Recent Developments in Adaptive MPI

Sam White & Evan Ramos
Overview

● Introduction to AMPI
● Recent Work
  ○ Communication Optimizations (Sam)
  ○ Automatic Global Variable Privatization (Evan)
Introduction
Motivation

● Variability in various forms (SW and HW) is a challenge for applications moving toward exascale
  ○ Task-based programming models address these issues

● How to adopt task-based programming models?
  ○ Develop new codes from scratch
  ○ Rewrite existing codes, libraries, or modules (and interoperate)
  ○ Implement other programming APIs on top of tasking runtimes
Background

- AMPI virtualizes the ranks of MPI_COMM_WORLD
  - AMPI ranks are user-level threads (ULTs), not OS processes
Background

- AMPI virtualizes the ranks of MPI_COMM_WORLD
  - AMPI ranks are user-level threads (ULTs), not OS processes
  - Cost: virtual ranks in each process share global/static variables
  - 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
AMPI Benefits

- Communication Optimizations
  - Overlap of computation and communication
  - Communication locality of virtual ranks in shared address space

- Dynamic Load Balancing
  - Balance achieved by migrating AMPI virtual ranks
  - Many different strategies built-in, customizable
  - Isomalloc memory allocator serializes all of a rank’s state

- Fault Tolerance
  - Automatic checkpoint-restart within the same job
AMPI Benefits: LULESH-v2.0

No overdecomposition or load balancing (8 VPs on 8 PEs):

With 8x overdecomposition, after load balancing (7 VPs on 1 PE shown):
Migratability

- **Isomalloc** memory allocator reserves a globally unique slice of virtual memory space in each process for each virtual rank.
  - Benefit: no user-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
Communication Optimizations
Communication Optimizations

- AMPI exposes opportunities to optimize for communication locality:
  - Multiple ranks on the same PE
  - Many ranks in the same OS process
Communication Optimizations

- Recent work: optimize for point-to-point messaging within a process
  - No need for kernel-assisted interprocess copy mechanism
  - Motivated the Charm++ Zero Copy communication APIs
Communication Optimizations

- Application study: XPACC’s *PlasCom2* code
  - Now seeing AMPI outperform MPI (+OMP) even without LB
Communication Optimizations

- New virtualization-aware collective implementations avoid $O(VP)$ message creation and copies
  - Next: further shared-memory awareness
Communication Optimizations

- Recent study of memory usage by AMPI applications
  - Led to hoisting AMPI’s read-only memory storage to node-level
  - Future work: support for in-place rank migration via RDMA
Automatic Privatization
Privatization Problem

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 shared 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
    - Portable
  - Refactoring tools (Photran)
  - GOT (global offset table) swapping (Swapglobals)
    - Doesn’t handle statics
    - Requires ELF and old GNU ld linker version (< 2.24 w/o patch, < ~2.29 w/ patch)
  - Thread-local storage segment pointer swapping (TLSglobals)
    - Need to tag variable declarations (but not accesses)
    - Linux: Only works with GCC and new Clang
    - macOS: Works with Apple Clang and GCC (newly implemented in AMPI)
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, 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 each namespaced handle’s entry point
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:

- \textit{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 + 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:
#define AMPI_FUNC(return_type, function_name, ...) 
  extern return_type (* function_name)(__VA_ARGS__);
#endif
#include "ampi_functions.h"

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

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"
}

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"
}
```

```c
ampi_modules.h:
AMPI_FUNC(int, MPI_Send, const void *msg, int count, 
          MPI_Datatype type, int dest, int tag, MPI_Comm comm)
```
AMPI + Filesystem Globals

Similar to PiPglobals, but copies PIE binary on filesystem per-rank, then \texttt{dlopen}.

- Does not depend on GNU/Linux-specific \texttt{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 increasingly valuable for a growing set of applications
  ○ Not just those with load imbalance

● Recent work spans the full stack of AMPI
  ○ Conformance to the MPI-3.1 standard
  ○ Communication performance improvements in AMPI/Charm++/LRTS
  ○ More automated tooling for conversion of legacy code
  ○ Working closely with more application developers
Questions?
## AMPI Standard Compliance

<table>
<thead>
<tr>
<th>MPICH-3.2 Test Suite</th>
<th># Passing in 2014</th>
<th># Passing in 2019</th>
<th>Total # of Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-to-Point</td>
<td>28</td>
<td>40</td>
<td>42</td>
</tr>
<tr>
<td>Collectives</td>
<td>41</td>
<td>69</td>
<td>72</td>
</tr>
<tr>
<td>Datatypes</td>
<td>32</td>
<td>64</td>
<td>68</td>
</tr>
<tr>
<td>Communicators</td>
<td>24</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Groups</td>
<td>4</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Topologies</td>
<td>10</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>Attributes</td>
<td>8</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>Infos</td>
<td>7</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Init</td>
<td>6</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>