Parallel Computing with the Message Passing Interface (MPI)
October 30, 2013

“The best things in life aren’t things.”
-- Art Buchwald, Pulitzer Prize winner
Before We Get Started…

- Last time: OpenMP
  - reduce operations in OpenMP
  - Closing comments on the OpenMP API

- Today:
  - Basic concepts related to computing on clusters of CPUs
  - Getting started on the Message Passing Interface (MPI) standard

- Miscellaneous
  - HW posted online, due on Mo. Last hard assignment
  - Midterm Project: If you don’t hear from me by midnight tonight, it means all is ok
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
  - 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]→
MPI: Textbooks, Further Reading...

- **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
Shared Memory Systems

- Memory resources are shared among processors
  - Typical scenario, on a budget: one node with four CPUs, each with 16 cores

- Relatively easy to program since there is a single unified memory space

- Two issues:
  - Scales poorly with system size due to the need for cache coherence
  - Most often, you need more memory than available on the typical multi-core node

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

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

[A. Jacobs]→
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

- Software development: 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
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

Euler
Euler
~ Hardware Configurations ~
**Hardware Relevant in the Context of MPI**

**Two Components of Euler that are Important**

- **CPU**: AMD Opteron 6274 Interlagos 2.2GHz
  - 16-Core Processor (four CPUs per node → 64 cores/node)
  - 8 x 2MB L2 Cache per CPU
  - 2 x 8MB L3 Cache per CPU
  - Thermal Design Power (TDP): 115W

- **HCA**: 40Gbps Mellanox Infiniband interconnect
  - Bandwidth comparable to PCIe2.0 x16 (~32Gbps), yet the latency is rather poor (~1 microsecond)
  - Ends up being the bottleneck in cluster computing
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 definition [in ME759]: a program counter and address space

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

- Interprocess communication typically of
  - Synchronization, followed by…
  - … 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
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**)

![Diagram](image-url)
```c
#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;
}
```

A First MPI Program

Has to be called first, and once

Has to be called last, and once
Program Output

```
[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]$
```
Why Care about MPI?

- Today, MPI is what enables supercomputers to run at PFlops rates
  - Some of these supercomputers might use GPU acceleration though

- Examples of architectures relying on MPI for HPC:
  - IBM Blue Gene L/P/Q (Argonne National Lab – “Mira”)
  - Cray supercomputers (Oakridge National Lab – “Titan”, also uses K20X GPUs)

- MPI has FORTRAN, C, and C++ bindings – widely used in Scientific Computing
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

- One of the early common implementations: MPICH
  - The CH comes from Chameleon, the portability layer used in the original MPICH to provide portability to the existing message-passing systems
  - OpenMPI: a new kid on the block, joint effort of three or four groups (Los Alamos, Tennessee, Indiana University, Europe)
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
MPI vs. CUDA

- 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)
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

$$\int_{0}^{1} \frac{4}{1 + x^2} \approx \sum_{i=1}^{n} \frac{1}{n} f((i - 0.5) \cdot h)$$
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 * ((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;