Distributed Memory ...

- Programming Model
- MPI *
- Hardware Issues (network topologies, communication costs)

*thanks to David Bindel and Bill Gropp for a lot of today’s slides
Distributed Memory Model

- \( p \) processes, each with its own local memory to which it has exclusive access. (No global memory)
- All data must be explicitly partitioned and placed
- Exchange of data by explicit communication operations that transfers messages from between participating processors (there are point-to-point and global communication operations)
- Use barriers for coordination (no need for locks)
• Message Passing Interface - specification of message passing libraries.
  • Started with MPI Forum in 1992; (about 40 organizations). MPI-1 released in 1994; MPI-2 in 1997 (not universally implemented).
  • Before the standard there was PVM, P4, Parmacs, ..

• Bindings to C, C++, Fortran

• can be used behind the scenes to implement shared memory models on distributed architectures; can also be implemented on shared memory architectures
Basic Idea

- Usually programmed with SPMD model (single program, multiple data)
- In MPI-1 number of tasks is static - cannot dynamically spawn new tasks at runtime. Enhanced in MPI-2.
- No assumptions on type of interconnection network; all processors can send a message to any other processor.
- All parallelism explicit - programmer responsible for correctly identifying parallelism and implementing parallel algorithms
#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("Hello world from %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
Hello World

To compile: Need to load “MPI” wrappers in addition to the compiler modules (OpenMPI, MPICH,...)

```bash
module load openmpi/intel/1.3.3
```

To compile: `mpicc hello.c`

To run: need to tell how many processes you are requesting

```bash
mpiexec -n 10 a.out (mpirun -np 10 a.out)
```

```
Hello from 1 of 10
Hello from 2 of 10
Hello from 6 of 10
Hello from 8 of 10
Hello from 9 of 10
Hello from 0 of 10
Hello from 5 of 10
Hello from 3 of 10
Hello from 7 of 10
Hello from 4 of 10
```
Basic MPI Elements

- Note the include file `<mpi.h>` (mpif.h in Fortran)
- Every program must start with `MPI_Init` before other MPI calls, and end with `MPI_Finalize`. (`MPI_Init` can pass command line args to other processes).
- Each process has its own unique integer id assigned by system when the process initializes. Contiguous numbers starting at 0.
- Processes do their own work, interspersed with communication (sending and receiving of messages)
- MPI uses communicators to define which collection of processes may communicate with each other. Most MPI routines have the communicator as an argument. `MPI_COMM_WORLD` is a pre-defined communicator that includes all your MPI processes.
Sending and Receiving

Need to specify:

- What’s the data? Different machines use different encodings (e.g. endian-ness).
- For the sender, where to send it to?
- For the receiver, where to put the data, how much of it, where did it come from? Does the tag match?
- What does it mean to ”complete” a send/recv?
Sending and Receiving

- Not sufficient to specify (Address, Length) of message buffer - very restrictive. What if data not contiguous in memory? What if sending structures, not simple data types?

- MPI uses higher-level description of message buffers: (address, count, datatype) where datatype can be defined by user, in addition to pre-defined types such as MPI_REAL, MPI_INT, etc.

- MPI tags are integers that label messages to:
  - distinguish between different message types
  - screen messages with wrong tag
  - MPI_ANY_TAG is a wild card
Basic **blocking** point-to-point communication:

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

int MPI_Recv(void *buf, int count,
              MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status);
```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]){
    int numtasks, rank, dest, source, rc, count, tag=1;
    char inmsg, outmsg='x';
    MPI_Status stat;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (0 == rank){
        source = 1;
        dest = 1;
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &stat);
    }
    else if (1 == rank){
        source = 0;
        dest = 0;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &stat);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }

    rc = MPI_Get_count(&stat, MPI_CHAR, &count);
    printf("proc %d received %d chars from proc %d with tag %d\n",
            rank, count, stat.MPI_SOURCE, stat.MPI_TAG);

    MPI_Finalize();
}
MPI Send/Recv

- **Recv** ignores messages that don’t match source and tag. 
  \texttt{MPI\_ANY\_SOURCE} and \texttt{MPI\_ANY\_TAG} are wildcards.

- **Recv status** contains more info (tag, source, actual size of message - can be smaller but not larger than count):
  
  - \texttt{status.MPI\_SOURCE} - returns rank of sending process
  - \texttt{status.MPI\_TAG} - returns tag of message just received
  - \texttt{status.MPI\_ERROR} - returns error code
  - \texttt{int MPI\_Get\_count = (&status, datatype, &int)} returns number of elements received
Sending and Receiving

- Where do messages go if send operation has occurred, but receiver not yet ready? Should the sending process
  - copy to buffer?
    As long as space available to hold a copy why not?
  - wait?
    If no buffer space available don’t want computation to fail. Better to wait for matching receive. If cannot guarantee enough space to store a copy of arbitrarily large message, why not a policy that says never copy a message into internal storage? (If MPI implementation does copy the send buffer into internal storage, it buffers the data). But MPI_Send should not require buffering.
- same problem if multiple sends arrive at same receiver, which can only accept one at a time.
Sending and Receiving

1d finite difference example, each process only has part of array.

Need to exchange values at boundaries

Updating at \( i = \text{iend} \) requires \( u(\text{istart} + 1) \)
Updating at \( i = \text{istart} \) requires \( u(\text{istart} - 1) \)

Example with \( n = 15 \) interior points (plus boundaries)

Process 0 has \( \text{istart} = 1, \text{iend} = 5 \)
Process 1 has \( \text{istart} = 6, \text{iend} = 10 \)
Process 2 has \( \text{istart} = 11, \text{iend} = 15 \)

If everybody executes

\[
\text{MPI\_Send}(&u, 1, \text{MPI\_FLOAT}, (\text{myrank} + 1) \% \text{nProc}, 1, \text{MPI\_COMM\_WORLD})
\]
\[
\text{MPI\_Recv}(&u, 1, \text{MPI\_FLOAT}, (\text{myrank} - 1 + \text{nProc}) \% \text{nProc}, 1, \text{MPI\_COMM\_WORLD,})
\]

could deadlock if MPI\_Send is blocking w/o buffering
(or if recv’s come first)
Blocking Sending and Receiving

• Receive will block until a matching send. If one is never posted, it will hang indefinitely.

• Send may block until a matching receive. Or it may copy the message and return and MPI will transfer it in due course.

• Blocking send only “returns” after it is safe to modify your application buffer for reuse. Does not mean data was received - may be in system buffer.

• Blocking receive only “returns” after data has arrived and is ready for use by program.
Sending and Receiving

It is up to the MPI implementation (not MPI standard) to decide what to do with data. Typically a system buffer area is reserved to hold data in transit.

Need to make sure your program will work either way.

There are additional send and receive routines that specify other options.
Non-blocking Sending and Receiving

MPI_ISend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_COMM comm, MPI_Request *request)

MPI_IRecv

- Non-blocking sends and receives return almost immediately; they do not wait for any communication events to complete.

- Unsafe to modify your application buffer until requested operation actually performed. Must check wait routines. MPI_Waitall, MPI_Test,... using MPI_Request handle.

- Non-blocking communication can be used to overlap computation with communication, for better performance.
Sending and Receiving

- **MPI_SSend/MPI_SRecv** - synchronous (blocking) send and receive. Does not complete until receive has begun. (asynchronous communication has no coordination between sender and receiver).

- **MPI_BSend/MPI_BRecv** - buffered send and receive (you supply the buffer, must attach and detach it from MPI process)

- Any send can be matched with any receive.

MPI guarantees that two messages from sender to receiver with same tag will be delivered in the same order they were sent. No guarantee if more than two processors involved (unless you make it happen by matching the expected to/from processors and message tags).
Sending and Receiving

combine this common operation into one to avoid deadlock.

MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status);

Process 0 has istart = 1, iend = 5
Process 1 has istart = 6, iend = 10
Process 2 has istart = 11, iend = 15

MPI_Sendrecv(&u[istart], 1, MPI_INT, leftProc, 123, &u[iend+1], 1, MPI_INT, rightProc, 123, MPI_COMM, &status)
Collective Operations

- Unlike the send/recv examples above, which are one-to-one (or point-to-point) communication, collective operations involve all processes in communicator.

- Basic classes of collective operations:
  - Data movement (e.g. broadcast)
  - Collective Computation (e.g. reduce)
  - Synchronization (e.g. MPI_Barrier(comm))

- Collective operations are blocking (extended in MPI-2), can only be used with MPI predefined datatypes, not with derived types.
MPI Broadcast

MPI_Bcast(&buf, count, datatype, root, comm)

All processors must call MPI_Bcast with the same root value.
#include "mpi.h"
#include <stdio.h>

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

    int myRank, numProcs, n=-1;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    if (0 == myRank){
        printf("Enter n: ");
        fflush(stdout);
        scanf("%d", &n);
    }

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

    printf("proc %d has value n = %d\n",myRank,n);
    return 0;
}
MPI Scatter/Gather

P0 | A | B | C | D |
---|---|---|---|---|
P1 |   |   |   |   |
P2 |   |   |   |   |
P3 |   |   |   |   |

Scatter

P0 | A |
P1 | B |
P2 | C |
P3 | D |

Gather
MPI Allgather

P0
A

P1
B

P2
C

P3
D

A B C D

Allgather

P0
A B C D

P1
A B C D

P2
A B C D

P3
A B C D
Reduction operators can be min, max, sum, multiply, logical ops, max value and location ... Must be associative (commutative optional)
MPI Allreduce

Suppose each processor working on its own subdomain
Each computes max norm and L1 norm of a soln. update

MPI_Allreduce(&localMax, &globalMaxResid, 1,
            MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD)

MPI_Allreduce(&localL1, &globalL1Resid, 1,
            MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD)

MPI_Allreduce is equivalent to MPI_Reduce followed by MPI_Bcast from root.
MPI Scan

P0 A
P1 B
P2 C
P3 D

Scan

P0 A
P1 AB
P2 ABC
P3 ABCD
Communicators

- Processes form *groups*

- Identify process by its rank in group. A process can belong to many different groups.

- Messages sent in *contexts*, and must be received in the same context.
  
  context - assigned by system for each communicator; makes separate communication possible for libraries.

- Group + context = communicator

- Default is `MPI_COMM_WORLD` - includes all processes.

- Others (e.g. a *Cartesian “virtual topology”* - logical arrangement of processors) provided by MPI.
Some MPI Primitive Data Types

<table>
<thead>
<tr>
<th>MPI data type</th>
<th>C data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>8 bits</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>data packed with MPI_Pack()</td>
</tr>
</tbody>
</table>

MPI allows for user derived data types (contiguous, strided, or indexed arrays, arbitrary structures.)
MPI-2

• One-sided **RMA** (remote memory access) communication
  - potential for greater efficiency, easier programming.
  - Use ”windows” into memory to expose regions for access
  - Race conditions now possible.

• **Parallel I/O** like message passing but to file system not other processes.

• Allows for **dynamic number of processes** and **inter-communicators** (as opposed to intra-communicators)

• Cleaned up MPI-1
RMA

- Processors can designate portions of its address space as available to other processors for read/write operations (
  MPI\_Get, MPI\_Put, MPI\_Accumulate).

- RMA window objects created by collective window-creation fns. (MPI\_Win\_create must be called by all participants)

- Before accessing, call MPI\_Win\_fence (or other synchr. mechanisms) to start RMA access epoch; fence (like a barrier) separates local ops on window from remote ops

- RMA operations are no-blocking; separate synchronization needed to check completion. Call MPI\_Win\_fence again.