Distributed Memory

- Each CPU has its own (local) memory

This needs to be fast for parallel scalability (e.g. Infiniband, Myrinet, etc.)
Hybrid Model

- Shared-memory within a node
- Distributed-memory across nodes

e.g. a compute node of the Hoffman2 cluster
Today’s Topics

- What is MPI
- Message passing basics
- Point to point communication
- Collective communication
- Derived data types
- Examples
MPI = **Message Passing Interface**

- API for distributed-memory programming
  - parallel code that runs across multiple computers (nodes)
- De facto industry standard
  - available on (almost) every parallel computer for scientific computing
- Use from C/C++, Fortran, Python, R, ...
- More than 200 routines
- Using only 10 routines are enough in many cases
  - Problem dependent
Clarification

- You can mix MPI and OpenMP in one program
- You *could* run multiple MPI processes on a single CPU
  - e.g. debug MPI codes on your laptop
  - An MPI job can span across multiple computer nodes (distributed memory)
- You *could* run multiple OpenMP threads on a single CPU
  - e.g. debug OpenMP codes on your laptop
MPI Facts

- High-quality implementation available for free
  - Easy to install one on your desktop/laptop
  - OpenMPI: http://www.open-mpi.org/

- Installation Steps
  - download the software
  - (assuming you already have C/C++/Fortran compilers)
  - On Mac or Linux: “configure, make, make install”
Communicator

- A group of processes
  - processes are numbered 0,1,.. to N-1
- Default communicator
  - MPI_COMM_WORLD
  - contains all processes
- Query functions:
  - How many processes in total?
    MPI_Comm_size(MPI_COMM_WORLD, &nproc)
  - What is my process ID?
    MPI_Comm_rank(MPI_COMM_WORLD, &rank)

...
Hello world (C)

```c
#include "mpi.h"    // MPI header file
#include <stdio.h>
main(int argc, char *argv[])
{
    int np, pid;
    MPI_Init(&argc, &argv);    // initialize MPI

    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    printf("N. of procs = %d, proc ID = %d\n", np, pid);

    MPI_Finalize();        // clean up
}
```
Hello world (Fortran)

```fortran
program hello
    Use mpi
    integer :: ierr,np,pid
    call mpi_init(ierr)
    call mpi_comm_size(MPI_COMM_WORLD,np,ierr)
    call mpi_comm_rank(MPI_COMM_WORLD,pid,ierr)
    write(*,'("np = ",i2,2x,"id = ",i2)') np,pid
    call mpi_finalize(ierr)
end program hello
```

When possible, use “use mpi”, instead of “include ‘mpif.h’”
Error checking

- Most MPI routines returns an error code
  - C routines as the function value
  - Fortran routines in the last argument

Examples

- Fortran
  ```
  MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
  ```
- C/C++
  ```
  int ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  ```
### MPI built-in data types

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>MPI_CHARACTER</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>MPI_INTEGER</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>MPI_REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>MPI_DOUBLE_PRECISION</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- See MPI standard for a complete list
- New types can be (recursively) created/defined
  - based on existing types
  - called “derived data type”
  - discussed later
Today’s Topics

- Message passing basics
- Point to point communication
- Collective communication
- Derived data types
- Examples
Point to point communication

process 0

MPI_Send(…, dest=1)

process 2

data

send

MPI_Recv(…, src=2)

process 1

process 3
MPI_Send: send data to another process

MPI_Send(buf, count, data_type, dest, tag, comm)

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>starting address of send buffer</td>
</tr>
<tr>
<td>count</td>
<td># of elements</td>
</tr>
<tr>
<td>data_type</td>
<td>data type of each send buffer element</td>
</tr>
<tr>
<td>dest</td>
<td>processor ID (rank) destination</td>
</tr>
<tr>
<td>tag</td>
<td>message tag</td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
</tr>
</tbody>
</table>

Examples:

C/C++: MPI_Send(&x,1,MPI_INT,5,0,MPI_COMM_WORLD);
Fortran: MPI_Send(x,1,MPI_INTEGER,5,0,MPI_COMM_WORLD,ierr)
# MPI_Recv: receive data from another process

The function `MPI_Recv` is used to receive data from another process. The function signature is:

```c
MPI_Recv(buf, count, datatype, src, tag, comm, status)
```

## Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>buf</code></td>
<td>Starting address of send buffer</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Number of elements</td>
</tr>
<tr>
<td><code>datatype</code></td>
<td>Data type of each send buffer element</td>
</tr>
<tr>
<td><code>src</code></td>
<td>Processor ID (rank) destination</td>
</tr>
<tr>
<td><code>tag</code></td>
<td>Message tag</td>
</tr>
<tr>
<td><code>comm</code></td>
<td>Communicator</td>
</tr>
<tr>
<td><code>status</code></td>
<td>Status object (an integer array in Fortran)</td>
</tr>
</tbody>
</table>

## Examples:

**C/C++:**

```c
MPI_Recv(&x, 1, MPI_INT, 5, 0, MPI_COMM_WORLD, &stat);
```

**Fortran:**

```fortran
MPI_Recv(x, 1, MPI_INTEGER, 5, 0, MPI_COMM_WORLD, stat, ierr)
```
Notes on MPI_Recv

- A message is received when the followings are matched:
  - Source (sending process ID/rank)
  - Tag
  - Communicator (e.g. MPI_COMM_WORLD)
- Wildcard values may be used:
  - MPI_ANY_TAG
    (don’t care what the tag value is)
  - MPI_ANY_SOURCE
    (don’t care where it comes from; always receive)
Send/recv example (C)

- Send an integer array f[N] from process 0 to process 1

```c
int f[N], src=0, dest=1;
MPI_Status status;
// ...
MPI_Comm_rank( MPI_COMM_WORLD, &rank);

if (rank == src)  // process “dest” ignores this
    MPI_Send(f, N, MPI_INT, dest, 0, MPI_COMM_WORLD);

if (rank == dest)  // process “src” ignores this
    MPI_Recv(f, N, MPI_INT, src, 0, MPI_COMM_WORLD, &status);
//...
```
Send/recv example (F90)

- Send an integer array f(1:N) from process 0 to process 1

```fortran
integer f(N), status(MPI_STATUS_SIZE), rank, src=0, dest=1,ierr
// ...
call MPI_Comm_rank( MPI_COMM_WORLD, rank,ierr);
if (rank == src) then  !process "dest" ignores this
    call MPI_Send(f, N, MPI_INT, dest, 0, MPI_COMM_WORLD,ierr)
end if
if (rank == dest) then  !process "src" ignores this
    call MPI_Recv(f, N, MPI_INT, src, 0, MPI_COMM_WORLD,
                  status,ierr)
end if
//...
```
Send/Recv example (cont’d)

- Before

<table>
<thead>
<tr>
<th>process 0 (send)</th>
<th>process 1 (recv)</th>
</tr>
</thead>
<tbody>
<tr>
<td>f[0]=0</td>
<td>f[0]=0</td>
</tr>
<tr>
<td>f[1]=1</td>
<td>f[1]=0</td>
</tr>
</tbody>
</table>

- After

<table>
<thead>
<tr>
<th>process 0 (send)</th>
<th>process 1 (recv)</th>
</tr>
</thead>
<tbody>
<tr>
<td>f[0]=0</td>
<td>f[0]=0</td>
</tr>
<tr>
<td>f[1]=1</td>
<td>f[1]=1</td>
</tr>
</tbody>
</table>
Blocking

- Function call does not return until the communication is complete
- MPI_Send and MPI_Recv are blocking calls
- Calling order matters
  - it is possible to wait indefinitely, called “deadlock”
  - improper ordering results in serialization (loss of performance)
Deadlock

- This code always works:

```c
MPI_Comm_rank(comm, &rank);

if (rank == 0) {
    MPI_Send(sendbuf, cnt, MPI_INT, 1, tag, comm);
    MPI_Recv(recvbuf, cnt, MPI_INT, 1, tag, comm, &stat);
} else {  // rank==1
    MPI_Recv(recvbuf, cnt, MPI_INT, 0, tag, comm, &stat);
    MPI_Send(sendbuf, cnt, MPI_INT, 0, tag, comm);
}
```
Deadlock

- This code deadlocks:

```c
MPI_Comm_rank(comm, &rank);

if (rank == 0) {
    MPI_Recv(recvbuf, cnt, MPI_INT, 1, tag, comm, &stat);
    MPI_Send(sendbuf, cnt, MPI_INT, 1, tag, comm);
} else { /* rank==1 */
    MPI_Recv(recvbuf, cnt, MPI_INT, 0, tag, comm, &stat);
    MPI_Send(sendbuf, cnt, MPI_INT, 0, tag, comm);
}
```

reason: MPI_Recv on process 0 waits indefinitely and never returns.
Non-blocking

- Function call returns immediately, without completing data transfer
  - Only “starts” the communication (without finishing)
  - MPI_Isend and MPI_Irecv
  - Need an additional mechanism to ensure transfer completion (MPI_Wait)
- Avoid deadlock
- Possibly higher performance
- Examples: MPI_Isend & MPI_Irecv
MPI_Isend

MPI_Isend(buf, count, datatype, dest, tag, comm, request)

- Similar to MPI_Send, except the last argument “request”
- Typical usage:

```c
MPI_Request request_X, request_Y;
MPI_Isend(..., &request_X);
MPI_Isend(..., &request_Y);

//... some ground-breaking computations ...

MPI_Wait(&request_X, ...);
MPI_Wait(&request_Y,...);
```
MPI_Irecv

MPI_Irecv(buf, count, datatype, src, tag, comm, request)

- Similar to MPI_Recv, except the last argument “request”
- Typical usage:

```
MPI_Request request_X, request_Y;
MPI_Irecv(..., &request_X);
MPI_Irecv(..., &request_Y);

//... more ground-breaking computations ...

MPI_Wait(&request_X, ...);
MPI_Wait(&request_Y,...);
```
Caution about MPI_Isend and MPI_Irecv

- The sending process should not access the send buffer until the send completes

```
MPI_Isend(data, ..., &request);
// ... some code
MPI_Wait(..., &request);
// ready to use data here
```

DO NOT write to “data” in this region

OK to use “data” from here on
MPI_Wait

MPI_Wait(MPI_Request, MPI_Status)

- Wait for an MPI_Isend/recv to complete
- Use the same “request” used in an earlier MPI_Isend or MPI_Irecv
- If they are multiple requests, one can use
  MPI_Waitall(count, request[], status[]);
  request[] and status[] are arrays.
Other variants of MPI Send/Recv

- MPI_Sendrecv
  - send and receive in one call
- Mixing blocking and non-blocking calls
  - e.g. MPI_Isend + MPI_Recv
- MPI_Bsend
  - buffered send
- MPI_Ibsend
- ... (see MPI standard for more)
Today’s Topics

- Message passing basics
  - communicators
  - data types
- Point to point communication
- Collective communication
- Derived data types
- Examples
Collective communication

- One to all
  - MPI_Bcast, MPI_Scatter

- All to one
  - MPI_Reduce, MPI_Gather

- All to all
  - MPI_Alltoall
MPI_Bcast

MPI_Bcast(buffer, count, datatype, root, comm )

Broadcasts a message from “root” process to all other processes in the same communicator
MPI_Bcast Example

- Broadcast 100 integers from process “3” to all other processes

```c/c++
MPI_Comm comm;
int array[100];
//...
MPI_Bcast( array, 100, MPI_INT, 3, comm);
```

```fortran
INTEGER comm
integer array(100)
//...
call MPI_Bcast( array, 100, MPI_INTEGER, 3, comm,ierr)
```
MPI_Gather & MPI_Scatter

MPI_Gather (sbuf, scnt, stype, rbuf, rcnt, rtype, root, comm )
MPI_Scatter(sbuf, scnt, stype, rbuf, rcnt, rtype, root, comm )

When gathering, make sure the root process has big enough memory to hold the data (especially when you scale up the problem size).
MPI_Gather Example

MPI_Comm comm;
int np, myid, sendarray[N], root;
double *rbuf;
MPI_Comm_size( comm, &np);  // # of processes
MPI_Comm_rank( comm, &myid);  // process ID
if (myid == root)  // allocate space on process root
    rbuf = new double [np*N];

MPI_Gather( sendarray, N, MPI_INT, rbuf, N, MPI_INT, root, comm);
Variations of MPI_Gather/Scatter

- Variable data size
  - MPI_Gatherv
  - MPI_Scatterv

- Gather + broadcast (in one call)
  - MPI_Allgather
  - MPI_Allgatherv
MPI_Alltoall

MPI_Alltoall( send_buf, send_count, send_data_type, recv_buf, recv_count, recv_data_type, comm)

The j-th block send_buf from process i is received by process j and is placed in the i-th block of rbuf:
**MPI_Reduce**

MPI_Reduce (send_buf, recv_buf, data_type, OP, root, comm)

- Apply operation OP to send_buf from all processes and return result in the recv_buf on process “root”.
- Some predefined operations:

<table>
<thead>
<tr>
<th>Operations (OP)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>maximum value</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>minimum value</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>products</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

(see MPI standard for more predefined reduce operations)
MPI_Reduce example

- Parallel vector inner product:

\[ a \leftarrow x \cdot y \]

```c
// loc_sum = local sum
float loc_sum = 0.0; // probably should use double
for (i = 0; i < N; i++)
    loc_sum += x[i] * y[i];

// sum = global sum
MPI_Reduce(&loc_sum, &sum, 1, MPI_FLOAT, MPI_SUM,
            root, MPI_COMM_WORLD);
```
Today’s Topics

- Message passing basics
  - communicators
  - data types
- Point to point communication
- Collective communication
- Derived data types
- Examples
Derived Data Type

- Define data objects of various sizes and shapes (memory layout)
- Example
  - The send and recv ends have same data size but different memory layouts
### Data Type Constructors

<table>
<thead>
<tr>
<th>Constructors</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contiguous</td>
<td>contiguous chunk of memory</td>
</tr>
<tr>
<td>Vector</td>
<td>strided vector</td>
</tr>
<tr>
<td>Hvector</td>
<td>strided vector in bytes</td>
</tr>
<tr>
<td>Indexed</td>
<td>variable displacement</td>
</tr>
<tr>
<td>Hindexed</td>
<td>variable displacement in bytes</td>
</tr>
<tr>
<td>Struct</td>
<td>fully general data type</td>
</tr>
</tbody>
</table>
MPI_Type_contiguous

MPI_Type_contiguous(count, old_type, newtype)
- Define a contiguous chunk of memory
- Example – a memory block of 10 integers

```c
int a[10];
MPI_Datatype intvec;
MPI_Type_contiguous(10, MPI_INT, &intvec);
MPI_Type_commit(&intvec);
MPI_Send(a, 1, intvec, ...); /* send 1 10-int vector */
```

is equivalent to

```c
MPI_Send(a, 10, MPI_INT,...); /* send 10 ints */
```
MPI_Type_vector

MPI_Type_vector(count, blocklen, stride, old_type, newtype)

To create a strided vector (i.e. with “holes”):

```c
MPI_Datatype yellow_vec;
MPI_Type_vector(3, 4, 6, MPI_FLOAT, &yellow_vec);
MPI_Type_commit(&yellow_vec);
```
Commit and Free

- A new type needs to be committed before use
  
  \texttt{MPI\_Type\_commit(datatype)}

- Once committed, it can be used many times

- To destroy a data type, freeing the memory:
  
  \texttt{MPI\_Type\_free(datatype)}

If you repeatedly (e.g. in iterations) create MPI types, make sure you free them when they are no longer in use. Otherwise you may have memory leak.
Examples

- Poisson equation
- Fast Fourier Transform (FFT)
Poisson equation (or any elliptic PDE)

\[
\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = R(x, y)
\]

Computational grid:
Poisson equation

Jacobi iterations (as an example)

\[ f_{i,j}^{k+1} = \frac{1}{4} (f_{i+1,j}^k + f_{i-1,j}^k + f_{i,j+1}^k + f_{i,j-1}^k) \]

One solution is to introduce “ghost points” (see next slide)
Ghost points

Redundant copy of data held on neighboring processes
Update ghost points in one iteration

- 2-step process

- Repeat for many iterations until convergence
Poisson solution

Dirichlet boundary conditions

\[ \phi(x, 1) = 1, \, \phi(x, 0) = \phi(0, y) = \phi(1, y) = 0 \]
“Parallel” FFT

\[ \hat{X}(k_x, k_y) = \sum \sum X(x, y) \exp^{-i(k_x x + k_y y)} \]

Doing multiple (sequential) FFT in parallel
Timing

- MPI_Wtime
  - elapsed wall-clock time in seconds
  - Note: wall-clock time is not CPU time
- Example

```c
double t1,t2;
t1 = MPI_Wtime();
//... some heavy work ...
t2 = MPI_Wtime();
printf("elapsed time = %f seconds\n", t2-t1);
Parallel
```
How to run an MPI program

- **Compile**
  - C: `mpicc foo.c`
  - C++: `mpicxx foo.cpp`
  - F90: `mpif90 foo.f90`

- **Run**
  - `mpiexec -n 4 [options] a.out`
    - The options in mpiexec are implementation dependent
    - Check out the user’s manual

*mpicc, mpicxx and mpif90 are sometimes called the MPI compilers (wrappers)*
Summary

- MPI for distributed-memory programming
  - works on shared-memory parallel computers too
- Communicator
  - a group of processes, numbered 0,1,…,to N-1
- Data Types
  - derived types can be defined based on built-in ones
- Point-to-point Communication
  - blocking (Send/Recv) and non-blocking (Isend/Irecv)
- Collective Communication
  - gather, scatter, alltoall
Online Resources

- MPI-1 standard

- MPI-2 standard

- MPI-3 standard
  http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf