Introduction to Parallel Computing

Xizhou Feng

Information Technology Services
Marquette University

MUGrid Bootcamp, 2010
Outline

1 Overview of Parallel Computing
   - What is Parallel Computing?
   - Parallel Computing Hardware
   - Parallel Programming Paradigms
   - Parallel Algorithm Design

2 Parallel Computing with MPI (Message Passing Interface)
   - Background
   - Getting Started
   - MPI Programs and Environments
   - Point-to-point Communication
   - Collective Communication
   - Other Topics

3 Parallel Computing with OpenMP
Announcement

If you are a Marquette user, you may send the Pere account request form to its-rcs@marquette.edu to have your account activated on pere.
Parallelism is Universal

Definition

Sequential processes are those that occur in strict order, where it is not possible to the next step until the current one is completed.

- Professor teaches classes
- Player plays chess
- Child play violin

Definition

Parallel processes are those in which many events can happen simultaneously or exhibit concurrencies.

- Students take classes
- Players play football
- Musicians perform orchestra
What is Parallel Computing?

- Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem.
- A parallel computer platform can be:
  - A single computer with multiple processing elements (PEs);
  - An arbitrary number of computers connected by a network;
  - A combination of both.
- The computational problem exhibits large degree of parallelism:
  - Can be decomposed into concurrent discrete pieces of work;
  - Can execute multiple program instructions at any moment;
  - Can run much faster when using multiple compute resources.
Parallel computing is usually targeted at applications that perform computations on large datasets or large equations, examples include:

- Meteorologists: Weather prediction
- Biologists: DNA sequences analysis
- Pharmaceutical companies: Drug design
- Oil companies: Seismic exploration
- Wall Street: financial markets
- NASA: aerospace vehicle design
- Entertainment industry: special effects in movies and commercials
Flynn’s Taxonomy of Computer

- Flynn’s taxonomy categorizes computers by how streams of instructions interact with streams of data.

![Diagram showing SISD, MISD, SIMD, and MIMD architectures]
SIMD and MIMD

- **SIMD (Single Instruction, Multiple Data)**
  - All PEs execute the same instruction at any given clock cycle
  - Each PE can operate on a different data element
  - Most modern processors support SIMD operations

- **MIMD (Multiple Instruction, Multiple Data)**
  - The most common type of modern parallel computers.
  - Every processor may be executing a different instruction stream.
  - Every processor may be working with a different data stream
The Memory Model of MIMD Computers

- Shared Memory (UMA and NUMA)
- Distribute Memory
- Hybrid Distributed-Shared Memory
Comparison of Different Memory Models

- Shared memory
  - Easy to program
  - Fast data sharing
  - Lack of scalability
  - Memory conflicts
  - Expensive

- Distributed memory
  - Scalable
  - Fast local memory access
  - Cost effectiveness
  - Hard to program
  - Non-uniform memory access

- Hybrid distributed shared memory
  - Combine pros/cons of shared memory and distributed memory
  - Map modern parallel computer architecture
Parallel Programming Paradigms

- Exists as an abstraction above hardware and memory architecture
- Parallel programming paradigms in common use
  - Message Passing
  - Data parallel
  - Shared memory
  - Partitioned global memory space
  - Threads
Programs create multiple tasks;
- Each task has its local data and is identified by a unique name;
- Task interact by send and receive message to and from other named tasks.
- MPI is the de facto standard that implements message passing model
Parallel Programming Paradigms
Data Parallel Model

- Exploits the concurrency that exits in performing the same operation to multiple elements of a data structure;
- Multiple tasks work collectively on the same data structure, but each task works on a different partition
- Fortran 90/95 and HPF (High Performance Fortran) support data parallel model
Tasks share a common global address space, which they read and write asynchronously;

Various mechanisms (e.g., lock and semaphores) control how task access to the shared memory;

OpenMP implements both shared memory model and threads model.
Parallel Programs Design
Considerations

- Efficiency
- Scalability
- Locality
- Load-balancing
- Communication cost and latency hiding
Parallel Programs Design
Program Patterns

- Data-Parallel
- Task Graph
- Work Pool
- Master-Slave
- Pipeline/Producer-Consumer
- Map-Reduce
- Hybrid
What is MPI

**Definition**

**MPI** is a message-passing library specification that was designed to be a portable, efficient, and flexible standard for writing parallel programs using message passing.

1. Specification, not an implementation
2. Library, not a language
3. Classical message-passing model

- Defined in 1994 by MPI Forum.
- Supported by vendors on all parallel machines.
- Free, portable implementations. (MPICH, LAM, and OpenMPI)
Features of MPI
Included in MPI-1

- Thread safety
- User-defined Datatypes and Packing
- Collective operations
- Process groups and communication domains
- Point-to-point communication
- Application-oriented process topologies
- Profiling interface
- Environmental management and inquiry
- Binings for Fortran, C
Features of MPI
Added in MPI-2

- Dynamic process management
- Parallel I/O
- Remote memory operations
- Bindings for C++ and Fortran90
MPI Resources

- The Standard
  - MPI Forum [http://www.mpi-forum.org/docs/docs.html](http://www.mpi-forum.org/docs/docs.html)
  - MPI official releases, in both form of html and postscript
- The documents of the implementations (MPICH/OpenMPI/LAM MPI/Vendor-provided MPI)
- Books
  - Gropp et al, Using MPI: Portable Parallel Programming with the Message-Passing Interface (1994)
  - Foster, Designing and Building Parallel Programs (1995)
  - Pacheco, Parallel Programming with MPI (1997)
  - Quinn, Parallel Programming in C with MPI and OpenMP, McGraw (2003)
- Tutorials: Google ”MPI Tutorial”
Why and When to Use MPI?

- MPI is the only "standard" for message passing library
- MPI is supported on virtually all HPC platforms
- Write portable parallel codes across platforms
- Exploit high performance of the applications
- Deal with irregular or dynamic data structure
When Not to Use MPI?

- You can use HPF or a parallel Fortran 90
- You don’t need parallelism at all
- You can use libraries which may be written in MPI
- You can use other computing paradigm (e.g. multiple threading or job level parallelism)
Is MPI Large or Small?

- MPI is large. MPI-1 has 128 functions. MPI-2 has 152 functions.
  - MPI extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity
- MPI is small (6 functions)
  - Many parallel programs can be written with just 6 basic functions.
    - MPI_Init
    - MPI_Comm_size
    - MPI_Comm_rank
    - MPI_Send
    - MPI_Recv
    - MPI_Finalize
- MPI is just right
  - One can access flexibility when it is required.
  - One need not master all parts of MPI to use it.
Writing MPI Programs

```c
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello world\n" );
    MPI_Finalize();
    return 0;
}
```
Compiling MPI Programs

- **Method 1: Using MPI compiler wrapper**
  1. `mpicc`: for C code
  2. `mpicxx/mpic++/mpiCC`: for C++ code
  3. `mpif77/mpif90`: for FORTRAN code
  4. **Example:**
     ```
     mpicc -o hello hello.c
     ```

- **Method 2: Using standard compilers and MPI library**
  1. MPI is just a library, so you can link the library to your code to get the executables.
  2. **Example:**
     ```
     gcc -o ping ping.c \
     -I/usr/mpi/gcc/openmpi-1.2.8/include \
     -L/usr/mpi/gcc/openmpi-1.2.8/lib64 -lmpi
     ```

- **Method 3: Vendor-specific compilers (PGI compilers, Intel compilers, etc)**
**Makefile** files and the **make** utility enable automatically builds executable programs and libraries from source code

**Example:**

```plaintext
CC = mpicc
CLINKER = mpicc
CFLAGS = -O3
MATH_LIB = -lm

hello: hello.o
    $(CLINKER) -o hello hello.o $(MATH_LIB)

.c.o:
    $(CC) $(CFLAGS) -c $<

clean:
    rm -f *.o hello
```
#!/bin/sh

#PBS -N bootcamp-1
#PBS -l nodes=2:ppn=8
#PBS -l walltime=12:00:00
#PBS -q batch
#PBS -j oe
#PBS -o $PBS_JOBNAME-$PBS_JOBID.log

cd $PBS_O_WORKDIR
module load openmpi/gcc/1.2.8
PROG=hello
NP=16
mpiexec -np $NP $PROG
Running MPI jobs with PBS
Using a fixed set of hosts

#!/bin/sh

#PBS -N bootcamp-2
#PBS -l nodes=cn47:ppn=8+cn48:ppn=8
#PBS -l walltime=12:00:00
#PBS -q batch
#PBS -j oe
#PBS -o $PBS_JOBNAME-$PBS_JOBID.log

cd $PBS_O_WORKDIR
module load openmpi/gcc/1.2.8
PROG=hello
NP=16
mpiexec -np $NP $PROG
Commonly Used PBS Commands

- qsub: submit a job to
  - qsub job-scripts: submit a single job
  - qsub -t id1,id2-id3 job-scripts: submit an array of jobs
- qstat: get status on the system, queue and jobs
  - qstat -a: list all jobs on the system
  - qstat job-id: list a specific jobs
  - qstat -f job-id: list all information on a specific jobs
- qdel: delete a batch job
- pbsnode: show current node status
# an example condor submit file for MPI jobs
universe = parallel
executable = mpirun_condor
arguments = simple
machine_count = 4
output = simple2.out
error = simple2.err
log = simple2.log
should_transfer_files = IF_NEEDED
when_to_transfer_output = on_exit
queue
Explanations of the Condor Submit Description Files

1. Use the parallel universe
2. Use an mpirun wrapper for condor executable
3. Make the original executable and arguments as the new arguments
4. Specify the number of hosts needed to run the job
5. Add options to instruct how to transfer files
Exercise 1: Compiling and Running MPI Programs Using PBS

1. Copy the example files to your home directory
   
   $ cd Session4/openmpi/example1
   $ ls

2. Compile an MPI executable
   
   $ module load openmpi/gcc/1.2.8
   $ make
   mpicc -o hello hello.c

3. Submit the job to the queue
   
   $ qsub mpi.qsub
   $ qstat

4. View the output after the program finished
   
   $ more *.out
Exercise 2: Compiling and Running MPI Programs
Using Condor

1. Copy the example files to your home directory
   
   ```
   $ cd Session4/mpi/example2
   $ ls
   ```

2. Compile an MPI executable
   
   ```
   $ make
   mpicc -o hello hello.c
   ```

3. Submit the job to the queue
   
   ```
   $ condor_submit mpi.condor
   $ condor_q
   ```

4. View the output after the program finished
   
   ```
   $ more *.out
   ```
The Basic Structure of MPI programs

1. Include MPI header file
   ```c
   #include "mpi.h"
   ```

2. Declarations, prototypes

3. Program Begin

4. Serial code

5. Initialize MPI environment
   ```c
   MPI_Init(&argc, &argv);
   ```

6. Compute and make message passings

7. Terminate MPI environment
   ```c
   MPI_Finalize();
   ```

8. Serial code

9. Program Ends
Format of MPI Calls

- MPI loosely follow an object oriented design
- Most MPI routines act on MPI objects
- C-binding format
  
  \[
  rc = \text{Class}\_\text{Action}(\text{parameters})
  \]
  \[
  rc = \text{Class}\_\text{Action\_subset}(\text{parameters});
  \]

- C++-binding format
  
  \[
  rc = \text{MPI::Class::Action\_subset}(\text{parameters});
  \]
  \[
  rc = \text{MPI::Action\_subset}(\text{parameters});
  \]

- Fortran-binding format
  
  \[
  \text{CLASS\_ACTION\_SUBSET}(\text{parameters}, \text{rc});
  \]
  \[
  \text{CLASS\_ACTION}(\text{parameters}, \text{rc});
  \]
Communication Groups and Process Identity

- **Communicator**: an MPI object that defines a group of processes which communicate. 
  
  *MPI_COMM_WORLD*: the predefined communicator that includes all MPI processes.

- **Rank (task ID)**: an own unique, integer identifier assigned by the system to each process within a communicator.
Some Environment Management Routines

- **MPI_Init** - Initializes the MPI execution environment
- **MPI_Comm_size** - Determines the number of processes in the group associated with a communicator
- **MPI_Comm_rank** - Determines the rank of the calling process within the communicator.
- **MPI_Finalize** - Terminates the MPI execution environment.
- **MPI_Abort** - Terminates all MPI processes associated with the communicator.
- **MPI_Wtime()** - Returns an elapsed wall clock time in seconds (double precision) on the calling processor.
Example: Using Environment Management Routines

```c
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int nprocs, myrank;
    MPI_Init( &argc, &argv );
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("Task %d of %d!\n", myrank, nprocs);
    MPI_Finalize();
    return 0;
}
```
Exercise 3: Embarrassingly Parallel
Parallel Processing many files

- The Problem
  Write an MPI program to process a large number of data files

- The Solution
  1. Implement a `processFile` function
  2. Get filename and number of files from arguments
  3. Assign files to process a round-robin order

- Implement the solution as a program
- Test and use the program
Exercise 3: Embarrassingly Parallel
The Solution

```c
#include "mpi.h"
#include <stdio.h>
#include <string.h>

#define MAX_FILENAME_LEN 127

int nprocs, myrank;

void processFile(char *filename) {
    printf("Processing %s by task %d\n", filename, myrank);
}
```

Feng, Xizhou (Marquette University)
Introduction to Parallel Computing
Bootcamp 2010
Exercise 3: Embarrassingly Parallel

The Solution

```
12 int main( int argc, char *argv[] )
13 {
14   int numFiles;
15   char filename[MAX_FILENAME_LEN+1];
16   char currentFile[MAX_FILENAME_LEN+1];
17
18   MPI_Init( &argc, &argv );
19   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
20   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
21   if ( argc <= 2 ) {
22     if (myrank==0) {
23       printf("usage: %s data_file num_files\n", argv[0]);
24     }
25     MPI_Abort(MPI_COMM_WORLD, -1);
26   } else {
```
Exercise 3: Embarrassingly Parallel
The Solution

```c
26 } else {
27     strncpy(filename, argv[1], MAX_FILENAME_LEN);
28     filename[MAX_FILENAME_LEN] = '\0';
29     numFiles = strtol(argv[2]);
30 }
31
32 int i;
33 for(i=myrank; i<numFiles; i+=nprocs) {
34     sprintf(currentFile, "%s-%d", filename, i);
35     processFile(currentFile);
36 }
37
38 MPI_Finalize();
39 return 0;
40 }
```
The Message: Data + Envelope

**The Data**
- Data types
- Number of elements
- Buffer (Where to store it)

**The Envelope**
- The rank of the receiver
- The rank of the sender
- A tag
- A communicator
Point-to-Point Communication

- Blocking message passing
  - `MPI_Send`
  - `MPI_Recv`
  - `MPI_Sendrecv`
  - `MPI_Wait`
  - `MPI_Probe`
  - ...

- Non-Blocking message passing
  - `MPI_Isend`
  - `MPI_Irecv`
  - `MPI_Test`
  - `MPI_Iprobe`
  - ...
Exercise 4: Messaging Passing in a Ring

The Problem

- The tasks is organized as a ring topology
- Each task has an initial key
- At each time step, it passes the key to its right neighbor and receives a key from its left neighbor
- The program terminates after $N$ time steps

The Communications

- Task $i$ send key to task $(i + 1) \mod P$, $P$ is the number of tasks
- Task $i$ receive key to task $(i - 1 + P) \mod P$
Exercise 4: Messaging Passing in a Ring
The Solution

19 int key, key_received;
20 key = myrank;
21 int step;
22 source = (myrank - 1 + nprocs) % nprocs;
23 dest = (myrank + 1 ) % nprocs;
24
25 for (step=0; step<nprocs; step++) {
26   printf("step=%d task=%d key=%d\n", step, myrank, key);
27   MPI_Send(&key, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
28   MPI_Recv(&key_received, 1, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
29   key = key_received;
30 }

Feng, Xizhou (Marquette University)
Collective Communications

- Collective communications involve all processes in the scope of a communicator.

- Types of collective communication
  - Synchronization: processes wait all members reach the synchronization point
  - Data movement: broadcast, scatter/gather, all-to-all
  - Collective computation (reduction)

- Collective operations are blocking

- Can only be used with MPI predefined types
Collective Communications Routines

- *MPI_Barrier*
- *MPI_Bcast*
- *MPI_Scatter*
- *MPI_Gather*
- *MPI_Reduce*
- *MPI_Allgather*
- *MPI_Allreduce*
- *MPI_Alltoall*
- ...
Other MPI Topics

- User-defined Datatypes
- Application-oriented process topologies
- Profiling interface
- Parallel I/O
- Binings for Fortran, C++
- Debugging MPI programs
What is OpenMP (Open Multi-Processing)?

An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism.

Comprised of three primary API components:

1. Compiler Directives
2. Runtime Library Routines
3. Environment Variables

Portable and Standardized
The Fork-Join Model

Diagram:
- **Master thread**:
  - Forks into parallel regions
  - Joins back together
- **Parallel regions**:
  - Simultaneous execution
  - Coordinating with master thread
Sample OpenMP programs
an OpenMP version of Hello

```c
#include <omp.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    int th_id, nthreads;
    #pragma omp parallel private(th_id)
    {
        th_id = omp_get_thread_num();
        printf("Hello World from thread %d\n", th_id);
        #pragma omp barrier
        if ( th_id == 0 ) {
            nthreads = omp_get_num_threads();
            printf("There are %d threads\n", nthreads);
        }
    }
    return 0;
}
```
Sample OpenMP programs
Parallel Matrix Multiplication

```c
#pragma omp parallel for private(tmp, i, j, k)
for (i=0; i<Ndim; i++){
    for (j=0; j<Mdim; j++){
        tmp = 0.0;
        for(k=0;k<Pdim;k++){
            /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
            tmp += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));
        }
        *(C+(i*Ndim+j)) = tmp;
    }
}
```
Exercise 5: Running OpenMP Jobs on Pere

1. Copy the example files to your home directory
   
   ```
   $ cd Session4/openmp/example1
   $ ls
   ```

2. Compile an openMP executable
   
   ```
   $ make
   gcc44 -O3 -fopenmp -o hello hello.c
   ```

3. Submit the job to the queue
   
   ```
   $ qsub openmp.qsub
   $ qstat
   ```

4. View the output after the program finished
   
   ```
   $ more test*.out
   ```