ADVANCED MPI

Hajime Fujita and Wesley Bland

04/30/2019

Senior Software Development Engineer, Intel Corporation
Last Time...

We covered:

• What is MPI (and the implementations of it)?
• Startup & Finalize
• Blocking Send & Receive
• Non-blocking Send & Receive
```c
int rank, size, numbers[100];
MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (rank == 0) {
    for (i = 1; i < size; i++)
        MPI_Send(&numbers[(100/size)*i], (100/size), ...
} else
  MPI_Recv(&numbers[0], (100/size), ...

sort_numbers(numbers, (100/size));

if (rank == 0) {
    for (i = 1; i < size; i++)
        MPI_Recv(&numbers[(100/size)*i], (100/size), ...
} else
  MPI_Send(&numbers[0], (100/size), ...

combine_arrays(&numbers[0], &numbers[(100/size)], (100/size));
MPI_Finalize();
```

Basic Send/Recv

```
8 23 19 67 45 35 1 24 13 30 3 5
```

```
8 19 23 35 45 67 1 3 5
```

```
1 3 5 8 13 19 23 24 30 35 45 67
```

```
13 24 30
```

```
Process1  Process2  Process3  Process4
Process1  Process2  Process3  Process4
Process1  Process2  Process3  Process4
Process1  Process2  Process3  Process4
```
A Non-Blocking communication example

...snip...

/* Compute each data element and send it out */
if (rank == 0) {
    for (i=0; i< 100; i++) {
        data[i] = compute(i);
        MPI_Isend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request[i]);
    }
    MPI_Waitall(100, request, MPI_STATUSES_IGNORE)
} else if (rank == 1) {
    for (i = 0; i < 100; i++)
        MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
...snip...
This Time

- Collectives (continued)
- Communicators
- Datatypes
- Brief look at advanced topics (RMA, threads)
- Analyzing the performance of MPI
COLLECTIVES (CONT.)
What Are Collectives?

A group of processes works together to accomplish something.

• E.g. Calculate a value, distribute some data, gather a result, synchronize operations, etc.

Instead of sending a value to each process in a for loop, use one collective call and let MPI optimize doing that for you.
Integer Sorting Example Revisited

```c
if (rank == 0) {
    for (i = 1; i < size; i++)
        MPI_Send(&numbers[(100/size)*i], (100/size), ..., i, ...);
else
    MPI_Recv(&numbers[0], (100/size), ...,);

int root = 0;
MPI_Scatter(numbers, 100/size, MPI_INT, rank == root ? MPI_IN_PLACE : numbers,
100/size, MPI_INT, root, MPI_COMM_WORLD);
```

![Diagram showing the process of scattering and gathering data among processes P0, P1, P2, and P3.](image)
Example: Calculating π

Calculating π via numerical integration

- Divide interval up into subintervals
- Assign subintervals to processes
- Each process calculates partial sum
- Add all the partial sums together to get π

1. Width of each segment \( w \) will be \( 1/n \)
2. Distance \( d(i) \) of segment “i” from the origin will be “i * w”
3. Height of segment “i” will be \( \sqrt{1 - [d(i)]^2} \)
Example: Pi in C

```c
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[]) {
    double mypi = 0.0;
    [...snip...]
    MPI_Bcast(&num_segs, 1, MPI_INT, 0, MPI_COMM_WORLD);
    double width = 1.0 / (double) num_segs;
    for (int i = rank + 1; i <= n; i += size)
        mypi += width * sqrt(1 - (((double) i / num_segs) * ((double) i / num_segs)));
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               4 * pi, fabs((4 * pi) - PI25DT));
    [...snip...]
}
```
```c
#include <mpi.h>
#include <math.h>

int main(int argc, char *argv[]) {
    double mypi = 0.0;
    [...snip...]

    MPI_Bcast(&num_segs, 1, MPI_INT, 0, MPI_COMM_WORLD);

    double width = 1.0 / (double) num_segs;
    for (int i = rank + 1; i <= n; i += size)
        mypi += width * sqrt(1 - (((double) i / num_segs) * ((double) i / num_segs)));

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0)
        printf("pi is approximately %.16f, Error is %.16f\n", 4 * pi, fabs((4 * pi) - PI25DT));
    [...snip...]
}
```

**Example: Pi in C**

Tell all processes how many rectangles there are.

Calculate my share of pi.

Send the result to rank 0 and calculate the total at the same time.
Non-blocking Collectives

Recently (MPI 3.0), non-blocking collectives were added to the MPI Standard. Now you can use MPI_IREDUCE, MPI_IBCAST, and even MPI_IBARRIER.
COMMUNICATORS
Let's Take a Closer Look at MPI_SEND

```c
int MPI_Send(
    const void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm)
```

- **Pointer to the data buffer**
- **Description of the data buffer**
- **Rank of message destination**
- **Tag to differentiate messages**
- **MPI Communicator**
Let’s Take a Closer Look at MPI_SEND

```c
int MPI_Send(
    const void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm
)
```

- **Pointer to the data buffer**
- **Description of the data buffer**
- **Rank of message destination**
- **Tag to differentiate messages**
- **MPI Communicator**
What is an MPI Communicator?

Groups of MPI processes that can talk to each other

• Start with MPI_COMM_WORLD that includes everyone that started with your job

• Can divide MPI_COMM_WORLD to be smaller communicators like comm1, comm2, comm3.

Each process has a rank in each communicator

Each communicator has its own size

Communicator could even be a copy of a previous communicator
How Do They Overlap?

Can be thought of as independent communication layers over a group of processes

Messages in one layer will not affect messages in another
How Do I Divide a Communicator?

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

Communicator to be divided into subcommunicators
Colors to match to create new communicators
Keys to order ranks in new communicators
Output communicator (potentially different on each rank)
Communicator Splitting Example

MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

color = rank / 3;
key = rank % 3;

MPI_Comm_split(MPI_COMM_WORLD, color, key, &newcomm);

...snip...
Communicator Splitting Example

MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

color = rank / 3;
key = rank % 3;

MPI_Comm_split(MPI_COMM_WORLD, color, key, &newcomm);

[...snip...]
Why Would I Want To Use Communicators?

• Being able to communicate with just a subset of processes is very helpful for certain kinds of algorithms.

• For example, with matrix factorization, doing row and column-based communication is important.

• You can use the smaller set of ranks to make it easy to calculate message sources and destinations.

• Perform a collective operation on a smaller set of ranks
DATATYPES
What About the Rest of MPI_SEND?

```c
int MPI_Send(
    const void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm
)
```
What is a Datatype?

Data representation is always tricky.

- Particularly when you want your program to work on multiple architectures at the same time.
- E.g. Combine Big Endian and Little Endian in the same job.

If you just want to send an int from one process to another, do you really want to deal with this?
So How Do We Solve It?

MPI defined a set of datatypes that users can work with and not have to worry about different architectures, etc.

The user can (usually) just use their language’s built in datatypes and they will translate correctly.

<table>
<thead>
<tr>
<th>MPI Type</th>
<th>C Type</th>
<th>Fortran Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR/MPI_CHARACTER</td>
<td>char</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>MPI_INT/MPI_INTEGER</td>
<td>int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_DOUBLE / MPI_DOUBLE_PRECISION</td>
<td>double</td>
<td>DOUBLE PRECISION</td>
</tr>
</tbody>
</table>
Halo Exchange Example

Provide access to remote data through a *halo* exchange (called a stencil).

Remember that our send function has a count field:

```c
int MPI_Send(const void *buf,
             int count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm)
```

So our row send could look like this:

```c
MPI_Send(&buf[0], 10, MPI_DOUBLE, neighbor, 0, MPI_COMM_WORLD);
```

But what if we want to send a column?
Vector Datatype

Specify strided blocks of data of oldtype

Very useful for Cartesian arrays

```c
MPI_Type_vector(int count, int blocklen, int stride,
                 MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```c
MPI_Type_vector(6, 1, 6, MPI_DOUBLE, &column)
```
Use Datatype in Halo Exchange

\[
\text{MPI\_Type\_contiguous}(\text{count}=bx, \text{MPI\_DOUBLE}, \ldots) \text{ or count with MPI\_DOUBLE}
\]

\[
\text{MPI\_Type\_vector}(\text{count}=by, \text{blocklen}=1, \text{stride}=bx+2, \text{MPI\_DOUBLE}, \ldots)
\]
ADVANCED TOPICS: ONE-SIDED COMMUNICATION
One-sided Communication

Decouple data movement with process synchronization

- Should be able to move data without requiring synchronization
- Each process exposes a part of its memory to other processes
- Other processes can directly read from or write to this memory
Two-sided Communication Example

![Diagram showing two-sided communication example with two processors and multiple memory segments. Each processor communicates with memory segments, demonstrating the concept of MPI implementation.]
One-sided Communication Example
Comparing One-sided and Two-sided Programming

Even the sending process is delayed

Delay in process 1 does not affect process 0

Even the sending process is delayed
Creating Public Memory

Any memory used by a process is, by default, only locally accessible.

- \( X = \text{malloc}(100); \)

Once the memory is allocated, the user has to make an explicit MPI call to declare a memory region as remotely accessible.

- MPI terminology for remotely accessible memory is a “\textit{window}”

- A group of processes collectively create a “window”

Once a memory region is declared as remotely accessible, all processes in the window can read/write data to this memory without explicitly synchronizing with the target process.
MPI_WIN_CREATE

Expose a region of memory in an RMA window

- Only data exposed in a window can be accessed with RMA ops.

Arguments:

- base - pointer to local data to expose
- size - size of local data in bytes (nonnegative integer)
- disp_unit - local unit size for displacements, in bytes (positive integer)
- info - info argument (handle)
- comm - communicator (handle)
- win - window (handle)

```c
MPI_Win_create(void *base, MPI_Aint size,
               int disp_unit, MPI_Info info,
               MPI_Comm comm, MPI_Win *win)
```
Data movement

MPI provides ability to read, write and atomically modify data in remotely accessible memory regions

- MPI_PUT
- MPI_GET
- MPI_ACCUMULATE (atomic)
- MPI_GET_ACCUMULATE (atomic)
- MPI_COMPARE_AND_SWAP (atomic)
- MPI_FETCH_AND_OP (atomic)
Data movement: *Put*

Move data from origin, to target

Separate data description triples for origin and target

**MPI_Put**

```
MPI_Put(const void *origin_addr, int origin_count,
        MPI_Datatype origin_dtype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_dtype, MPI_Win win)
```
Data movement: Get

MPI_Get(void *origin_addr, int origin_count,
         MPI_Datatype origin_dtype, int target_rank,
         MPI_Aint target_disp, int target_count,
         MPI_Datatype target_dtype, MPI_Win win)

Move data to origin, from target

Separate data description triples for origin and target
RMA Synchronization Models

RMA data access model

- When is a process allowed to read/write remotely accessible memory?
- When is data written by process X available for process Y to read?
- RMA synchronization models define these semantics

Three synchronization models:

- Fence (active target)
- Post-start-complete-wait (generalized active target)
- Lock/Unlock (passive target)

Data accesses occur within “epochs”

- Access epochs: contain a set of operations issued by an origin process
- Exposure epochs: enable remote processes to update a target’s window
- Epochs define ordering and completion semantics
- Synchronization models provide mechanisms for establishing epochs
  - E.g., starting, ending, and synchronizing epochs
MPI + X: THREADS
Why Hybrid MPI+X?

Strong scaling applications are increasing in importance

- Hardware limitations: not all resources scale at the same rate as cores (e.g., memory capacity, network resources)
- Desire to solve the same problem faster on a bigger machine

Strong scaling pure MPI applications is getting harder

- On-node communication is costly compared to load/stores
- \(O(P^x)\) communication patterns (e.g., All-to-all) costly

MPI+X benefits (\(X=\) {threads, MPI shared-memory, etc.})

- Less memory hungry (MPI runtime consumption, \(O(P)\) data structures, etc.)
- Load/stores to access memory instead of message passing
- \(P\) is reduced by constant \(C\) (#cores/process) for \(O(P^x)\) communication patterns

Evolution of the memory capacity per core in the Top500 list (Peter Kogge. Pim & memory: The need for a revolution in architecture.)

Strong scaling = fixed problem size
Weak scaling = variable problem size
MPI + Threads: How To?
MPI + Threads: How To?

Interoperation or thread levels:

- **MPI_THREAD_SINGLE**
  - No additional threads

- **MPI_THREAD_FUNNELED**
  - Master thread communication only

- **MPI_THREAD_SERIALIZE**
  - Threaded communication serialized

- **MPI_THREAD_MULTIPLE**
  - No restrictions
**MPI_THREAD_SINGLE**

There are no additional user threads in the system

- E.g., there are no OpenMP parallel regions

```c
int buf[100];
int main(int argc, char ** argv)
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */
    MPI_Finalize();
    return 0;
}
```
**MPI_THREAD_SERIALIZED**

Only one thread can make MPI calls at a time

- Protected by OpenMP critical regions
- Essentially the same as MPI_THREAD_FUNNELED

```c
int buf[100];
int main(int argc, char ** argv) {
    int provided;
    
    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    if (provided < MPI_THREAD_FUNNELED)
        MPI_Abort(MPI_COMM_WORLD,1);
    
    #pragma omp parallel
    {
        /* Do non-MPI stuff */
        #pragma omp single
        {
            /* Do MPI stuff */
        }
        /* Do non-MPI stuff */
    }
    MPI_Finalize();
    return 0;
}
```
MPI_THREAD_MULTIPLE

Any thread can make MPI calls any time (restrictions apply)

```c
int buf[100];
int main(int argc, char ** argv)
{
    int provided;

    MPI_Init_thread(&argc, &argv,
                     MPI_THREAD_FUNNELED,
                     &provided);

    if (provided < MPI_THREAD_FUNNELED)
        MPI_Abort(MPI_COMM_WORLD,1);

    #pragma omp parallel
    {
        /* Do non-MPI stuff */
        /* Do MPI stuff */
        /* Do non-MPI stuff */
    }

    MPI_Finalize();
    return 0;
}
```
INTEL CLUSTER TOOLS

Tuning MPI Application Performance
Intel® Trace Analyzer and Collector (ITAC)

Automatic Performance Assistant
Detect common MPI performance issues
Automated tips on potential solutions

Automatically detect performance issues and their impact on runtime
Intel® VTune™ Amplifier XE
Tune hybrid parallelism using ITAC + VTune Amplifier

Tune OpenMP performance of high impact ranks in VTune Amplifier

Top OpenMP Processes by MPI Communication Spin Time

This section lists processes sorted by MPI Communication Spin time. The lower MPI Communication Spin time is a critical path of MPI application execution. Explore OpenMP efficiency metrics by MPI processes relying on the process.

<table>
<thead>
<tr>
<th>Process</th>
<th>MPI Communication Spinning</th>
<th>OpenMP Potential Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>heart_demo (rank 7)</td>
<td>5.122s 8.1%</td>
<td>19.929s 11.3%</td>
</tr>
<tr>
<td>heart_demo (rank 10)</td>
<td>5.463s 8.6%</td>
<td>19.482s 30.6%</td>
</tr>
<tr>
<td>heart_demo (rank 11)</td>
<td>5.593s 8.8%</td>
<td>20.183s 31.7%</td>
</tr>
<tr>
<td>heart_demo (rank 6)</td>
<td>6.264s 9.8%</td>
<td>19.429s 30.5%</td>
</tr>
<tr>
<td>heart_demo (rank 9)</td>
<td>6.595s 10.4%</td>
<td>19.379s 30.5%</td>
</tr>
</tbody>
</table>

Per-rank OpenMP Potential Gain and Serial Time metrics

Ranks sorted by MPI Communication Spins – ranks on the critical path are on the top

Process names link to OpenMP metrics

Detailed OpenMP metrics per MPI ranks
Checking MPI Application Correctness

**Warnings** indicate potential problems that could cause unexpected behavior (e.g., incomplete message requests, overwriting a send/receive buffer, potential deadlock, etc.).

**Errors** indicate problems that violate the MPI standard or definitely cause behavior not intended by the programmer (e.g., incomplete collectives, API errors, corrupting a send/receive buffer, deadlock, etc.).
WRAPPING UP
What did we learn?

• What is MPI (and the implementations of it)?
• Startup & Finalize
• Basic Send & Receive
• Communicators
• Collectives
• Datatypes
• Non-blocking communication

• Brief look at advanced topics (RMA, threads)
• Analyzing the performance of MPI
What is Next for MPI?

- **Fault Tolerance**
  - Always under research, coming in incremental changes
- **Persistent Collective Communication**
  - Avoid setting up collective communication channels multiple times
- **“Sessions”**
  - Improve interaction between applications, libraries, and tools
  - Allow better scalability for sparse applications
- **Improved Tool Support**
  - Allow multiple tools to attach at once
  - Get event-driven callbacks
Where Can I Learn More?

Standard Documentation: http://mpi-forum.org/docs

Stack Overflow (tags: mpi, mpich, open-mpi)

Websites: http://mpitutorial.com
Tutorial Books on MPI

Using MPI
Portable Parallel Programming
with the Message-Passing Interface
third edition

William Gropp
Ewing Lusk
Anthony Skjellum

Basic MPI

Using Advanced MPI
Modern Features of the
Message-Passing Interface

William Gropp
Torsten Hoeffer
Rajeev Thakur
Ewing Lusk

Advanced MPI, including MPI-3