Lecture 5

MPI Programming (II)

Two famous laws in parallel computing
More on collective communication

Outline

- Two Laws in Parallel Computing
- More on collective communication

Two 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:

$$T_1 = T_s + T_p$$

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

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

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

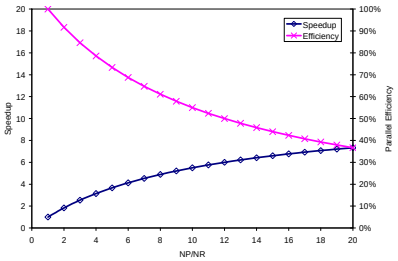
Amdahl's Law

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

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

Amdahl's Law

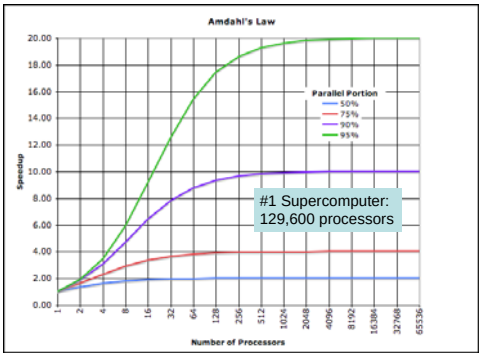
- If T_s=0 → speedup = P, efficiency=1
- If Tp/Ts=10 →



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

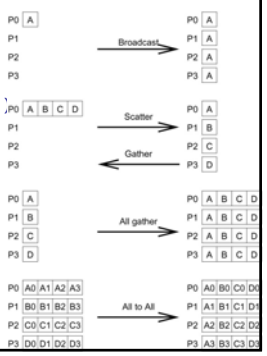
- Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph.
- Amdahl's Law approximately suggests:
  - No matter how fast you drive the last half, it is impossible to achieve 90 mph average before reaching the second city. Since 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

Collective Communication (II)

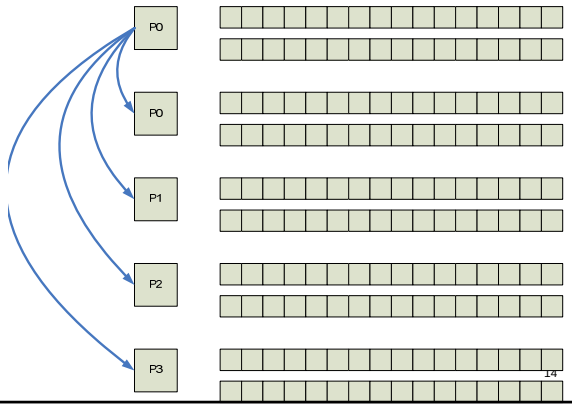
Collective Communication

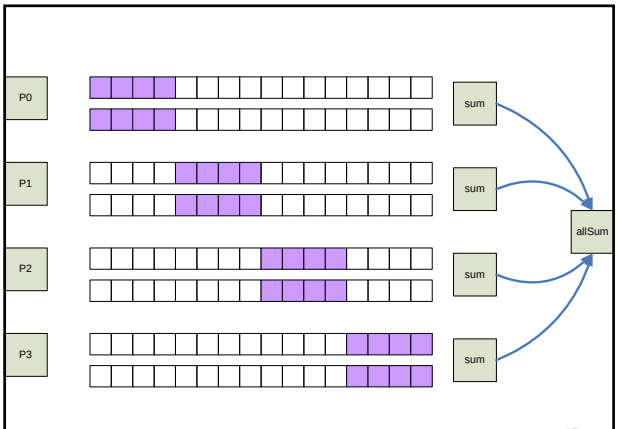
- Collective communication
  - `MPI_Bcast(), MPI_Reduce()`
  - `MPI_Scatter(), MPI_Gather()`
  - `MPI_Allgather(), MPI_Allreduce()`
  - `MPI_Alltoall()`
  - `MPI_Barrier(), MPI_Scan()`



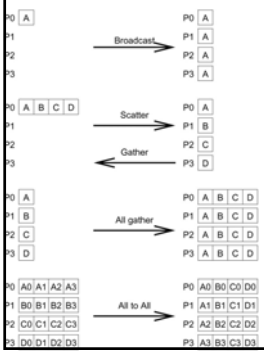
MPI_Bcast() / MPI_Reduce

- 06.cpp
  performs vector inner product
- Broadcast revisited:
  - MPI_Bcast() to broadcast the vector to all nodes
  - Each node decides which portion of the vector to work on
  - Perform calculation
  - MPI_Reduce() to sum up the inner dot from different portions of the vector
- Thus this is a bad parallel algorithm for performing vector inner product.
- Should really use MPI_Scatterv()!





MPI Collective Communications



All collective communications can be used to transmit
- equal-sized arrays
- unequal-sized arrays

Effect = gather + broadcast, but better

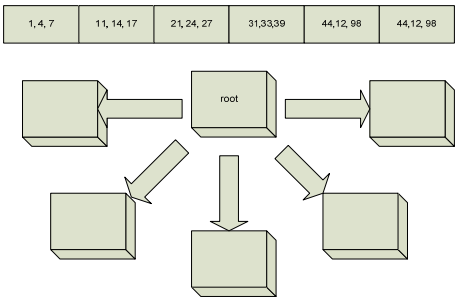
MPI_Scatter() / MPI_Gather()

- For dividing/grouping and distributing/gathering arrays or vectors (1-D array) to/from all nodes within the specified communicator.
- Each node only receives part of the array
- Each node receives/sends equal amount of data

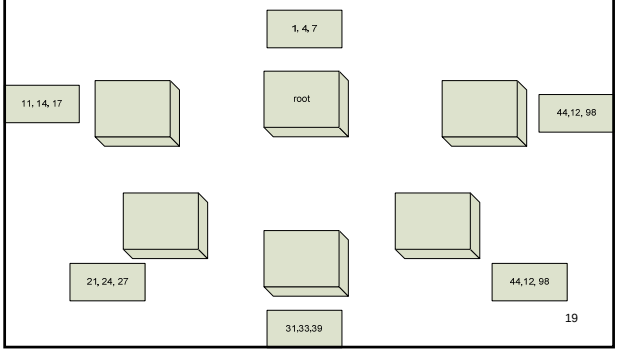
```
int MPI_Scatter (
  void *sendbuf, int sendcnt, MPI_Datatype sendtype,
  void *recvbuf, int recvcnt, MPI_Datatype recvtype,
  int root, MPI_Comm comm )

int MPI_Gather (
  void *sendbuf, int sendcnt, MPI_Datatype sendtype,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm )
```


Before Scatter Operation



After Scatter Operation



07.cpp

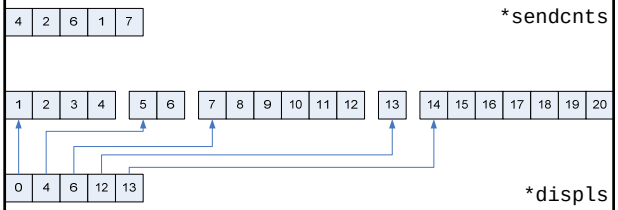
- This is an example demonstrating the use of `MPI_Scatter() / MPI_Gather()`
  - Generate some numbers on the root node
  - Scatter generated numbers onto all nodes
  - Each node prints out what they have
  - Each node calculate summation of the data the own
  - Gather summations from all nodes
  - Root prints out the data after gathering

MPI_Scatterv()/ MPI_Gatherv()

- For dividing/grouping and distributing/gathering arrays or vectors (1-D array) to/from all nodes within the specified communicator
- Each node only receives part of the array
- Each node does not necessarily receive/send equal amount of data

```
int MPI_Scatterv (
 void *sendbuf, int *sendcnts, int *displs, MPI_Datatype sendtype,
 void *recvbuf, int recvcnt, MPI_Datatype recvtype,
 int root, MPI_Comm comm )

int MPI_Gatherv (
 void *sendbuf, int sendcnt, MPI_Datatype sendtype,
 void *recvbuf, int *recvcnts, int *displs, MPI_Datatype recvtype,
 int root, MPI_Comm comm )
```

08.cpp

- This is a program performing vector normalization (make the length of the vector to be unity).

Other MPI collective functions

```
int MPI_Alltoall(
   void* sendbuf, int scnts, MPI_Datatype sendtype,
   void* recvbuf, int rcnts, MPI_Datatype recvtype,
   MPI_Comm comm)

int MPI_Alltoallv(
   void* sendbuf, int *scnts, int *sdispls, MPI_Datatype sendtype,
   void* recvbuf, int *rcnts, int *rdispls, MPI_Datatype recvtype,
   MPI_Comm comm)

int MPI_Allgather(
   void* sendbuf, int scnts, MPI_Datatype sendtype,
   void* recvbuf, int rcnts, MPI_Datatype recvtype,
   MPI_Comm comm)

int MPI_Allgatherv(
   void* sendbuf, int  scnts, MPI_Datatype sendtype,
   void* recvbuf, int *rcnts, int *rdispls, MPI_Datatype recvtype,
   MPI_Comm comm)

int MPI_Reduce_scatter(
   void* sendbuf, void* recvbuf, int *rcnts,
   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Other MPI collective functions

```
void MPI_Barrier(MPI_Comm comm)

int MPI_Scan(
  void* sendbuf, void* recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Op_create(
  MPI_User_function *function, int commute, MPI_Op *op)
int MPI_Op_free(MPI_Op *op)
```

commutative :
1: a#b = b#a
0: a#b != b#a

Synchronization

- `MPI_Barrier()` → used to synchronize all processes have called this subroutine.
  `int MPI_Barrier(MPI_Comm comm)`
- Processes started up on different machines run independently from each other. Therefore, different machines may be running different portion of a code in an instance, and running at different speed.
- It is sometimes necessary to ensure all processes are at the same point or at the same pace.
- For example, when friends go out for a long trip in different cars or motorcycles, it is necessary to set up some "synchronization point" so that everyone will reach the destination. (especially when there are drivers who doesn't know how to get there).
- Blocking communication usually results in synchronization.

Examples: 09a_noBarrier.c vs. 09b_barrier.c (compare the output)

MPI_Scan()

- Perform a scan ("partial reduction") of data
  - Also called "all-prefix-sums";

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

```
P0: [ 0  1  2]                          P0: [ 0  1  2]
P1: [ 3  4  5]        Count=3           P1: [ 3  5  7]
P2: [ 6  7  8]        MPI_SUM           P2: [ 9 12 15]
P3: [ 9 10 11]                          P3: [18 22 26]
```

Example: 10_scan.cpp

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()`
  - `MPI_Scatter()`, `MPI_Gather()`
  - `MPI_Allgather()`, `MPI_Allreduce()`
  - `MPI_Barrier()`, `MPI_Scan()`
  - `MPI_Alltoall()`