MPI from the Ground Up: From Operations to Implementations Part I

Derek Schafer (derek-schafer@utc.edu)
Tony Skjellum (Tony-Skjellum@utc.edu)

University of Tennessee at Chattanooga
October 1st, 2021

Overview

- Aspects of parallel programming
- Goals & realities of message passing
- Four deep topic areas:
  1. Non-blocking sending and receiving
  2. Completion
  3. Persistence
  4. Collectives
- One more bonus abstract focus area:
  5. Communicators & Groups
- Each area will focus on concepts then implementation details

BSP – Bulk Synchronous Parallel

- Processes ‘own’ their respective data
- Programs execute by successive iterations of
  - Local computation
  - Synchronization, data exchange, data reorganization, reductions
- Simplest programs: all messages/operations known in advance
- Adaptive programs: messages generated are data dependent
- “Data parallel” programs are BSP [e.g., BMR matrix multiplication]

Performance, Productivity, Predictability and Portability

- Parallel programs want all of these
- There are always fundamental and practical trade-offs
- Performance-portability is an important term for:
  - Scalable applications that can run on different platforms
  - Have reasonable/affordable re-tuning requirements when ported

Overheads of Parallel Programs

- Communication
- Synchronization
- Indexing
- Load Imbalance

Goals of a message passing system

- Overall goal: Implement message passing between peer processes
  - Communicating sequential processes (CSP)
  - Same program text that differs only on a value (process rank)
  - Generalizations thereof (MPMD, multithreaded processes)
- Move data while computing
- Move data proactively without extra prompts from the user application
- Reliability
- Scalability
- Fast
- FIFO, pair-wise ordering between processes

Sample Program Target

Notes:

- C++ Syntax
- Return codes are ignored
- Not all communication modes utilized
- No collectives demonstrated
- Output order will be "random"

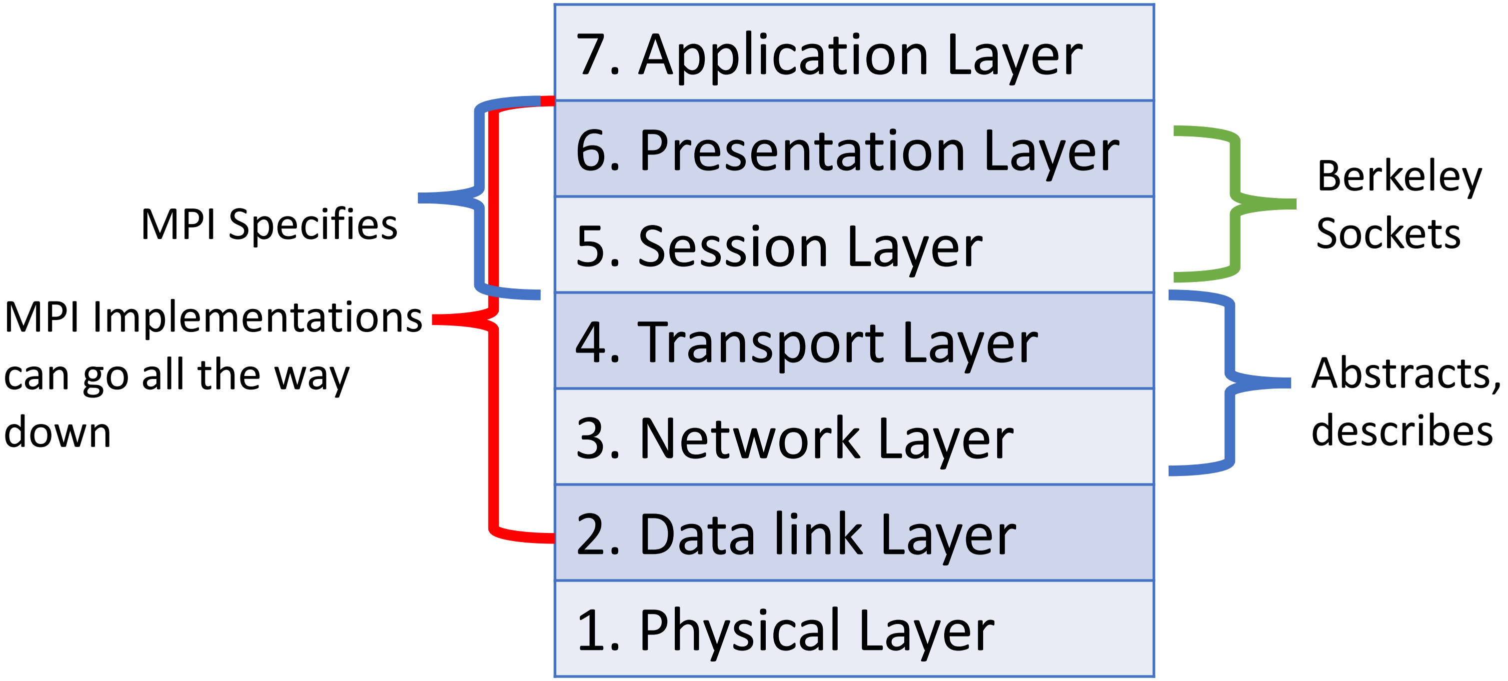
```
#include "mpi.h"
#include <iostream>

int main(int argc, char ** argv)
{
    // Specific to world model. Could be replaced with
    // Sessions model with no changes in rest of code
    MPI_Init(&argc, &argv);
    MPI_Comm our_comm = MPI_COMM_WORLD;
    // -------------------------------------------

    int process_rank = -1;
    MPI_Comm_rank(our_comm, &process_rank);

    int important_buffer = -1;
    MPI_Request my_request = MPI_REQUEST_NULL;

    if(0 == (process_rank % 2))
    {
        important_buffer = 1337;
        MPI_Isend(&important_buffer, 1, MPI_INT, process_rank+1, 0, our_comm, &my_request);
    }
    else
    {
        MPI_Irecv(&important_buffer, 1, MPI_INT, process_rank-1, 0, our_comm, &my_request);
    }

    MPI_Wait(&my_request, MPI_STATUS_IGNORE);

    std::cout << "My process rank: " << process_rank << " got: " << important_buffer << std::endl;

    // Also specific to world model.
    MPI_Finalize();
    //-------------------------------------------
    return 0;
}
```

Realities

- Network/middleware may not move (or be able to) data proactively
- Multiple networks or mechanisms to move data
- Multiple memory regions (and requirements) on system
- Finite amount of buffering in a system
- Sends may be started (and arrive) before receives are posted
- Matching on the receive-side is not strictly FIFO (wildcards)
- Non-contiguous transfers are common
- Faults happen

OSI Model & MPI

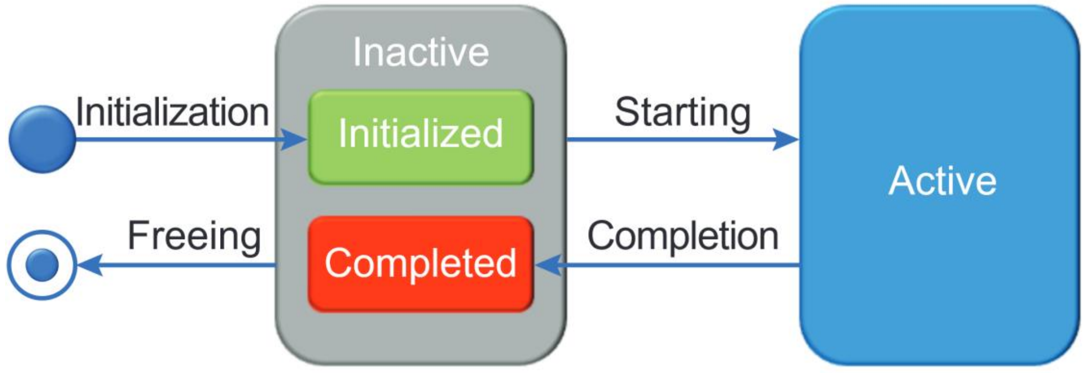
- MPI Standard focuses on data movement and synchronization, not on protocol
- MPI Implementations can take on several layers depending on target architecture
- Kernel/user level barriers can vary too



Semantic Terms

- Initialization
- Starting / Initiation
- Completing
- Freeing

- Operations vs Procedures



From the MPI 4.0 Standard, Figure 2.3

MPI Prefixes

- MPI uses several variations for its functions
  - Qualifiers: I, S, R, B, P
  - Collective suffixes: _V, _W
- MPI also has other naming conventions:
  - _c = big count
  - _x = old name for big count
  - _init = initializing function [persistent]
- MPIX_ - experimental, prestandard
- MPI_T_ - tools (not used by user apps most of the time)
- PMPI_ - profiling interface (not used by user apps most of the time)

1. Basic Point to Point (Nonblocking)

- Assumptions:
  - Only two processes (for now)
  - Networking is established before this point
  - FIFO messages (no overtaking)
  - No specific assumption of buffering
- With this connection, we can send or receive a message from other process
- Function starts operation, with the assumption that it will eventually complete

Example Program so far

- New functions:
  - nonblocking_send
  - nonblocking_rev
- How to decide how many bytes to receive?
- Overall, not too different from using a networking library

```
#include "ourlibrary.h"
#include <iostream>

int main(int argc, char ** argv)
{
    /* Message Passing Library setup */

    int process_rank = /* Magic */;

    int important_buffer = -1;

    if(0 == process_rank )
    {
        important_buffer = 1337;
        nonblocking_send(&important_buffer, sizeof(important_buffer), 1);
    }
    else
    {
        nonblocking_recv(&important_buffer, sizeof(important_buffer), 0);
    }

    std::cout << "My process rank: " << process_rank << " got: " << important_buffer << std::endl;

    return 0;
}
```

Resources in MPI

- Could be from a variety of areas
  - Network resources
    - Special connection objects
    - Resources provided by network itself
  - Implementation resources
    - MPI objects given to user to use
    - Internal data structures used for various infrastructure
    - Threads, runtime services and processes
  - User resources:
    - Allocated memory
    - MPI objects given to user (or created by user)
    - Array objects, buffers

Implementation Insights – Resource Control

- When is it safe to return control to the user?
- When do we want to?
- A few options for returning:
  1. Return control ASAP, but continue ownership of user resources
  2. When data has been buffered (but not sent)
  3. When data has been queued to network
  4. When data has been matched on receiving side at MPI level
- *MPI TIP*: The standard, blocking send in MPI can do any of 2- 4, but often tries to do #3. All modes are provided through other MPI functions.

Implementation Insights – Progress

- “Eventually complete” also requires some sort of progress design
- Weak progress options (no extra threads):
  - Do all sending/receiving at communication call
    - – Not really non-blocking, or “starting”
    - + Could work well for small messages
  - Do all sending/receiving at wait
    - – no overlap of computation and communication
  - Progress all pending network communication during various completing MPI function calls
    - Could get some overlap, but no guarantees

Implementation Insights – Progress (Cont.)

- Strong progress options:
  - Offload to a dedicated thread
  - Offload to network interface
- Thread works on messages until program is complete
  - But how would thread know program is complete?
  - And when does thread start?
- Could do 1 thread per send/recv!
  - Not very scalable…

Implementation Insights – Library Initialization

- Since we hinted at the use of behind-the-scenes implementation resources:
  - Where do those get made?
  - Where do such resources get destroyed?
- What else could be done here?
  - Network connections
  - Now we can have more than two processes!
- But how do we know which message comes from who…?
  - Give each process a rank!
  - Use something unique to each process (such as process ids)
  - Determined by library in init function

Updated Example Program

- Initialization changes:
  - Library_init (MPI_Init)
  - Library_finalize (MPI_Finalize)
- Changes from adding ranks
  - get_my_rank (~MPI_Comm_rank)
  - Can get size too
  - Adjust lines 12, 15, and 19

```
1  #include "ourlibrary.h"
2  #include <iostream>
3
4  int main(int argc, char ** argv)
5  {
6      Library_init(&argc, &argv);
7
8      int process_rank = get_my_rank();
9
10     int important_buffer = -1;
11
12     if(0 == (process_rank % 2))
13     {
14         important_buffer = 1337;
15         nonblocking_send(&important_buffer, sizeof(important_buffer), process_rank+1);
16     }
17     else
18     {
19         nonblocking_recv(&important_buffer, sizeof(important_buffer), process_rank-1);
20     }
21
22     std::cout << "My process rank: " << process_rank << " got: " << important_buffer << std::endl;
23
24     Library_finalize();
25
26     return 0;
27 }
```

2. Completion

- Need some way to ensure action is “done done”
  - This can be difficult to know
  - Instead, having a way to ensure that user resources are safe to use
- Solution:
  - A wait function that returns only when MPI is done with user resources
  - Waiting can be restrictive, could we do something during all that waiting time and check in periodically?
- Also add a testing function!
  - *MPI TIP*: The MPI_Test function is destructive for non-blocking (non-persistent) requests!

But wait, what to wait on?!

- Upon starting the operation, the library should provide an object representing the action we requested.
  - AKA MPI Requests

- Once waited on, the library can go ahead and reclaim resources associated with this object

```cpp
#include "ourlibrary.h"
#include <iostream>

int main(int argc, char ** argv)
{
    Library_init(&argc, &argv);

    int process_rank = get_my_rank();

    int important_buffer = -1;
    Request my_request;

    if(0 == (process_rank % 2))
    {
        important_buffer = 1337;
        nonblocking_send(&important_buffer, sizeof(important_buffer), process_rank+1, &my_request);
    }
    else
    {
        nonblocking_recv(&important_buffer, sizeof(important_buffer), process_rank-1, &my_request);
    }

    wait(&my_request);

    std::cout << "My process rank: " << process_rank << " got: " << important_buffer << std::endl;

    Library_finalize();

    return 0;
}
```

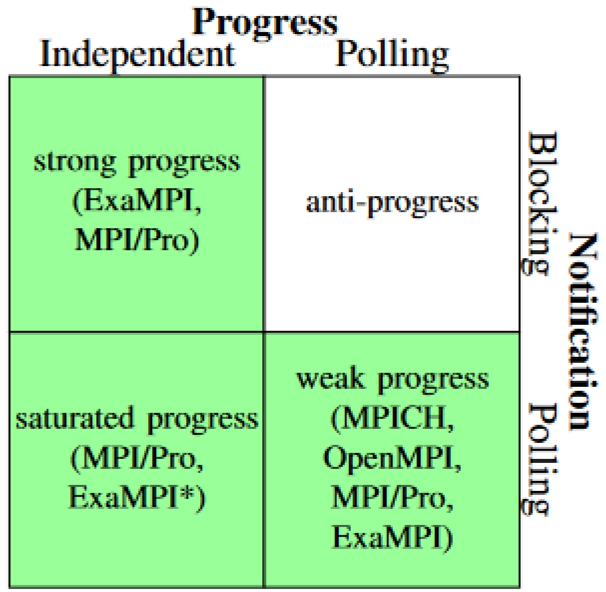
Basic Point to Point (Blocking)

- Call non-blocking version
- Immediately call wait function
- ???
- Profit

```c
int MPI_Send(const void * buf,
             int          count,
             MPI_Datatype datatype,
             int          dest,
             int          tag,
             MPI_Comm     comm)
{
    MPI_Request temp = MPI_REQUEST_NULL;
    MPI_Isend(buf, count, datatype, dest, tag, comm, &temp);
    MPI_Wait(&temp, MPI_STATUS_IGNORE);
}
```

Dimitrov’s Model of MPI Progress and Completion

- What kinds of programs need what kinds of progress?
- Many small message – strong progress, polling notification
- Many large messages – strong progress, blocking notification
- Hybrid polling/blocking notification “optimal”
- Overlap poor when notification polling



Implementation Insights – Waiting

- In weak progress, user thread completes action itself through library call
- In strong progress, user thread waits for completion notification from progress thread/network resource
- For strong progress, use classic threading tools:
  - Mutexs + locks, semaphores, conditional variables, signals, etc

3. Persistence (Init, Start, Free)

- Suppose the user wanted to repeat an operation several times
  - Reallocating resources for the same operation over and over seems wasteful.
  - What if we made them stick around? (persistent)
- Initialization functions:
  - Allocate any resource that will be used for communication
  - Nothing is started
- Starting function:
  - Only works on inactive requests
- Freeing functions:
  - An explicit way for user to get rid of implementation resources
  - Wait/test will still work as normal but won't destroy a request when it is done

Looking at an Example

```cpp
#include "ourlibrary.h"
#include <iostream>

int main(int argc, char ** argv)
{
    Library_init(&argc, &argv);

    int process_rank = get_my_rank();

    int important_buffer = -1;
    Request my_request;

    if(0 == (process_rank % 2))
    {
        important_buffer = 1337;
        send_init(&important_buffer, sizeof(important_buffer), process_rank+1, &my_request);
    }
    else
    {
        recv_init(&important_buffer, sizeof(important_buffer), process_rank-1, &my_request);
    }

    for(int iterations = 0; i < 10; i++)
    {
        start(&my_request);
        /* Do Computation Stuff! */
        wait(&my_request);
        std::cout << "My process rank: " << process_rank << " got: " << important_buffer << std::endl;
        /* Update buffer, if required.*/
    }

    library_request_free(&my_request);

    Library_finalize();

    return 0;
}
```

Implementation Insights

- Layers don't add performance
- Applications don't want to pay for features they don't use
- MPI operations often have generalized feature sets
- Persistence is:
  - An example abstraction on how to capture more performance
  - Like caching
- Certain operations have many possible implementations and protocols

4. Collectives

- With more than one other process, how to optimally talk to everyone?
- Two kinds of collectives:
  - All to one
  - One to all
- Let the user build some pattern out of sending and receiving?
  - Requiring users to roll their own is restrictive
  - Let's hide away complex algorithms by providing convenience functions
  - Follow same blocking/nonblocking/persistence semantics
- Introduces new "synchronizing semantics"

Implementation Insights – Collective Algorithm Design

- Users can certainly make their own design with basic MPI functions, but lack ability to interact with network resources
- Implementations have the power to optimize collective:
  - Size of message is small? Do X pattern
  - Only N number of processes? Do Y pattern
  - Topologically aware of where everyone is? Do Z pattern
- Implementations could still be built off a schedule of basic sends and receives underneath

5. Groups, Communicators, and Contexts

- Separate the communication from different parts of your program
  - For example, your main program can have different messages from your library, even if they use the same group
- Two kinds:
  - Who you can address (Group)
  - Where the messages can go (Context)
- Communicator provides both concepts at the same time
  - Adding contexts will add small overhead to message
  - Adding tags will also add more overhead

MPI subsets are often enough

Minimum

- MPI Init
- MPI Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Isend
- MPI_Irecv
- MPI_Wait (MPI_Test)

More

- Collectives:
  - e.g.: MPI_Igather, MPI_Ibcast
  - MPI_Iallreduce
- Manage sub-group communication
  - MPI_Comm_idup
  - MPI_Comm_split (no _isplit yet)
  - MPI_Comm_free
- More operation modes
  - MPI_bsend, MPI_ssend

Other Resources

- "Light" reading:
  - Skjellum, Anthony, et al. 2019. "ExaMPI: A Modern Design and Implementation to Accelerate Message Passing Interface Innovation."
  - Laguna, Ignacio, et al. 2019. "A large-scale study of MPI usage in open-source HPC applications."
- "Darker" corners:
  - Holmes, Dan J., et al. 2020. "Why is MPI (perceived to be) so complex? Part 1—Does strong progress simplify MPI?"
  - Bangalore, Purushotham V., et al. 2019. "Exposition, clarification, and expansion of MPI semantic terms and conventions: is a nonblocking MPI function permitted to block?"
  - Other MPI implementations
- The MPI Standard itself has many insights
- Ask questions!

Any Questions?

Thank you!