Quote of the Day

“Do what you can, with what you have, where you are.”
-- Theodore Roosevelt, US President
1958-1919
Before We Get Started

- Issues covered last time:
  - SSE and AVX quick overview
  - Parallel computing w/ MPI

- Today’s topics
  - Examples, MPI-enabled parallel computing
  - Point-to-point message passing

- Other issues:
  - HW08, due on Th, Nov. 12, at 11:59 PM
  - New assignment: HW09, due on Wd, Nov. 18, at 11:59 PM. Posted online today.
  - Final Project proposal due on 11/13 in Learn@UW
  - Second and last exam: coming up on 11/23 at 7:15 PM (Room TBA)
    - Review during regular lecture hours, on 11/23
MPI: A Second Example Application

- Example out of Pacheco’s book:
  - “Parallel Programming with MPI”
  - Good book, newer edition available

/* greetings.c -- greetings program
 *
 * Send a message from all processes with rank != 0 to process 0.
 *    Process 0 prints the messages received.
 *
 * Input: none.
 * Output: contents of messages received by process 0.
 *
 * See Chapter 3, pp. 41 & ff in PPMPI.
 */
#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]$
MPI, a Third Example: Approximating $\pi$

\[
\int_0^1 \frac{4}{1 + x^2} = 4 \cdot \tan^{-1}(1) = \pi
\]

Numerical Integration: Midpoint rule

\[
\int_0^1 \frac{4}{1 + x^2} \approx \sum_{i=1}^{n} \frac{1}{n} f \left((i - 0.5) \cdot h\right)
\]
MPI, a Third Example: Approximating $\pi$

- Use 4 MPI processes (rank 0 through 3)
- In the picture, $n=13$
- Sub-intervals are assigned to ranks in a round-robin manner
  - Rank 0: 1, 5, 9, 13
  - Rank 1: 2, 6, 10
  - Rank 2: 3, 7, 11
  - Rank 3: 4, 8, 12
- Each rank computes the area in its associated sub-intervals
- **MPIReduce** is used to sum the areas computed by each rank yielding final approximation to $\pi$
// Code for Approximating π

// 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 * (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
MPI, Practicalities
MPI on Euler

[Selecting MPI Distribution]

- What’s available: OpenMPI, MVAPICH, MVAPICH2
- OpenMPI is default on Euler

- To load OpenMPI environment variables:
  - Typically not needed, should be done automatically for you

$ module load openmpi
MPI on Euler:
[Compiling MPI Code by Hand]

- Most MPI distributions provide wrapper scripts named `mpicc` or `mpicxx`
  - Adds in `-L`, `-l`, `-I`, etc. flags for MPI
  - Passes any options to your native compiler (`gcc`)
  - Very similar to what `nvcc` did for CUDA – it’s a compile driver…

```
$ mpicxx -o integrate_mpi integrate_mpi.cpp
```
Running MPI Code on Euler

```
mpirun [-np #] [-machinefile file] <program> [<args>]
```

- `-np` will be set automatically by SLURM. Do not use it.
- `-machinefile` will be set automatically by SLURM. Do not use it.
- See the `mpirun` manpage for more options.

Number of processors. Inside SLURM, this is handled automatically.

List of hostnames to use. Inside SLURM, this is handled automatically.

Your program and its arguments.
### BEGINNING OF submit_mpi.sh SCRIPT ###

```
#!/bin/bash
#SBATCH -t 0-5:0:0
#SBATCH -o output.txt

cd $SLURM_SUBMIT_DIR
mpirun ./integrate_mpi
```

### END OF SCRIPT ###

```
euler $ sbatch -N 2 -n 4 submit_mpi.sh
euler $ cat output.txt
8 32.121040666358297 in 2.171963s

euler $ sbatch -N 2 -n 2 submit_mpi.sh
euler $ cat output.txt
4 32.121040666358297 in 4.600204s

euler $ sbatch -N 1 -n 1
euler $ cat output.txt
1 32.121040666358297 in 15.163330s
```
MPI Nuts and Bolts
The Rank & The Communicator
[As Facilitators for Data and Work Distribution]

- To communicate with each other MPI processes need identifiers: $\text{rank} = \text{identifying number}$

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

![Diagram showing communication network and program distribution based on rank]
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
#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 */
[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

\[
\text{MPI_Comm_create}(\text{MPI_COMM_WORLD}, \text{new\_group}, \&\text{DANS_COMM_WORLD});
\]
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**
  - `newcomm` - new communicator (handle)
[New Topic]

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

<p>| 2345 | 654  | 96574 | -12  | 7676 |</p>
<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:

```
count=5
int arr[5]
datatype=MPI_INT
```
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 runtime (daemon) can buffer the content and actually carry out the transaction later on when the receiving process asks for data
    - Can’t play though this trick if you attempt to move around a huge chunk of data
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 collection 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

A **blocking** alternative: MPI\_Send

- Several other blocking flavors exist, to be discussed later

- 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
Example: Send & Receive
A non-blocking alternative: MPI_Isend

- If non-blocking, the data “lives” in your array – 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 a *blocking* version that comes pretty close to MPI_Isend in terms of performance
  - Called MPI_Bsend
    - “B” stands for “buffered”

  - *You* need to provide an additional staging buffer that stages the data transfer
    - Interesting question: how large should *that* staging buffer be?

  - Adding another twist to the story: if you keep posting MPI_Bsend sends that are not matched by corresponding “MPI_Recv” operations, you are going to overflow this staging buffer
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 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
  - 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.
- \texttt{status.MPI\_SOURCE, status.MPI\_TAG, count} via \texttt{MPI\_Get\_count()}

```c
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count);
```

For a communication to complete fine:

- 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