ME964
High Performance Computing for Engineering Applications

Parallel Computing with MPI: Introduction
March 29, 2012

“The empires of the future are the empires of the mind.”
-Winston Churchill
Before We Get Started…

- Last lecture
  - Wrap up GPU programming with **thrust**
  - GPU computing with compiler directives
  - Brief overview of CUDA ecosystem
  - GPU programming with CUDA wrap-up

- Today
  - Short discussion, what comes next
  - Message Passing Interface (MPI) standard
  - Learn how to run an MPI executable on Euler

- Other issues
  - Please take care of the reading assignments, they are related to CUDA programming and optimization
What Comes Next

- After Spring Break
  - Three more lectures on MPI
  - Three lectures on OpenMP
  - One lecture on parallel design patterns
  - Last week of class: Two guest lectures, both on CUDA
    - CUDA Optimization, with Hammad Mazhar – Mech. Eng. grad student
    - CUDA for image processing, with Brian David – Medical Physics Grad student

- Midterm exam on April 17
  - Andrew and the TA will proctor it – Dan is out of town

- Assignment 9 due on Sunday, 11:59 PM
- Assignment 10 posted this weekend, due on 04/12
Acknowledgments

- Parts of MPI material covered draws on a set of slides made available by the Irish Centre for High-End Computing (ICHEC) - [www.ichec.ie](http://www.ichec.ie)
  - These slides will contain “ICHEC” at the bottom
  - In turn, the ICHEC material was based on the MPI course developed by Rolf Rabenseifner at the High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart in collaboration with the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh

- Individual or institutions are acknowledged at the bottom of the slide, like [A. Jacobs]→
Shared Memory Systems

- Memory resources are shared among processors
- Relatively easy to program for since there is a single unified memory space
- Scales poorly with system size due to the need for cache coherence

Example:
- Symmetric Multi-Processors (SMP)
  - Each processor has equal access to RAM

Traditionally, this represents the hardware setup that supports OpenMP-enabled parallel computing
Distributed Memory Systems

- Individual nodes consist of a CPU, RAM, and a network interface
  - A hard disk is typically not necessary; mass storage can be supplied using NFS

- Information is passed between nodes using the network

- No cache coherence and no need for special cache coherency hardware

- More difficult to write programs for distributed memory systems since the programmer must keep track of memory usage

- Traditionally, this represents the hardware setup that supports MPI-enabled parallel computing

[A. Jacobs]→
Overview of Large Multiprocessor Hardware Configurations

Larger multiprocessors

- Shared address space
  - Symmetric shared memory (SMP)
    - Examples: IBM eserver, SUN Sunfire
  - Distributed shared memory (DSM)
- Distributed address space
  - Commodity clusters: Beowulf and others
  - Custom cluster

- Cache coherent: ccNUMA
  - SGI Origin/Altix
- Noncache coherent: Cray T3E, X1

- Uniform cluster: IBM BlueGene
- Constellation cluster of DSMs or SMPs
  - SGI Altix, ASC Purple

© 2007 Elsevier, Inc. All rights reserved.
Euler
~ Hardware Configurations ~

File Server Architecture
- CPU: Intel Xeon 5620
- RAM: 16 GB DDR3
- Infiniband HCA
- RAID 6
- 24x 2TB Hard Disks

CPU/GPU Node Architecture
- CPU 0: Intel Xeon 5520
- CPU 1: Intel Xeon 5520
- RAM: 48 GB DDR3
- Hard Disk
- Infiniband HCA
- GPU 0
- GPU 1
- GPU 2
- GPU 3
- GTX480
- 3.5GB RAM
- 448 Cores
- PCIe x16 2.0

AMD Node Architecture
- CPU 0: AMD Opteron 6276
- CPU 1: AMD Opteron 6276
- RAM: 128 GB DDR3
- Infiniband HCA
- SSD

Legend, Connection Type:
- Gigabit Ethernet
- 4x QDR Infiniband

Remote Collaborators
- Internal Users
- Internet
- Gigabit Ethernet Switch
- 4x QDR Infiniband Switch
- CPU/GPU Node 1
- CPU/GPU Node 2
- CPU/GPU Node 14
- AMD Node 1

File Server

Head Node
Hardware Relevant in the Context of MPI
Three Components of Euler that are Important

- **CPU**: Intel Xeon E5520 Nehalem 2.26GHz
  - Quad-Core Processor
  - 4 x 256KB L2 Cache
  - 8MB L3 Cache
  - LGA 1366 (Intel CPU Socket B)
  - 80W

- **HCA**: 40Gbps Mellanox MHQH19B-XTR Infiniband interconnect
  - Pretty large bandwidth compared to PCI-e X 16, yet the latency is poor
  - This is critical factor in a cluster

- **Switch**: QLogic Switch
The same program is launched for execution independently on a collection of cores.

Each core executes the program.

What differentiates processes is their rank: processes with different ranks do different things ("branching based on the process rank")

Very similar to GPU computing, where one thread did work based on its thread index.
The Message-Passing Model

- One starts many process on different cores but on each core the process is spawned by launching the same program
  - Process [in ME964]: a program counter and address space

- Message passing enables communication among processes that have separate address spaces

- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process’s address space to another’s

- Execution paradigm embraced in MPI: Single Program Multiple Data (SPMD)
The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**
  - A processor may run many processes

- **Message-Passing Programming Paradigm**
  - Distributed memory
  - Parallel processors
  - Communication network
Our View: A process is a program performing a task on a processor.

Each processor/process in a message passing program runs a instance/copy of a program:

- Written in a conventional sequential language, e.g., C or Fortran,
- The variables of each sub-program have the same name but different locations (distributed memory) and different data!
- Communicate via special send & receive routines (message passing)
Why Care about MPI?

- Today, MPI is what makes the vast majority of the supercomputers tick at TFlops and PFlops rates

- Examples of architectures relying on MPI:
  - IBM Blue Gene L/P/Q
  - Cray supercomputers (Jaguar, etc.)

- MPI known for portability and performance

- MPI has FORTRAN, C, and C++ bindings
MPI is a Standard

- MPI is an API for parallel programming on distributed memory systems. Specifies a set of operations, but says nothing about the implementation
  - MPI is a standard

- Popular because it many vendors support (implemented) it, therefore code that implements MPI-based parallelism is very portable

- Probably the most common implementation: MPICH
  - The CH comes from Chameleon, the portability layer used in the original MPICH to provide portability to the existing message-passing systems
  - First MPICH implementation overseen at Argonne National Lab by Gropp and Lusk
MPI Forum

- MPI-1 Forum
  - First message-passing interface standard.
  - Sixty people from forty different organizations.
  - Users and vendors represented, from US and Europe.
  - Two-year process of proposals, meetings and review.
  - MPI 1.0 — June, 1994.
  - MPI 1.1 — June 12, 1995.

- MPI-2 Forum July 18, 1997
Where Can We Use Message Passing?

- Message passing can be used wherever it is possible for processes to exchange messages:
  - Distributed memory systems
  - Networks of Workstations
  - Even on shared memory systems
CUDA vs. MPI

- When would you use CPU/GPU computing and when would you use MPI-based parallel programming?

  - Use CPU/GPU
    - If your data fits the memory constraints associated with GPU computing
    - You have parallelism at a fine grain so that you the SIMD paradigm applies
    - Example:
      - Image processing

  - Use MPI-enabled parallel programming
    - If you have a very large problem, with a lot of data that needs to be spread out across several machines
    - Example:
      - Solving large heterogeneous multi-physics problems

- In large scale computing the future likely to belong to heterogeneous architecture
  - A collection of CPU cores that communicate through MPI, each or which farming out work to an accelerator (GPU)
#include "mpi.h"
#include <iostream>

int main(int argc, char **argv) {
    int my_rank, n;
    char hostname[128];

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

    gethostname(hostname, 128);
    if (my_rank == 0) {
        /* master */
        printf("I am the master: %s\n", hostname);
    } else {
        /* worker */
        printf("I am a worker: %s (rank=%d/%d)\n", hostname, my_rank, n-1);
    }

    MPI_Finalize();
    return 0;
}
Program Output

```bash
[negrut@euler04 CodeBits]$ mpiexec -np 8 ./a.out
I am a worker: euler04 (rank=1/7)
I am a worker: euler04 (rank=5/7)
I am a worker: euler04 (rank=6/7)
I am a worker: euler04 (rank=3/7)
I am a worker: euler04 (rank=4/7)
I am the master: euler04
I am a worker: euler04 (rank=2/7)
I am a worker: euler04 (rank=7/7)
[negrut@euler04 CodeBits]$
[negrut@euler04 CodeBits]$
```
MPI: A Second Example Application

- Example out of Pacheco’s book:
  - “Parallel Programming with MPI”
  - Good book, on reserve at Wendt

```c
/* 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
", 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 on Euler

[Selecting MPI Distribution]

- MVAPICH, MVAPICH2, OpenMPI available
- OpenMPI is default on Euler
  - This is the only one we’ll support in ME964 this year

- 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(SBELUtils.cmake)

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

With the template

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})

Without the template
Replaces include(SBELUtils.cmake) and enable_mpi_support() above
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`)

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

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

- **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

[A. Seidl]→
Example

```bash
testenv euler $ qsub -I -l nodes=8:ppn=4,walltime=5:00
testenv qsub: waiting for job 15246.euler to start
testenv qsub: job 15246.euler ready
testenv euler07 $ cd $PBS_O_WORKDIR
testenv euler07 $ mpiexec -machinefile $PBS_NODEFILE ./integrate_mpi
32 32.121040666358297 in 0.998202s
testenv euler07 $ mpiexec -np 16 -machinefile $PBS_NODEFILE ./integrate_mpi
16 32.121040666359455 in 1.524001s
testenv euler07 $ mpiexec -np 8 -machinefile $PBS_NODEFILE ./integrate_mpi
8 32.121040666359136 in 2.171963s
testenv euler07 $ mpiexec -np 4 -machinefile $PBS_NODEFILE ./integrate_mpi
4 32.121040666360585 in 4.600204s
testenv euler07 $ mpiexec -np 2 -machinefile $PBS_NODEFILE ./integrate_mpi
2 32.121040666366788 in 7.615060s
testenv 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: A Message-Passing Interface Standard** (1.1, June 12, 1995)

• **MPI-2: Extensions to the Message-Passing Interface** (July 18, 1997)


• **Parallel Programming with MPI**, Peter S. Pacheco, Morgan Kaufmann Publishers, 1997 - very good introduction.

• **Parallel Programming with MPI**, Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti. Training handbook from EPCC which can be used together with these slides
  
Example: Approximating $\pi$

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

Numerical Integration: Midpoint rule

$$\int_0^1 \frac{4}{1 + x^2} \approx \sum_{i=1}^{n} \frac{1}{n} f\left((i - 0.5) \cdot h\right)$$
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, giving final approximation to $\pi$
Code for Approximating $\pi$

```cpp
// 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;
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
Broadcast

- A one-to-many communication.
Reduction Operations

- Combine data from several processes to produce a single result

\[ \text{sum} = ? \]

\[ \begin{align*}
200 & \\
15 & \\
10 & \\
300 & \\
30 & \\
\end{align*} \]
Barriers

- Used to synchronize processes
MPI Nuts and Bolts
Goals and Scope of MPI

● MPI’s prime goals
  ● To provide a message-passing interface
  ● To provide source-code portability
  ● To allow efficient implementations

● It also offers:
  ● A great deal of functionality
    ● Algorithms, various communication modes, ways to package data for communication, etc.
MPI Implementations

- MPI implementations available for free
  - Appleseed (UCLA)
  - CRI/EPCC (Edinburgh Parallel Computing Centre)
  - LAM/MPI (Indiana University)
  - MPI for UNICOS Systems (SGI)
  - MPI-FM (University of Illinois); for Myrinet
  - MPICH (ANL)
  - MVAPICH (Infiniband)
  - SGI Message Passing Toolkit
  - OpenMPI

- NOTE: MPI is available in Microsoft Visual Studio
  - The implementation offered by VS is the MPICH one

- A detailed list of MPI implementations with features can be found at http://www.lam-mpi.org/mpi/implementations/
Data and Work Distribution

- To communicate together mpi-processes need identifiers: \( rank = \) identifying number

- All distribution decisions are based on the \( rank \)
  - Helps establish which process works on which data
  - Just like we had thread and block IDs in CUDA
Message Passing

- Messages are packets of data moving between different processes
- Necessary information for the message passing system:
  - sending process + receiving process \( \{ \text{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 system](attachment://message_passing_diagram.png)
Point-to-Point Communication

- Simplest form of message passing

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

- Sends and receives can be
  - Blocking
  - Non-blocking
  - More on this shortly
MPI_Send & MPI_Recieve: 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, you (the sender) wait 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 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 place a send with 10 GB of data :-(
MPI_Send & MPI_Recvieve: Blocking vs. Non-blocking

- NOTE: Each implementation of MPI has a default value (which might change during 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 it too large…

- 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 before it is received and the send-receive transaction completes

- This is where the blocking & non-blocking issue comes into play
  - 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 the data has been sent
    - The data “lives” in your buffer, so problem solved
  - 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: but before you modify the content of the buffer you better make sure (through a MPI status call) that the send actually completed
MPI_Send & MPI_Recieve: Blocking vs. Non-blocking

- 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”

- However, there is a second way of you providing a buffer region, where your sent message is buffered so that the fact that it’s non-blocking is still safe
  - This typically 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_Receive” operations, you are going to overflow this staging buffer
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 problem can hang

There are several other flavors of send/receive operations, to be discussed later, that can help with concerns 1 and 2 above