Lecture 4

MPI Programming (I)

1

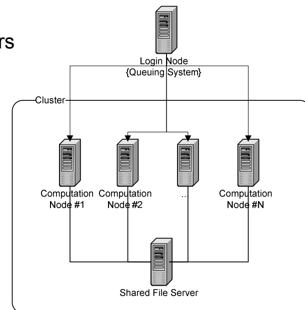
Topics

- Our Cluster
- MPI
  - Introduction
  - Information inquery
  - Broadcast / Reduce

2

What is a cluster?

- A cluster is a dedicated resource for running computational tasks.
  - A collection of computers



IT Group Cluster2 (1/2)

- Base system: Gentoo Linux (http://www.gentoo.org)
  - 8 Pentium D930 (Dual core, 3.0GHz) nodes
  - 4 Core 2 dual E6320 (dual core, 1.86GHz) nodes
  - All equipped with 4GB RAM (available memory varies)
- Queuing system: SLURM (http://www.llnl.gov/linux/slurm/)
  - 2 public queues (Public, Core) with 10-minute limits
  - 1 private queue without time limit
  - First come, first serve.
- Monitoring system: Ganglia (http://ganglia.sourceforge.net/)
  - http://140.118.5.6:8000

4

IT Group Cluster2 (2/2)

- **Programming environment**
  - Compilers
    - g++, gcc: GNU Compiler Collection suite
    - icpc, icc: Intel C/C++/Fortran Compilers
    - *pathCC, pathcc PathScale*
  - Auxiliary utilities
    - make
    - gdb - debugger
    - valgrind – tool identifying memory accessing problems
    - gprof – profiler, used to identify performance bottlenecks
  - Supporting programming libraries
    - MPI: OpenMPI, MPICH2
    - Numerical libraries: Intel MKL, AMD ACML, …
    - OpenCV, vtk, …

5

Important Commands / Queuing

- **sinfo** → get information about the queue

```
ymhsieh@n00 ~ $ sinfo
PARTITION AVAIL  TIMELIMIT NODES  STATE NODELIST
Public*      up      30:00     8   idle n[00-07]
Core         up      30:00     4   idle c[00-03]
```

```
ymhsieh@n00 ~ $ sinfo
PARTITION AVAIL  TIMELIMIT NODES  STATE NODELIST
Public*      up      30:00     4  alloc n[01-04]
Public*      up      30:00     4   idle n[00,05-07]
Core         up      30:00     4   idle c[00-03]
```

alloc: allocated / occupied

6

Important Commands / Queuing

- **squeue** → information about the queue

```
ymhsieh@n00 ~/Work/04_Test/MPI $ squeue
  JOBID PARTITION     NAME     USER  ST       TIME  NODES
NODELIST(REASON)
   1008    Public  startup  ymhsieh  PD       0:00      3 (Resources)
   1009    Public  startup  ymhsieh  PD       0:00      3 (Resources)
   1010    Public  startup  ymhsieh  PD       0:00      3 (Resources)
   1006    Public  startup  ymhsieh   R       0:04      3 n[01-03]
   1007    Public  startup  ymhsieh   R       0:03      3 n[04-06]
```

```
PD: Pending
R: Runing
```

7

Important Commands / Queuing

- **scancel** → cancel a job in the queue

```
ymhsieh@n00 ~/Work/04_Test/MPI $ scancel 1011
ymhsieh@n00 ~/Work/04_Test/MPI $ squeue
  JOBID PARTITION     NAME     USER  ST       TIME  NODES NODELIST(REASON
   1010    Public  startup  ymhsieh  PD       0:00      3 (Resources)
   1012    Public  startup  ymhsieh  PD       0:00      3 (Resources)
   1013    Public  startup  ymhsieh  PD       0:00      3 (Resources)
   1014    Public  startup  ymhsieh  PD       0:00      3 (Resources)
   1008    Public  startup  ymhsieh   R       0:38      3 n[01-03]
   1009    Public  startup  ymhsieh   R       0:37      3 n[04-06]
```

8

Important Commands / Queuing

- **sbatch** (batch processing) → submit *a job script* for later execution. The script will typically contain one or more srun commands to launch parallel tasks.
  - sbatch -n8 /opt/mpich2/startup ./myProg.exe
- **srun** (interactive processing) → submit a job for execution or initiate job steps in interactive mode.
  - srun ./myProg.exe
  - srun -n8 ./myProg.exe
  - srun -n8 ./myProg.exe 200
- Complete Reference: https://computing.llnl.gov/linux/slurm/quickstart.html

9

Summary

- sinfo
- squeue
- scancel
- sbatch
- srun

10

MPI Programming

Information Enquiry

Basic Collective Communication

Point-to-Point Communication

11

MPI: Message Passing Interface

http://www-unix.mcs.anl.gov/mpi/

- It is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.
  - The MPI standard is available.
  - MPI was designed for high performance on both massively parallel machines and on workstation clusters.
  - MPI is widely available, with both free available and vendor-supplied implementations.
  - MPI was developed by a broadly based committee of vendors, implementors, and users.
  - Several versions are now available: 1.0, 1.1, 1.2, 1.3, 2.0, **2.1, 2.2**
    - http://www.mpi-forum.org
- Mainly for programming distributed memory computers

12

MPI Implementations (1)

- Two major free implementations
  - √ MPICH2: http://www-unix.mcs.anl.gov/mpi/mpich2/
  - √ Open-MPI: http://www.open-mpi.org/
- Differences between different implementations
  - Different algorithms (for collective communications)
  - Different features (e.g. fault tolerance, …)
  - Different MPI-version support (MPI-1.1 vs.2.0)
  - SMP utilizations

MPI Online References

- Online References
  - Online version of "MPI, the complete reference"
    - http://www.netlib.org/utk/papers/mpi-book/mpi-book.html
  - Tutorial material on MPI available on the Web
    - http://www-unix.mcs.anl.gov/mpi/tutorial/
  - List of MPI APIs
    - http://www-unix.mcs.anl.gov/mpi/www/
  - Google Keyword: MPI tutorial

Our First MPI Program 01.cpp

Available in our cluster system: /home/course/Lecture04

```
#include <iostream>
#include "mpi.h"
using namespace std;

int main(int argc, char **argv){
   char name[1024];
   int length=1024, major, minor;

   MPI_Init(&argc, &argv);
   MPI_Get_processor_name(name, &length);
   cout << "\nHello from " << name;
   MPI_Get_version(&major, &minor);
   cout << "\nMPI Version " << major << "." << minor;
   cout << "\nFrom header: Version " << MPI_VERSION << "."
  << MPI_SUBVERSION;
   MPI_Finalize();
   return 0;
}
```


Running MPI Programs *in our cluster*

- Compile:
  - C: mpicc 01.c -o 01.exe
  - C++: mpic++ 01.cpp –o 01.exe
- Open-MPI, Interactive mode
  salloc -n8 *[Allocate resource with 8 processors]*
  mpiexec -np 8 ./01.exe *[execute your MPI code]*
  exit *[release allocation]*
- Open-MPI, Batch mode
  sbatch -n8 /opt/openmpi/startup ./01.exe

/opt/openmpi/startup is a shell script I made to make your life easier …

MPI API Naming Rules

- All MPI functions start with MPI_
- Followed by a capital letter
- The rest indicates the purpose of the API
- Words are separated by underscore _
  - MPI_Init(&argc, &argv);
    - Initialize the MPI execution environment
      - Create "MPI_COMM_WORLD"
      - Pass argc & argv to all processes
  - MPI_Get_processor_name(name, &length);
    - Get processor name (hostname) for the current processor

Our Second MPI Program 02.cpp

Available in our cluster system: /home/course/Lecture04

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

int main(int argc, char **argv) {
  int rank, size;
  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::cout << "\nSize=" << size << ", MyRank:" << rank;

  MPI_Finalize();
  return 0;
}
```

MPI

- Each process in the MPI environment is given a integer "rank" starts from 0 !
  - Thus, for N processes, rank ranges between 0 and N-1
- The rank is like an IP address on the Internet so each process can be uniquely identified for point-to-point communication.
- A "communicator" is a group of processes that can be defined by programmers to communicate between a group of processes
  - Communicator can be regarded to as the identifier for groups
  - MPI_COMM_WORLD: an automatically created communicator that contains ALL MPI processes

MPI

- Rank is used for point-to-point communication
- Communicator is used for collective communication
- We have introduced 5 functions so far, there are more than 120 functions defined for MPI-1 (up to 1.2)
  - Fortunately, we don't need to use all of them in a single program
  - In fact, basic programs can be written with only six functions
    - MPI_Init(), MPI_Finalize()
    - MPI_Comm_rank(), MPI_Comm_size()
    - MPI_Send(), MPI_Recv()
- http://www-unix.mcs.anl.gov/mpi/www/

Timing
MPI_Wtime()

```
double MPI_Wtime();
```

Return value
Time in seconds since an arbitrary time in the past.

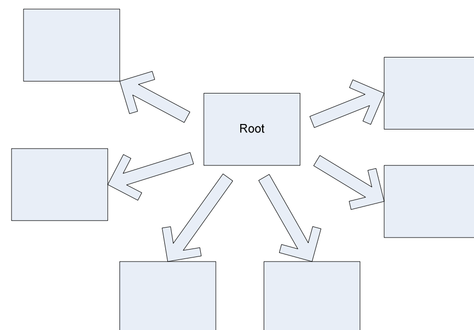
Notes
This is intended to be a high-resolution, elapsed (or wall) clock. See MPI_WTICK to determine the resolution of MPI_WTIME. If the attribute MPI_WTIME_IS_GLOBAL is defined and true, then the value is synchronized across all processes in MPI_COMM_WORLD.

MPI
Information Enquiry

- Initialization & Finalization
  - `MPI_Init(&argc, &argv);`
  - `MPI_Finalize();`
- Get information about a process or a communicator
  - `MPI_Get_processor_name(name, &length);`
  - `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`
  - `MPI_Comm_size(MPI_COMM_WORLD, &size);`
- Other Information
  - `MPI_Get_version(&major, &minor);`
  - `MPI_Wtime();`

Basic
Collective Communication

Broadcast



Broadcast

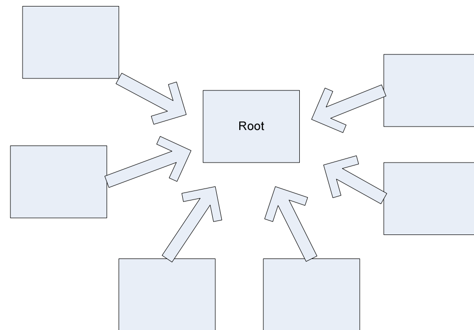
```
int MPI_Bcast (void *buffer, int count,
   MPI_Datatype datatype, int root, MPI_Comm comm);
```

Input/output Parameters

buffer	starting address of buffer (choice)
count	number of entries in buffer (integer)
datatype	data type of buffer (handle)
root	rank of broadcast root (integer)
comm	communicator (handle)

25

Reduction



26

Reduction

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
   int count, MPI_Datatype datatype, MPI_Op op,
   int root, MPI_Comm comm )
```

Input Parameters

sendbuf	address of send buffer (choice)
count	number of elements in send buffer (integer)
datatype	data type of elements of send buffer (handle)
op	reduce operation (handle)
root	rank of root process (integer)
comm	communicator (handle)

Output Parameter

recvbuf	address of receive buffer (choice, significant only at root)

27

MPI_Datatype

```
MPI Basic Datatypes for C
----------------------------------------
MPI Datatype          C datatype
----------------------------------------
MPI_CHAR              signed char
MPI_SHORT             signed short int
MPI_INT               signed int
MPI_LONG              signed long int
MPI_UNSIGNED_CHAR     unsigned char
MPI_UNSIGNED_SHORT    unsigned short int
MPI_UNSIGNED          unsigned int
MPI_UNSIGNED_LONG     unsigned long int
MPI_FLOAT             float
MPI_DOUBLE            double
MPI_LONG_DOUBLE       long double
MPI_BYTE
MPI_PACKED
```

28

MPI_Op op

MPI_MAX	return the maximum
MPI_MIN	return the minimum
MPI_SUM	return the sum
MPI_PROD	return the product
MPI_LAND	return the logical and
MPI_BAND	return the bitwise and
MPI_LOR	return the logical or
MPI_BOR	return the bitwise of
MPI_LXOR	return the logical exclusive or
MPI_BXOR	return the bitwise exclusive or
MPI_MINLOC	return the minimum and the location (actually, the value of the second element of the structure where the minimum of the first is found)
MPI_MAXLOC	return the maximum and the location

29

Examples

- 03.cpp: Brute-force method to calculate summation from 1 to a specified number
- 04.cpp: Integration of a function using trapezoidal rule

- All these examples are known as "embarrassingly/pleasingly parallel", which exchange little information at beginning, and exchange little information at the end. These examples demonstrate excellent parallel efficiency, as will be demonstrated.

30

A shell script for running 03.cpp

```
#!/bin/bash

num=1000000000
run=./03.exe

for i in `seq 1 20`; do
  mpiexec -np $i $run $num
done
```

```
sbatch -n16 ./03.sh
```

31

Outputs from 03.cpp

```
1, sum(1..100000000)= 987459712, 0.0332892 seconds.
2, sum(1..100000000)= 987459712, 0.0208299 seconds.
3, sum(1..100000000)= 987459712, 0.011507 seconds.
4, sum(1..100000000)= 987459712, 0.00897193 seconds.
5, sum(1..100000000)= 987459712, 0.00850511 seconds.
6, sum(1..100000000)= 987459712, 0.00971389 seconds.
7, sum(1..100000000)= 987459712, 0.010541 seconds.
8, sum(1..100000000)= 987459712, 0.00654697 seconds.
9, sum(1..100000000)= 987459712, 0.030273 seconds.
10, sum(1..100000000)= 987459712, 0.00907707 seconds.
11, sum(1..100000000)= 987459712, 0.00620389 seconds.
12, sum(1..100000000)= 987459712, 0.00781703 seconds.
13, sum(1..100000000)= 987459712, 0.0085721 seconds.
14, sum(1..100000000)= 987459712, 0.00981593 seconds.
15, sum(1..100000000)= 987459712, 0.0127621 seconds.
16, sum(1..100000000)= 987459712, 0.015419 seconds.
17, sum(1..100000000)= 987459712, 0.00857997 seconds.
18, sum(1..100000000)= 987459712, 0.00824785 seconds.
19, sum(1..100000000)= 987459712, 0.00717402 seconds.
20, sum(1..100000000)= 987459712, 0.0120881 seconds.
```

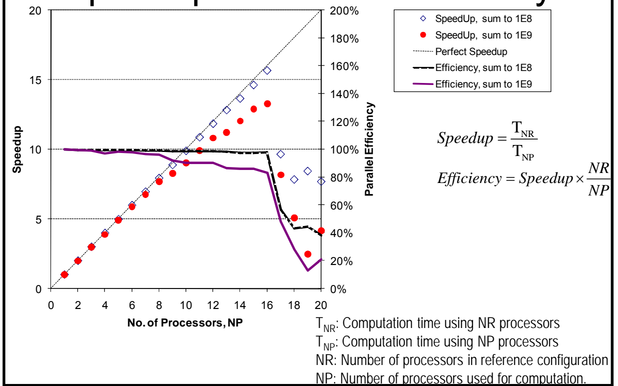
32

Outputs from 03.cpp

```
1, sum(1..1000000000)= 4051657984, 0.332772 seconds.
2, sum(1..1000000000)= 4051657984, 0.166482 seconds.
3, sum(1..1000000000)= 4051657984, 0.114991 seconds.
4, sum(1..1000000000)= 4051657984, 0.083952 seconds.
5, sum(1..1000000000)= 4051657984, 0.115212 seconds.
6, sum(1..1000000000)= 4051657984, 0.0566759 seconds.
7, sum(1..1000000000)= 4051657984, 0.048713 seconds.
8, sum(1..1000000000)= 4051657984, 0.051435 seconds.
9, sum(1..1000000000)= 4051657984, 0.0392969 seconds.
10, sum(1..1000000000)= 4051657984, 0.036885 seconds.
11, sum(1..1000000000)= 4051657984, 0.0325961 seconds.
12, sum(1..1000000000)= 4051657984, 0.031656 seconds.
13, sum(1..1000000000)= 4051657984, 0.0357301 seconds.
14, sum(1..1000000000)= 4051657984, 0.0337989 seconds.
15, sum(1..1000000000)= 4051657984, 0.0287411 seconds.
16, sum(1..1000000000)= 4051657984, 0.0246971 seconds.
17, sum(1..1000000000)= 4051657984, 0.032264 seconds.
18, sum(1..1000000000)= 4051657984, 0.0383141 seconds.
19, sum(1..1000000000)= 4051657984, 0.0453541 seconds.
20, sum(1..1000000000)= 4051657984, 0.0406802 seconds.
```

33

Speedup & Parallel Efficiency



Observations

- The program seems correct!
  - The answer doesn't change with number of processors
- Very good parallel efficiency is observed!
- These examples (03, 04) are known as "embarrassingly (or pleasingly) parallel"!

35

Two famous laws in parallel computing

Amdahl's Law
Gustafson's Law

36

Amdahl's Law

- Maximum speedup is governed by the serial fraction (non-parallelizable part) of a program
- A task can be divided into parallel (p) and non-parallel (s, serial) fractions:

$$T_1 = T_s + T_p$$

$$T_{NP} = T_s + T_p \times \frac{1}{P}$$

$$Speedup = \frac{T_1}{T_{NP}} = \frac{T_s + T_p}{T_s + T_p \times \frac{1}{P}}$$

$$Efficiency = \frac{T_s + T_p}{T_s \times P + T_p}$$

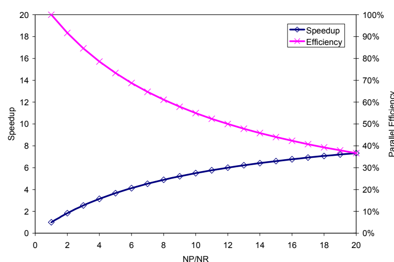
Amdahl's Law

$$Speedup = \frac{T_1}{T_{NP}} = \frac{T_s + T_p}{T_s + T_p \times \frac{1}{P}} = \frac{\alpha + 1}{\alpha + \frac{1}{P}} \qquad \alpha = \frac{T_s}{T_p}$$

$$Efficiency = \frac{T_s + T_p}{T_s \times P + T_p} = \frac{\alpha + 1}{\alpha P + 1}$$

Amdahl's Law

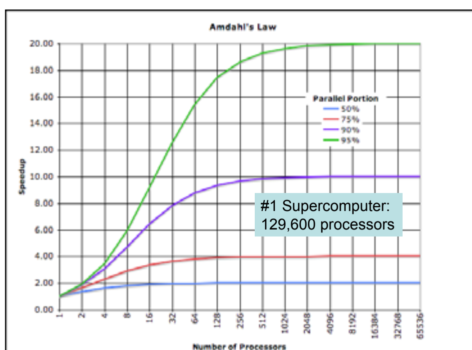
- If T_s=0 → speedup = P, efficiency=1
- If Tp/Ts=10 →



Amdahl's Law

- Thus, we need to minimize Ts as much as possible
  - Ts = $T_{\text{serial code}}$ + $T_{\text{communication}}$
  - $T_{\text{communication}}$: Communication overhead, may increase with NP
- One way to reduce Ts for communication is "overlap communication with computation"
  - To be covered next time when we talk about non-blocking communication

Amdahl's Law



http://upload.wikimedia.org/wikipedia/commons/6/6b/AmdahlsLaw.png

Gustafson's Law

- As the problem to be solved increases in size, the serial fraction decreases and parallel fraction increases → α decreases

$$Speedup = \frac{T_1}{T_{NP}} = \frac{T_s + T_p}{T_s + T_p \times \frac{1}{P}} = \frac{\alpha + 1}{\alpha + \frac{1}{P}}$$

$$Efficiency = \frac{T_s + T_p}{T_s \times P + T_p} = \frac{\alpha + 1}{\alpha P + 1}$$

A Driving Metaphor

- Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph.
- Amdahl's Law approximately suggests:
  - No matter how fast you drive the last half, it is impossible to achieve 90 mph average before reaching the second city. Since it has already taken you 1 hour and you only have a distance of 60 miles total; going infinitely fast you would only achieve 60 mph.
- Gustafson's Law approximately states:
  - Given enough time and distance to travel, the car's average speed can always eventually reach 90mph, no matter how long or how slowly it has already traveled. For example, in the two-cities case this could be achieved by driving at 150 mph for an additional hour.

43

http://en.wikipedia.org/wiki/Gustafson%27s_Law

MPI Summary

- Information Enquiry
  - `MPI_Initialize()`
  - `MPI_Get_processor_name()`
  - `MPI_Get_version()`
  - `MPI_Comm_size()`
  - `MPI_Comm_rank()`
  - `MPI_Wtime()`
  - `MPI_Finalize()`
- Collective communication
  - `MPI_Bcast()`
  - `MPI_Reduce()`

44

What you should know after today's class

- You should know how to use the queuing system to utilize our cluster system.
- You should be able to run interactive and batch jobs. (and you should understand what they are).
- Understand backgrounds of MPI
- Know how to use MPI APIs to
  - Get basic information (no. of processes, hostnames, …)
  - Communicate data between processes (collective communication)
  - Write a simple parallel program.
- Know what is “embarrassingly parallel” tasks.
- Two famous laws in parallel computing

45