Outline

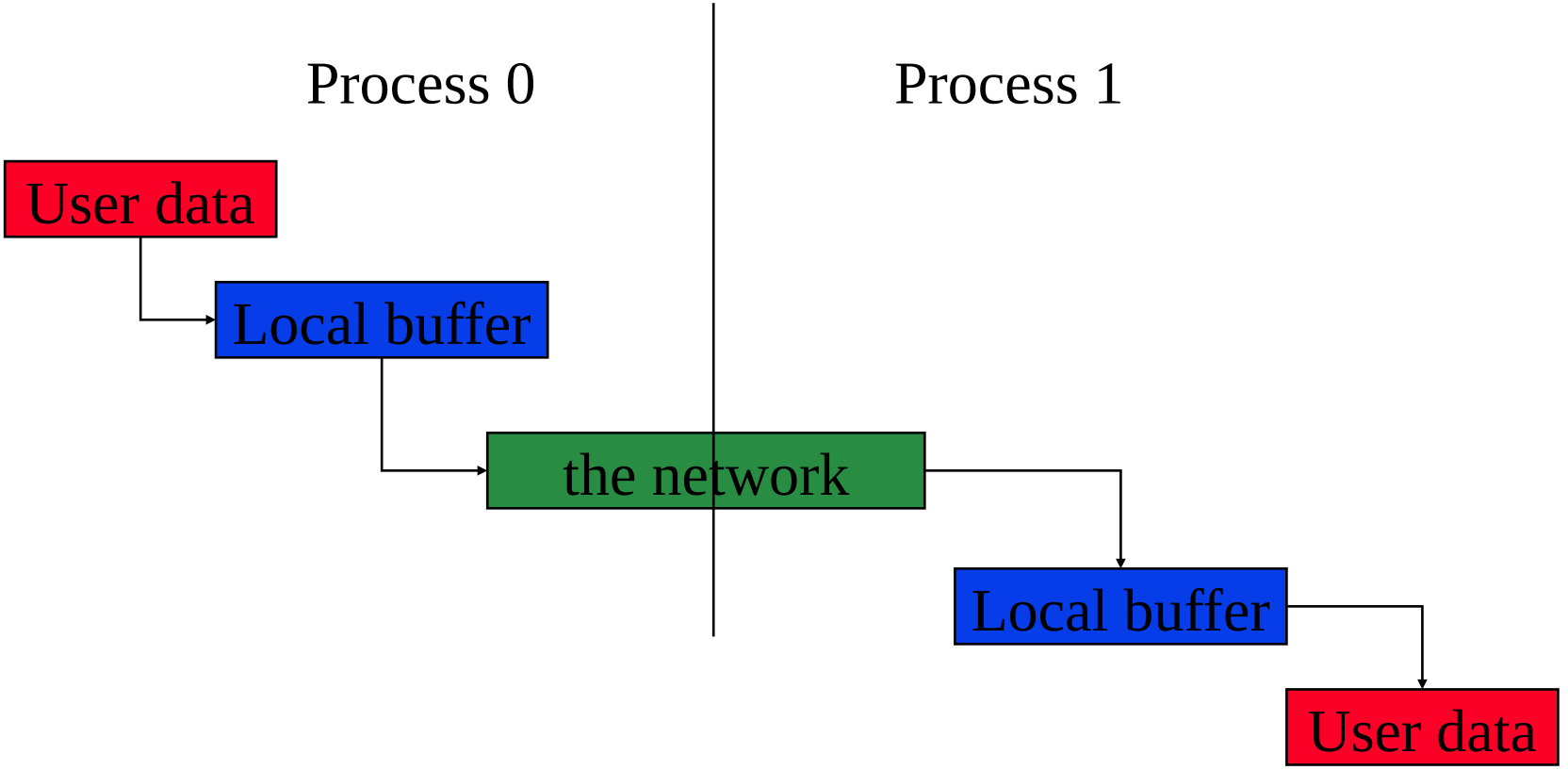
- Review of message passing
- Data movement via message passing
- Hybrid programming: thread + message passing
- RMA: Creating public memory

More on Message Passing

Message passing is a simple programming model, but there are issues

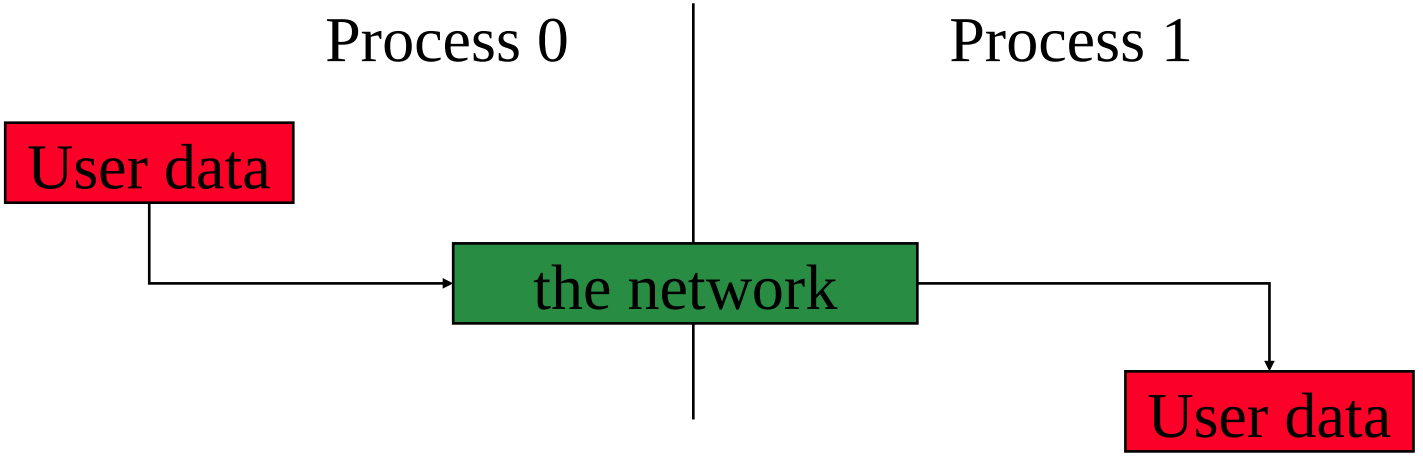
- Buffering and deadlock
- Deterministic execution
- Performance

Buffers: Where does data go?



Avoiding buffering

- Avoiding copies uses less memory
- May use more or less time



This requires that `MPI_Send` wait on delivery, or that `MPI_Send` return before transfer is complete, and we wait later.

Blocking and Non-blocking Communication

- So far we have been using *blocking* communication:
  - `MPI_Recv` does not complete until the buffer is full (available for use).
  - `MPI_Send` does not complete until the buffer is empty (available for use).
- Completion depends on size of message and amount of system buffering.

Sources of Deadlock

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- This is called “unsafe” because it depends on the availability of system buffers in which to store the data sent until it can be received

Two deadlock solutions

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	`Sendrecv(0)`

Two more deadlock solutions

- Supply own space as buffer for send

Process 0	Process 1
`Bsend(1)`	`Bsend(0)`
`Recv(1)`	`Recv(0)`

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`

MPI's Non-blocking Operations

- Non-blocking operations return (immediately) with "request handles" that can be tested and waited on:

```
MPI_Isend(start, count, datatype,
    dest, tag, comm, &request);
MPI_Irecv(start, count, datatype,
    dest, tag, comm, &request);
MPI_Wait(&request, &status);
```

(each request must be Waited on)

One can also test without waiting:

```
MPI_Test(&request, &flag, &status);
```

- Accessing the data buffer without waiting is undefined

Multiple Completions

- It is sometimes desirable to wait on multiple requests:

```
MPI_Waitall(count, array_of_requests,
array_of_statuses)
MPI_Waitany(count, array_of_requests,
 &index, &status)
MPI_Waitsome(count,array_of_requests,
 array_of indices, array_of_statuses)
```

- There are corresponding versions of **`test`** for each of these

Communication modes

- MPI provides multiple *modes* for sending messages:
  - Synchronous mode (**`MPI_Ssend`**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  - Buffered mode (**`MPI_Bsend`**): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.
  - Ready mode (**`MPI_Rsend`**): user guarantees that a matching receive has been posted.
    - Allows access to fast protocols
    - undefined behavior if matching receive not posted
- Non-blocking versions (**`MPI_Issend`**, etc.)
- **`MPI_Recv`** receives messages sent in *any* mode.

Other point-to-point features

- **`MPI_Sendrecv`**
  - Exchange data
- **`MPI_Sendrecv_replace`**
  - Exchange data in place
- **`MPI_Cancel`**
  - Useful for multibuffering
- Persistent requests
  - Useful for repeated communication patterns
  - Some systems can exploit to reduce latency and increase performance

MPI_Sendrecv

- Allows simultaneous send and receive
  - Send and receive datatypes (even type signatures) may be different
  - Can use Sendrecv with plain Send or Recv (or Irecv or Ssend_init, …)

What MPI Functions are in Use?

For simple applications, these are common:

- Point-to-point communication
  - MPI_Irecv, MPI_Isend, MPI_Wait, MPI_Send, MPI_Recv
- Startup, Shutdown
  - MPI_Init, MPI_Finalize
- Information on the processes
  - MPI_Comm_rank, MPI_Comm_size, MPI_Get_processor_name
- Collective communication
  - MPI_Allreduce, MPI_Bcast, MPI_Allgather

Not mentioned...

- Topologies: map a communicator onto, say, a 3D Cartesian processor grid
  - Implementation can provide ideal logical-to-physical mapping
- Rich set of I/O functions: individual, collective, blocking and non-blocking
  - Collective I/O can lead to many small requests being merged for more efficient I/O
- One-sided communication: puts and gets with various synchronization schemes
  - Implementations not well-optimized and rarely used
  - Redesign of interface is underway
- Task creation and destruction: change number of tasks during a run
  - Few implementations available

Implementing Synchronous Message Passing

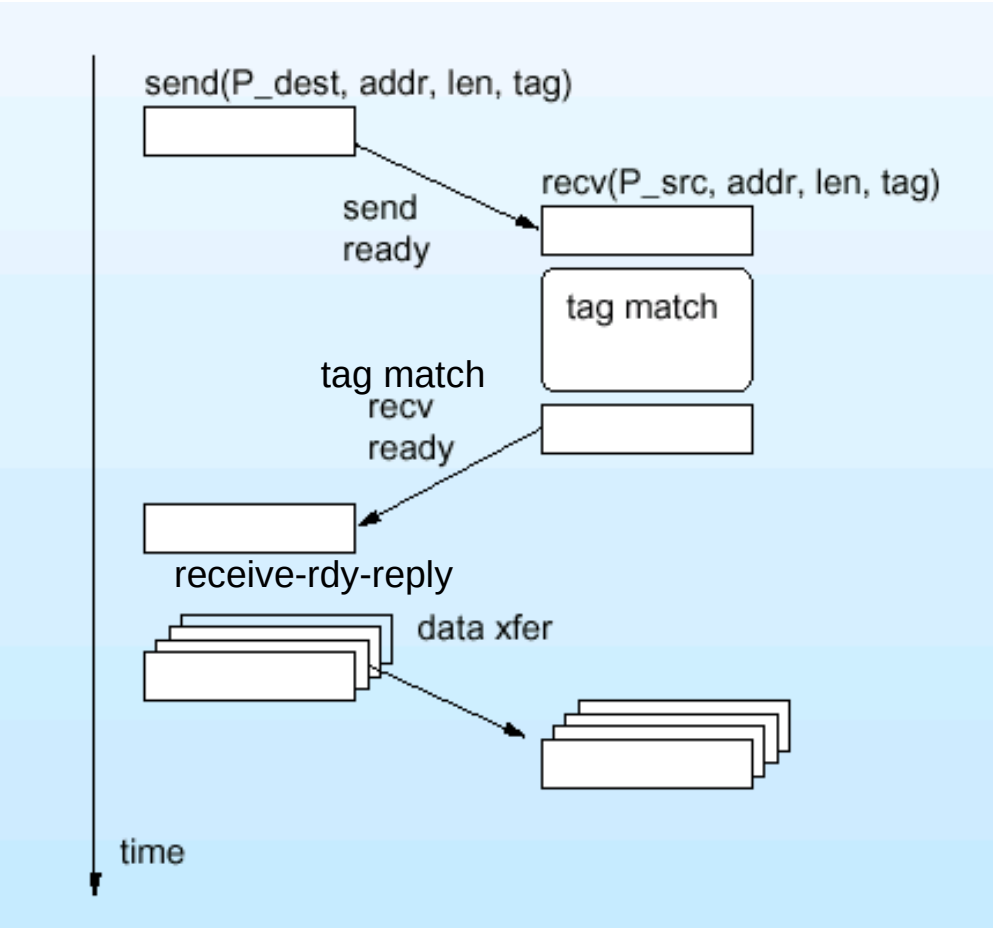
- Send operations complete after matching receive and source data has been sent
- Receive operations complete after data transfer is complete from matching send

1) Initiate send
2) Address translation on P_{dest}
3) Send-Ready Request

4) Remote check for posted receive

5) Reply transaction

6) Bulk data transfer

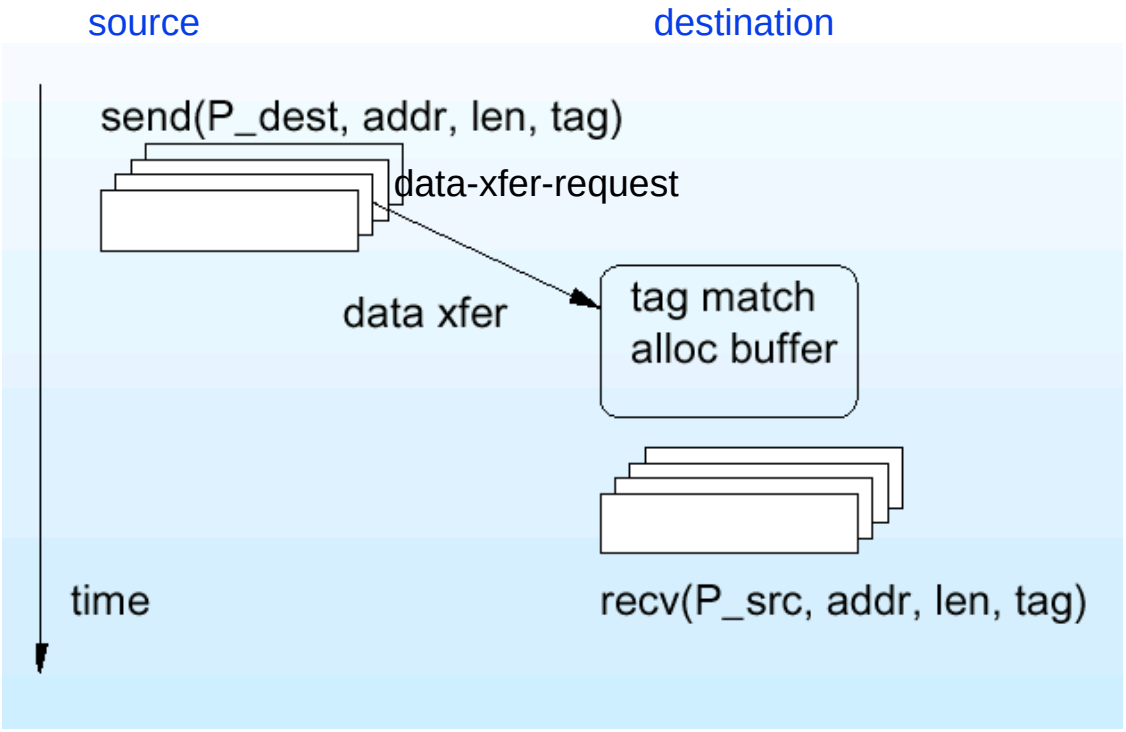


Implementing Asynchronous Message Passing

- Optimistic single-phase protocol assumes the destination can buffer data on demand

1) Initiate send
2) Address translation on P_{dest}
3) Send Data Request

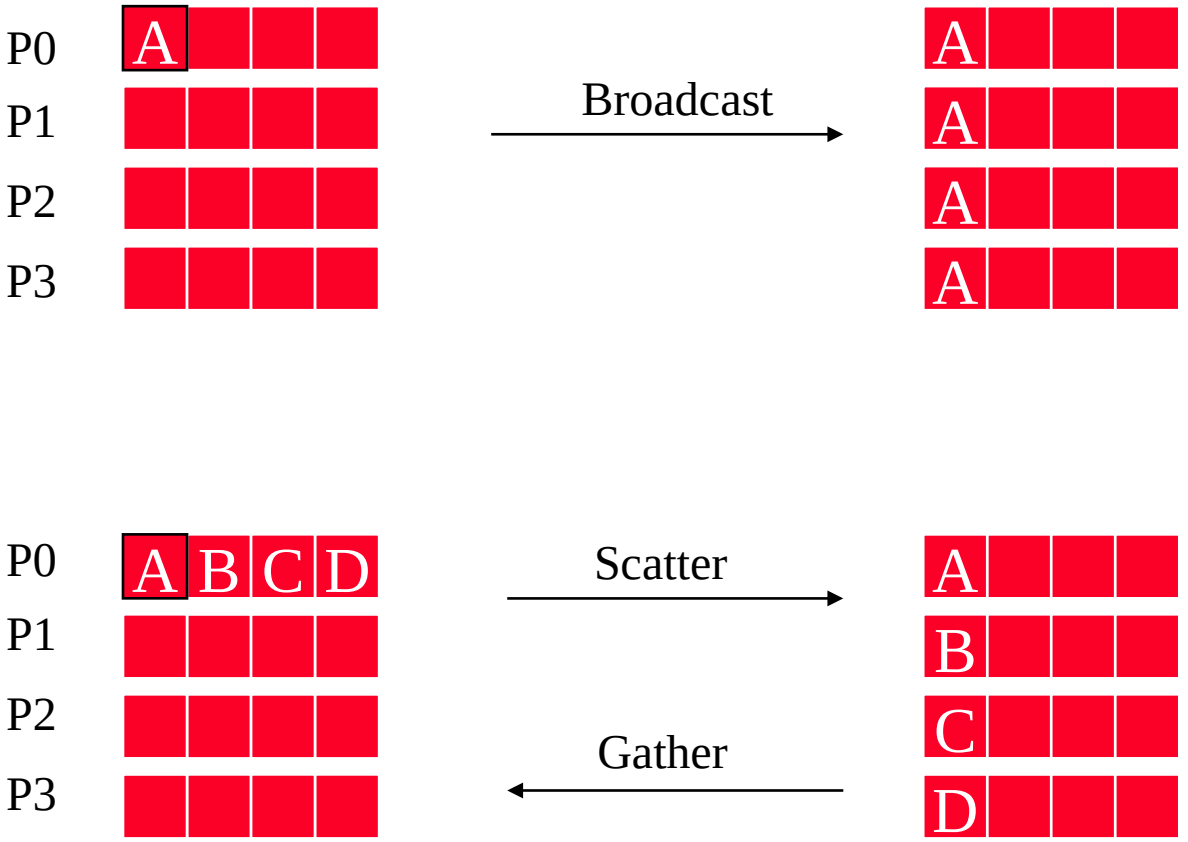
4) Remote check for posted receive
5) Allocate buffer (if check failed)
6) Bulk data transfer



Synchronization

- **`MPI_Barrier( comm )`**
- Blocks until all processes in the group of the communicator **`comm`** call it.
- Almost never required in a parallel program
  - Occasionally useful in measuring performance and load balancing

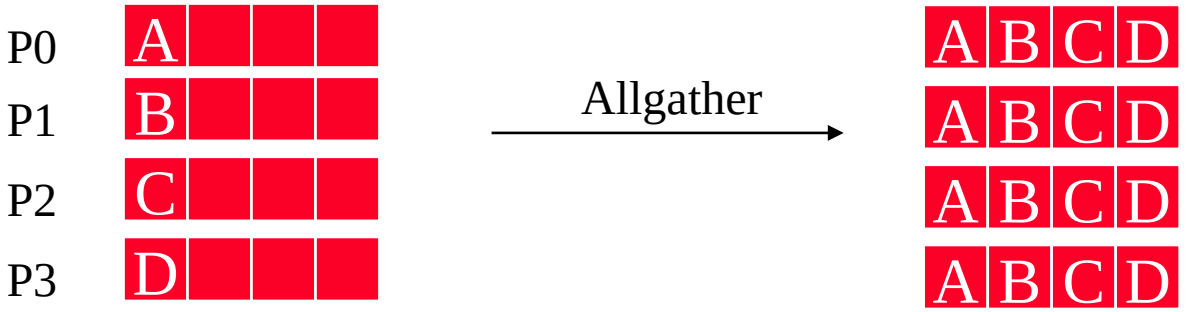
Collective data movement



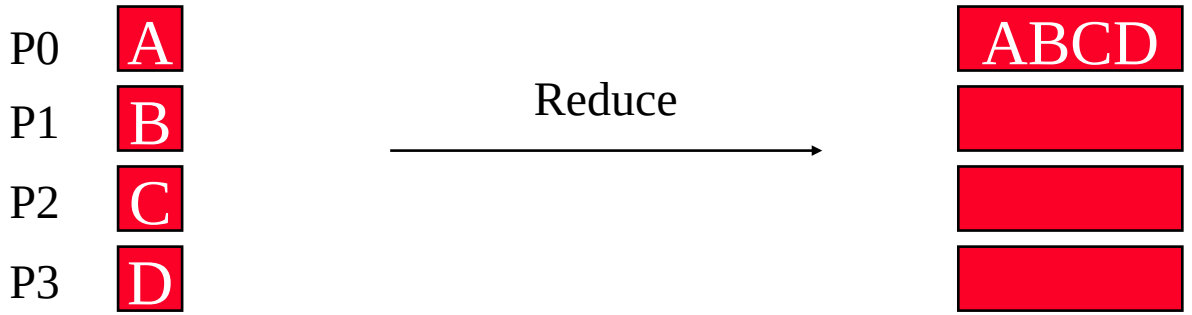
On broadcast...

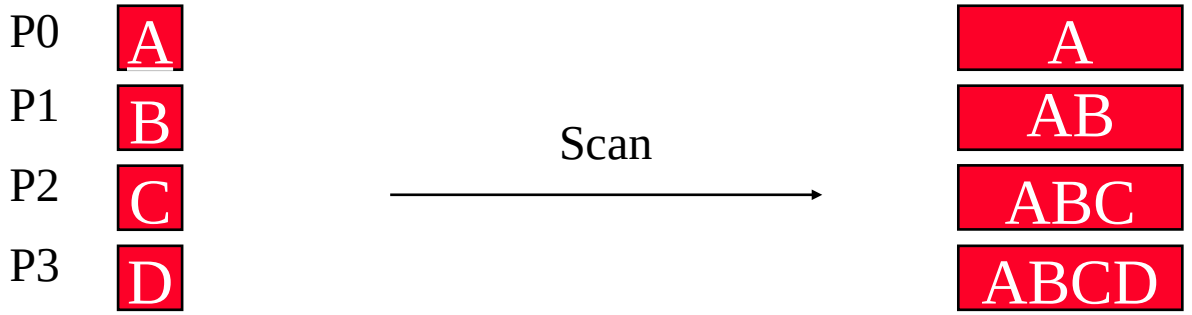
- All collective operations must be called by *all* processes in the communicator
- MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast
  - MPI_Bcast is not a “multi-send”
  - “root” argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive

More collective data movement



Collective computation





MPI Collective Routines

- Many Routines: **`Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv`**
- **`All`** versions deliver results to all participating processes.
- V versions allow the hunks to have variable sizes.
- **`Allreduce`**, **`Reduce`**, **`Reduce_scatter`**, and **`Scan`** take both built-in and user-defined combiner functions.
- MPI-2 adds **`Alltoallw`**, **`Exscan`**, intercommunicator versions of most routines

MPI Built-in Collective Computation Operations

• **MPI_MAX**	Maximum
• **MPI_MIN**	Minimum
• **MPI_PROD**	Product
• **MPI_SUM**	Sum
• **MPI_LAND**	Logical and
• **MPI_LOR**	Logical or
• **MPI_LXOR**	Logical exclusive or
• **MPI_BAND**	Binary and
• **MPI_BOR**	Binary or
• **MPI_BXOR**	Binary exclusive or
• **MPI_MAXLOC**	Maximum and location
• **MPI_MINLOC**	Minimum and location

The Collective Programming Model

- One style of higher level programming is to use *only* collective routines
- Provides a “data parallel” style of programming
  - Easy to follow program flow

Hybrid programming models

- MPI describes parallelism between processes (with separate address spaces)
- Thread parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives
  - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user

Hybrid programming models

Common options for programming *multicore* clusters

- All MPI
  - MPI between processes both within a node and across nodes
  - MPI internally uses shared memory to communicate within a node
- MPI + OpenMP
  - Use OpenMP within a node and MPI across nodes
- MPI + Pthreads
  - Use Pthreads within a node and MPI across nodes

Hybrid programming models

- In MPI-only programming, each MPI process has a single program counter
- In MPI+threads hybrid programming, there can be multiple threads executing simultaneously
  - All threads share all MPI objects (communicators, requests)
  - The MPI implementation might need to take precautions to make sure the state of the MPI stack is consistent

MPI's Four Levels of Thread Safety

- MPI defines four levels of thread safety -- these are commitments the application makes to the MPI
  1) **MPI_THREAD_SINGLE**: only one thread exists in the application
  2) **MPI_THREAD_FUNNELED**: multithreaded, but only the main thread makes MPI calls (the one that called MPI_Init_thread)
  3) **MPI_THREAD_SERIALIZED**: multithreaded, but only one thread at a time makes MPI calls
  4) **MPI_THREAD_MULTIPLE**: multithreaded and any thread can make MPI calls at any time (with some restrictions to avoid races)
- Thread levels are in increasing order
  - If an application works in FUNNELED mode, it can work in SERIALIZED
- MPI defines an alternative to MPI_Init
  - **MPI_Init_thread(requested, provided)**
- Application gives level it needs; MPI implementation gives level it supports

MPI_THREAD_SIN

- There are no threads in the system
  - E.g., there are no OpenMP parallel regions

```
int main(int argc, char ** argv)
{
    int buf[100];
    MPI_Init(&argc, &argv);
    for (i = 0; i < 100; i++)
        compute(buf[i]);
    /* Do MPI stuff */
   MPI_Finalize();
    return 0;
}
```

MPI_THREAD_FUNNELED

- All MPI calls are made by the master thread
  - Outside the OpenMP parallel regions
  - In OpenMP master regions

```
int main(int argc, char ** argv)
{
    int buf[100], provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    if (provided < MPI_THREAD_FUNNELED)
        MPI_Abort(MPI_COMM_WORLD, 1);
#pragma omp parallel for
    for (i = 0; i < 100; i++)
        compute(buf[i]);
    /* Do MPI stuff */
   MPI_Finalize();
    return 0;
}
```

MPI_THREAD_SERIALIZED

- Only one thread can make MPI calls at a time
  - Protected by OpenMP critical regions

```
iint main(int argc, char ** argv)
{
    int buf[100], provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
    if (provided < MPI_THREAD_SERIALIZED)
        MPI_Abort(MPI_COMM_WORLD, 1);
#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
#pragma omp critical
        /* Do MPI stuff */
    }
MPI_Finalize();
    return 0;
}
```

MPI_THREAD_MULTIPLE

- Any thread can make MPI calls any time (with restrictions)

```
int main(int argc, char ** argv)
{
    int buf[100], provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    if (provided < MPI_THREAD_MULTIPLE)
        MPI_Abort(MPI_COMM_WORLD, 1);
#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
        /* Do MPI stuff */
    }
MPI_Finalize();
return 0;
}
```

Threads and MPI

- An implementation is not required to support levels higher than MPI_THREAD_SINGLE; that is, an implementation is *not* required to be thread safe
- A fully thread-safe implementation will support MPI_THREAD_MULTIPLE
- A program that calls MPI_Init (instead of MPI_Init_thread) should assume that only MPI_THREAD_SINGLE is supported
- A threaded MPI program that does not call MPI_Init_thread is an *incorrect program*

Specification of MPI_THREAD_MULTIPLE

- *Ordering*: When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
  - Ordering is maintained within each thread
  - User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
    - E.g., cannot call a broadcast on one thread and a reduce on another thread on the same communicator
  - It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
    - E.g., accessing an info object from one thread and freeing it from another thread
- *Blocking*: Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions

MPI threads in action

- All MPI implementations support MPI_THREAD_SINGLE
- They probably support MPI_THREAD_FUNNELED even if they don't admit it.
  - Does require thread-safe malloc
  - Probably OK in OpenMP programs
- Many (but not all) implementations support MPI_THREAD_MULTIPLE
  - Hard to implement efficiently though (lock granularity issue)
- "Easy" OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need FUNNELED
  - So don't need "thread-safe" MPI for many hybrid programs

What we need to know in MPI RMA

- How to create remote accessible memory?
- Reading, Writing and Updating remote memory
- Data Synchronization
- Memory Mode

Creating Public Memory

- Any memory used by a process is, by default, only locally accessible
  - X = malloc(100);
  - One of the benefits of MPI is that by making each process's memory private by default, it ensures "locality"
- Once the memory is allocated, the user has to make an explicit MPI call to declare a memory region as remotely accessible
  - MPI terminology for remotely accessible memory is a "window"
  - A group of processes collectively create a "window"
- Once a memory region is declared as remotely accessible, all processes in the window can read/write data to this memory without explicitly synchronizing with the target process

Window creation models

- **MPI_WIN_CREATE**
  - You already have an allocated buffer that you would like to make remotely accessible
- **MPI_WIN_ALLOCATE**
  - You want to create a buffer and directly make it remotely accessible
- **MPI_WIN_CREATE_DYNAMIC**
  - You don’t have a buffer yet, but will have one in the future
  - You may want to dynamically add/remove buffers to/from the window
- **MPI_WIN_ALLOCATE_SHARED**
  - You want multiple processes on the same node share a buffer

MPI_WIN_ALLOCATE

- int **MPI_Win_allocate**(MPI_Aint size, int disp_unit,
  MPI_Info info, MPI_Comm comm, void *baseptr,
  MPI_Win *win)
- Create a remotely accessible memory region in an RMA window
- Only data exposed in a window can be accessed with RMA ops.
- Arguments:
  - size - size of local data in bytes (nonnegative integer)
  - disp_unit - local unit size for displacements, in bytes (positive integer)
  - info - info argument (handle)
  - comm - communicator (handle)
  - baseptr - pointer to exposed local data
  - win - window (handle)

MPI_Win_allocate example

```
int main(int argc, char ** argv)
{
    int *a;  MPI_Win win;
    MPI_Init(&argc, &argv);
    /* collectively create remote accessible memory in a window */
    MPI_Win_allocate(1000*sizeof(int), sizeof(int), MPI_INFO_NULL,
                     MPI_COMM_WORLD, &a, &win);
   /* Array ‘a’ is now accessible from all processes in
    * MPI_COMM_WORLD
    */
    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
```

MPI_WIN_CREATE_DYNAMIC

int **MPI_Win_create_dynamic**(MPI_Info info, MPI_Comm comm, MPI_Win *win)

- Create an RMA window, to which data can later be attached
  - Only data exposed in a window can be accessed with RMA ops
- Initially “empty”
  - Application can dynamically attach/detach memory to this window by calling MPI_Win_attach/detach
  - Application can access data on this window only after a memory region has been attached
- Window origin is MPI_BOTTOM
  - Displacements are segment addresses relative to MPI_BOTTOM
  - Must tell others the displacement after calling attach

MPI_Win_allocate example

```
iint main(int argc, char ** argv)
{
    int *a;    MPI_Win win;
    MPI_Init(&argc, &argv);
    MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &win);
    /* create private memory */
    a = (int *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;
    /* locally declare memory as remotely accessible */
    MPI_Win_attach(win, a, 1000*sizeof(int));   /* Array ‘a’ is now accessible to all processes */
    /* undeclare remotely accessible memory */
    MPI_Win_detach(win, a);  free(a);
    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```

Data movement in RMA

MPI provides ability to read, write and atomically modify data in remotely accessible memory regions

- **MPI_PUT**
- **MPI_GET**
- **MPI_ACCUMULATE**
- **MPI_GET_ACCUMULATE**
- **MPI_COMPARE_AND_SWAP**
- **MPI_FETCH_AND_OP**

Data movement: Put

MPI_Put(void * origin_addr, int origin_count,
MPI_Datatype origin_datatype, int target_rank,
MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Win win)

- Move data from origin to target
- Separate data description triples for origin and target

Data movement: Get

MPI_Get(void * origin_addr, int origin_count,
MPI_Datatype origin_datatype, int target_rank,
MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Win win)

- Move data to origin from target

RMA Synchronization Models

- RMA data access model
  - When is a process allowed to read/write remotely accessible memory?
  - When is data written by process X is available for process Y to read?
  - RMA synchronization models define these semantics
- Three synchronization models provided by MPI:
  1) Fence (active target)
  2) Post-start-complete-wait (generalized active target)
  3) Lock/Unlock (passive target)
- Data accesses occur within “epochs”
  - Access epochs: contain a set of operations issued by an origin process
  - Exposure epochs: enable remote processes to update a target’s window
  - Epochs define ordering and completion semantics
  - Synchronization models provide mechanisms for establishing epochs
    - E.g., starting, ending, and synchronizing epochs

Ordering of Operations in MPI RMA

- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the same location undefined
- Result of Get concurrent Put/Accumulate undefined
  - Can be garbage in both cases
- Result of concurrent accumulate operations to the same location are defined according to the order in which the occurred
  - Atomic put: Accumulate with op = MPI_REPLACE
  - Atomic get: Get_accumulate with op = MPI_NO_OP
- Accumulate operations from a given process are ordered by default
  - User can tell the MPI implementation that (s)he does not require ordering as optimization hint
  - You can ask for only the needed orderings: RAW (read-after-write), WAR, RAR, or WAW

One-sided communication

- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes (*shared memory)*
  - Other processes can directly read from or write to this memory
- Passive mode: One-sided, asynchronous communication
  - Target does not participate in communication operation

Passive Target Synchronization

MPI_Win_lock(int locktype, int rank, int assert, MPI_Win win)

MPI_Win_unlock(int rank, MPI_Win win)

- Begin/end passive mode epoch
  - Target process does not make a corresponding MPI call
  - Can initiate multiple passive target epochs to different processes
  - Concurrent epochs to same process not allowed (affects threads)
- Lock type
  - SHARED: Other processes using shared can access concurrently
  - EXCLUSIVE: No other processes can access concurrently

Simulation outline

- Discrete event systems
  - Time and space are discrete
- Particle systems
  - Important special case of lumped systems
- Lumped systems (ODEs)
  - Location/entities are discrete, time is continuous
- Continuous systems (PDEs)
  - Time and space are continuous

Particle Systems

- A particle system has
  - a finite number of particles moving in space according to Newton's Laws (i.e. $F = ma$)
  - time is continuous
- Examples
  - stars in space with laws of gravity
  - electron beam in semiconductor manufacturing
  - atoms in a molecule with electrostatic forces
  - neutrons in a fission reactor
  - cars on a freeway with Newton's laws plus model of driver and engine
  - balls in a pinball game
- Reminder: many simulations combine techniques such as particle simulations with some discrete events

Forces in particle systems

- Force on each particle can be subdivided: force = external_force + nearby_force + far_field_force
- External force
  - ocean current in sharks and fish world
  - externally imposed electric field in electron beam
- Nearby force
  - sharks attracted to eat nearby fish
  - balls on a billiard table bounce off of each other
  - Van der Waals forces in fluid ($1/r^6$)
- Far-field force
  - fish attract other fish by gravity-like ($1/r^2$) force
  - gravity, electrostatics, radiosity in graphics
  - forces governed by elliptic PDE

Particle example: Fish and current

```
%    fishp = array of initial fish positions (stored as complex numbers)
%    fishv = array of initial fish velocities (stored as complex numbers)
%    fishm = array of masses of fish
%    tfinal = final time for simulation (0 = initial time)
%   Algorithm: integrate using Euler's method with varying step size
%   Initialize time step, iteration count, and array of times
     dt = .01;   t = 0;
%  loop over time steps
     while t < tfinal,
        t = t + dt;
        fishp = fishp + dt*fishv;
        accel = current(fishp)./fishm;       % current depends on position
        fishv = fishv + dt*accel;
%       update time step (small enough to be accurate, but not too small)
        dt = min(.1*max(abs(fishv))/max(abs(accel)),1);
    end
```

Parallelism in external forces

- These are the simplest
  - The force on each particle is independent
- Called “embarrassingly parallel”
- Evenly distribute particles on processors: Any distribution works
- Locality is not an issue, no communication
- For each particle on processor, apply the external force
- Also called “map”; May need to “reduce” (eg compute maximum) to compute time step, other data

Parallelism in nearby forces

- Nearby forces require interaction and therefore communication.
- Force may depend on other nearby particles:
  - Example: collisions.
  - simplest algorithm is $O(n^2)$: look at all pairs to see if they collide.
- Usual parallel model is domain decomposition of physical region in which particles are located
  - O(n/p) particles per processor if evenly distributed.

Parallelism in nearby forces: Interactions

- Challenge 1: interactions of particles near processor boundary:
  - need to communicate particles near boundary to neighboring processors.
    - Region near boundary called “ghost zone”
  - Low surface to volume ratio means low communication
  - Use squares, not slabs, to minimize ghost zone sizes

Parallelism in nearby forces: Load imbalance

- Challenge 2: load imbalance, if particles cluster:
  - galaxies, electrons hitting a device wall.
- To reduce load imbalance, divide space unevenly.
  - Each region contains roughly equal number of particles.
  - Data structure to use: Quad-tree in 2D, oct-tree in 3D.

Parallelism in far-field forces

- Far-field forces involve all-to-all interaction and therefore communication
- Force depends on all other particles:
  - Examples: gravity, protein folding
- Simplest algorithm is $O(n^2)$
- Just decomposing space does not help since every particle needs to “visit” every other particle.
- Use more clever algorithms to reduce communication
- Use more clever algorithms to beat $O(n^2)$

Parallelism in far-field forces: Particle meshes

- Based on approximation:
  - Superimpose a regular mesh.
  - “Move” particles to nearest grid point.
- Exploit fact that the far-field force satisfies a PDE that is easy to solve on a regular mesh:
  - FFT, multigrid (described in future lectures)
  - Cost drops to O(n log n) or O(n) instead of $O(n^2)$
- Accuracy depends on the fineness of the grid is and the uniformity of the particle distribution.

Parallelism in far-field forces: Tree decomposition

- Based on approximation
  - Forces from group of far-away particles “simplified” -- resembles a single large particle.
  - Use tree; each node contains an approximation of descendants.
- Also O(n log n) or O(n) instead of $O(n^2)$.
- Several Algorithms
  - Barnes-Hut.
  - Fast multipole method (FMM) of Greengard/Rohklin.
  - Anderson’s method

Summary of particle methods

- Model contains discrete entities, namely, particles
- Time is continuous – must be discretized to solve
- Simulation follows particles through timesteps
  - Force = external _force + nearby_force + far_field_force
  - All-pairs algorithm is simple, but inefficient, $O(n^2)$
  - *Particle-mesh* methods approximates by moving particles to a regular mesh, where it is easier to compute forces
  - *Tree-based* algorithms approximate by treating set of particles as a group, when far away