MPI for Exascale Systems

(How I Learned to Stop Worrying about Exascale and Love MPI)

Pavan Balaji

Computer Scientist and Group Lead

Argonne National Laboratory
Argonne National Laboratory

About Argonne

- $675M operating budget
- 3,200 employees
- 1,450 scientists and engineers
- 750 Ph.D.s
Major Scientific User Facilities

- Advanced Photon Source
- Argonne Tandem Linear Accelerator System
- Center for Nanoscale Materials
- Argonne Leadership Computing Facility
- Electron Microscopy Center
Let’s talk MPI!
Standardizing Message-Passing Models

- Early vendor systems (Intel’s NX, IBM’s EUI, TMC’s CMMD) were not portable (or very capable)

- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of message-passing issues
  - Lacked vendor support
  - Were not implemented at the most efficient level

- The MPI Forum was a collection of vendors, portability writers and users that wanted to standardize all these efforts
What is MPI?

- **MPI: Message Passing Interface**
  - The MPI Forum organized in 1992 with broad participation by:
    - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
    - Portability library writers: PVM, p4
    - Users: application scientists and library writers
    - MPI-1 finished in 18 months
  - Incorporates the best ideas in a “standard” way
    - Each function takes fixed arguments
    - Each function has fixed semantics
      - Standardizes what the MPI implementation provides and what the application can and cannot expect
      - Each system can implement it differently as long as the semantics match

- **MPI is not...**
  - a language or compiler specification
  - a specific implementation or product
MPI Forum Organizational Structure

- Steering committee
- Organizers
  - Martin Schulz (chair)
- Bunch of working groups
  - One-sided communication (chairs: Bill Gropp, Rajeev Thakur)
  - Hybrid programming (chair: Pavan Balaji)
  - Collective communication (chair: Torsten Hoefler)
  - Communicators, Contexts and Groups (chair: Pavan Balaji)
  - Tools (chair: Kathryn Mohror)
  - Fortran (chair: Rolf Rabenseifner)
  - Fault Tolerance (chairs: Wesley Bland, Aurelien)
MPI-1

- MPI-1 supports the classical message-passing programming model: basic point-to-point communication, collectives, datatypes, C/Fortran bindings, etc.
- MPI-1 was defined (1994) by a broadly based group of parallel computer vendors, computer scientists, and applications developers.
  - 2-year intensive process
- Implementations appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.
- Free, portable implementations exist for clusters and other environments (MPICH, Open MPI)
Following MPI Standards

- MPI-2 was released in 2000
  - Several additional features including MPI + threads, MPI-I/O, remote memory access, dynamic processes, C++/F90 bindings and others

- MPI-2.1 (2008) and MPI-2.2 (2009) were recently released with some corrections to the standard and small features

- MPI-3 (2012) added several new features to MPI

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML

- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of material including tutorials, a FAQ, other MPI pages
Overview of New Features in MPI-3

- **Major new features**
  - Nonblocking collectives
  - Neighborhood collectives
  - Improved one-sided communication interface
  - Tools interface
  - Fortran 2008 bindings

- **Other new features**
  - Matching Probe and Recv for thread-safe probe and receive
  - Noncollective communicator creation function
  - “const” correct C bindings
  - Comm_split_type function
  - Nonblocking Comm_dup
  - Type_create_hindexed_block function

- C++ bindings removed; previous deprecated functions removed
MPI in the Exascale Era

- Under a lot of scrutiny (good!)
  - Lots of myths floating around (bad!)
- Push to get new programming models designed and developed for exascale
- The truth is that MPI today is a new programming model (compared to 2004), and MPI in 2020 will be a new programming model (compared to today)

Strengths of MPI

- Composability
  - Ability to build tools and libraries above and around MPI
  - No “do everything under the sun” attitude
- Continuous evolution
  - The standard incorporates best research ideas
MPI: A Philosophical Perspective

- Debate in the community that MPI is too hard to program
  - We don’t disagree
  - It’s meant to be a low-level portable runtime on top of which higher-level programming models should be developed

- A programming model has to pick a tradeoff between programmability, portability, and performance
  - MPI has chosen to be a high-performance/portable programming model
  - Focus has been on completeness and ability to help real and complex applications meet their computational needs

- **MPI’s goal is not to make simple programs easy to write, but to make complex programs possible to write**
MPI Ecosystem

- Concentrates on minimal required features (not minimal functions!) and aims to build a large ecosystem that provides easier/more-specific interfaces
  - High-level libraries for tasks, global address, math, visualization, I/O
The vast majority of DOE’s production parallel scientific applications today use MPI

- Increasing number use (MPI + OpenMP) hybrid
- Some exploring (MPI + accelerator) hybrid

Today’s largest systems in terms of number of regular cores (excluding GPU cores)

- Sequoia (LLNL) 1,572,864 cores
- Mira (ANL) 786,432 cores
- K computer 705,024 cores
- Jülich BG/Q 393,216 cores
- Blue Waters 386,816 cores
- Titan (ORNL) 299,008 cores

MPI already runs in production on systems with up to 1.6 million cores
Current Situation with Production Applications (2)

- IBM has successfully scaled the LAMMPS application to over 3 million MPI ranks
- Applications are running at scale on LLNL’s Sequoia and achieving 12 to 14 petaflops *sustained* performance
- HACC cosmology code from Argonne (Salman Habib) achieved **14 petaflops** on Sequoia
  - Ran on full Sequoia system using MPI + OpenMP hybrid
  - Used 16 MPI ranks * 4 OpenMP threads on each node, which matches the hardware architecture: 16 cores per node with 4 hardware threads each
  - SC12 Gordon Bell prize finalist
Current Situation with Production Applications (3)

- Cardioid cardiac modeling code (IBM & LLNL) achieved **12 petaflops** on Sequoia
  - Models a beating human heart at near-cellular resolution
  - Ran at scale on full system (96 racks)
  - Used MPI + threads hybrid: 1 MPI rank per node and 64 threads
  - OpenMP was used for thread creation only; all other thread choreography and synchronization used custom code, not OpenMP pragmas
    - [http://nnsa.energy.gov/mediaroom/pressreleases/sequoia112812](http://nnsa.energy.gov/mediaroom/pressreleases/sequoia112812)
    - SC12 Gordon Bell Prize finalist

- And there are other applications running at similar scales...
On the path to Exascale (based on Bill Dally’s talk at ISC 2013)

- **Software Improvements:** 1.7 – 4X
- **Logic Circuit Design:** 3X
- **Device Technology Fabrication Process:** 2.2X
- **Device Technology Fabrication Process:** 11.1 - 26.3X

Top500
Green500
Exaflop
New Features in MPI-3
Nonblocking Collectives

- Nonblocking versions of all collective communication functions have been added
  - MPI_Ibcast, MPI_Ireduce, MPI_Iallreduce, etc.
  - There is even a nonblocking barrier, MPI_Ibarrier

- They return an MPI_Request object, similar to nonblocking point-to-point operations

- The user must call MPI_Test/MPI_Wait or their variants to complete the operation

- Multiple nonblocking collectives may be outstanding, but they must be called in the same order on all processes
Nonblocking Collectives Overlap

- Software pipelining
  - More complex parameters
  - Progression issues
  - Not scale-invariant

Hoefler: Leveraging Non-blocking Collective Communication in High-performance Applications
Dynamic Sparse Data Exchange (DSDE)

- **Main Problem: metadata**
  - Determine who wants to send how much data to me
  (I must post receive and reserve memory)
  
  **OR:**
  - Use MPI semantics:
    - Unknown sender
      - MPI_ANY_SOURCE
    - Unknown message size
      - MPI_PROBE
    - Reduces problem to counting the number of neighbors
    - Allow faster implementation!

T. Hoefler et al.: Scalable Communication Protocols for Dynamic Sparse Data Exchange
**MPI_Ibarrier (NBX)**

- Complexity - census (barrier): \( \Theta(\log(P)) \)
  - Combines metadata with actual transmission
  - Point-to-point synchronization
  - Continue receiving until barrier completes
  - Processes start coll. synch. (barrier) when p2p phase ended
    - barrier = distributed marker!
  - Better than PEX, PCX, RSX!
Parallel Breadth First Search

- On a clustered Erdős-Rényi graph, weak scaling
  - 6.75 million edges per node (filled 1 GiB)

- HW barrier support is significant at large scale!
Neighborhood Collectives

- New functions MPI_Neighbor_allgather, MPI_Neighbor_alltoall, and their variants define collective operations among a process and its neighbors.
- Neighbors are defined by an MPI Cartesian or graph virtual process topology that must be previously set.
- These functions are useful, for example, in stencil computations that require nearest-neighbor exchanges.
- They also represent sparse all-to-many communication concisely, which is essential when running on many thousands of processes.
  - Do not require passing long vector arguments as in MPI_Alltoallv.
Cartesian Neighborhood Collectives

- Buffer ordering example:
Improved RMA Interface

- Substantial extensions to the MPI-2 RMA interface

- New window creation routines:
  - MPI_Win_allocate: MPI allocates the memory associated with the window (instead of the user passing allocated memory)
  - MPI_Win_create_dynamic: Creates a window without memory attached. User can dynamically attach and detach memory to/from the window by calling MPI_Win_attach and MPI_Win_detach
  - MPI_Win_allocate_shared: Creates a window of shared memory (within a node) that can be accessed simultaneously by direct load/store accesses as well as RMA ops

- New atomic read-modify-write operations
  - MPI_Get_accumulate
  - MPI_Fetch_and_op (simplified version of Get_accumulate)
  - MPI_Compare_and_swap
One-sided Communication

- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory
Use Case: Distributed Shared Arrays

- **Quantum Monte Carlo: Ensemble data**
  - Represents initial quantum state
  - Spline representation, cubic basis functions
  - Large (100+ GB), read-only table of coeff.
  - Accesses are random

- **Coupled cluster simulations**
  - Evolving quantum state of the system
  - Very large, tables of coefficients
  - $Table_t$ read-only, $Table_{t+1}$ accumulate-only
  - Accesses are non-local/overlapping

- **Global Arrays PGAS programming model**
  - Can be supported with passive mode RMA [Dinan et al., IPDPS’12]
Case-study: NWChem over MPI-3

CCSD Performance

Iteration Number

Wallclock Time (seconds)

CCSD(T) Performance

Average Iteration Time

Wallclock Time (seconds)

Courtesy Jeff Hammond, Argonne National Laboratory
Hybrid Programming with Shared Memory

- MPI-3 allows different processes to allocate shared memory through MPI
  - MPI_Win_allocate_shared
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads
Creating Shared Memory Regions in MPI

- **MPI_COMM_WORLD**
- `MPI_Comm_split_type` (COMM_TYPE_SHARED)
- **Shared memory communicator**
  - `MPI_Win_allocate_shared`
  - **Shared memory window**
- **Shared memory communicator**
  - **Shared memory window**
- **Shared memory communicator**
  - **Shared memory window**
Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory
- E.g., $x[100] = 10$

All of the existing RMA functions can also be used on such memory for more advanced semantics such as atomic operations

Can be very useful when processes want to use threads only to get access to all of the memory on the node
- You can create a shared memory window and put your shared data
Case Study: Genome Assembly

- Largest genome assembly to date: 2.3TB dataset performed with MPI-3 shared memory capability
  - First terascale genome assembly
- Very simple optimization: place all of the node dataset in shared memory and access as read-only data
- Could not use threads because all MPI calls face lock overheads
Tools Interface (1)

- An extensive interface to allow tools (performance analyzers, etc.) to portably extract information about MPI processes

- Enables the setting of various control variables within an MPI implementation, such as algorithmic cutoff parameters
  - e.g., eager v/s rendezvous thresholds
  - Switching between different algorithms for a collective communication operation

- Provides portable access to performance variables that can provide insight into internal performance information of the MPI implementation
  - e.g., length of unexpected message queue

- Note that each implementation defines its own performance and control variables; MPI does not define them
Tools Interface (2)

- Standardized model for debuggers to query for MPI library information
- Totalview, DDT, etc., can peek inside the MPI library and provide information on queues and other operations
- This has been supported by almost every MPI implementation for many years; it was just blessed by the Forum now
- (This is a companion standard, not included in MPI-3)
Fortran 2008 Bindings

- An additional set of bindings for the latest Fortran specification
- Supports full and better quality argument checking with individual handles
- Support for choice arguments, similar to (void *) in C
- Enables passing array subsections to nonblocking functions
- First parallel programming library to actually be Fortran compliant 😊
  - Well, not exactly; requires some Fortran TS 29113 extensions
Support for Very Large Data

- Ability to communicate very large data (more than 2GB)
- Model is a little round-about for backward compatibility reasons, but not too hard
  - If < 2GB, use MPI communication directly
  - If >= 2GB, create derived datatype that represents large data
- Added a few extra functions that would typically be used by high-level libraries to query for such information
## Status of MPI-3 Implementations

<table>
<thead>
<tr>
<th>Feature</th>
<th>MPICH</th>
<th>MVAPICH</th>
<th>Cray</th>
<th>Tianhe</th>
<th>Intel</th>
<th>IBM PE</th>
<th>IBM Platform</th>
<th>Open MPI</th>
<th>SGI</th>
<th>Fujitsu</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NB collectives</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Q1 ‘14</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Neighborhood collectives</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Q1 ‘14</td>
<td>Q3 ‘14</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>RMA</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Q1 ‘14</td>
<td>Q3 ‘14</td>
<td>Q1</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Shared memory</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Q1 ‘14</td>
<td>Q2 ‘15</td>
<td>Q1</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Tools Interface</td>
<td>✔</td>
<td></td>
<td>Q1 ‘14</td>
<td>Q2 ‘14</td>
<td>Q1 ‘14</td>
<td>Q1 ‘14</td>
<td>Q2 ‘15</td>
<td>✔</td>
<td>Q1</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Non-collective comm. create</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Q1 ‘14</td>
<td>Q2 ‘15</td>
<td>Q4</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>F08 Bindings (MPI-3 + errata)</td>
<td>(Q1 ‘14)</td>
<td>(Q2 ‘14)</td>
<td>(Q2 ‘14)</td>
<td>(Q1 ‘14)</td>
<td>(Q1 ‘14)</td>
<td>(Q3 ‘14)</td>
<td>(✔️)</td>
<td>(Q1 ‘14)</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>New Datatypes</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Q1 ‘14</td>
<td>Q2 ‘15</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Large Counts</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Q1 ‘14</td>
<td>Q2 ‘15</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Matched Probe</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Q1 ‘14</td>
<td>Q3 ‘14</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Release dates are estimates and are subject to change at any time. Empty cells indicate no publicly announced plan to implement/support that feature.
Beyond MPI-3
MPI Research Activities beyond MPI-3

- Improved interoperability with Threads and Shared memory
  - Some of this is being considered for MPI-4

- Fault Tolerance
  - Tolerating faults in processes (fatal crashes), memory, network
  - Some of it is being considered for MPI-4

- Tasking Models
  - Message-driven execution, irregular computations
  - Some of it (Active Messages) is being considered for MPI-4

- Interacting with Heterogeneous Memory
  - Non-DRAM memories (NVRAM, scratchpad, on-chip memory)
  - Definitely beyond MPI-4
MPI+Threads Hybrid Programming

- One of the most successful models in used today
- Hybrid programming vs. a single unified programming model
  - The number of models we program to should not be too large, but a small collection of standardized programming models which interoperate with each other is not a bad thing
  - MPI+OpenMP has demonstrated this successfully

Why is this: better than this?
Four levels of MPI Thread Safety

- **MPI_THREAD_SINGLE**
  - MPI only, no threads

- **MPI_THREAD_FUNNELED**
  - *Outside* OpenMP parallel region, or OpenMP **master** region

- **MPI_THREAD_SERIALIZED**
  - *Outside* OpenMP parallel region, or OpenMP **single** region, or **critical** region

- **MPI_THREAD_MULTIPLE**
  - Any thread is allowed to make MPI calls at any time

```c
#pragma omp parallel for
for (i = 0; i < N; i++) {
    uu[i] = (u[i] + u[i - 1] + u[i + 1])/5.0;
}

MPI_Function ();
```

```c
#pragma omp parallel
{
    /* user computation */
    #pragma omp single
    MPI_Function ();
}
```

```c
#pragma omp parallel
{
    /* user computation */
    #pragma omp critical
    MPI_Function ();
}
```
Problem: Idle Resources during MPI Calls

- Threads are only active in the computation phase
- Threads are **IDLE** during MPI calls

```c
#pragma omp parallel for
for (i = 0; i < N; i++) { 
    uu[i] = (u[i] + u[i - 1] + u[i + 1])/5.0;
}
MPI_Function ( );
```

(a) Funneled mode

```
#pragma omp parallel
{ 
    /* user computation */
    #pragma omp single
    MPI_Function ( );
}
```

(b) Serialized mode
Our Approach: Sharing Idle Threads with Application inside MPI

```c
#pragma omp parallel
{
    /* user computation */

    #pragma omp single
    MPI_Function () {
        #pragma omp parallel
        {
            /* MPI internal task */
        }
    }
}
```

(b) Serialized mode

Master

MPI CALL
Challenges

- Some parallel algorithms are not efficient with insufficient threads, need tradeoff, but the number of available threads is **UNKNOWN**!

- Nested parallelism
  - Simply creates new Pthreads
  - Offloads thread scheduling to OS, caused threads **OVERRUNNING** issue

```c
#pragma omp parallel
{
    /* user computation */
    #pragma omp single
    MPI Function()
    { ...
    }
}

#pragma omp parallel
{
    #pragma omp single
    { ...
    }
    #pragma omp parallel
    { ...
    }
}
```

(a) Unknown number of IDLE threads          (b) Threads overrunning
Derived Data Type Packing Processing

- **MPI_Pack / MPI_Unpack**
- **Communication using Derived Data Type**
  - Transfer *non-contiguous* data
  - Pack / unpack data internally

```c
#pragma omp parallel for
for (i=0; i<count; i++){
    dest[i] = src[i * stride];
}
```

<table>
<thead>
<tr>
<th>blocklength</th>
<th>count</th>
<th>stride</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
</tbody>
</table>

![Graph showing speedup vs. number of threads]

**Hybrid MPI+OpenMP NAS Parallel MG benchmark**
Parallel shared memory communication

- Get **as many available cells as we can**
- Parallelizing large data movement

(a) Sequential Pipelining

(b) Parallel pipelining
Intra-node Large Message Communication

- OSU MPI micro-benchmark

Caused by too small Eager/Rendezvous communication threshold on Xeon Phi. Not by MT-MPI!

Poor pipelining but worse parallelism

Similar results of **Latency** and **Message rate**
Parallel InfiniBand Communication

- **Structures**
  - IB context
  - Protection Domain
  - Queue Pair (critical)
    - 1 QP per connection
  - Completion Queue (critical)
    - Shared by 1 or more QPs

- **RDMA communication**
  - Post RDMA operation to QP
  - Poll completion from CQ

- Internally supports Multi-threading
- OpenMP contention issue
One-sided Graph500 benchmark

- Every process issues many MPI Accumulate operations to the other processes in every breadth first search iteration.
- Scale $2^{22}$, 16 edge factor, 64 MPI processes
MPI_THREAD_MULTIPLE optimizations

- **Global**
  - Use a single global mutex, held from function enter to exit
  - Existing implementation

- **Brief Global**
  - Use a single global mutex, but reduce the size of the critical section as much as possible

- **Per-Object**
  - Use one mutex per data object: lock *data* not *code sections*

- **Lock-Free**
  - Use no mutexes
  - Use lock-free algorithms
Several Optimizations Later...

- Reduction of lock granularities
- Thread-local pools to reduce sharing
- Per-object locks
- Some atomic operations (reference counts)
- But the performance scaling was still suboptimal
From Pure-MPI to Multithreaded-MPI: Multi-Node Communication Throughput

- Example: circular communication pattern
- Onesided communications
- No intra-node communication
- 17 nodes x 16 cores/node
- Global mutex
- Throughput drops severely when moving to threads

Problems:
- Safer algorithms?
- Lock contention?
# Contention in a Multithreaded MPI Model

**Multithreaded MPI**
- Threads can make MPI calls concurrently
- Thread-safety is necessary

```c
MPI_Init_thread(..., MPI_THREAD_MULTIPLE, ...);

#pragma omp parallel
{
    /* Do Work */
    MPI_Put();
    /* Do Work */
}
```

- Thread-safety can be ensured by:
  - **Critical Sections** (Locks)
    - Possible Contention!
  - Using **Lock-Free** algorithms
    - Non trivial!
  - Still does memory barriers

![Diagram showing contention in a multithreaded MPI model.](image)
Hidden Evil: Lock Monopolization (Starvation)

- Implementing critical sections with spin-locks or mutexes
- Watch out: no fairness guarantee!

Starvation measurement with 16 processes and 16 threads/nodes

```
int waiting_threads = 0;
int last_holder;

acquire_lock(L)
{
  bool lock_acquired = false;
  try_lock(L, lock_acquired)
  if ((lock_acquired) &&
      (my_thread_id == last_holder) &&
      (waiting_threads > 0))
    STARVATION_CASE;
  else if (!lock_acquired)
  {
    atomic_incr(waiting_threads);
    lock(L);
    atomic_decr(waiting_threads);
  }
  last_holder = my_thread_id;
  return;
}
```
How to fix Lock Monopolization?

- Use locks that ensure **fairness**
- Example: Ticket Spin-Lock
- Basics:
  - Get my ticket and Wait my turn
  - Ensures **FIFO** acquisition

**2D Stencil, Hallo=2MB/direction, Message size=1KB, 16Threads/Node**

Simplified Execution flow of a Thread-safe MPI implementation with critical sections
Priority Locking Scheme

- 3 basic locks:
  - One for mutual exclusion in each priority level
  - Another for high priority threads to block lower ones

- Watch out: do not forget fairness in the same priority level
  - Use exclusively FIFO locks (Ticket)

2D Stencil, Hallo=2MB/direction, Message size=1KB, 16Threads/Node

![Graph showing speed-up over Mutex for different numbers of nodes](image-url)
Does Fixing Lock Contention Solve the Problem?

- Spin-lock based critical sections
- Contention metric: **Wasted Polls**
- Test scenarios:
  - Micro-benchmarks
  - HPC applications

```c
#pragma omp parallel
{
    for(i=0; i< NITER; i++)
    {
        MPI_Put();
        /*Delay X us*/
        Delay(X)
    }
}
```

“When you have eliminated the impossible, whatever remains, however improbable, must be the truth.” – Sherlock Holmes, Sign of Four, Sir Arthur Conan Doyle
Hybrid MPI+OpenMP (or other threading models)

- Thread execution model exposed to applications is too simplistic
- OpenMP threads can be pthreads (i.e., can execute concurrently) or user-level threads such as qthreads (i.e., might or might not execute concurrently)
  - Not exposed to users/MPI library
- What does this mean to MPI?
  - MPI runtime never knows when two threads can execute concurrently and when they cannot
  - Always has to perform locks and memory consistency calls (memory barriers) even when switching between user-level threads
Argobots: Integrated Computation and Data Movement with Lightweight Work Units

**Execution Model**

- **Execution stream**: a thread executed on a hardware processing element
- **Work units**: a user level thread or a tasklet with a function pointer

**Memory Model**

- **Memory Domains**: A memory consistency call on a big domain also impacts all internal domains
- **Synchronization**: explicit & implicit memory consistency calls
- **Network**: PUT/GET, atomic ops
Argobots Ongoing Works: Fine-grained Context-aware Thread Library

- **Two Level of Threads**
  - execution stream: a normal thread
  - work unit: a user level thread or a tasklet with a function pointer

- **Avoid Unnecessary lock/unlock**
  - Case 1: switch the execution to another work unit in the same execution stream **without unlock**
  - Case 2: switch to another execution stream, call unlock

- **Scheduling Work Units in Batch Order**
  - work units in the same execution stream will be batch executed
MPI + Shared memory with Partitioned Address Space

- Partitioned virtual address space allows us to create different virtual address images instead a single OS virtual address space
- Each process has a virtual address space, but they can look into each others virtual address space
- All memory is semi-private

Fig.2: VM management

*Courtesy Atsushi Hori, University of Tokyo*
MPI + Shared memory with User-level processes

- Extending the PVAS concept to “user-level processes”
- Multiple “MPI processes” inside a single OS process
- Each MPI process has its own address space, but the MPI implementation can look into other process’ memory and cooperatively yield to them

Collaborative work with Atsushi Hori, University of Tokyo
Fault Tolerance: What failures do we expect?

- **CPU (or full system)**
  - Total Failure

- **Memory**
  - ECC Errors (corrected/uncorrected)
  - Silent Errors

- **Network**
  - Dropped packets
  - Lost routes
  - NIC Failure

- **GPU/Accelerator**
  - Memory Errors
  - Total Failure

- **Disk**
  - R/W Errors
  - Total Failure
CPU (Total System) Failures

- Generally will result in a process failure from the perspective of other (off-node) processes
- Need to recover/repair lots of parts of the system
  - Communication library (MPI)
  - Computational capacity (if necessary)
  - Data
    - C/R, ABFT, Natural Fault Tolerance, etc.
- MPI-3 has the theoretical ability to deal with faults, but the user has to do a bunch of bookkeeping to make that happen
  - New communicator has to be created
  - All requests have to be kept track of and migrated to the new communicator
  - Need to watch out for failure messages from other processes
MPIXFT: MPI-3 based library for FT bookkeeping

- Lightweight virtualization infrastructure
  - Gives users virtual communicators and requests and internally manages the real ones

- Automatically repairs MPI communicators as failures occur
  - Handles running in n-1 model

- Virtualizes MPI Communicator
  - User gets an MPIXFT wrapper communicator
  - On failure, the underlying MPI communicator is replaced with a new, working communicator
MPIXFT Design

- Possible because of new MPI-3 capabilities
  - Non-blocking equivalents for (almost) everything
  - MPI_COMM_CREATE_GROUP
MIPXFT Results

- **MCCK Mini-app**
  - Domain decomposition communication kernel
  - Overhead within standard deviation

---

**Halo Exchange**

- Halo Exchange (1D, 2D, 3D)
  - Up to 6 outstanding requests at a time
  - Very low overhead
User Level Failure Mitigation

- Proposed change to MPI Standard for MPI-4
- Repair MPI after process failure
  - Enable more custom recovery than MPIXFT
- Does not pick a particular recovery technique as better or worse than others
- Introduce minimal changes to MPI
- Works around performance problems with MPIXFT
- Treat process failure as fail-stop failures
  - Transient failures are masked as fail-stop
- Ability to notify remaining processes on errors
Recovery with only notification
Master/Worker Example

- Post work to multiple processes
- MPI_Recv returns error due to failure
  - MPI_ERR_PROC_FAILED if named
  - MPI_ERR_PROC_FAILED_PENDING if wildcard
- Master discovers which process has failed with ACK/GET_ACKED
- Master reassigns work to worker 2
Failure Propagation

- When necessary, manual propagation is available.
  - **MPI_Comm_revoke**(MPI_Comm comm)
    - Interrupts all non-local MPI calls on all processes in comm.
    - Once revoked, all non-local MPI calls on all processes in comm will return MPI_ERR_REVOKED.
      - Exceptions are MPI_COMM_SHRINK and MPI_COMM_AGREE (later)
    - Necessary for deadlock prevention

- Often unnecessary
  - Let the application discover the error as it impacts correct completion of an operation.
GVR (Global View Resilience) for Memory Errors

- Multi-versioned, distributed memory
  - Application commits “versions” which are stored by a backend
  - Versions are coordinated across entire system
- Different from C/R
  - Don’t roll back full application stack, just the specific data.
Memory Fault Tolerance

- Memory stashing
- Multi-versioned memory
- Implementation-defined block-level retrieval across versions

- Silent-data corruption handled through application hooks (only works if applications can detect “silent errors”)

- Nondeterministic methods can naturally deal with them because of some common hardware properties
  - Exponents are less error prone than mantissa, because of hardware logic models (mantissa uses simpler hardware)
  - Instruction caches are more reliable than data caches
**MPI Memory Resilience (Planned Work)**

- Integrate memory stashing within the MPI stack
  - Data can be replicated across different kinds of memories (or storage)
  - On error, repair data from backup memory (or disk)

---

**Traditional**

```
MPI_PUT
```

**Replicated**

```
MPI_PUT
```

---

**Traditional**

```
MPI_PUT
```

**Replicated**

```
MPI_PUT
```
VOCL: Transparent Remote GPU Computing

- Transparent utilization of remote GPUs
- Efficient GPU resource management:
  - Migration (GPU / server)
  - Power Management: pVOCL
VOCL-FT (Fault Tolerant Virtual OpenCL)

Synchronous Detection Model

User App.
- bufferWrite
- launchKernel
- bufferRead
- sync

VOCL FT Functionality
- ECC Query
- Checkpointing

Asynchronous Detection Model

User App.
- bufferWrite
- detecting thread
- launchKernel
- bufferRead
- sync

VOCL FT Thread
- ECC Query

Double- (uncorrected) and single-bit (corrected) error counters may be queried in both models.

Minimum overhead, but double-bit errors will trash whole executions.
VOCL-FT: Single and Double Bit Error Detection Overhead

Matrix Transpose

- Execution time (ms)
- #threads: 512*512, 1024*1024, 2048*2048, 4096*4096
- VOCL, Async, Sync

Smith-Waterman

- Execution time (ms)
- Query size: 1K, 2K, 3K, 4K, 5K, 6K
- VOCL, Async, Sync

Nbody

- Execution time (ms)
- #of bodies: 15360, 23040, 30720, 38400
- VOCL, Async, Sync

DGEMM

- Execution time (ms)
- Matrix block: 512*512, 1024*1024, 2048*2048, 4096*4096
- VOCL, Async, Sync
Data-Intensive Applications

- Graph Algorithm in Social Network Analysis

- DNA Sequence Assembly

- Common Characteristics
  - Organized around sparse structures
  - Communication-to-computation ratio is high
  - Irregular communication pattern
Tasking Models

- Several tasking models built on top of MPI (ADLB, Scioto)
- Provide a dynamic task model, potentially with dependencies, for applications to use
  - MPI processes are static; application view of computation/data is dynamic
- Need to be taken with a grain of salt
  - Asynchronous execution not always great: can screw up cache locality
  - Synchronization does not necessarily mean idleness
  - Idleness does not necessarily mean worse performance or energy usage
- Big problem with these and other tasking models is MPI interoperability
  - Libraries built on top of MPI are better in this regard, but still not perfect, because there is no support for it from the runtime system
  - E.g., if I spawn off tasks to execute on a remote process, while the remote process is waiting in an MPI_BARRIER, the tasking library does not have process control to execute the tasks
**Message Passing Models**

- **Current MPI is not well-suitable to data-intensive applications**

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (data)</td>
<td>Receive (data)</td>
</tr>
<tr>
<td>Receive (data)</td>
<td>Send (data)</td>
</tr>
</tbody>
</table>

  - two-sided communication (explicit sends and receives)

- **Active Messages**
  - Sender explicitly sends message
  - Upon message’s arrival, message handler is triggered, receiver is not explicitly involved
  - User-defined operations on remote process

  ![Diagram showing active messages](image)

  - origin
  - target
  - messages
  - messages handler
  - reply
  - reply handler
Generalized and MPI-Interoperable AM

**MPI-AM**: an MPI-interoperable framework that can dynamically manage data movement and user-defined remote computation.

- **Streaming AMs**
  - define “segment”—minimum number of elements for AM execution
  - achieve pipeline effect and reduce temporary buffer requirement

- **Explicit and implicit buffer management**
  - **system buffers**: eager protocol, not always enough
  - **user buffers**: rendezvous protocol, guarantee correct execution

- **Correctness semantics**
  - **Memory consistency**
    - MPI runtime must ensure consistency of window
  - **Three different type of ordering**
    - **Concurrency**: by default, MPI runtime behaves “as if” AMs are executed in sequential order. User can release concurrency by setting MPI assert.

---

**ICPADS 2013**: X. Zhao, P. Balaji, W. Gropp, R. Thakur, “MPI-Interoperable and Generalized Active Messages”, in proceedings of ICPADS’ 13

---

**Pavan Balaji, Argonne National Laboratory**

**Mardi Gras Conference (02/28/2014)**
Asynchronous and MPI-Interoperable AM

- Supporting asynchronous AMs internally from MPI library
  - Inter-node messages: **spawn a thread in network module**
    - Block waiting for AM
    - Separate sockets for AM and other MPI messages
  - Intra-node messages: **“origin computation”**
    - Processes on the same node allocate window on a shared-memory region
    - Origin process directly fetches data from target process’s memory, completes computation locally and writes data back to target process’s memory

Graph500 results, strong scaling (gathered on Fusion cluster at ANL, 320 nodes, 2560 cores, QDR InfiniBand)

Design of asynchronous AMs

Tasking Models based on MPI AM

- Arbitrary graphs can be created where tasks have dependencies on computation and communication
- Spawn off an AM when some collection of MPI requests are complete
  - Each MPI request can correspond to communication operations or other AMs
Data Movement for Heterogeneous Memory

Research areas:

- Programming constructs
- Performance and power/energy
- Transparency through introspection tools
- Evaluation through an integrated test suite and various DOE applications

Relevance
miniMD default dataset - 6 sec execution in 1 core - 7G data accesses
- 54Gb of data accesses

DMEM: exploring efficient techniques for data movement on heterogeneous memory architectures

Hierarchical Memory View

Heterogeneous Memory as First-class Citizens
DMEM Inside the MPICH MPI implementation

**MPI-ACC**: an integrated and extensible approach to data movement in accelerator-based systems

- Enables GPU pointers in MPI calls
  - (CUDA & OpenCL); generic support for heterogeneous memory subsystems is coming
- Coding productivity + performance
- Generic approach, feature independent (UVA not needed)
- Datatype and communicator attrs
  - Pointer location
  - Streams / queues

**Evaluating Epidemiology Simulation with MPI-ACC**

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>MPI + CUDA</th>
<th>MPI-ACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-D Copy (Packing)</td>
<td>0</td>
<td>0.003</td>
</tr>
<tr>
<td>GPU Receive Buffer Init</td>
<td>0</td>
<td>0.024</td>
</tr>
<tr>
<td>H-D Copy</td>
<td>0.382</td>
<td>0</td>
</tr>
<tr>
<td>H-H Copy (Packing)</td>
<td>2.570</td>
<td>0</td>
</tr>
<tr>
<td>CPU Receive Buffer Init</td>
<td>2.627</td>
<td>0</td>
</tr>
</tbody>
</table>


AM Aji, P Balaji, JS Dinan, W Feng, and RS Thakur. “Synchronization and ordering semantics in hybrid MPI+GPU programming”. In AsHES 2013.


F Ji, AM Aji, JS Dinan, DT Buntinas, P Balaji, RS Thakur, W Feng, and X Ma. DMA-assisted, intranode communication in GPU accelerated systems. In HPCC 2012.

Understanding the Relevance of Heterogeneous Memory

- Per-memory object data access pattern analysis
  - Statically allocated
  - Dynamically allocated

Strategy:
- HW emulator
  - Instruction execution
  - Per-object differentiation
- Cache simulator
- Memory simulator

Goals:
- To estimate the number of stall cycles per object
- To propose the most suitable memory for them
Access Pattern Overview (w/o Cache Effects) - MiniAPP

MiniMD

CIAN Coupling

MCCK

Nekbone

Pavan Balaji, Argonne National Laboratory

Mardi Gras Conference (02/28/2014)
Take Away

- MPI has a lot to offer for Exascale systems
  - MPI-3 and MPI-4 incorporate some of the research ideas
  - MPI implementations moving ahead with newer ideas for Exascale
  - Several optimizations inside implementations, and new functionality

- The work is not done, still a long way to go
  - But a start-from-scratch approach is neither practical nor necessary
  - Invest in orthogonal technologies that work with MPI (MPI+X)

- I don’t know what tomorrow’s scientific computing language will look like, but I know it will be called Fortran

- I don’t know what tomorrow’s social networking site will look like, but I know it will be called Facebook

- I don’t know what tomorrow’s parallel programming model will look like, but I know it will be called MPI (+X)