Recent Progress on Adaptive MPI

Sam White & Evan Ramos
Overview

- Introduction to AMPI
- Recent Work
  - Collective 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):
Migratibility

- **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
Point-to-Point Communication

- Past work: optimize for point-to-point messaging within a process
  - No need for kernel-assisted interprocess copy mechanism
  - Motivated generic Charm++ Zero Copy APIs
Point-to-Point Communication

- Application study: XPACC’s *PlasCom2* code
  - AMPI outperforms MPI (+ OpenMP), even without LB

![Graphs comparing MPI vs. MPI+OMP and MPI vs. AMPI](image-url)
Collective Communication

- Virtualization-aware collective implementations avoid $O(VP)$ message creation and copies
  - [nokeep] optimized to avoid msg copies on recv-side of bcasts
  - Zero Copy APIs to match MPI’s buffer ownership semantics
  - For reductions, avoid CkReductionMsg creation & copy
  - Revamping Sections/CkMulticast for subcommunicator collectives
Collective Communication

- Node-aware reductions: small msg optimizations
  - Sender-side streaming: no intermediate CkReductionMsg creation & copy
  - Dedicated shared buffer per node per comm

<table>
<thead>
<tr>
<th>Version</th>
<th>CrayMPI VP=1 (usec)</th>
<th>AMPI VP=1 (usec)</th>
<th>AMPI VP=16 (usec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>1.24</td>
<td>5.32</td>
<td>9.81</td>
</tr>
<tr>
<td>Sender-side streaming</td>
<td>---</td>
<td>5.35</td>
<td>5.71</td>
</tr>
<tr>
<td>… + dedicated shared buffer</td>
<td>---</td>
<td>1.77</td>
<td>3.18</td>
</tr>
</tbody>
</table>
Collective Communication

- Node aware reductions: large msg optimizations
Memory Usage

- Recent study of memory usage by AMPI applications
  - User-space zero copy communication b/w ranks in shared address space -> lower rendezvous threshold
    - Avoid overheads of kernel-assisted IPC
  - Led to hoisting AMPI’s read-only memory storage to node-level
    - Predefined datatype objects, reduction ops, groups, etc.
  - Developed in-place rank migration support via RDMA
    - Zero copy PUP API for large buffer migration (Isomalloc)
Memory Usage

Total Memory Usage on PE 0 of Jacobi-3D on Stampede2 (TACC)

- AMPI
- AMPI-new
Automatic Privatization
Privatization Problem

Illustration of unsafe global/static variable accesses:

```c
int rank_global;

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

    MPI_Barrier(MPI_COMM_WORLD);

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

- Manual refactoring
  - Developer encapsulates mutable global state into struct
  - Allocate struct on stack or heap, pass pointer as part of control flow
  - Most portable strategy
  - Can require extensive developer effort and invasive changes
Privatization Method Goals

- **Ease of use:** Method should be as automated as possible
- **Portability**
  - Portable across OSes, compilers
  - Require few/no changes to OS, compiler, or system libraries
- **Feature support**
  - Handle both `extern` and `static` global variables
  - Support for static and shared linking
  - Support for runtime migration of virtual ranks (using Isomalloc)
- **Optimizable:** Can share read-only state across virtual ranks in node
Privatization Methods

- First-generation automated methods
  - Swapglobals: GOT (global offset table) swapping
    - No changes to code: AMPI runtime walks ELF table, updating pointers for each variable
    - Does not handle *static* variables
    - Requires obsolete *GNU ld* linker version (< 2.24 w/o patch, < ~2.29 w/ patch)
    - O(n) context switching cost
    - Deprecated
  - TLSglobals: Thread-local storage segment pointer swapping
    - Add *thread_local* tag to global variable declarations and definitions (but not accesses)
    - Supported with migration on Linux (GCC, Clang 10+), macOS (Apple Clang, GCC)
    - O(1) context switching cost
    - Good balance of ease of use, portability, and performance
Privatization Solutions

● Source-to-source transformation tools
  ○ Camfort, Photran, ROSE tools explored in the past
  ○ Clang/Libtooling-based tools are promising
    ■ Prototype C/C++ TLSglobals transformer created at Charmworks
    ■ Interested in building encapsulation transformer (more complex)
    ■ Flang/F18 merged into LLVM 11, hope to see Fortran Libtooling support
  ○ Some bespoke scripting efforts
Privatization Methods

- Second-generation automated methods
  - PiPglobals: Process-in-Process Runtime Linking (thanks RIKEN R-CCS)
  - FSglobals: Filesystem-Based Runtime Linking
- How they work
  - `ampicc` builds the MPI program as a PIE shared object (process-independent executable)
  - PIE binaries store and access globals relative to instruction pointer
  - AMPI runtime uses dynamic loader to instantiate a copy for each rank
    - PiPglobals: Call glibc extension `dlmopen` with unique `Lmid_t` namespace index per-rank
    - FSglobals: Make copies of `.so` on disk for each rank, call `dlopen` on them normally
- Integrated into Charm’s nightly unit testing on production machines
Privatization Methods

● PiPglobals and FSglobals have drawbacks
  ○ PiPglobals requires patched PiP-glibc for >11 virtual ranks per process
  ○ FSglobals slams the filesystem making copies
  ○ FSglobals does not support programs with their own shared objects
  ○ Neither supports migration: Cannot Isomalloc code/data segments

● How to resolve drawbacks?
  ○ Patch ld-linux.so to intercept mmap allocations of segments?
  ○ Get hands dirty at runtime... new method: PIEglobals
Privatization Methods: PIEglobals

- PIEglobals: Position-Independent Executable Runtime Relocation
  - Leverage existing .so loading infrastructure from PiP/FSglobals
  - AMPI processes the shared object at program start
    - `dlopen`: dynamically load shared object once per node
    - `dl_iterate_phdr`: get list of program segments in memory
    - duplicate code & data segments for each virtualized rank w/ Isomalloc
    - scan for and update PIC (position-independent code) relocations in data segments and global constructor heap allocations to point to new privatized addresses
    - calculate privatized location of entry point for each rank and call it
  - Global variables become *privatized* and *migratable*
Privatization Methods: PIEglobals

● Pitfalls
  ○ Program startup overhead (ex. miniGhost: ~2 seconds)
  ○ Debugging is difficult: debug symbols don’t apply to copied segments
    ■ Debug without PIEglobals (no virtualization) as much as possible
    ■ Helpful GDB commands: call pieglobalsfind($rip) or call pieglobalsfind((void *)0x...)
  ○ Relocation scanning can incur false positives
    ■ Solution in development: Open two copies using dlmopen, scan contents pairwise
  ○ Machine code duplication causes icache bloat and migration overhead
    ■ Solutions: posix_memfd mirroring within nodes; extend Isomalloc bookkeeping
  ○ Requires Linux and glibc v2.2.4 or newer (v2.3.4 for dlmopen)

● Successes: miniGhost, Nekbone
● Frontiers: OpenFOAM, mpi4py
Conclusion

● AMPI is increasingly valuable for a growing set of applications
  ○ Benefits apparent even in applications without load imbalance
  ○ Close to running complex legacy codes with virtualization easily

● Recent work spans the full stack of AMPI
  ○ Conformance to the MPI standard and conventions of other MPIs
  ○ Communication and memory improvements
  ○ More automation for privatization of legacy code
  ○ Working closely with more application developers

● Rebranding as Charm MPI to emphasize underlying technology
Questions?

white67@illinois.edu
evan@hpccharm.com
Privatization Methods

• Proposed Methods
  ○ MPC (Multi-Processor Computing) -fmpc-privatize: requires compiler and linker support
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 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