Lecture 14:

Message Passing Programming

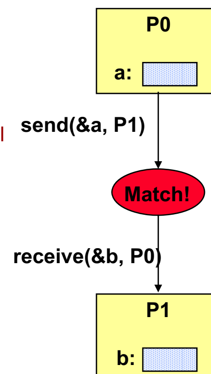
Department of Electrical Engineering
Stanford University

`http://eeclass.stanford.edu/ee282`

Announcements

Message-passing Programs

- Characteristics
  - Separate threads or processes
    - They execute independently and concurrently
  - Separate address spaces
    - Distributed memory model assumed
    - Can be implemented on top of shared memory as well
- Processes transfer data cooperatively
  - Sends & receive library calls
  - Point-to-point communication
- Computation models
  - Single Program Multiple Data (SPMD)
    - All processes are the same program, but act on different data
  - Multiple Program Multiple Data (MPMD)
    - Each process may be a different program

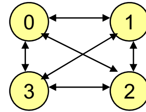


MPI: Message Passing Interface

- Software standard for distributed memory parallel programs
  - Library of routines: communication, utilities
  - MPI is a specification, not a specific implementation
  - Portable & fast
  - Parallel model: MPMD with explicit distributed memory
- MPI covers
  - Point-to-point communication (send/receive)
  - Collective communication
  - Support for library development
- MPI does not cover
  - Fault tolerance
  - Parallel/distributed operating system
- MPI tutorials: http://www-unix.mcs.anl.gov/mpi/tutorial/index.html
- MPI standard: http://www-unix.mcs.anl.gov/mpi/

MPI Application Environment

- The elements of an application are
  - N processes, numbered 0 through N-1
  - Communication paths between them



- MPI_COMM_WORD
  - The set of processes plus the communication paths
  - MPI_COMM_SIZE(): the number of processes (N)
  - MPI_COMM_RANK(): the specific number of a process
- Compiling & running (not standard)
  - mpicc -o hello hello.c
  - mpirun -np 4 ./hello
    - 4 processes but not necessarily 4 processors…

Hello MPI

```
#include <mpi.h>
main(int argc, char *argv[])
{
   int me, nprocs;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &me);
   printf("Hello from node %d of %d\n", me, nprocs);
   MPI_Finalize();
}
```

Output (4 processes)

```
   Hello from node 2 of 4
   Hello from node 0 of 4
   Hello from node 1 of 4
   Hello from node 3 of 4
```


MPI_Send(): sending a message

- The first important MPI function:
  - **MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**
- Buf: address of send buffer (where the message is)
- Count: number of elements
- Datatype: data type of send buffer elements
- Dest: process id of destination process
- Tag: message tag
- Comm: communicator (group of communicating processes)

MPI_Recv(): resceiving a message

- The other important MPI function:
  - **MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)**
- Buf: address of receive buffer
- Count: size of receive buffer in elements
- Datatype: data type of receive buffer elements
- Source: source process id or MPI_ANY_SOURCE
- Tag: message tag or MPI_ANY_TAG
- Status: status object (indicates sender and tag)
  - Receiver can recover source, length, error, etc
- Note: sender & receiver must match
  - Count and datatype
  - Tag and communicator

Simple Example

- Process 1 sends array "A" to process 2 which receives it as "B"
- Process 1:

```
#define TAG 123
double A[10];
MPI_Send(A, 10, MPI_DOUBLE, 1, TAG, MPI_COMM_WORLD);
```

- Process 2:

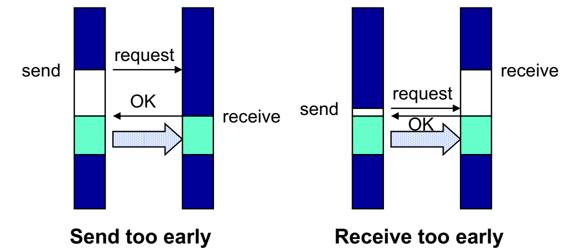
```
#define TAG 123
double B[10];
MPI_Recv(B, 10, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD,
   &status);
```

- or

```
MPI_Recv(B, 10, MPI_DOUBLE, MPI_ANY_SOURCE,
   MPI_ANY_TAG, MPI_COMM_WORLD, &status)
```

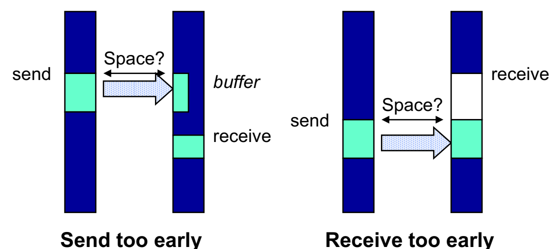

Send/Receive Model Basics I

- MPI doesn't specify whether these calls implement:
  - A fully blocking send *and* receive, OR
  - A *buffered* send and blocking receive
- Double-blocking model doesn't require any buffering:



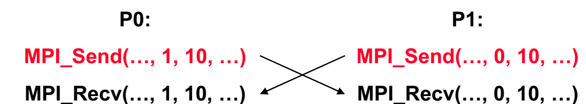
Send/Receive Model Basics II

- Receive-only blocking model requires buffering
  - Sender can hold data until receiver ready
  - Receiver can buffer data in a temporary location
  - Handshake overhead only used to determine buffer overflow

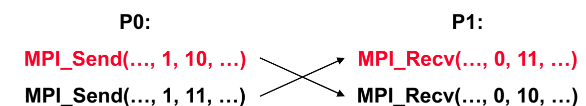


Send/Receive Model Deadlocks I

- But key difference is in deadlock behavior
- These deadlock on double-block but not receive-block:
  - Both block-on-send waiting for receiver:

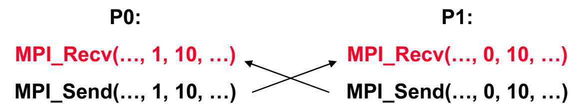


  - Crossed tags block each other:



Send/Receive Model Deadlocks II

- Double-receive deadlocks with either environment:



- Either way, must be CAREFUL to avoid deadlock
  - Must carefully sequence all messages in MP

- One way to avoid common traps: **`MPI_SendRecv`**
  - Exchanging data is a common and dangerous operation
  - Combines a “crossed” send & receive together safely
  - Versions with separate *or* same (**`replace`**) buffers

Side Note: Deadlocks & Livelocks

- Deadlock
  - Threads wait for some event or condition that will <u>never</u> happen
  - Typically two or more threads wait for each other
    - Circular dependency
- Livelock
  - Threads change state in response to each other
  - But no forward progress from the applications point of view
- Starvation
  - Some thread gets deferred forever
  - Milder case: lack of fairness

- General advice
  - Watch out for ordering requirements in your programs
    - At multiple levels
  - Stick to canonical/simple orderings

MPI Program Template

```
main(int argc, char *argv[])
{
  MPI_Init (&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  /* Data distribution */ ...
  /* Computation & Communication*/ ...
  /* Result gathering */ ...
  MPI_Finalize();
}
```


Example: Array Increment (1/3)

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, int argv ) {
   int rank, size. i;
   MPI_Status status;
   int A[100];
   MPI_INIT( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (rank == 0) {
        read_array(A, "input.file", 100); /* read array A*/
        MPI_send(A+25, 25, MPI_INT, 1, 0, MPI_COMM_WORD);
        MPI_send(A+50, 25, MPI_INT, 2, 0, MPI_COMM_WORD);
        MPI_send(A+75, 25, MPI_INT, 3, 0, MPI_COMM_WORD);
   } else {
    MPI_Recv(A, 25, MPI_INT, 0, 0, MPI_COMM_WORD, &status);
 }
```

Example: Array Increment (1/2)

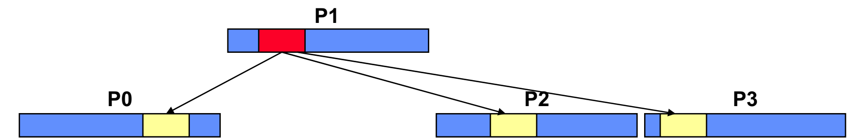
```
  /* computation without any communication */
  for (i=0; i<25 i++) A[i]++;

  if (rank != 0) MPI_send(A, 25, MPI_INT, 0, 1, MPI_COMM_WORD);
  else {
     MPI_Recv(A+25, 25, MPI_INT, 1, 1, MPI_COMM_WORD, &status);
     MPI_Recv(A+50, 25, MPI_INT, 2, 1, MPI_COMM_WORD, &status);
     MPI_Recv(A+75, 25, MPI_INT, 3, 1, MPI_COMM_WORD, &status);
     write_array(A, "output.file", 100); /* write array A*/
  }

  MPI_Finalize( );
  return 0;
}
```

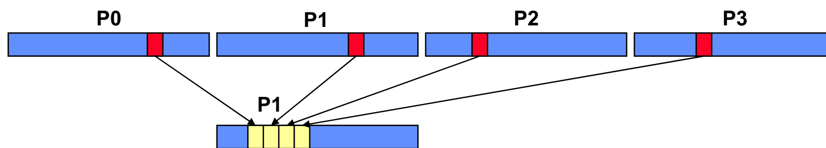

Barrier & Broadcast

- MPI_Barrier(MPI_Comm comm)
  - Global barrier synchronization
  - All processes wait at barrier until all have arrived
- MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)
  - Allows a one-to-all send of a single data block
  - All processors in comm must call this
  - Processor source does a send
  - Other processors do a receive



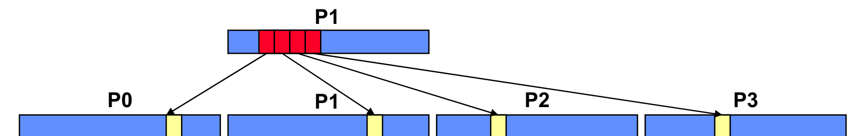
Gather

- MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype datatype, void *recvbuf, …, int target, MPI_Comm comm)
  - Collects together sendbufs from all in comm
    - Each input is sendcount•sizeof(datatype)
  - Outputs all into target's recvbuf in rank order
    - Output size must be p•sendcount•sizeof(datatype)
    - MPI_Allgather lacks a target input, outputs to everyone
  - Good for collecting together bits of a larger structure



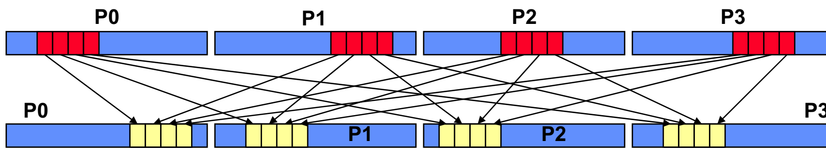
Scatter

- MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype datatype, void *recvbuf, …, int source, MPI_Comm comm)
  - Spreads a sendbuf to all in comm
    - Input size must be p•sendcount•sizeof(datatype)
  - Puts sendcount/p into each p's recvbuf
    - Outputs each get part of input in rank order
    - Each output is sendcount•sizeof(datatype)
  - Good for spreading out bits of a larger structure



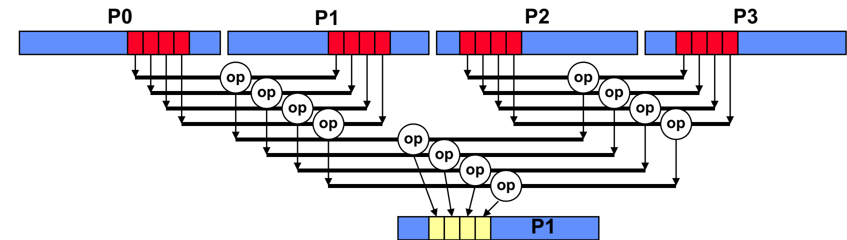
All-to-All

- MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype datatype, void *recvbuf, …, int source, MPI_Comm comm)
  - Trades portions of sendbuf to all in comm
    - Input size must be p•sendcount•sizeof(datatype)
  - Each p puts sendcount/p into each p's recvbuf
    - Outputs each get part of ranked input in rank order
    - Each output is p•sendcount•sizeof(datatype)
  - Good for exchanging changes to a distributed data object



Reduction

- MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int target, MPI_Comm comm)
  - Implements count reductions in parallel:
    - Combines results across comm processors
    - Handles count reductions element-wise from the same buffers
  - target processor's recvbuf holds the result at the end
  - datatype and op control the actual reduction (add, max, etc)



Blocking vs. Non-Blocking Messages

- Blocking messages
  - Sender waits until message is transmitted
    - Send buffer becomes empty
  - Receiver waits until message is received
    - Receive buffer becomes full
  - Remember the deadlock cases

- Non-blocking messages
  - Sender proceeds even if message has not been sent
  - Receiver proceeds even if message has not be received
  - Either can check on the status of the message
  - Non-blocking avoids idle time and (some) deadlocks
    - Overlap computation with communication

Non-blocking Communication in MPI

- Split communication operations into two parts
  - First part initiates the operation & does not block
  - Second part waits for the operation to complete

```
MPI_Request request;
MPI_Recv(buf, count, type, dest, tag, comm, status)
=
MPI_Irecv(buf, count, type, dest, tag, comm, &request) +
MPI_Wait(&request, &status)

MPI_Send(buf, count, type, dest, tag, comm)
=
MPI_Isend(buf, count, type, dest, tag, comm, &request) +
MPI_Wait(&request, &status)
```

Using Non-blocking Communication

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
```

- Process 0:

```
MPI_Irecv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request)
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD)
MPI_Wait(&request, &status)
```

- Process 1:

```
MPI_Irecv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request)
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD)
MPI_Wait(&request, &status)
```

- No deadlock
- Data may be transferred concurrently

Using Non-blocking Communication

- Also possible to use nonblocking send:

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
p=1-me; /* calculates partner in 2 process exchange */
```

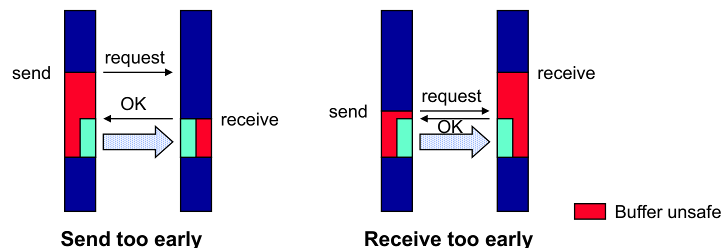
- Process 0 and 1:

```
MPI_Isend(A, 100, MPI_DOUBLE, p, MYTAG, WORLD, &request)
MPI_Recv(B, 100, MPI_DOUBLE, p, MYTAG, WORLD, &status)
MPI_Wait(&request, &status)
```

- No deadlock
  - "status" argument to MPI_Wait doesn't return useful info here
  - Better to use Irecv instead of Isend if only using one.

Non-blocking & Buffering

- Nonblocking calls return with the send/receive incomplete
  - Sends don't immediately flush out the buffer
    - We can't immediately write into the buffer
  - Receives may not be complete
    - We just posted the *address* . . . data comes later



Check for if Communication has Completed

- Problem: How do we know when operations are complete?
  - Use the request record to check the access
  - MPI_Test(MPI_Request *request, int *flag, MPI_Status *status) to check for completion
    - There are also MPI_TestAll, MPI_TestAny, MPI_TestSome
  - MPI_Wait(MPI_Request *request, MPI_Status *status) to block for completion (turn non-block into block)
    - There are also MPI_WaitAll, MPI_WaitAny, MPI_WaitSome
  - Warning: Must MPI_Test = TRUE, MPI_Wait, or MPI_Request_free all request flags!

Other Communication Modes

- Standard mode (MPI_Send): send may not complete until matching receive is posted
- Synchronous mode (MPI_Ssend): the send does not complete until a matching receive has begun
- Buffered mode (MPI_Bsend): the user supplies the buffer to system
- Ready mode (MPI_Rsend): user guarantees that matching receive has been posted
- Non-blocking versions are MPI_Issend, MPI_Irsend, MPI_Ibsend

Other Features of MPI to Keep in Mind

- Communicator topologies
  - Allows MPI to understand logical topology of processes
    - Cartesian, graph, …
  - Sometimes make it easier to write code
- Timing
  - MPI_Wtime()
- Communicators
  - Create private communication domains
- Resources
  - http://www.mpi-forum.org and http://www.mcs.anl.gov/mpi
  - Newsgroups: comp.parallel.mpi
  - Books:
    - Using MPI, by Gropp, Lusk, Skjellum
    - MPI: The Complete Reference, by Snir, Otto, Huss-Lederman, Walker, Dongarra
    - MPI: The Complete Reference, Volume 2, by Gropp, Lederman, Nitzberg, Saphir, Snir
    - Parallel Programming with MPI, by Pacheco. Morgan

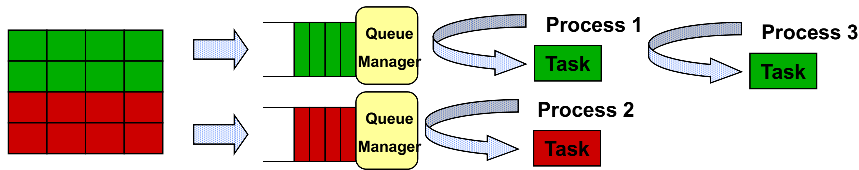
Performance Issues for MPI Programs: Communication Latency & Overhead

- Too much time spend on send() & recv()
- Solution: overlap computation with communication
  - Use non-blocking communication primitives
  - Attempt to send/receive while doing other parallel work
  - Results depend on implementation
- Alternative: grouped communication (granularity change)
  - Use single physical message for many logical messages
  - Amortize the latency of send/receive
  - Same effect with larger messages…
- Alternative: use a "pull model" for communication
  - Do not initiate before the receiver has requested it
- Alternative: replicate work to replace communication
  - Some amount of work may be cheaper than communication

Performance Issues for MPI Programs: Load Imbalance

- Work not perfectly divided between processes
  - Idle time for some process, while other processes are slow
  - Makes communication latency worse
- Solutions:
  - Static balanced work distribution (ok for regular applications)
    - Based on work/data not on data size!
  - Dynamic repartitioning in the algorithm
    - If process done, try to "steal work" from another process (at random)
  - Work queue based algorithm
    - Work queued in one of more locations
    - Fast processes grab work from queues
- Note: these solutions may be tricky to implement in MPI
  - They require irregular communication

Work-queue Model



- Queue manager
  - Handles queue request for processes
  - Make sure it does not become the bottleneck
- When is the work-queue model better
  - High variance in work per task (irregular problem)
  - High variance in processor speed

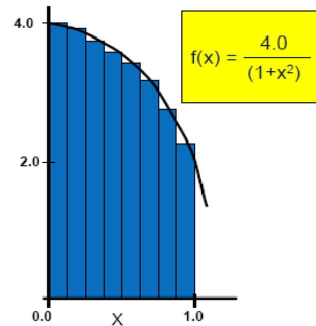
Fine Vs Coarse Grain Parallelization Granularity

- Fine-grain Parallelism
  - Low computation to communication ratio
  - Small amounts of computational work between communication stages
  - Less opportunity for performance enhancement
  - High communication overhead
- Coarse-grain Parallelism
  - High computation to communication ratio
  - Large amounts of computational work between communication events
  - More opportunity for performance increase
  - Harder to load balance efficiently

Another Example: Computing PI with Integration

```
static long num_steps = 100000;
void main() {
   int i;
   double pi, x, step, sum = 0.0;

   step = 1.0 / (double) num_steps;
   for (i = 0; i < num_steps; i++){
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n", pi);
}
```



$$f(x) = \frac{4.0}{(1+x^2)}$$

Computing PI with Integration (MPI)

```
static long num_steps = 100000;
void main(int argc, char* argv[]) {
   int i_start, i_end, i, myid, numprocs;
   double pi, mypi, x, step, sum=0.0;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);

   MBI_BCAST(&num_stemps, 1, MPI_INT, 0, MPI_COMM_WORLD);
   i_start = my_id * (num_steps/numprocs);
   i_end = i_start + (num_steps/numprocs)
   step = 1.0 / (double) num_steps;
   for (i = i_start; i < i_end; i++) {
         x = (i + 0.5) * step;
         sum = sum + 4.0 / (1.0 + x*x);
   }
   mypi = step*sum;

   MPI_REDUCE(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
   if (myid==0) printf("Pi = %f\n", pi);
   MPI_Finalize();
}
```