“There are three rules to follow when parallelizing large codes. Unfortunately, no one knows what these rules are.”

W. Somerset Maugham and Gary Montry
Before We Get Started…

- Last time
  - Start discussion of Message Passing Interface (MPI) standard
  - Mentioned the components that you need to install on your machine to run MPI code

- Today
  - Learn how to run an MPI executable on Newton
  - Point-to-Point Communication with MPI
  - Collective Communication in MPI
  - Wrap up discussion of Message Passing Interface (MPI) standard

- Other issues
  - Midterm Project: progress report due tonight
  - Syllabus has been updated, up-to-date info regarding deadlines, topics covered, etc.
Running on Local Machine

\texttt{mpiexec \textasciitilde n 4 MPI\textunderscore PI\textunderscore exe 13}

(number of processes command line argument (number of sub-intervals))

(run this in the directory where you have your executable)

Credit: Toby Heyn
Running on Newton (1 node)
Running on Newton (2 nodes)

Request exactly two nodes
Running on Newton (2 nodes)

Two cores per node

Use exactly two nodes
Running on Newton (2 nodes)

Note 4 processes on two nodes
(NEWTON04 and NEWTON06)
Debugging MPI Code
Under VS 2008, 32 Bit

- Set MPIRun Command to “C:\Program Files\Microsoft HPC Pack 2008 R2\Bin\mpiexec.exe” (or wherever your mpiexec.exe is stored)

- Set MPIShim Location to “C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\Remote Debugger\x86\mpishim.exe” (or wherever your mpishim.exe is stored)
Compiling MPI Code, Known Issue...

- Why do I get a compilation error "catastrophic error: #error directive: SEEK_SET is #defined but must not be for the C++ binding of MPI" when I compile C++ application?
  - Define the MPICH_IGNORE_CXX_SEEK macro at compilation stage to avoid this issue. For instance,
    - mpicc -DMPICH_IGNORE_CXX_SEEK

- Why?
  - There are name-space clashes between stdio.h and the MPI C++ binding. MPI standard requires SEEK_SET, SEEK_CUR, and SEEK_END names in the MPI namespace, but stdio.h defines them to integer values. To avoid this conflict make sure your application includes the mpi.h header file before stdio.h or iostream.h or undefine SEEK_SET, SEEK_CUR, and SEEK_END names before including mpi.h.
Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Closing Remarks
Example: Sending and Receiving
[Point-To-Point Communication]

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

int main(int argc, char **argv) {
    int i, my_rank, nprocs, x[4];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    if (my_rank == 0) { /* master */
        for (i=1;i<nprocs;i++)
            MPI_Send(x, 4, MPI_INT, i, 0, MPI_COMM_WORLD);
    }
    else { /* worker */
        MPI_Status status;
        MPI_Recv(x, 4, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        if ( my_rank==1 ) {
            printf("I am worker rank=%d/%d\n", my_rank,nprocs-1);
            printf("Here's the data I recently received: %d, %d, %d, %d", x[0], x[1], x[2], x[3]);
        }
    }

    MPI_Finalize();
    exit(0);
}
```

Contains material from Allan Snavely's presentation
Program Output

```
C:\Users\negrut\Temp\MPI\Debug>
C:\Users\negrut\Temp\MPI\Debug>
C:\Users\negrut\Temp\MPI\Debug> mpiexec.exe -n 8 MPI.exe
I am worker rank=1/8
Here’s the data I recently received: 42, 43, 44, 45
C:\Users\negrut\Temp\MPI\Debug>
```
MPI Basic Send/Receive
(Point-to-Point Communication)

- We need to fill in the details in

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(data)</td>
<td>Receive(data)</td>
</tr>
</tbody>
</table>

- 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?

Credit: Rusty Lusk
Point-to-Point Communication

- Data to be communicated is described by three attributes:
  - address
  - data type of the message
  - length of the message

- Involved processes are described by two attributes
  - communicator
  - rank

- A message is identified by a user chosen “tag” (integer)
MPI Communicators

MPI_COMM_WORLD

User-created Communicator

Credit: Allan Snavely
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 means of a specific tag, or not screened by using `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 data types.

Credit: Rusty Lusk
Tags and Contexts

- Separation of messages used to be 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

- MPI_Comm_split can be used to create new communicators
MPI Point-to-Point Communication

- Two **modes** of communication:
  - Blocking: Communication does not complete until the message has been received
  - Non-blocking: Completes as soon as the message is “on its way”, and hopefully it gets to destination

- MPI provides four **versions** for these two modes
  - synchronous, buffered, standard, ready

Contains material from Allan Snavely’s presentation
Synchronous/Buffered Sending in MPI

- **Synchronous with MPI_Ssend**
  - In synchronous mode, a send will not complete until a matching receive is posted.
  - Copy data to the network, wait for an acknowledgement
  - The sender has to wait for a receive to be posted
  - No buffering of data

- **Buffered with MPI_Bsend**
  - The send completes once the message has been buffered internally by MPI
  - Buffering incurs an extra memory copy
  - Does not require a matching receive to be posted
  - May cause buffer overflow if many bsends and no matching receives have been posted yet

Contains material from Allan Snavely’s presentation
Standard/Ready Send

- **Standard with MPI_Send**
  - Up to MPI to decide whether to do synchronous or buffered, for performance reasons
  - The rationale is that a correct MPI program should not rely on buffering to ensure correct execution

- **Ready with MPI_Rsend**
  - May be started *only* if the matching receive has been posted
  - Can be done efficiently on some systems as no hand-shaking is required

Contains material from Allan Snavely’s presentation
MPI_RECV

- There is only one MPI_Recv, which returns when the data has been received
  - Performs a blocking receive operation
  - C Function prototype:

```c
void MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status);
```

- buf - the initial address of the receive buffer (choice) (OUT)
- count - the number of elements to be received (integer) (IN)
- datatype - the data type of each receive buffer element (handle) (IN)
- source - the rank of the source task in comm or MPI_ANY_SOURCE (integer) (IN)
- tag - the message tag or MPI_ANY_TAG (positive integer) (IN)
- comm - the communicator (handle) (IN)
- status - the status object (INOUT).
Deadlock Issues

- Deadlock situations: appear when due to a certain sequence of commands the execution hangs

Contains material from Allan Snavely’s presentation
Deadlocking, Another Example

- Recall that MPI_Send is either synchronous or buffered
- Example, on a certain machine running MPICH v1.2.1:

```
PROCESS 0
...
MPI_Send()
MPI_Recv()
...

Deadlock
Data size > 127999 bytes
Data size < 128000 bytes

No Deadlock

PROCESS 1
...
MPI_Send()
MPI_Recv()
...
```
Avoiding Deadlocking

- Easy way to eliminate deadlock is to pair MPI_Ssend and MPI_Recv operations the right way:

  **PROCESS 0**
  
  ...  
  MPI_Ssend()  
  MPI_Recv()  
  ...  

  **PROCESS 1**
  
  ...  
  MPI_Recv()  
  MPI_Ssend()  
  ...

  ![No Deadlock](image)

- Conclusion: understand how the implementation works and what its pitfalls/limitations are
The Buffered Send

- This subroutine is a blocking buffered mode send operation

- It is a local operation. It does not depend on the occurrence of a matching receive in order to complete

- If a send operation is started and no matching receive is posted, the outgoing message is buffered to allow the send call to complete

- Return from an MPI_Bsend does not guarantee the message was sent

- Message may remain in the buffer until a matching receive is posted. MPI_Buffer_Detach() will block until all messages are received
The Buffered Send

[Cntd.]

- Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is not enough buffer space.

- The amount of buffer space needed to be safe depends on the expected peak of pending messages. The sum of the sizes of all of the pending messages at that point plus \((\text{MPI\_BSEND\_OVERHEAD} \times \text{number\_of\_messages})\) should be sufficient. You attach a buffer to a process using the following function:

```c
int MPI_Buffer_attach(void* buffer, int size);
```

- This subroutine provides MPI a buffer in the user's memory which is used for buffering outgoing messages. This buffer is used only by messages sent in buffered mode, and only one buffer is attached to a task at any time.

- MPI_Bsend adds overhead because it requires an extra memory-to-memory copy of the outgoing data.
Non-blocking Communications

- So far only blocking communication have been discussed:
  - The call returns whenever its operation is complete (MPI_SSEND returns once the message has been received, MPI_BSEND returns once the message has been buffered, etc.).

- MPI provides non-blocking communication: the call returns immediately and there is another call that can be used to check on completion.

- Rationale: Non-blocking calls let the sender/receiver do something useful while waiting for completion of the operation (without playing with threads, etc.).

Contains material from Allan Snavely’s presentation
Non-blocking Communication

- MPI_Issend, MPI_Ibsend, MPI_Isend, MPI_Irsend, MPI_Irecv

  ```c
  MPI_Request request;
  MPI_Isend(&x,1,MPI_INT,dest,tag,communicator,&request);
  MPI_Irecv(&x,1,MPI_INT,src,tag,communicator,&request);
  ```

- Functions to check on completion: MPI_Wait, MPI_Test, MPI_Waitany, MPI_Testany, MPI_Waitall, MPI_Testall, MPI_Waitsome, MPI_Testsome.

  ```c
  MPI_Status status;
  MPI_Wait(&request, &status) /* blocks */
  MPI_Test(&request, &status) /* doesn’t block */
  ```

Contains material from Allan Snively’s presentation
MPI Barrier

- Synchronization of the calling processes
  - The MPI_Barrier call blocks until all of the processes belonging to certain communicator have placed the call

- No data is exchanged

```c
... MPI_Barrier(MPI_COMM_WORLD) ...
...```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char **argv) {
    int my_rank, x;
    MPI_Status status;
    MPI_Request request;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

    if (my_rank == 0) { /* Process 0 */
        x=42;
        MPI_Isend(&x,1,MPI_INT,1,0,MPI_COMM_WORLD,&request);
        //.. // Do some work here, but don't change x (since might not have been sent yet)
    }

    else if (my_rank == 1) { /* Process 1 */
        int y;
        // Do some work here, you don't need x here
        MPI_Irecv(&x,1,MPI_INT,0,0,MPI_COMM_WORLD,&request);
        // Do some work here, don't count on x yet, might not be here yet
        //...
        // Now I need x, i have to have it
        MPI_Wait(&request, &status);
        y = x/6;
        printf("I am worker rank=%d. The value of y: %d\n", my_rank, y);
    }

    MPI_Finalize();
    return 0;
}
Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Closing Remarks
Collective Communication

- Operations that allow more than two processes to communicate simultaneously
  - MPI broadcast
  - MPI scatter
  - MPI gather
  - MPI reduce

- All these can be built using point-to-point communications, but typical MPI implementations have optimized them, and it’s a good idea to use them

- In all of these, all processes place the same call (in good SPMD fashion), although depending on the process, some arguments may not be used
MPI Broadcast

- One-to-many communication

- Note that multicast can be implemented via the use of communicators (i.e., to create processor groups)

```c
MPI_Bcast(x, 4, MPI_INT, 0, MPI_COMM_WORLD)
```

Rank of the root
MPI Scatter

- One-to-many communication
- Not sending the same message to all

\[ \text{MPI Scatter}(x, 100, \text{MPI_INT}, y, 100, \text{MPI_INT}, 0, \text{MPI_COMM_WORLD}) \]

Credit: Allan Snavely
MPI Gather

- Many-to-one communication
- Not sending the same message to the root

```
MPI_Scatter(x, 100, MPI_INT, y, 100, MPI_INT, 0, MPI_COMM_WORLD)
```

Credit: Allan Snively
Gather-to-all

- Many-to-many communication
- Each process sends the same message to all
- Different processes send different messages

```c
MPI_Allgather(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)
```

Credit: Allan Snively
All-to-all

- Many-to-many communication
- Each process sends a different message to each other process

```
MPI_Alltoall(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)
```

Credit: Allan Snively
MPI Reduction Operations

- Used to compute a result from data that is distributed across processes
  - Often what a user wants to do anyway - why not provide the functionality as a single API call rather than having people keep re-implementing the same things

- Example: you have an large array and the first process stores the first 1000 entries, the second stores the next 1000, etc.
  - You are interested in the sum of all the elements in the array

- Predefined operations:
  - MPI_MAX, MPI_MIN, MPI_SUM, etc.

- It is possible to have user-defined operations
MPI_Reduce, MPI_Allreduce

- **MPI_Reduce**: result is sent out to the root
  - The operation is applied element-wise for each element of the input arrays on each processor
- **MPI_Allreduce**: result is sent out to everyone

```c
... MPI_Reduce(x, r, 10, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD) ...
... MPI_Allreduce(x, r, 10, MPI_INT, MPI_MAX, MPI_COMM_WORLD) ...
```

Credit: Allan Snively
MPI_Reduce example

\[
\text{MPI\_Reduce}(\text{sbuf}, \text{rbuf}, 6, \text{MPI\_INT}, \text{MPI\_SUM}, 0, \text{MPI\_COMM\_WORLD})
\]
**MPI_Scan: Prefix reduction**

- Process i receives data reduced on process 0 to i.

<table>
<thead>
<tr>
<th></th>
<th>sbuf</th>
<th>rbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>3 4 2 8 12 1</td>
<td>P0 3 4 2 8 12 1</td>
</tr>
<tr>
<td>P1</td>
<td>5 2 5 1 7 11</td>
<td>P1 8 6 7 9 19 12</td>
</tr>
<tr>
<td>P2</td>
<td>2 4 4 10 4 5</td>
<td>P2 10 10 11 19 23 17</td>
</tr>
<tr>
<td>P3</td>
<td>1 6 9 3 1 1</td>
<td>P3 11 16 12 22 24 18</td>
</tr>
</tbody>
</table>

**`MPI_Scan(sbuf, rbuf, 6, MPI_INT, MPI_SUM, MPI_COMM_WORLD)`**

Credit: Allan Snavely
And more...

- Most broadcast operations come with a version that allows for a stride (so that blocks do not need to be contiguous)
  - MPI_Gatherv(), MPI_Scatterv(), MPI_Allgatherv(), MPI_Alltoallv()

- MPI_Reduce_scatter(): functionality equivalent to a reduce followed by a scatter

- All the above have been created as they are common in scientific applications and save code

- All details on the MPI Webpage

Credit: Allan Snively
Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Closing Remarks
In some MPI implementations there are more than 300 MPI functions. Not all of them part of the MPI standard though, some vendor specific.

Recall the 20/80 rule: six basic calls is probably what you need to implement a decent MPI code:

- MPI_Init, MPI_Comm_Size, MPI_Comm_Rank, MPI_Send, MPI_Recv, MPI_Finalize
The PETSc Library

- PETSc: Portable, Extensible Toolkit for Scientific Computation
  - One of the most successful libraries built on top of MPI
  - Intended for use in large-scale application projects,
  - Developed at Argonne National Lab (Barry Smith)

- PETSc provides routines for the parallel solution of systems of equations that arise from the discretization of PDEs
  - Linear systems
  - Nonlinear systems
  - Time evolution

- PETSc also provides routines for
  - Sparse matrix assembly
  - Distributed arrays
  - General scatter/gather (e.g., for unstructured grids)
Structure of PETSc

PETSc PDE Numerical Solution Utilities

- ODE Integrators
- Visualization
- Nonlinear Solvers, Unconstrained Minimization
- Interface
- Linear Solvers
- Preconditioners + Krylov Methods
- Grid Management
- Object-Oriented Matrices, Vectors, Indices
- Profiling Interface

Computation and Communication Kernels
- MPI, MPI-IO, BLAS, LAPACK
## PETSc Numerical Components

### Nonlinear Solvers
- Newton-based Methods
- Line Search
- Trust Region
- Other

### Time Steppers
- Euler
- Backward Euler
- Pseudo Time Stepping
- Other

### Krylov Subspace Methods
- GMRES
- CG
- CGS
- Bi-CG-STAB
- TFQMR
- Richardson
- Chebychev
- Other

### Preconditioners
- Additive Schwartz
- Block Jacobi
- Jacobi
- ILU
- ICC
- LU (Sequential only)
- Others

### Matrices
- Compressed Sparse Row (AIJ)
- Blocked Compressed Sparse Row (BAIJ)
- Block Diagonal (BDIAG)
- Dense
- Matrix-free
- Other

### Distributed Arrays

### Index Sets
- Indices
- Block Indices
- Stride
- Other

### Vectors
Flow Control for PDE Solution

Main Routine

Timestepping Solvers (TS)

Nonlinear Solvers (SNES)

Linear Solvers (SLES)

PETSc

Application Initialization

Function Evaluation

Jacobian Evaluation

Post-Processing

PC

KSP

User code  PETSc code