Leveraging MPI's One-Sided Communication Interface for Shared-Memory Programming

Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, Rajeev Thakur

The Shared Memory Reality

- Multi- and manycore is ubiquitous
- They offer shared memory that allows:

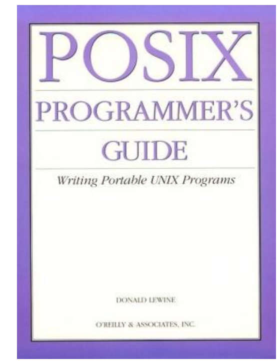
1. Sharing of data structures
   - ✓ Reduce copies/effective memory consumption
   - x NUMA accesses
2. Fast in-memory communication
   - ✓ May be faster than MPI
   - x Performance model is very complex

State of the Art Programming

- MPI offers shared memory optimizations
  - But real zero copy is impossible
- MPI+X to utilize shared memory
  - X={OpenMP, pthreads, UPC …}
- Complex interactions between models
  - Deadlocks possible
  - Race conditions made eas
  - Slowdown due to higher MPI thread level
- Requirements are often simple
  - Switching programming models not necessary?

Why not just use OS tools?

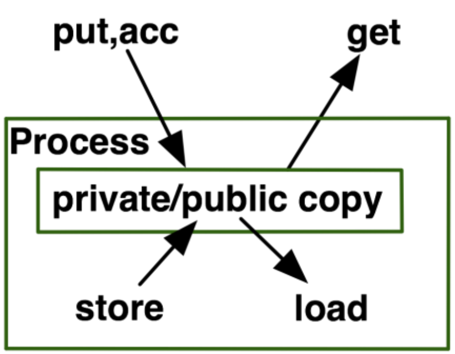
- One may use POSIX shm calls to create shared memory segments?



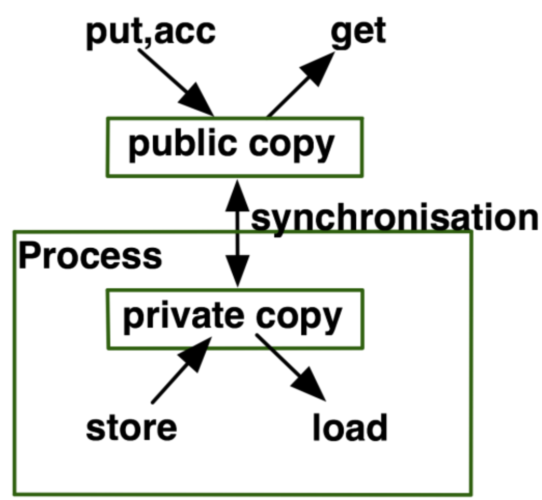
- Several issues:
  1. Allocation is not collective and users would have to deal with NUMA intricacies
  2. Cleanup of shm regions is problematic in the presence of abnormal termination
  3. MPI's interface allows easy support for debuggers and performance tools

MPI-3.0 One Sided Memory Models

- MPI offers two memory models:
  - Unified: public and private window are identical
  - Separate: public and private window are separate
- Type is attached as attribute to window
  - MPI_WIN_MODEL



MPI_UNIFIED



MPI_SEPARATE

Creating a Shared Memory Window

- MPI_WIN_ALLOCATE_SHARED(
  - size - in Bytes (of calling process)
  - disp_unit - addressing offset in Bytes
  - info - specify optimization hints
  - comm - input communicator
  - baseptr - returned pointer
  - win – returned window

)

➢ The creation call is collective

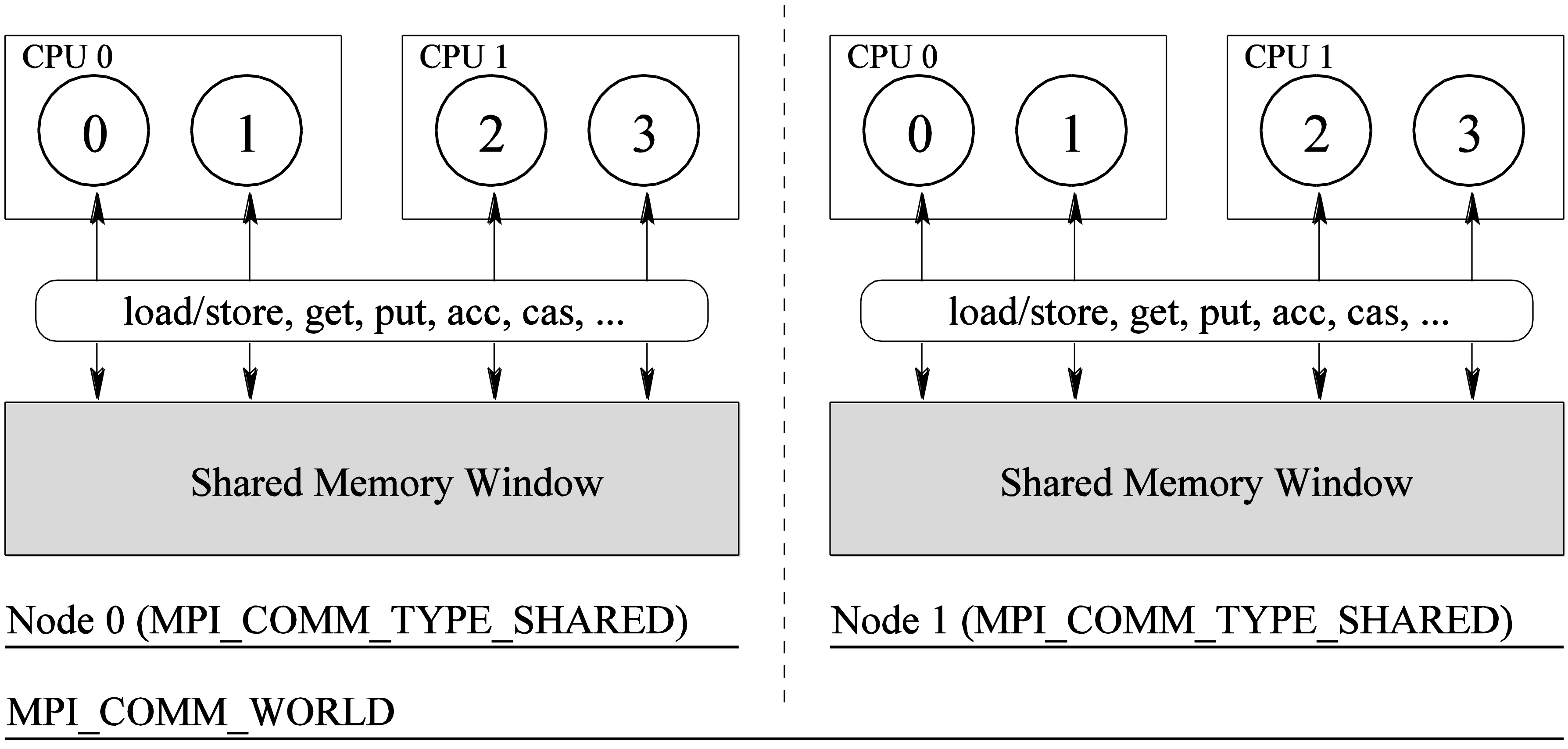
How do I use it?

- All processes in comm must be in shared memory
  - Fulfilling the unified window requirements
- Each process gets a pointer to its segment
  - Does not know other processes' pointer
- Query function:
  - MPI_WIN_SHARED_QUERY(win, rank, size, disp_unit, baseptr)
  - ➢ Query rank's size, disp_unit, and baseptr

Creating shared memory communicators

- Or: "How do I know which processes share memory"?
- MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)
  - split_type = MPI_COMM_TYPE_SHARED
  - Splits communicator into maximum shared memory islands
  - Portable

SCHEMATIC OVERVIEW

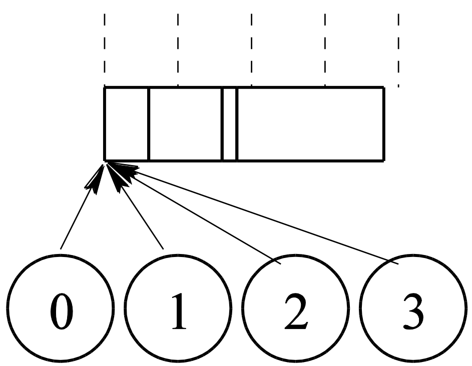


Memory Layout

- "Principle of least surprise" (default)
  - Memory is consecutive across ranks
  - Allows for inter-rank address calculations
  - i.e., rank i's first Byte starts right after rank i-1's last Byte
- "Optimizations allowed"
  - Specify info "alloc_shared_noncontig"
  - May create non-contiguous regions
  - Must use win_shared_query

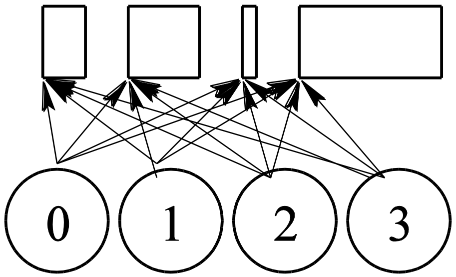
Implementation Options I

- Contiguous (default)
  - **Reduce** total size to rank 0
  - Rank 0 creates shared memory segment
  - **Broadcast** address and key
  - **Exscan** to get local offset
  - O(log P) time and O(P) total storage

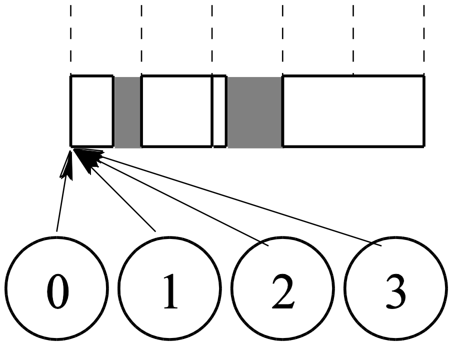


Implementation Options II

- Noncontiguous (specify alloc_shared_noncontig)
- Option 1:
  - Each rank creates his own segment



- Option 2:
  - Rank 0 creates one segment but pads to page boundaries



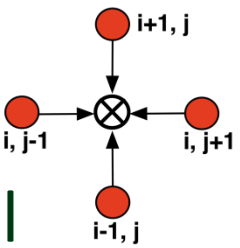
USE CASES

1. Share data structures
   - Use hybrid programming where it is efficient
     - E.g., OpenMP at the loop level
   - Have MPI processes share common memory
     - Retain all MPI features, e.g., collective etc.
2. Improve communication performance
   - Enables direct access to “remote” data
   - No need for halo zones (but they often help!)
   - True zero copy in this sense

FAST SHARED MEMORY COMMUNICATION

- Two fundamental benefits:
  1. Avoid tag matching and MPI stack
  2. Avoid expensive fine-grained synchronization
- Full interface implemented in Open MPI and MPICH2
  - Similar implementation and performance
- Evaluated on 2.2 GHz AMD Opteron
  - Six cores

Five-Point Stencil Example



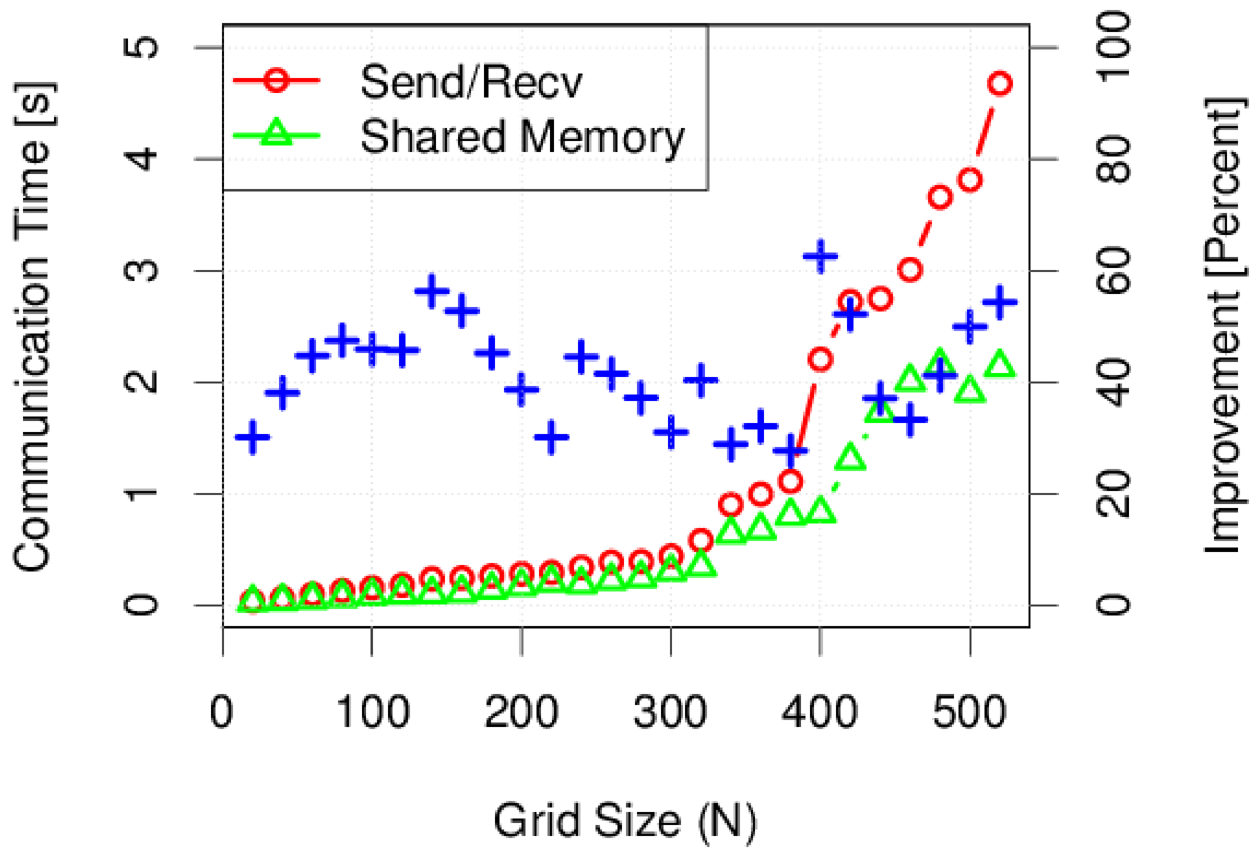
- NxN grid decomposed in 2D
  - Dims_create, cart_create, isend/irecv, waitall

```
MPI_Comm_split_type(comm, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &shmcomm);

MPI_Win_allocate_shared(size*sizeof(double), info, shmcomm, &mem, &win);
MPI_Win_shared_query(win, north, &sz, &northptr);
// ... south, east, west directions

for(iter=0; iter<niters; ++iter) {
  MPI_Win_fence(0, win); // start new access and exposure epoch
  if(north != MPI_PROC_NULL) // the "communication"
    for(int i=0; i<bx; ++i) a2[ind(i+1,0)] = northptr[ind(i+1,by)];
  // ... south, east, west directions
  update_grid(&a1, &a2); // apply operator and swap arrays
}
```

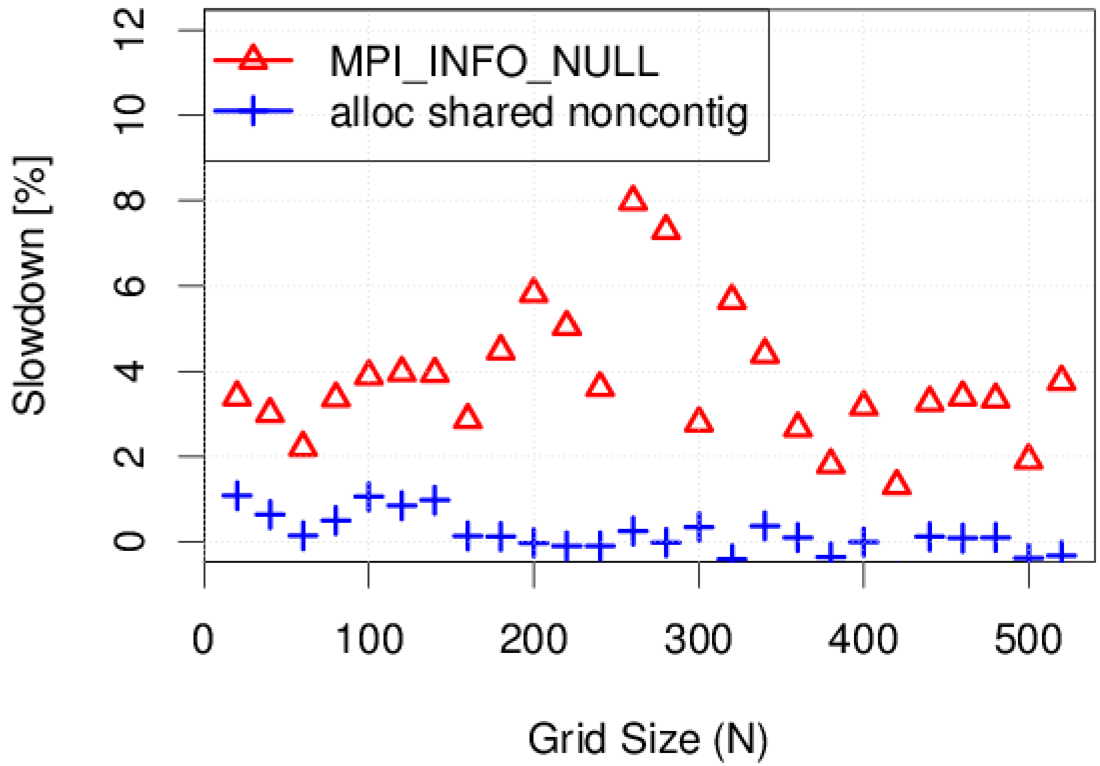
Communication Times



- 30-60% lower communication overhead!

NUMA EFFECTS?

- The whole array is allocated in shared memory



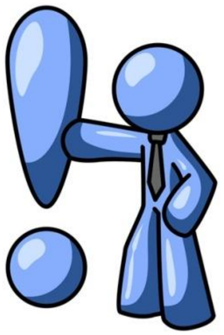
- Significant impact of alloc_shared_noncontig

SUMMARY

- MPI-3.0 offers support for shared memory
  - Ratified last week, standard available online
- MPICH2 as well as Open MPI implement the complete interface
  - Should be in official releases soon
- We demonstrated two use-cases
  - Showed application speedup for a simple code
  - Performance may vary (depends on architecture)

Conclusions & Future Work

- MPI is (more) ready for multicore!
  - Supports coherent shared memory
  - Offers easy-to-use and portable interface
  - Mix&match with other MPI functions
- We plan to evaluate
  - Different use-cases and applications
- The Forum continues discussion
  - Non-coherent shared memory?



Acknowledgments

- The MPI Forum
  - Especially the RMA working group!

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

BLUE WATERS
SUSTAINED PETASCALE COMPUTING

Sandia National Laboratories