“A state-of-the-art calculation requires 100 hours of CPU time on the state-of-the-art computer, independent of the decade.” -- Edward Teller
Before We Get Started…

- Last lecture
  - CUDA Optimization Tips
  - Marked the end of the GPU Computing segment of the course
  - Summary of important dates

- Today
  - Start discussion of Message Passing Interface (MPI) standard
    - Discussion to be wrapped up in one more lecture
  - Learn how to run an MPI executable on Newton

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

Includes material from Adam Jacobs presentation
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

Includes material from Adam Jacobs presentation
Overview of Large Multiprocessor Hardware Configurations

Shared address space
- Symmetric shared memory (SMP)
  Examples: IBM eserver, SUN Sunfire

Distributed address space
- Distributed shared memory (DSM)
- Commodity clusters: Beowulf and others
- Custom cluster
  - Cache coherent: ccNUMA
    - SGI Origin/Altix
  - Noncache coherent: Cray T3E, X1
  - Newton

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

© 2007 Elsevier, Inc. All rights reserved.
Hardware Relevant in the Context of MPI
Three Components of Newton 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-Ex16, yet the latency is poor
  - This is critical factor in a cluster

- **Switch**: QLogic Switch
MPI: The 30,000 Feet Perspective

- A pool of processes is started on a collection of cores
- The code executes on the CPU core (no direct role for the GPU)
- The approach is characteristic of the SPMD (Single Program Multiple Data) computing paradigm
- 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
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

- 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 Bill Gropp and Rusty Lusk
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/](http://www.lam-mpi.org/mpi/implementations/)

Includes material from Adam Jacobs presentation
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
      - The typical application: nuclear physics (DOE’s playground)

- 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)
The Message-Passing Model

- A *process* is (traditionally) a program counter and address space
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space
- Message passing is for communication among processes, which have separate address spaces
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process’s address space to another’s
A First MPI Program

```c
#include "mpi.h"
#include <iostream>
#include <winsock2.h>

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

Includes material from Allan Snively presentation
Program Output

```
C:\Users\negrut\Temp\MPI\Debug>mpiexec.exe -n 4 MPI.exe
I am a worker: teddy (rank=1/3)
I am a worker: teddy (rank=3/3)
I am a worker: teddy (rank=2/3)
I am the master: teddy
```

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.
*/
```
MPI: A Second Example Application

[Cntd.]

```
#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
MPI can be thought of as a small specification, because any complete implementation need only provide the following operations:

- MPI_INIT
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_SEND
- MPI_RECV
- MPI_FINALIZE
MPI Data Types

MPI specifies data types explicitly in the message. This is needed since you might have a cluster of heterogeneous machines ⇒ word sizes and data formats may differ.

- MPI_INT
- MPI_FLOAT
- MPI_BYTE
- MPI_CHAR
- etc…
Example: Approximating $\pi$

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

Numerical Integration: Midpoint rule

$$\int_0^1 \frac{4}{1 + x^2} \approx \sum_{i=1}^{n} \frac{1}{n} f((i - 0.5) \cdot h)$$
Example: Approximating $\pi$

- Use 4 MPI process (rank 0 through 3)
- Here, $n=13$
- 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
- $\text{MPI}_\text{Reduce}$ is used to sum the areas computed by each rank, giving final approximation to $\pi$

Credit: Toby Heyn
// 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;
Compiling/Running MPI under Windows with VS 2008

- On a Windows machine with VS2008, in this order, do the following:
  - Download/Install HPC Pack 2008 R2 Client Utilities Redistributable Package with Service Pack 1
  - Download/Install HPC Pack 2008 R2 SDK with Service Pack 1
  - Download/Install HPC Pack 2008 R2 MS-MPI Redistributable Package with Service Pack 1
Compiling Code

- Compile like any other project in MS Visual Studio

- Additional Include Directories:
  - C:\Program Files\Microsoft HPC Pack 2008 R2\Inc

- Additional Library Directories:
  - C:\Program Files\Microsoft HPC Pack 2008 R2\Lib\i386

- Additional Dependencies:
  - msmpi.lib

- Compile in Release mode
  - MPI_PI.exe