“As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications.”

Dave Parnas
Before We Get Started…

- Last time: Started the MPI segment of the course
  - Basic concepts related to computing on clusters of CPUs
  - Getting started on the Message Passing Interface (MPI) standard

- Today:
  - MPI practicalities
  - Point-to-point communication in MPI

- Miscellaneous
  - I provided feedback to all students who uploaded a project proposal
  - Email me if you uploaded a proposal yet haven’t heard from me

  Choose your Final Project presentation time slot - see post http://sbel.wisc.edu/Forum/viewtopic.php?f=15&t=508
// MPI_PI.cpp : Defines the entry point for the console application.
//
#include "mpi.h"
#include <math.h>
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
    int n, rank, size, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int namelen;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name(processor_name, &namelen);

    cout << "Hello from process " << rank << " of " << size << " on " << processor_name << endl;
}
if (rank == 0) {
    //cout << "Enter the number of intervals: (0 quits) \";
    //cin >> n;
    if (argc<2 || argc>2)
        n=0;
    else
        n=atoi(argv[1]);
}

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n>0) {
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = rank + 1; i <= n; i += size) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
        cout << "pi is approximately " << pi << ", Error is " << fabs(pi - PI25DT) << endl;
}

MPI_Finalize();
return 0;
Broadcast
[ MPI function used in Example]

- A one-to-many communication.
Collective Communications

- Collective communication routines are higher level routines
- Several processes are involved at a time
- May allow optimized internal implementations, e.g., tree based algorithms
  - Require $O(\log(N))$ time as opposed to $O(N)$ for naïve implementation
Reduction Operations

[MPI function used in Example]

- Combine data from several processes to produce a single result
Barriers

- Used implicitly or explicitly to synchronize processes
MPI, Practicalities
MPI on Euler
[Selecting MPI Distribution]

- What’s available: OpenMPI, MVAPICH, MVAPICH2
- OpenMPI is default on Euler
  - This is the only one we’ll support in ME759

- To load OpenMPI environment variables:
  - (This should have been done automatically)

```bash
$ module load mpi/gcc/openmpi
```
# Minimum version of CMake required.
cmake_minimum_required(VERSION 2.8)

# Set the name of your project
project(ME964-mpi)

# Include macros from the SBEL utils library
Include(ParallelUtils.cmake)

# Example MPI program
enable_mpi_support()
add_executable(integrate_mpi integrate_mpi.cpp)
target_link_libraries(integrate_mpi ${MPI_CXX_LIBRARIES})

find_package("MPI" REQUIRED)
list(APPEND CMAKE_C_COMPILE_FLAGS ${MPI_C_COMPILE_FLAGS})
list(APPEND CMAKE_C_LINK_FLAGS ${MPI_C_LINK_FLAGS})
include_directories(${MPI_C_INCLUDE_PATH})
list(APPEND CMAKE_CXX_COMPILE_FLAGS ${MPI_CXX_COMPILE_FLAGS})
list(APPEND CMAKE_CXX_LINK_FLAGS ${MPI_CXX_LINK_FLAGS})
include_directories(${MPI_CXX_INCLUDE_PATH})
Most MPI distributions provide wrapper scripts named \texttt{mpicc} or \texttt{mpicxx}:

- Adds in \texttt{-L}, \texttt{-l}, \texttt{-I}, etc. flags for MPI
- Passes any options to your native compiler (\texttt{gcc})
- Very similar to what \texttt{nvcc} did for CUDA – it’s a compile driver…

\begin{verbatim}
$ mpicxx -o integrate_mpi integrate_mpi.cpp
\end{verbatim}
Running MPI Code on Euler

\begin{verbatim}
mpiexec [-np #] [-machinefile file] <program> [<args>]
\end{verbatim}

- Number of processors. Optional if using a machinefile
- List of hostnames to use. Inside Torque, this file is at $PBS_NODEFILE
- Your program and its arguments

- The machinefile/nodefile is required for multi-node jobs with the version of OpenMPI on Euler
- `-np` will be set automatically from the machinefile; can select lower, but not higher
- See the `mpiexec` manpage for more options
Example

euler $ qsub -I -l nodes=8:ppn=4:amd,walltime=5:00
qsub: waiting for job 15246.euler to start
qsub: job 15246.euler ready

euler07 $ cd $PBS_O_WORKDIR
euler07 $ mpiexec -machinefile $PBS_NODEFILE ./integrate_mpi
32 32.121040666358297 in 0.998202s

euler07 $ mpiexec -np 16 -machinefile $PBS_NODEFILE ./integrate_mpi
16 32.121040666359455 in 1.524001s

euler07 $ mpiexec -np 8 -machinefile $PBS_NODEFILE ./integrate_mpi
8 32.121040666359136 in 2.171963s

euler07 $ mpiexec -np 4 -machinefile $PBS_NODEFILE ./integrate_mpi
4 32.121040666360585 in 4.600204s

euler07 $ mpiexec -np 2 -machinefile $PBS_NODEFILE ./integrate_mpi
2 32.121040666366788 in 7.615060s

euler07 $ ./integrate_mpi
1 32.121040666353437 in 15.163330s
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`. 
MPI Nuts and Bolts
Goals/Philosophy of MPI

- MPI’s prime goals
  - Provide a message-passing interface for parallel computing
  - Make source-code portability a reality
  - Provide a set of services (building blocks) that increase developer’s productivity

- The philosophy behind MPI:
  - Specify a standard and give vendors the freedom to go about its implementation
  - Standard should be hardware platform & OS agnostic – key for code portability
The Rank, as a Facilitator for Data and Work Distribution

- To communicate together MPI processes need identifiers: 
  \( \text{rank} = \text{identifying number} \)

- Work distribution decisions are based on the \textit{rank}
  - Helps establish which process works on which data
  - Just like we had thread and block indices in CUDA

![Diagram showing rank distribution](image-url)
Message Passing

- Messages are packets of data moving between different processes
- Necessary information for the message passing system:
  - sending process + receiving process \{ i.e., the two “ranks” \}
  - source location + destination location
  - source data type + destination data type
  - source data size + destination buffer size

![Diagram of message passing](communication-network-diagram)
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int my_rank;    /* rank of process */
    int p;         /* number of processes */
    int source;    /* rank of sender */
    int dest;      /* rank of receiver */
    int tag = 0;   /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p);        // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
Program Output

[negrut@euler CodeBits]$ mpiexec -np 8 ./greetingsMPI.exe
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
[negrut@euler CodeBits]$
Communicator  MPI_COMM_WORLD

- All processes of an MPI program are members of the default communicator MPI_COMM_WORLD

- MPI_COMM_WORLD is a predefined handle in mpi.h

- Each process has its own rank in a given communicator:
  - starting with 0
  - ending with (size-1)

- You can define a new communicator in case you find it useful
  - Use MPI_Comm_create call. Example creates the communicator DANS_COMM_WORLD

```
MPI_Comm_create(MPI_COMM_WORLD, new_group, &DANS_COMM_WORLD);
```
MPI_Comm_create

● Synopsis

```c
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm);
```

● Input Parameters
  ● `comm` - communicator (handle)
  ● `group` - subset of the family of processes making up the `comm` (handle)

● Output Parameter
  ● `comm_out` - new communicator (handle)
Point-to-Point Communication

- Simplest form of message passing

- One process sends a message to another process
  - MPI_Send
  - MPI_Recv

- Sends and receives can be
  - Blocking
  - Non-blocking
  - More on this shortly
Point-to-Point Communication

- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator, e.g., DANS_COMM_WORLD
- Processes are identified by their ranks in the communicator
The Data Type

- A message contains a number of elements of some particular data type

- MPI data types:
  - Basic data type
  - Derived data types – more on this later

- Data type handles are used to describe the type of the data moved around

Example: message with 5 integers

```
| 2345 | 654 | 96574 | -12 | 7676 |
```
<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</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_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

```
2345  654  96574  -12  7676
```

count=5

datatype=MPI_INT

int arr[5]
MPI_Send & MPI_Recv: The Eager and Rendezvous Flavors

- If you send small messages, the content of the buffer is sent to the receiving partner immediately
  - Operation happens in “eager mode”

- If you send a large amount of data, the sender function waits for the receiver to post a receive before sending the actual data of the message

- Why this eager-rendezvous dichotomy?
  - Because of the size of the data and the desire to have a safe implementation
  - If you send a small amount of data, the MPI implementation can buffer the content and actually carry out the transaction later on when the receiving process asks for data
    - Can’t play this trick if you attempt to move around a huge chunk of data though
MPI_Send & MPI_Recv: The Eager and Rendezvous Flavors

- **NOTE:** Each implementation of MPI has a default value (which might change at run time) beyond which a larger `MPI_Send` stops acting “eager”
  - The MPI standard doesn’t provide specifics
  - You don’t know how large is too large…

- Does it matter if it’s Eager or Rendezvous?
  - In fact it does, sometimes the code can hang – example to come

- Remark: In the message-passing paradigm for parallel programming you’ll always have to deal with the fact that the data that you send needs to “live” somewhere during the send-receive transaction
MPI_Send & MPI_Recv: Blocking vs. Non-blocking

- Moving away from the Eager vs. Rendezvous modes → they only concern the MPI_Send and MPI_Recv pair.

- Messages can be sent with other vehicles than plain vanilla MPI_Send.

- The class of send-receive operations can be classified based on whether they are blocking or non-blocking.
  
  - Blocking send: upon return from a send operation, you can modify the content of the buffer in which you stored data to be sent since a copy of the data has been sent.
  
  - Non-blocking: the send call returns immediately and there is no guarantee that the data has actually been transmitted upon return from send call.
    
    - Take home message: before you modify the content of the buffer you better make sure (through a MPI status call) that the send actually completed.
Example: Send & Receive

Non-blocking Alternative: MPI_Isend

- If non-blocking, the data “lives” in your buffer – that’s why it’s not safe to change it since you don’t know when transaction was closed
  - This typically realized through a MPI_Isend
    - “I” stands for “immediate”

- NOTE: there is another way for providing a buffer region but this alternative is blocking
  - Realized through MPI_Bsend
    - “B” stands for “buffered”
  - The problem here is that *you* need to provide this additional buffer that stages the transfer
    - Interesting question: how large should *that* staging buffer be?
  - Adding another twist to the story: if you keep posting non-blocking sends that are not matched by corresponding “MPI_Recv” operations, you are going to overflow this staging buffer
Example: Send & Receive Blocking Options (several of them)

- The plain vanilla MPI_Send & MPI_Recieve pair is blocking
  - It’s safe to modify the data buffer upon return

- The problem with plain vanilla:
  - 1: when sending large messages, there is no overlap of compute & data movement
    - This is what we strived for when using “streams” in CUDA
  - 2: if not done properly, the processes executing the MPI code can hang

- There are several other flavors of send/receive operations, to be discussed later, that can help with concerns 1 and 2 above
The Mechanics of P2P Communication: Sending a Message

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

- **buf** is the starting point of the message with **count** elements, each described with **datatype**
- **dest** is the rank of the destination process within the communicator **comm**
- **tag** is an additional nonnegative integer piggyback information, additionally transferred with the message
  - The **tag** can be used to distinguish between different messages
  - Rarely used
The Mechanics of P2P Communication: Receiving a Message

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

- `buf/count/datatype` describe the receive buffer
- Receiving the message sent by process with rank `source` in `comm`
- Only messages with matching `tag` are received
- Envelope information is returned in the `MPI_Status` object `status`
MPI_Recv: The Need for an MPI_Status Argument

- The MPI_Status object returned by the call settles a series of questions:
  - The receive call does not specify the size of an incoming message, but only an upper bound.
  - If multiple requests are completed by a single MPI function, a distinct error code may need to be returned for each request.
  - The source or tag of a received message may not be known if wildcard values were used in a receive operation.
The Mechanics of P2P Communication: Wildcarding

- Receiver can wildcard
  - To receive from any source – source = MPI_ANY_SOURCE
  - To receive from any tag – tag = MPI_ANY_TAG
  - Actual source and tag returned in receiver’s status argument
The Mechanics of P2P Communication: Communication Envelope

- Envelope information is returned from MPI_RECV in status.

- status.[MPI_SOURCE](status.[MPI_TAG](count via [MPI_Get_count()](int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count));

[ICHEC]
For a communication to succeed:

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Message data types must match
- Receiver’s buffer must be large enough
Blocking Type: Communication Modes

- Send communication modes:
  - Synchronous send → MPI_SSEND
  - Buffered [asynchronous] send → MPI_BSEND
  - Standard send → MPI_SEND
  - Ready send → MPI_RSEND

- Receiving all modes → MPI_RECV
## Cheat Sheet, Blocking Options

<table>
<thead>
<tr>
<th>Sender modes</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous send</td>
<td>Only completes when the receive has started</td>
<td></td>
</tr>
<tr>
<td><code>MPI_SSEND</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffered send</td>
<td>Always completes (unless an error occurs), irrespective of receiver</td>
<td>needs application-defined buffer to be declared with <code>MPI_BUFFER_ATTACH</code></td>
</tr>
<tr>
<td><code>MPI_BSEND</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td>Standard send</td>
<td></td>
</tr>
<tr>
<td><code>MPI_SEND</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ready send</td>
<td>May be started only if the matching receive is already posted!</td>
<td>avoid, might cause unforeseen problems...</td>
</tr>
<tr>
<td><code>MPI_RSEND</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>Completes when a the message (data) has arrived</td>
<td></td>
</tr>
<tr>
<td><code>MPI_RECV</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>