CSCI 4850/5850 High-Performance Computing

Spring 2018

Tae-Hyuk (Ted) Ahn

Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
Learning Objectives

- Learn about point-to-point operation of MPI more in detail.
MPI Blocking Send & Recv

● C++ Syntax

MPI::COMM_WORLD.Send( data, count, datatype, dest, tag)
MPI::COMM_WORLD.Recv( data, count, datatype, dest, tag, status)

● Example:

```cpp
int tag1=0;
MPI::Status status;
MPI::COMM_WORLD.Send ( &value, 100, MPI::FLOAT, dest, tag1 );
MPI::COMM_WORLD.Recv ( &value, 100, MPI::FLOAT, MPI::ANY_SOURCE, tag1, status );
```
Return status

- The source or tag of a received message may not be known if wildcard values were used in the receive operation.
- Also, if multiple requests are completed by a single MPI function, a distinct error code may need to be returned for each request.
- The information is returned by the `status` argument of `MPI_RECV`. The type of status is MPI-defined.
- Status variables need to be explicitly allocated by the user, that is, they are not system objects.
Return status

- In C, status is a structure that contains three fields named `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`; the structure may contain additional fields. Thus, `status.MPI_SOURCE`, `status.MPI_TAG` and `status.MPI_ERROR` contain the source, tag, and error code, respectively, of the received message.

- In C++, the status object is handled through the following methods:

```cpp
int MPI::Status::Get_source() const
void MPI::Status::Set_source(int source)
int MPI::Status::Get_tag() const
void MPI::Status::Set_tag(int tag)
int MPI::Status::Get_error() const
void MPI::Status::Set_error(int error)
```
Lab with status

```c++
MPI::Status status;

if (myid == 0) {
    val = 100;
    MPI::COMM_WORLD.Send(&val, 1, MPI::INT, 1, 0);
}
// rank 1 receive message from rank 0
else if (myid == 1) {
    // MPI::COMM_WORLD.Recv(&val, 1, MPI::INT, 0, 0, status);
    MPI::COMM_WORLD.Recv(&val, 1, MPI::INT, MPI::ANY_SOURCE, MPI::ANY_TAG, status);
    // cout << "Process 1 received number " << val << " from process 0" << endl;
    cout << "Process " << myid << " received number " << val << " from process " << status.Get_source() << endl;
    int len_message = status.Get_count(MPI::INT);
    cout << "Length of message (count) is " << len_message << endl;
}
```
## P2P_vector.cpp

```cpp
#include <iostream>
#include <cstdlib>              // has exit(), etc.
#include <mpi.h>                // MPI header file
using namespace std;

int main(int argc, char **argv) {

    int i, dest, count, tag;
    float data[100], val[200];
    MPI::Status status;

    // Initialize MPI.
    MPI::Init(argc,argv);

    // get number of processes
    int nprocs = MPI::COMM_WORLD.Get_size();

    // get this process's number (ranges from 0 to nprocs - 1)
    int myid = MPI::COMM_WORLD.Get_rank();

    // vector send & recv: next page
    // Terminate MPI.
    MPI::Finalize();

    return EXIT_SUCCESS;
}
```
Lab: Send & Recv Vector

P2P_vector.cpp

// Process 0 expects to get up to 200 real values, from any source.
if (myid == 0) {
    tag = 55;
    MPI::COMM_WORLD.Recv(&val, 200, MPI::FLOAT, MPI::ANY_SOURCE, tag, status);
    cout << "P:" << myid << " Got data from process " << status.Get_source() << endl;
    count = status.Get_count(MPI::FLOAT);
    cout << "P:" << myid << " Got " << count << " elements." << endl;
}

// Process 1 sends 100 real values to process 0.
else if (myid == 1) {
    for (i = 0; i < 100; i++) {
        data[i] = i;
    }

    dest = 0;
    tag = 55;
    MPI::COMM_WORLD.Send(data, 100, MPI::FLOAT, dest, tag);
    cout << "P:" << myid << " Sent data to process " << dest << endl;
}

// Any other process is idle.
else {
    cout << "P:" << myid << " MPI has no work for me!" << endl;
}
MPI_Send & MPI_Recv

- MPI_Send and MPI_Recv are blocking! operation must be completed before jump to next instruction.
- **Asynchronous** communication: possible delay between Send and Receive, sent data could be buffered. Even if Send is completed, it doesn’t mean that message has already been received.
- Be cautious with **Deadlocks**: two processes waiting for a message that never come
MPI_Send & MPI_Recv : Deadlocks

● This code hangs!

```c
if( myrank == 0 ) {
    MPI::COMM_WORLD.Recv ( &value, 100, MPI::FLOAT, 1, tag, status );
    MPI::COMM_WORLD.Send ( &value, 100, MPI::FLOAT, 1, tag );
}
else if (myrank == 1) {
    MPI::COMM_WORLD.Recv ( &value, 100, MPI::FLOAT, 0, tag, status );
    MPI::COMM_WORLD.Send ( &value, 100, MPI::FLOAT, 0, tag );
}
```
Lab: Sum of the first N Integers

```
// serial solution
int main() {
    int N = 1000, sum = 0, i;
    for (i = 1; i <= N; i++) {
        sum = sum + i;
    }
    cout << "The sum from 1 to " << N << " is: " << sum;
}
```

- How to parallelize?
Magic Formula

- \( \text{start\_val} = N \times \frac{\text{myrank}}{\text{proc\_size}} + 1 \)
- \( \text{end\_val} = N \times \frac{(\text{myrank}+1)}{\text{proc\_size}} \)
- Process of rank 0 receive partial sums and add.

```cpp
if (myrank != 0) {
    MPI::COMM_WORLD.Send ( partial_sum, 1, MPI::INT, 0, tag );
}
else {
    for ( j=1; j<proc_size; j=j+1) {
        MPI::COMM_WORLD.Recv (temp_sum, 1, MPI::INT, 0, tag, status)
        partial_sum += temp_sum
    }
    if (myrank == 0) {
        cout << partial_sum << endl;
    }
```
Blocking vs. Non-blocking

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

- Non-blocking:
  - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
  - Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
  - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.
MPI Non-Blocking Send & Recv

- **C++ Syntax**
  
  ```cpp
  MPI::COMM_WORLD.Isend( data, count, datatype, dest, tag)
  MPI::COMM_WORLD.Irecv( data, count, datatype, dest, tag, status)
  Request::Wait(Status& status)
  Request::Wait()
  ```

- **Example:**
  ```cpp
  MPI::Request req;
  req = MPI::COMM_WORLD.Isend( val, count, MPI::INT, 1, tag)
  req.Wait();
  ```
MPI Blocking P2P

The threshold value (also called the “eager limit”) differs between systems.
MPI Non-Blocking P2P

MPI_ISEND (nonblocking standard send)
- size ≤ threshold

MPI_WAIT
- no delay even though message is not yet in user's buffer on receiving node

transfer to buffer on receiving node can be avoided if MPI_Irecv posted early enough

MPI_Irecv

MPI_WAIT
- no delay if MPI_WAIT is late enough
Detailed problem will be announced today night!

Consider the problem of computing in parallel the dot product of two float vectors \( x[N] \) and \( y[N] \) using \( p \) processes. Initially the data (the vectors \( x \) and \( y \)) are at the "master" processor \( P0 \). The computed dot product will be at \( P0 \), which will print the result. Consider all the elements of \( x \) and \( y \) to be equal to 1.0. The correct value of the dot product is \( N \).

1) Implement the parallel dot product using only point to point communication routines (send and recv functions).

2) Measure the execution time for \( N=100,1000,10000 \) and for \( p=2,4,8 \). You can use the function MPI::Wtime() or you can use the system "time" command at the master node.

Comment on how the cpu time scales with problem size \( N \) and with the number of processors \( p \).