Assignment #3

- How to count FLOP?
  - A = A + b * c ← 2 floating point operations

```
for(int i=0;i<10;i++) {
   for(int j=0;j<10;j++) {
      c[i] = c[i] + a[i]*b[j];
   }
}
```

- The statement above executes 100 times → 200 FLOP
- Count the total number of FLOP and measure the time spent → FLOPS
- i, j are integer operations, not counted in FLOP

Assignment #3

- How to estimate memory bandwidth?

```
double *a, *b, *c;
for(int i=0;i<10;i++) {
   for(int j=0;j<10;j++) {
    c[i] = c[i] + a[i]*b[j];
   }
}
```

- The above statement generates 4 memory references, each reference takes sizeof(double) → 400*8 = 3200 bytes
  1. Read from a[i]
  2. Read from b[i]
  3. Read from c[i]
  4. Write to c[i]

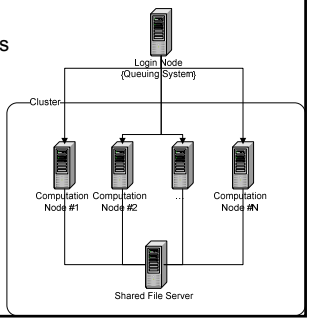
Assignment #3

- So, I would like to see discussions on memory bandwidth as well in your assignment #3, producing similar chart as the one shown last time!

IT Group Cluster2

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)
  - 7 Pentium D930 (Dual core, 3.0GHz) nodes
  - 3 Core 2 dual E6320 (dual core, 1.86GHz) nodes
  - 1 Xeon E5310 with NVidia GeForce 8800 Ultra for CUDA
  - All equipped with 4GB RAM (available memory varies)
- Queuing system: SLURM (http://www.llnl.gov/linux/slurm/)
  - 2 public queues (Public, Core) with 30-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

IT Group Cluster2 (2/2)

- **Programming environment**
  - Compilers
    - GNU Compiler Collection suite (gcc) / gcc, g++
    - Intel C/C++/Fortran Compilers
    - *PathScale, PGI*
  - Auxiliary utilities
    - **make**: help you compile your software projects
    - **gdb**: debugger
    - **valgrind**: detecting memory access problems, memory leaks, etc.
    - **gprof**: profiler for identifying performances in your code
    - Others…
  - Supporting programming libraries
    - MPI: OpenMPI, MPICH2
    - Numerical libraries: Intel MKL, AMD ACML, …

7

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

8

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

9

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]
```

10

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/openmpi/startup ./myProg.exe
  - sbatch -n20 -O /opt/openmpi/startup ./myProg.exe
- **srun** (interactive processing) → submit a job for execution or initiate job steps in interactive mode.
  - salloc -n8
  - mpiexec ./myProg.exe
  - mpiexec –np 4 ./myProg.exe
  - exit
- Complete Reference: https://computing.llnl.gov/linux/slurm/quickstart.html

11

Summary

- sinfo
- squeue
- scancel
- sbatch
- srun

12

Lecture 5

MPI Programming (I)

MPI Programming

Information Enquiry

Basic Collective Communication

Point-to-Point Communication

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, 2.0, **2.1, 2.2**
    - http://www.mpi-forum.org
- Mainly for programming distributed memory computers

MPI Implementations (1)

- Some main free implementations
  - √ MPICH2: http://www-unix.mcs.anl.gov/mpi/mpich2/
  - √ Open-MPI: http://www.open-mpi.org/
  - √ MPICH: http://www-unix.mcs.anl.gov/mpi/mpich1/
  - √ LAM: http://www.lam-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/Lecture05

```
#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/Lecture05

```
#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

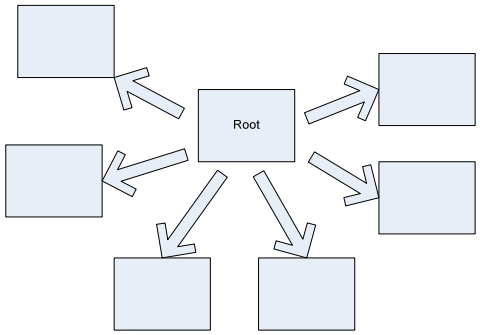
- Initialization & Finalization
  - `MPI_Init(&argc, &argv);`
  - `MPI_Finalize();`
- Get information about a process or a communicator
  - `MPI_Get_processor_name(name, &length);`
  - `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`
  - `MPI_Comm_size(MPI_COMM_WORLD, &size);`
- Other Information
  - `MPI_Get_version(&major,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

25

Basic
Collective Communication

26

Broadcast



27

Broadcast

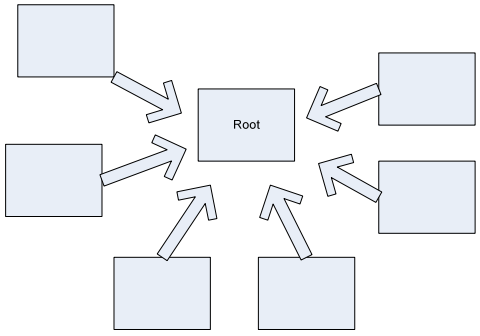
```
int MPI_Bcast (void *buffer, int count,
  MPI_Datatype datatype, int root, MPI_Comm comm);
```

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)

28

Reduction



29

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)

30

MPI_Datatype

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	

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

Examples

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

- 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.

A shell script for running 03.cpp

```
#!/bin/bash

num=1000000000
run=./03.exe

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

```
sbatch –n16 ./03.sh
```

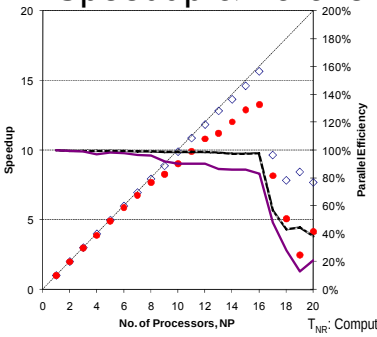

Outputs from 03.cpp

```
1sum(1..100000000= 987459712, 0.033323 seconds.
2sum(1..100000000= 987459712, 0.0167341 seconds.
3sum(1..100000000= 987459712, 0.011214 seconds.
4sum(1..100000000= 987459712, 0.00845981 seconds.
5sum(1..100000000= 987459712, 0.00692797 seconds.
6sum(1..100000000= 987459712, 0.00567198 seconds.
7sum(1..100000000= 987459712, 0.00492311 seconds.
8sum(1..100000000= 987459712, 0.00445008 seconds.
9sum(1..100000000= 987459712, 0.00733805 seconds.
10sum(1..100000000= 987459712, 0.00370121 seconds.
11sum(1..100000000= 987459712, 0.00325298 seconds.
12sum(1..100000000= 987459712, 0.00309396 seconds.
13sum(1..100000000= 987459712, 0.00291181 seconds.
14sum(1..100000000= 987459712, 0.00256586 seconds.
15sum(1..100000000= 987459712, 0.00244999 seconds.
16sum(1..100000000= 987459712, 0.00253606 seconds.
17sum(1..100000000= 987459712, 0.00673699 seconds.
18sum(1..100000000= 987459712, 0.00471187 seconds.
19sum(1..100000000= 987459712, 0.00984597 seconds.
20sum(1..100000000= 987459712, 0.00739312 seconds.
```


Outputs from 03.cpp

```
1sum(1..1000000000= 4051657984, 0.332862 seconds.
2sum(1..1000000000= 4051657984, 0.166736 seconds.
3sum(1..1000000000= 4051657984, 0.142624 seconds.
4sum(1..1000000000= 4051657984, 0.083472 seconds.
5sum(1..1000000000= 4051657984, 0.0670869 seconds.
6sum(1..1000000000= 4051657984, 0.055639 seconds.
7sum(1..1000000000= 4051657984, 0.047833 seconds.
8sum(1..1000000000= 4051657984, 0.0417891 seconds.
9sum(1..1000000000= 4051657984, 0.0376599 seconds.
10sum(1..1000000000= 4051657984, 0.0336511 seconds.
11sum(1..1000000000= 4051657984, 0.033962 seconds.
12sum(1..1000000000= 4051657984, 0.0280821 seconds.
13sum(1..1000000000= 4051657984, 0.026078 seconds.
14sum(1..1000000000= 4051657984, 0.0269802 seconds.
15sum(1..1000000000= 4051657984, 0.030549 seconds.
16sum(1..1000000000= 4051657984, 0.021384 seconds.
17sum(1..1000000000= 4051657984, 0.0329359 seconds.
18sum(1..1000000000= 4051657984, 0.0425711 seconds.
19sum(1..1000000000= 4051657984, 0.041537 seconds.
20sum(1..1000000000= 4051657984, 0.0570061 seconds.
```

Speedup & Parallel Efficiency



$$Speedup = \frac{T_{NR}}{T_{NP}}$$

$$Efficiency = Speedup \times \frac{NR}{NP}$$

T_{NR}: Computation time using NR processors
T_{NP}: Computation time using NP processors
NR: Number of processors in reference configuration
NP: Number of processors used for computation.

Observations

- The program seems correct!
  - The answer doesn't change with number of processors
- Very good parallel efficiency is observed!
- These examples (03, 04, 05.cpp) are known as "embarrassingly (or pleasingly) parallel"!

Two famous laws in parallel computing

Amdahl's Law
Gustafson's Law

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:

$$T_1 = T_s + T_p$$

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

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

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

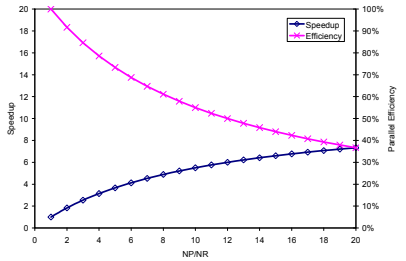
Amdahl's Law

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

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

Amdahl's Law

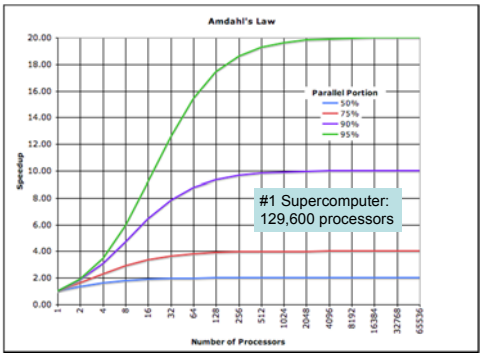
- If T_s=0 → speedup = P, efficiency=1
- If Tp/Ts=10 →



Amdahl's Law

- Thus, we need to minimize Ts as much as possible
  - Ts = $T_{serial\ code}$ + $T_{communication}$
  - $T_{communication}$: 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.

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()`

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