Lecture 4
Introduction to MPI

Jeremy Wei
Center for HPC, SJTU
Mar 13th, 2017
Recap of the last lecture (OpenMP)

• OpenMP is a standardized pragma-based intra-node parallel programming paradigm.

• Balance workload with “schedule” and eliminate false sharing with “reduce”.

• OpenMP 4.5 adds experimental support NVIDIA GPUs and Intel Xeon Phi accelerators.
Why MPI?

- A standardized message-passing programming interface for distributed-memory system.

- Good performance for both intra-node and inter-node programs with vendor MPI implementations exploring native hardware features. (used as the performance upper bound in experiments)

- Embrace cluster-like super computer in which millions of CPUs are connected by networks.
Goals of this lecture

• Introduce basic MPI concepts.

• Write real MPI program(s) on your laptop with GCC+OpenMPI.

• Some best practices for MPI communication.
Outline

• MPI programming models
• Blocking communication in MPI
• Non-blocking communication in MPI
• Group (collective) communication in MPI
The Message-Passing Model

• A process is (traditionally) a program counter and address space.
• Processes may have multiple threads (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
• Inter-process communication consists of
  – Synchronization (preparation)
  – movement of data from one process’s address space to another’s.
The Message-Passing Model (Parallel Sort)

- Each process has to send/receive data to/from other processes
- Example: Parallel Sort

**Process 1**
- 8 23 19 67 45 35 1 24 13 30 3 5

**Process 1**
- O(N log N)

**Process 2**
- 1 3 5 13 24 30

**Process 1**
- O(N/2 log N/2)

**Process 1**
- O(N)

8 19 23 35 45 67

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

1 3 5 8 13 19 23 24 30 35 45 67
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
What is in MPI-1

- Basic functions for communication (100+ functions)
- **Blocking sends, receives**
- **Nonblocking sends and receives**
- Variants of above
- **Rich set of collective communication functions**
  - Broadcast, scatter, gather, etc
  - Very important for performance; widely used
- Datatypes to describe data layout
- Process topologies
- C, C++ and Fortran bindings
- Error codes and classes
Following MPI Standards

• MPI-2 was released in 2000
  – Several additional features including MPI + threads, MPI-I/O, remote memory access functionality and many 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 (single-side communication)
• 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
Applications (> 100K processes per job)

• MPI is widely used in large scale parallel applications in science and engineering
  – Atmosphere, Earth, Environment
  – Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
  – Bioscience, Biotechnology, Genetics
  – Chemistry, Molecular Sciences
  – Geology, Seismology
  – Mechanical Engineering - from prosthetics to spacecraft
  – Electrical Engineering, Circuit Design, Microelectronics
  – Computer Science, Mathematics
Important considerations while using MPI

All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.
Compiling and Running MPI applications

• MPI is a library
  – Applications can be written in C, C++ or Fortran and appropriate calls to MPI can be added where required

• Compilation:
  – Regular applications:
    • gcc test.c -o test
  – MPI applications
    • mpicc test.c -o test

• Execution:
  – Regular applications
    • ./test
  – MPI applications (running with 16 processes)
    • mpirun -n 4 ./test
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
Outline

- MPI programming models
- Blocking communication in MPI
- Non-blocking communication in MPI
- Group (collective) communication in MPI
MPI Basic Send/Receive

• Simple communication model

  Process 0
  Send(data)

  Process 1
  Receive(data)

• Application needs to specify to the MPI implementation:
  1. How do you compile and run an MPI application?
  2. How will processes be identified?
  3. How will “data” be described?
Process Identification
communicator + rank

- MPI processes can be collected into groups
  - Each group can have multiple context (some times called color)
    - Group + context = communicator (it is like a name for the group)
  - When an MPI application starts, the group of all processes is initially given a predefined name called MPI_COMM_WORLD
    - The same group can have many names, but simple programs do not have to worry about multiple names

- A process is identified by a unique number within each communicator, called rank
  - For two different communicators, the same process can have two different ranks: so the meaning of a “rank” is only defined when you specify the communicator
Communicators

When you start an MPI program, there is one predefined communicator `MPI_COMM_WORLD`.

Communicators do not need to contain all processes in the system.

Communicators can be created “by hand” or using tools provided by MPI.

Simple programs typically only use the predefined communicator `MPI_COMM_WORLD`.

Every process in a communicator has an ID called as “rank”.

Can make copies of this communicator (same group of processes, but different “aliases”).

The same process might have different ranks in different communicators.

mpirun -n 16 ./test
MPI is Simple

• Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  – MPI_INIT – initialize the MPI library (must be the first routine called)
  – MPI_COMM_SIZE – get the size of a communicator
  – MPI_COMM_RANK – get the rank of the calling process in the communicator
  – MPI_SEND – send a message to another process
  – MPI_RECV – send a message to another process
  – MPI_FINALIZE – clean up all MPI state (must be the last MPI function called by a process)

• For performance, however, you need to use other MPI features
Data Communication

• Data communication in MPI is like email exchange
  – One process sends a copy of the data to another process (or a group of processes), and the other process receives it

• Communication requires the following information:
  – Sender has to know:
    • Whom to send the data to (receiver’s process \texttt{communicator+rank})
    • What kind of data to send (100 integers or 200 characters, etc)
    • A user-defined “tag” for the message (think of it as an email subject; allows the receiver to understand what type of data is being received)

  – Receiver “might” have to know:
    • Who is sending the data (OK if the receiver does not know; in this case sender rank will be \texttt{MPI\_ANY\_SOURCE}, meaning anyone can send)
    • What kind of data is being received (partial information is OK: I might receive \textit{up to} 1000 integers)
    • What the user-defined “tag” of the message is (OK if the receiver does not know; in this case tag will be \texttt{MPI\_ANY\_TAG})
MPI Basic (Blocking) Send

\texttt{MPI\_SEND(buf, count, datatype, dest, tag, comm)}

- The message buffer is described by \((\text{buf, count, datatype})\).
- The target process is specified by \text{dest} and \text{comm}.
  - \text{dest} is the rank of the target process in the communicator specified by \text{comm}.
- \text{tag} is a user-defined “type” for the message
- When this function returns, the data has been delivered to the system and the buffer can be reused.
  - The message may not have been received by the target process.
MPI Basic (Blocking) Receive

MPI_RECV(buf, count, datatype, source, tag, comm, status)

• Waits until a matching (on source, tag, comm) message is received from the system, and the buffer can be used.
• source is rank in communicator comm, or MPI_ANY_SOURCE.
• Receiving fewer than count occurrences of datatype is OK, but receiving more is an error.
• status contains further information:
  – Who sent the message (can be used if you used MPI_ANY_SOURCE)
  – How much data was actually received
  – What tag was used with the message (can be used if you used MPI_ANY_TAG)
  – MPI_STATUS_IGNORE can be used if we don’t need any additional information
More Details on Describing Data for Communication

• MPI Datatype is very similar to a C or Fortran datatype
  – int → MPI_INT
  – double → MPI_DOUBLE
  – char → MPI_CHAR

• More complex datatypes are also possible:
  – E.g., you can create a structure datatype that comprises of other datatypes → a char, an int and a double.
  – Or, a vector datatype for the columns of a matrix

• The “count” in MPI_SEND and MPI_RECV refers to how many datatype elements should be communicated
Simple Communication in MPI (Demo)

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    int rank, data[100];
    const int COUNT=100, TAG=0;
    const int MPI_MASTER=0, MPI_SLAVE=1;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == MPI_MASTER)
        MPI_Send(data, COUNT, MPI_INT, MPI_SLAVE, TAG, MPI_COMM_WORLD);
    else if (rank == MPI_SLAVE)
        MPI_Recv(data, COUNT, MPI_INT, MPI_MASTER, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    MPI_Finalize();
    return 0;
}
```
Parallel Sort using MPI Send/Recv

Rank 0

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

Rank 1

1 3 5 13 24 30

O(N log N)

Rank 0

8 19 23 35 45 67

Rank 0

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

Rank 0

1 3 5 8 13 19 23 24 30 35 45 67
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    int rank;
    int a[1000], b[500];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        MPI_Send(&a[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD);
        sort(a, 500);
        MPI_Recv(b, 500, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
        /* Serial: Merge array b and sorted part of array a */
    }
    else if (rank == 1) {
        MPI_Recv(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        sort(b, 500);
        MPI_Send(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize(); return 0;
}
Outline

• MPI programming models
• Blocking communication in MPI
• Non-blocking communication in MPI
• Group (collective) communication in MPI
Blocking vs. Non-blocking Communication

- **MPI_SEND/MPI_RECV** are blocking communication calls
  - Return of the routine implies completion
  - When these calls return the memory locations used in the message transfer can be safely accessed for reuse
  - For “send” completion implies variable sent can be reused/modified
  - Modifications will not affect data intended for the receiver
  - For “receive” variable received can be read

- **MPI_ISEND/MPI_IRecv** are non-blocking variants
  - Routine returns immediately – completion has to be separately tested for
  - These are primarily used to overlap computation and communication to improve performance
Blocking Send-Receive Diagram

T1: MPI_Send

T2: Internal completion is soon followed by return of MPI_Recv

sender returns @ T2, buffer can be reused

T0: MPI_Recv
Once receive is called @ T0, buffer unavailable to user

T3: Transfer Complete

T4: Receive returns @ T4, buffer filled

send side

receive side
Non-Blocking Send-Receive Diagram

High Performance Implementations
Offer Low Overhead for Non-blocking Calls

T0: MPI_Irecv
T1: Returns
T7: transfer finishes
T8

T2: MPI_Isend
T3
T4
sender
returns @ T3
buffer unavailable
sender
completes @ T5
buffer available
after MPI_Wait
T6: MPI_Wait
T5
T6
T9: Wait returns

send side

time

receive side

Internal completion is soon followed by return of MPI_Wait
Pitfalls of Blocking Communication

- In blocking communication.
  - `MPI_SEND` does not return until buffer is empty (available for reuse)
  - `MPI_RECV` does not return until buffer is full (available for use)
- A process sending data will be blocked until data in the send buffer is emptied
- A process receiving data will be blocked until the receive buffer is filled
- Exact completion semantics of communication generally depends on the message size and the system buffer size
- Blocking communication is simple to use but can be prone to deadlocks

```plaintext
If (rank == 0) Then
  Call mpi_send(..)
  Call mpi_recv(..)
Else
  Call mpi_send(..) \leftrightarrow UNLESS you reverse send/recv
  Call mpi_recv(..)
Endif
```
Non-Blocking Communication

- Non-blocking (asynchronous) operations return (immediately) “request handles” that can be waited on and queried
  - MPI_ISEND(start, count, datatype, dest, tag, comm, request)
  - MPI_IRECV(start, count, datatype, src, tag, comm, request)
  - MPI_WAIT(request, status)

- Non-blocking operations allow overlapping computation and communication
- One can also test without waiting using MPI_TEST
  - MPI_TEST(request, flag, status)
- Anywhere you use MPI_SEND or MPI_RECV, you can use the pair of MPI_ISEND/MPI_WAIT or MPI_IRECV/MPI_WAIT
- Combinations of blocking and non-blocking sends/receives can be used to synchronize execution instead of barriers
Multiple Completions

• It is sometimes desirable to wait on multiple requests:
  - MPI_Waitall(count, array_of_requests, array_of_statuses)
  - MPI_Waitany(count, array_of_requests, &index, &status)
  - MPI_Waitsome(count, array_of_requests, array_of_indices, array_of_statuses)

• There are corresponding versions of test for each of these
Message Completion and Buffering

• For a communication to succeed:
  – Sender must specify a valid destination rank
  – Receiver must specify a valid source rank (including MPI_ANY_SOURCE)
  – The communicator must be the same
  – Tags must match
  – Receiver’s buffer must be large enough

• A send has completed when the user supplied buffer can be reused

```c
*buf = 3;
MPI_Send(buf, 1, MPI_INT ...)
*buf = 4; /* OK, receiver will always receive 3 */
```

```c
*buf = 3;
MPI_Isend(buf, 1, MPI_INT ...)
*buf = 4; /* Not certain if receiver gets 3 or 4 or anything else */
MPI_Wait(...);
```

• Just because the send completes does not mean that the receive has completed
  – Message may be buffered by the system
  – Message may still be in transit
2D Poisson Problem
Sample Code

```
Do i=1, n_neighbors
    Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag,
                  comm, ierr)
Enddo

Do i=1, n_neighbors
    Call MPI_Recv(edge, len, MPI_REAL, nbr(i), tag,
                   comm, status, ierr)
Enddo
```

• What is wrong with this code?
Deadlocks!

• All of the sends may block, waiting for a matching receive (will for large enough messages)
• The variation of

```c
if (has down nbr)
    Call MPI_Send( ... down ... )
if (has up nbr)
    Call MPI_Recv( ... up ... )
...
```

sequentializes (all except the bottom process blocks)
Fix 1: Use Irecv

Do i=1, n_neighbors
    Call MPI_Irecv(edge, len, MPI_REAL, nbr(i), tag, comm, 
                   requests[i], ierr)
Enddo

Do i=1, n_neighbors
    Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag, comm, ierr)
Enddo

Call MPI_Waitall(n_neighbors, requests, statuses, ierr)

• Does not perform well in practice due to blocking in MPI_send.
A Non-Blocking communication example

```c
int main(int argc, char ** argv)
{
    [...snip...]
    if (rank == 0) {
        for (i=0; i< 100; i++) {
            /* Compute each data element and send it out */
            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 {
        for (i = 0; i < 100; i++)
            MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                      MPI_STATUS_IGNORE);
    }
    [...snip...]
}
```
Outline

• MPI programming models
• Blocking communication in MPI
• Non-blocking communication in MPI
• Group (collective) communication in MPI
Introduction to Collective Operations in MPI

• Collective operations are called by all processes in a communicator.
  
  • MPI_BCAST distributes data from one process (the root) to all others in a communicator.
  
  • MPI_REDUCE combines data from all processes in the communicator and returns it to one process.

• In many numerical algorithms, SEND/RECV can be replaced by BCAST/REDUCE, improving both simplicity and efficiency.
Synchronization

• **MPI_BARRIER**(comm)
  – Blocks until all processes in the group of the communicator **comm** call it
  – A process cannot get out of the barrier until all other processes have reached barrier
Collective Data Movement

Broadcast:

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scatter:

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

Gather:

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More Collective Data Movement

P0
A
B
C
D

P1
A0
B0
C0
D0

P2
A1
B1
C1
D1

P3
A2
B2
C2
D2

<table>
<thead>
<tr>
<th>Allgather</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alltoall</th>
<th>A0</th>
<th>B0</th>
<th>C0</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>P2</td>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td>D2</td>
</tr>
<tr>
<td>P3</td>
<td>A3</td>
<td>B3</td>
<td>C3</td>
<td>D3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alltoall</th>
<th>A0</th>
<th>B0</th>
<th>C0</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>P2</td>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td>D2</td>
</tr>
<tr>
<td>P3</td>
<td>A3</td>
<td>B3</td>
<td>C3</td>
<td>D3</td>
</tr>
</tbody>
</table>
Collective Computation

- **Reduce**
  - P0: A
  - P1: B
  - P2: C
  - P3: D
  - Result: ABCD

- **Scan**
  - P0: A
  - P1: B
  - P2: C
  - P3: D
  - Result: A
  - A
  - AB
  - ABC
  - ABCD
MPI Collective Routines

• Many Routines: `MPI_ALLGATHER`, `MPI_ALLGATHERV`, `MPI_ALLREDUCE`, `MPI_ALLTOALL`, `MPI_ALLTOALLV`, `MPI_BCAST`, `MPI_GATHER`, `MPI_GATHERV`, `MPI_REDUCE`, `MPI_REDUCESCATTER`, `MPI_SCAN`, `MPI_SCATTER`, `MPI_SCATTERV`

• “All” versions deliver results to all participating processes

• “V” versions (stands for vector) allow the hunks to have different sizes

• `MPI_ALLREDUCE`, `MPI_REDUCE`, `MPI_REDUCESCATTER`, and `MPI_SCAN` take both built-in and user-defined combiner functions
MPI Built-in Collective Computation Operations

- **MPI_MAX**  
  Maximum
- **MPI_MIN**  
  Minimum
- **MPI_PROD**  
  Product
- **MPI_SUM**  
  Sum
- **MPI_LAND**  
  Logical and
- **MPI_LOR**  
  Logical or
- **MPI_LXOR**  
  Logical exclusive or
- **MPI_BAND**  
  Bitwise and
- **MPI_BOR**  
  Bitwise or
- **MPI_BXOR**  
  Bitwise exclusive or
- **MPI_MAXLOC**  
  Maximum and location
- **MPI_MINLOC**  
  Minimum and location
Example: Calculating Pi

• Calculating pi 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 pi

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 (let us try writing this)

```c
#include <mpi.h>
#include <math.h>

int main(int argc, char *argv[])
{
    [...snip...]
    /* Tell all processes, the number of segments you want */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    w   = 1.0 / (double) n;
    mypi = 0.0;
    for (i = myid + 1; i <= n; i += numprocs)
        mypi += w * sqrt(1 - ((double) i / n) * ((double) i / n));
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0)
        printf("pi is approximately \%.16f, Error is \%.16f\n", 4 * pi, fabs((4 * pi) - PI25DT));
    [...snip...]
}
```
Conclusions

• Parallelism is critical today, given that that is the only way to achieve performance improvement with the modern hardware.
• MPI is an industry standard model for parallel programming
  – A large number of implementations of MPI exist (both commercial and public domain)
  – Virtually every system in the world supports MPI
• Gives user explicit control on data management
• Widely used by many scientific applications with great success
• Your application can be next!
Web Pointers

- MPI standard: http://www.mpi-forum.org/docs/docs.html
- MPICH: http://www.mpich.org
- MPICH mailing list: discuss@mpich.org
- MPI Forum: http://www.mpi-forum.org/

- Other MPI implementations:
  - MVAPICH2 (MPICH on InfiniBand): http://mvapich.cse.ohio-state.edu/
  - Microsoft MPI (MPICH derivative)
  - Open MPI: http://www.open-mpi.org/

- Several MPI tutorials can be found on the web
  - https://computing.llnl.gov/tutorials/mpi/

- Materials of this lecture come from Dr. Pavan Balaji’s MPI tutorials hosted on ETH Zurich.