Lecture 22: MPI Basics

William Gropp
www.cs.illinois.edu/~wgropp

Message Passing Features

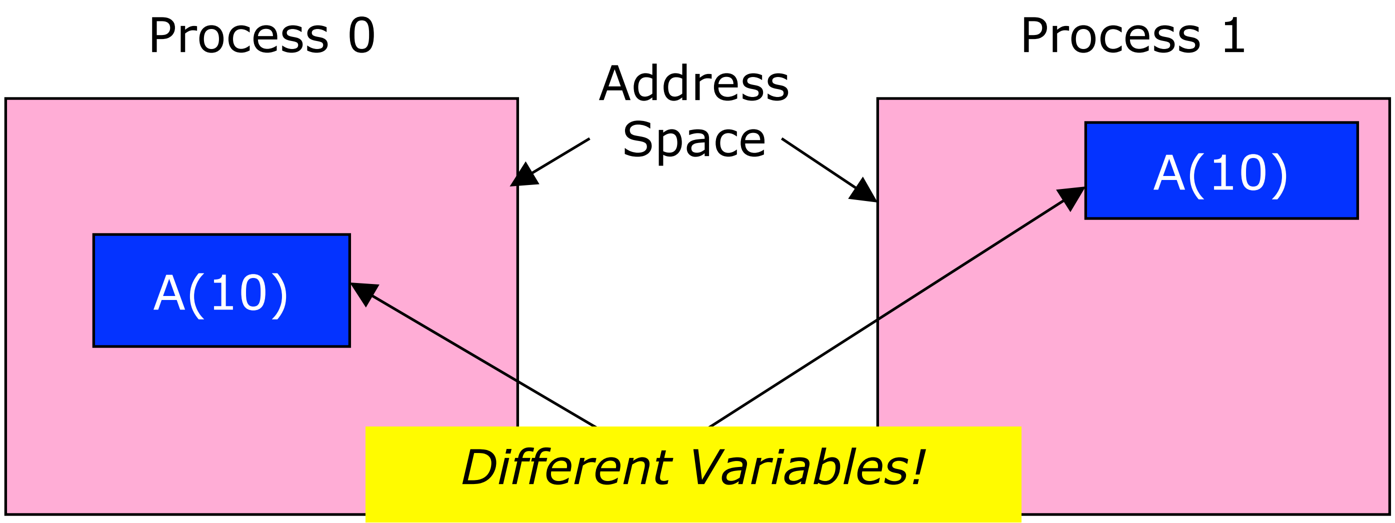
- Parallel programs consist of separate processes, each with its own address space
  - ♦ Programmer manages memory by placing data in a particular process
- Data sent explicitly between processes
  - ♦ Programmer manages memory motion
- Collective operations
  - ♦ On arbitrary set of processes
- Data distribution
  - ♦ Also managed by programmer
    - Message passing model doesn't get in the way
    - It doesn't help either

Types of Parallel Computing Models

- Data Parallel - the same instructions are carried out simultaneously on multiple data items (SIMD)
- Task Parallel - different instructions on different data (MIMD)
- SPMD (single program, multiple data) not synchronized at individual operation level
- SPMD is equivalent to MIMD since each MIMD program can be made SPMD (similarly for SIMD, but not in practical sense.)

Message passing (and MPI) is for MIMD/SPMD parallelism.

More on "Single Name Space"



- integer A(10)

  ...
  print *, A

- integer A(10)
  do i=1,10
    A(i) = i
  enddo
  ...

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.
    - MPI processes may have multiple threads
- Interprocess communication consists of
    - Synchronization
    - Movement of data from one process's address space to another's.

Programming With MPI

- MPI is a library
  - All operations are performed with routine calls
  - Basic definitions in
    - mpi.h for C
    - MPI or MPI_F08 module for Fortran
    - mpif.h for Fortran 77 (discouraged)
- First Program:
  - Create 4 processes in a simple MPI job
  - Write out process number
  - Write out some variables (illustrate separate name space)

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

Simple Program in Fortran

```
program main
use mpi
integer ierr, rank, size, I, provided
real A(10)
call MPI_Init_thread( MPI_THREAD_SINGLE, &
                       provided, ierr )
call MPI_Comm_size( MPI_COMM_WORLD, size, ierr )
call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
do i=1,10
    A(i) = i * rank
enddo
print *, 'My rank ', rank, ' of ', size
print *, 'Here are my values for A:'
print *, A
call MPI_Finalize( ierr )
end
```

Simple Program in C

```
#include "mpi.h"
int main(int argc, char *argv[])
{
  int rank, size, i, provided
  float A(10)
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE,
                          &provided);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for (i=0; i<10; i++)
    A[i] = i * rank;
  printf("My rank %d of %d\n", rank, size );
  printf("Here are my values for A\n");
  for (i=0; i<10; i++) printf("%f ", A[i]);
  printf("\n");
  MPI_Finalize();
}
```

Simple Program in C

```
#include "mpi.h"
int main(int argc, char *argv[])
{
  int rank, size, i, provided
  float A(10)
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE,
                              &provided);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for (i=0; i<10; i++)
    A[i] = i * rank;
  printf("My rank %d of %d\n", rank, size );
  printf("Here are my values for A\n");
  for (i=0; i<10; i++) printf("%f ", A[i]);
  printf("\n");
  MPI_Finalize();
}
```

Notes on Simple Program

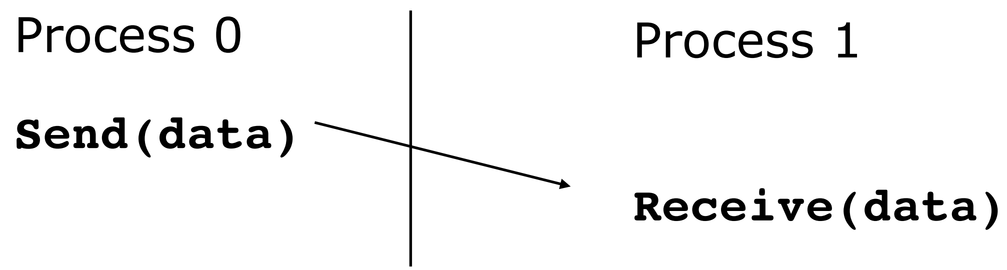
- All MPI programs begin with MPI_Init_thread and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or the MPI module (in Fortran) and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the `print` and `printf` statements
- I/O to standard output not part of MPI
  - output order undefined (may be interleaved by character, line, or blocks of characters)

Wait! What about MPI_Init?

- In MPI-1, MPI programs started with MPI_Init
  - MPI_Init(&argc, &argv) in C, MPI_INIT(ierr) in Fortran
- MPI-2 adds MPI_Init_thread so that programmer can request the level of *thread safety* required for the program
  - MPI_THREAD_SINGLE gives the same behavior as MPI_Init
- New programs should use MPI_Init_thread, and if more thread safety required, check on that (the provide arg).
  - Needed to use OpenMP with MPI

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.

MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive.
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes.

MPI Basic (Blocking) Send

MPI_SEND (start, count, datatype, dest, tag, comm)

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- 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.

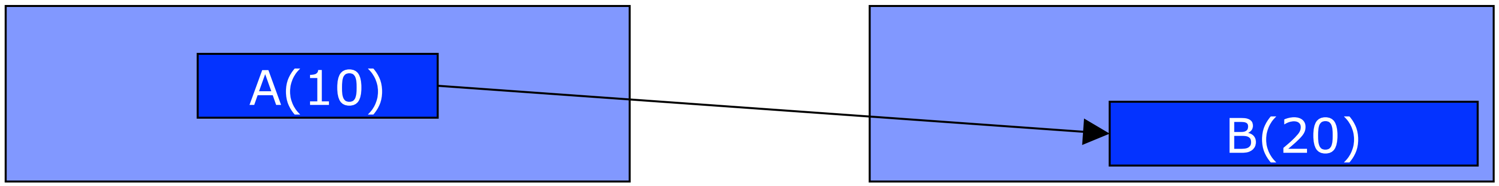
MPI Basic (Blocking) Receive

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

- Waits until a matching (on source and tag) message is received from the system, and the buffer can be used.
- source is rank in communicator specified by comm, or MPI_ANY_SOURCE.
- status contains further information
- Receiving fewer than count occurrences of datatype is OK, but receiving more is an error.

Send-Receive Summary

- Send to matching Receive



MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

- Datatype
  - Basic for heterogeneity
  - Derived for non-contiguous
- Contexts
  - Message safety for libraries
- Buffering
  - Robustness and correctness

Retrieving Further Information

- `Status` is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

Retrieving Further Information

- **`Status`** is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

Adding Communication

- Test yourself here. Take our original program and change it to do the following:
- Process 0 (i.e., the process with rank 0 from MPI_Comm_rank) sets the elements of A[i] to i, using a loop.
- Process 0 sends A to all other processes, one process at a time, using MPI_Send. The other processes receive A, using MPI_Recv.
  - The MPI datatype for "float" is MPI_FLOAT
  - You can ignore the status return in an MPI_Recv with MPI_STATUS_IGNORE
- The program prints rank, size, and the values of A on each process

One Answer to the Question in C (part 1)

```
#include "mpi.h"
int main(int argc, char *argv[])
{
  int rank, size, i, provided
  float A(10)
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE,
                          &provided);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

One Answer to the Question in C (part 2)

```
 if (rank == 0) {
    for (i=0; i<10; i++)
      A[i] = i;
    for (i=1, i<size; i++)
        MPI_Send(A, 10, MPI_FLOAT, i, 0,
                             MPI_COMM_WORLD);
 } else {
    MPI_Recv(A, 10, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
 }
 printf("My rank %d of %d\n", rank, size );
 printf("Here are my values for A\n");
 for (i=0; i<10; i++) printf("%f ", A[i]);
 printf("\n");
 MPI_Finalize();
}
```

Tags and Contexts

- In very early message passing systems, separation of messages was accomplished by use of tags, but
  - this requires libraries to be aware of tags used by other libraries.
  - this can be defeated by use of “wild card” tags.
- Contexts are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when a library sets up a communicator for its own use.
- User-defined tags still provided in MPI for user convenience in organizing application

Running MPI Programs

- The MPI Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
- **`mpiexec <args>`** is part of MPI, as a recommendation, but not a requirement, for implementors.
- For example, on Blue Waters, you'll need to use aprun and a batch script
    - Or do what I do – write a script that acts like mpiexec

Notes on C and Fortran

- C and Fortran bindings correspond closely
- In C:
  - mpi.h must be #included
  - MPI functions return error codes or **`MPI_SUCCESS`**
- In Fortran:
  - The mpi module should be included (use MPI); even better is the MPI_F08 module
  - Older programs may include the file mpif.h
  - Almost all MPI calls are to subroutines, with a place for the return code in the last argument.
- MPI-2 added and MPI-3 deleted a simple C++ binding

Error Handling

- By default, an error causes all processes to abort.
- The user can cause routines to return (with an error code) instead.
- A user can also write and install custom error handlers.
- Libraries can handle errors differently from applications.
  - MPI provides a way for each library to have its own error handler without changing the default behavior for other libraries or for the user's code

A Little More On Errors

- MPI has error *codes* and *classes*
  - MPI routines return error codes
  - Each code belongs to an error class
  - MPI defines the error *classes* but not codes
    - Except, all error classes are also error codes
- An MPI implementation can use error codes to return instance-specific information on the error
  - MPICH does this, providing more detailed and specific messages
- There are routines to convert an error code to text and to find the class for a code.

1867

PARALLEL@ILLINOIS

Timing MPI Programs

- The elapsed (wall-clock) time between two points in an MPI program can be computed using **MPI_Wtime**:

```
double t1, t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime();
printf( "time is %d\n", t2 - t1 );
```

- The value returned by a single call to **MPI_Wtime** has little value.
- The resolution of the timer is returned by **MPI_Wtick**
- Times in general are local, but an implementation might offer synchronized times.
  - For advanced users: see the MPI attribute **MPI_WTIME_IS_GLOBAL**.

Questions To Consider

- Find out how to compile and run MPI programs on your systems.
- MPI (both MPICH and Open MPI) can be installed on almost any machine, including many laptops. See if you can install on on your laptop.
- Add timing to the MPI programs in this lecture. Is the time taken by the communication operation what you expect?