, compilers
- User-level: no changes to OS, compiler, or system libraries preferably
- Handling of both global and static variables
- Support for static and dynamic linking
- Ability to share read-only state across virtual ranks
- Support for runtime migration of virtual processes (achieved with Isomalloc)
Privatization Methods

● Existing Methods
  ○ Manual refactoring
    ■ Developer encapsulates mutable global state
    ■ Can take days/weeks of developer effort
  ○ Refactoring tools (Photran)
  ○ GOT (global offset table) swapping (Swapglobals)
    ■ Doesn’t handle statics, requires ELF and a patched linker
  ○ Thread-local storage segment pointer swapping (TLSglobals)
    ■ Only works with GCC and new Clang
    ■ Need to tag variable declarations (but not accesses)
Privatization Methods

● In-Development Methods
  ○ Process-in-Process (PiPglobals): user-level library by Atsushi Hori (RIKEN R-CCS)
  ○ File-system Globals (FSglobals)
  ○ Clang/Libtooling-based source-to-source transformation

● Proposed Methods
  ○ MPC (Multi-Processor Computing) -fmpc-privatize: requires compiler and linker support
  ○ ROSE tool for source-to-source transformation
AMPI + PiP: Implementation Details

1. Compile MPI user binary as PIE (Position Independent Executable)
2. For each rank’s binary, call `dlmopen` with a unique namespace index (`lmid`)
   ○ `void *dlmopen (Lmid_t lmid, const char *filename, int flags);`
3. Use `dlsym` to look up and call the entry point for each rank
4. Global variables will be privatized with no modification to user program code
   ○ PIE binaries locate .data immediately following .text in memory
   ○ PIE global variables are accessed relative to the instruction pointer
   ○ `dlmopen` creates a separate copy of the binary in memory for each namespace
AMPI + PiP

Implementation Hurdles:

- *dlmopen* fails after 11 virtual ranks per process due to glibc limits
  - Requires patched glibc: PiP-glibc
- Runtime migration of virtual processes is difficult
  - Will require patched ld-linux.so to intercept mmap allocations of .data (and .text) segments
  - Allocations would be redirected through Isomalloc
AMPI + Filesystem Globals

Similar to PiPglobals, but copies PIE binary on filesystem per-rank, then `dlopen`

+ Does not depend on GNU/Linux-specific `dlmopen` extension
+ Does not have 11-rank per-process limit in the absence of patched glibc
+ Like PiPglobals, requires no modification of user program code
  - Wasteful, slow use of filesystem at startup
  - Same migration limitation as PiPglobals
Conclusion

- AMPI is valuable for a growing set of applications, though privatization remains a challenge for legacy MPI codes
  - Clang refactoring tool and PiPglobals are promising
  - Hurdle: migration support for AMPI + PiP

- Questions for you:
  - Use cases for process virtualization or for privatization?
  - Any JLESC applications interested in AMPI?
  - Ideas for other privatization techniques?
Questions?
AMPI + PiP Details

Implementation Hurdles:

- Cannot simply compile AMPI programs as PIE and call `dlmopen`
  - Depending on approach, would either
    - Privatize entire Charm++/AMPI runtime system
    - Runtime would not function
    - Waste of memory
    - Prevent `dlmopen`ed binary from seeing launcher’s AMPI symbols
  - Instead, restructure headers and link with a function pointer shim
  - Only user program needs to be PIE

```c
ampi_functions.h:
AMPI_FUNC(int, MPI_Send, const void *, int count, MPI_Datatype, int dest, int tag, MPI_Comm)
```

```c
mpi.h:
#ifdef AMPI_USE_FUNCPTR
#define AMPI_FUNC(return_type, function_name, ...) \
    extern return_type (* function_name)(__VA_ARGS__);
#else
#define AMPI_FUNC(return_type, function_name, ...) \
    extern return_type function_name(__VA_ARGS__);
#endif
#include "ampi_functions.h"
```

```c
ampi_funcptr.h:
struct AMPI_FuncPtr_Transport {
    #define AMPI_FUNC(return_type, function_name, ...) \
    return_type (* function_name)(__VA_ARGS__);
#include "ampi_functions.h"
};
```

```c
ampi_funcptr_loader.C (linked with AMPI runtime):
void AMPI_FuncPtr_Pack(struct AMPI_FuncPtr_Transport * x) {
    #define AMPI_FUNC(return_type, function_name, ...) \
    x->function_name = function_name;
#include "ampi_functions.h"
}
```

```c
ampi_funcptr_shim.C (linked with MPI user program):
void AMPI_FuncPtr_Unpack(struct AMPI_FuncPtr_Transport * x) {
    #define AMPI_FUNC(return_type, function_name, ...) \
    function_name = x->function_name;
#include "ampi_functions.h"
}