Introduction to MPI

Matt Turner and Bob Robey

Parallel Computing Summer Research Internship
June 10th, 2021
Acknowledgements

• Original MPI material by Kris Garrett, previous PCSRI co-lead
How MPI Works

Code

mpirun

Cluster

Node 0

Node 1

Node 2

Core 0  Core 1

Core 0  Core 1

Core 0  Core 1
MPI Allows Model Domain to be Decomposed Across Nodes
How MPI Works

• MPI = Message Passing Interface
• Executable is run in multiple processes
• Each process communicates with each other
  • Processes may be on the same computer
  • Processes may be on multiple nodes of a cluster
  • Multiple processes may be placed on a node to utilize multi-core processors
• C and Fortran library APIs are given by the standard
• Other 3rd party bindings exist (Python, C++, etc)
• Will concentrate on C library bindings here
MPI is an API

• MPI is just an Application Programming Interface (API)
• MPI standard specifies what a call to each routine should look like and how each routine should behave
  • Does not specify how each routine should be implemented
• Implementations are often library specific
• There are multiple open-source and proprietary implementations
  • IntelMPI (proprietary)
  • OpenMPI (open-source)
  • MPICH (open-source)
  • MVAPICH (open-source)
  • Cray MPI (proprietary, derived from MPICH)
First Program

```c
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);

    printf("%d of %d\n", rank, size);
    MPI_Finalize();

    return 0;
}
```
First Program

• Wrapper is used to compile MPI application (OpenMPI)
  • mpicc –o my_prog my_prog.c

• Wrapper is used to run application
  • mpirun -n 4 ./my_prog
  • For this example, 4 copies of my_prog are run
  • Each copy has an associated index called a rank
  • Processes are run in non-deterministic order

• MPI uses the concept of a communicator
  • Default MPI_COMM_WORLD for all MPI ranks
  • Can create subsets of ranks
    • Useful for libraries
Point-to-point communication
Point to Point Communication

• **MPI_Send** and **MPI_Recv** used for communication between 2 ranks

• **Parameters include**

  • Data to send (**MPI_Send**) – defined by triplet
    • Pointer to memory
    • Count
    • Datatype
  
  • Buffer to copy received data (**MPI_Recv**) – also defined by data triplet
  
  • Envelope – communication triplet
    • Rank to send to or receive from
    • Communicator and integer tag
      – Both must match in a send/recv
      – Can use MPI_ANY_TAG
Concept of a message

Message contains pointer to memory count datatype

Sending mailbox with message ready to go

Receiving mailbox with space available for receipt
### MPI_Send

```c
int MPI_Send(
    const void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm)
```

- **Data triplet**
  - `buf` – Buffer of data to send
  - `count` – Number of items to send
  - `datatype` – Built-in (MPI_INT, MPI_BYTE, MPI_DOUBLE, ...) or your own

- **Envelope triplet**
  - `dest` – Destination MPI rank
  - `tag` – Identifier for the data
  - `comm` – MPI communicator (MPI_COMM_WORLD or your own)
Example

Send array of double values to rank+1

double sendDoubles[2];
double recvDoubles[2];

if (rank < RANK_MAX) {
    MPI_Send(sendDoubles, 2, MPI_DOUBLE,
             rank+1, 0, MPI_COMM_WORLD);
}
if (rank > 0) {
    MPI_Recv(recvDoubles, 2, MPI_DOUBLE,
              rank-1, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
}
Example

• MPI_Send is hard to predict
  • After return from MPI_Send, you can reuse the data buffer
  • But you don’t know if the data has been sent when MPI_Send returns
    • Small messages: MPI returns before data is received by destination
    • Large messages: MPI returns after data is received by destination
• Can use MPI_Ssend to ensure message has been received by destination when function returns
• But, if MPI_Ssend works, MPI_Send should work and MPI_Send could yield higher performance
Example

- If MPI_Send does not return until data is received
  - All ranks except N-1 begin a send
  - Only rank N-1 gets to MPI_Recv statement
  - Rank N-1 receives data and rank N-2 finishes send
  - Rank N-2 receives data and rank N-3 finishes send
  - This continues *sequentially***
- This code may not parallelize
Example

Send array of double values to rank+1

double sendDoubles[2];
double recvDoubles[2];

if (rank < RANK_MAX) {
    MPI_Send(sendDoubles, 2, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD);
}
if (rank > 0) {
    MPI_Recv(recvDoubles, 2, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
Example

Send array of double values to rank+1

double sendDoubles[2];
double recvDoubles[2];

if (rank < RANK_MAX) {
    MPI_Send(sendDoubles, 2, MPI_DOUBLE,
             rank+1, 0, MPI_COMM_WORLD);
}
if (rank > 0) {
    MPI_Recv(recvDoubles, 2, MPI_DOUBLE,
              rank-1, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
}
Example

Send array of double values to rank+1

double sendDoubles[2];
double recvDoubles[2];

if (rank < RANK_MAX) {
    MPI_Send(sendDoubles, 2, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD);
}

if (rank > 0) {
    MPI_Recv(recvDoubles, 2, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
Example

Send array of double values to rank+1

double sendDoubles[2];
double recvDoubles[2];

if (rank < RANK_MAX) {
    MPI_Send(sendDoubles, 2, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD);
}
if (rank > 0) {
    MPI_Recv(recvDoubles, 2, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
}
Example

Send array of double values to rank+1

double sendDoubles[2];
double recvDoubles[2];

if (rank < RANK_MAX) {
    MPI_Send(sendDoubles, 2, MPI_DOUBLE,
             rank+1, 0, MPI_COMM_WORLD);
}
if (rank > 0) {
    MPI_Recv(recvDoubles, 2, MPI_DOUBLE,
              rank-1, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
}
Example

• Several solutions exist
• One solution is to use Isend/Irecv
  • These use nonblocking calls
  • Data buffer cannot be used after Isend/Irecv return
    • *The data in the buffer isn’t used yet*
  • Use MPI_Wait, Mpi_Waitall, MPI_Waitany to know when Isend/Irecv is done
    • Must have an MPI_Wait for each Isend/Irecv.
    • *Not having one creates a memory leak*

• Best practice
  • Post MPI_Irecv before MPI_Isend
Isend/Irecv

int MPI_Isend(
    const void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm,
    MPI_Request *request)

int MPI_Irecv(
    void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm,
    MPI_Request *request)

- request – Used by MPI_Wait
Example

Send array of double values to rank+1

double sendDoubles[2];
double recvDoubles[2];
MPI_Request requests[2] = {MPI_REQUEST_NULL,
                          MPI_REQUEST_NULL};

if (rank < RANK_MAX) {
    MPI_Isend(sendDoubles, 2, MPI_DOUBLE,
              rank+1, 0, MPI_COMM_WORLD, &requests[0]);
}
if (rank > 0) {
    MPI_Irecv(recvDoubles, 2, MPI_DOUBLE,
              rank-1, 0, MPI_COMM_WORLD, &requests[1]);
}
MPI_Waitall(2, requests, MPI_STATUSES_IGNORE);
Example

Send array of double values to rank+1

double sendDoubles[2];
double recvDoubles[2];
MPI_Request requests[2] = {MPI_REQUEST_NULL, MPI_REQUEST_NULL};

if (rank < RANK_MAX) {
    MPI_Isend(sendDoubles, 2, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD, &requests[0]);
}
if (rank > 0) {
    MPI_Irecv(recvDoubles, 2, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, &requests[1]);
}
MPI_Waitall(2, requests, MPI_STATUSES_IGNORE);
Example

Send array of double values to rank+1

double sendDoubles[2];
double recvDoubles[2];
MPI_Request requests[2] = {MPI_REQUEST_NULL,
                          MPI_REQUEST_NULL};

if (rank < RANK_MAX) {
    MPI_Isend(sendDoubles, 2, MPI_DOUBLE,
              rank+1, 0, MPI_COMM_WORLD, &requests[0]);
}
if (rank > 0) {
    MPI_Irecv(recvDoubles, 2, MPI_DOUBLE,
              rank-1, 0, MPI_COMM_WORLD, &requests[1]);
}
MPI_Waitall(2, requests, MPI_STATUSES_IGNORE);
Send/Recv is a common source of program hangs

- **Blocking Receive First**
  - Rank 0: Receive
  - Rank 1: Receive
  - Blocked waiting for data
  - Send

- **Blocking Send First**
  - Rank 0: Send
  - Rank 1: Send
  - If buffer is available, copy and continue
  - Receive

- **Alternating Send Receive**
  - Rank 0: Send
  - Rank 1: Receive
  - Operation completed, can continue
  - Receive

- **Non-blocking Send Receive**
  - Rank 0: Irecv
  - Rank 1: Irecv
  - All operations are posted and now we wait for completion
  - Waitall

---

Blocking communication calls

Non-blocking communication calls
Collective Routines
Broadcast and Collective Routines

• These functions include all ranks in a communicator
• Simplest is MPI_Barrier
  • Make all ranks wait until they hit the barrier
  • All ranks in communicator must call this before code moves forward
  • Be careful not to put this in a branching statement (like an if statement)
    • This can result in the simulation hanging
  • Can be useful for debugging
MPI_Scatter

Elements in array go to different ranks
(Bcast: 1 element sent to all ranks)

Rank 0  Rank 1  Rank 2  Rank 3

Rank 0
**MPI_Gather**

Element from each rank goes into one array
(Allgather: every rank gets the whole array)

Rank 0  Rank 1  Rank 2  Rank 3

Rank 0
MPI_Reduce

Binary operation of element sent to one rank
Built-in operations: max, min, sum, product, ...
Can define your own binary operation
(Allreduce: All ranks get answer)

Rank 0

x0

Rank 1

x1

Rank 2

x2

Rank 3

x3

x0 + x1 + x2 + x3

Rank 0
MPI_Scan

Binary operation on elements from each rank

Rank 0

\[ x_0 \]

\[ x_0 \to x_0 \]

Rank 1

\[ x_1 \]

\[ x_0 + x_1 \to x_0 + x_1 \]

Rank 2

\[ x_2 \]

\[ x_0 + x_1 + x_2 \to x_0 + x_1 + x_2 \]
MPI_Init_thread

- Use for threading with MPI (such as OpenMP, pthreads, ...)
- Four threading types
  - MPI_THREAD_SINGLE – No threading
  - MPI_THREAD_FUNNELED – All MPI calls made by master thread
  - MPI_THREAD_SERIALIZED – Only one thread makes an MPI call at a time, but calls can come from different threads
  - MPI_THREAD_MULTIPLE – Different threads may call MPI routines at the same time
MPI Datatypes
### MPI Datatypes

MPI defines commonly used datatypes, corresponding to datatype from the language.

<table>
<thead>
<tr>
<th>C/C++ Datatype</th>
<th>MPI Datatype</th>
<th>Fortran Datatype</th>
<th>MPI Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>MPI_CHAR</td>
<td>character(len=1)</td>
<td>MPI_CHARACTER</td>
</tr>
<tr>
<td>int</td>
<td>MPI_INT</td>
<td>integer</td>
<td>MPI_INTEGER</td>
</tr>
<tr>
<td>long int</td>
<td>MPI_LONG</td>
<td>integer(8)</td>
<td>MPI_INTEGER8</td>
</tr>
<tr>
<td>float</td>
<td>MPI_FLOAT</td>
<td>real</td>
<td>MPI_REAL</td>
</tr>
<tr>
<td>double</td>
<td>MPI_DOUBLE</td>
<td>double precision</td>
<td>MPI_DOUBLE_PRECISION</td>
</tr>
<tr>
<td>long double</td>
<td>MPI_LONG_DOUBLE</td>
<td>real(16)</td>
<td>MPI_REAL16</td>
</tr>
</tbody>
</table>
MPI Datatypes

You can create special datatypes. There are many different ways.

- **MPI_Type_contiguous** -- makes a block of contiguous data into a type.
- **MPI_Type_vector** -- this creates a type out of blocks of strided data.
- **MPI_Type_create_subarray** -- creates a rectangular subset of a larger array.
- **MPI_Type_indexed, MPI_Type_create_hindexed** - an irregular set of indices described by a set of block lengths and displacements. The hindexed version expresses the displacements in bytes instead of a datatypes for more generality.
- **MPI_Type_create_struct** - creates a datatype encapsulating the data items in a structure in a portable way that accounts for padding by the compiler.
Some Custom MPI Datatypes

Type diagrams for:
- **MPI_Type_contiguous**: Count
- **MPI_Type_vector**: Stride
- **MPI_Type_create_subarray**: 
  
Types are created with a MPI type function followed by a commit call. A free is needed when done.
- **MPI_Type_Commit** - initializes the new custom type with needed memory allocation or other setup.
- **MPI_Type_Free** - frees any memory or data structure entries from the creation of the datatype.
Ghost Cell Exchanges
Terminology

• Halo – set of cells surrounding a computational mesh domain
• Boundary cell halo – cells that impose a boundary condition
  • Avoids conditionals in sweeping through mesh calculations
    • Reflection
    • Outflow
    • Inflow
• Ghost cells – set of halo cells to hold values from adjacent computational domains
  • Only needed for parallel runs – won’t exist for a serial run
  • Ghost cells are a convenience to avoid communication with adjacent processes for each individual value needed
Ghost Cell Exchange
First in horizontal and then vertical
Ghost Exchange Methods

• **Pack technique**
  • Provides compatibility with older implementations
  • The MPI_Pack routine can pack any datatype. Unpack in the correct order on the other side. This makes it more general than other methods
  • Additional copy may make this method slower

• **Array assignment – copy into buffer**
  • Similar to pack but copies into a buffer (vector).
  • Still has an additional copy, but might vectorize the copy
  • Only can communicate a single datatype

• **MPI_Datatype**
  • Creates a vector MPI_Datatype for communication
Communication with Pack and Array Assignment

Pack communication

MPI_Irecv(&xbuf_rght_recv, bufsize, MPI_PACKED, nrght, 1001, MPI_COMM_WORLD, &request[0]);
MPI_Isend(&xbuf_left_send, bufsize, MPI_PACKED, nleft, 1001, MPI_COMM_WORLD, &request[1]);

MPI_Irecv(&xbuf_left_recv, bufsize, MPI_PACKED, nleft, 1002, MPI_COMM_WORLD, &request[2]);
MPI_Isend(&xbuf_rght_send, bufsize, MPI_PACKED, nrght, 1002, MPI_COMM_WORLD, &request[3]);
MPI_Waitall(4, request, status);

Array Assign

Similar communication but with bufcound and MPI_DOUBLE
Vector MPI_Datatype

Create Datatype once at start

```c
MPI_Datatype horiz_type;
MPI_Type_vector(jnum, nhalo, isize+2*nhalo, MPI_DOUBLE, &horiz_type);
MPI_Type_commit(&horiz_type);
```

Communicate with

```c
MPI_Irecv(&x[jlow][isize], 1, horiz_type, nrght, 1001, MPI_COMM_WORLD, &request[0]);
MPI_Isend(&x[jlow][0], 1, horiz_type, nleft, 1001, MPI_COMM_WORLD, &request[1]);
MPI_Irecv(&x[jlow][-nhalo], 1, horiz_type, nleft, 1002, MPI_COMM_WORLD, &request[2]);
MPI_Isend(&x[jlow][isize-nhalo], 1, horiz_type, nrght, 1002, MPI_COMM_WORLD, &request[3]);
```
MPI Topology Support
MPI Topology

- **Cartesian**
  - `MPI_Dims_create` – splits grid into dims
  - `MPI_Cart_create` – creates a new communicator
  - `MPI_Cart_coords` – maps from rank to coordinates
  - `MPI_Cart_rank` – gets rank for coordinates
  - `MPI_Cart_shift` – shifts by constant to get neighbors

- **Graph – use newer routines**
  - `MPI_Dist_graph_create`
  - `MPI_Dist_graph_neighbors`
  - `MPI_Dist_graph_neighbors_count`
  - `MPI_Dist_graph_create_adjacent`
Sample Cartesian Topology coding

```c
int dims[2] = {nprocy, nprocx}; // needs to be initialized
int periods[2]={0,0};
int coords[2];
MPI_Dims_create(nprocs, 2, dims);
MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &cart_comm);
MPI_Cart_coords(cart_comm, rank, 2, coords);

int nleft, nrght, nbot, ntop;
MPI_Cart_shift(cart_comm, 1, 1, &nleft, &nrght);
MPI_Cart_shift(cart_comm, 0, 1, &nbot, &ntop);
```
Cartesian neighbor communication

Can reduce communication to a single line

    int counts[4] = {1, 1, 1, 1};
    MPI_Neighbor_alltoallw (&x[-nhalo][-nhalo], counts, sdispls, sendtypes,
                        &x[-nhalo][-nhalo], counts, rdispls, recvtypes, cart_comm);

The arguments are defined once at start

    // displacements are from bottom left corner of memory block in bytes
    MPI_Aint sdispls[4] = {
        nhalo *(isize+2*nhalo)*8, // bottom row is nhalo above start
        jsize *(isize+2*nhalo)*8, // top row is jsize above start
        nhalo *8, // left column is nhalo right of start
        isize *8}; // right column is isize right of start

    MPI_Aint rdispls[4] = {
        0, // bottom ghost row is 0 above start
        (jsize+nhalo)*(isize+2*nhalo)*8, // top ghost row is jsize+nhalo above start
        0, // left ghost column is 0 right of start
        (isize+nhalo)*8}; // right ghost row is jsize+nhalo right of start

    MPI_Datatype sendtypes[4] = {vert_type, vert_type, horiz_type, horiz_type};
    MPI_Datatype recvtypes[4] = {vert_type, vert_type, horiz_type, horiz_type};
Advanced MPI: One-Sided Communication
One-Sided Communication

• This material is based on the Advanced MPI Programming tutorial at SC16, presented by:
  • Pavan Balaji
  • Torsten Hoefler
  • William Gropp
  • Rajeev Thakur
One-Sided Communication

• Basic idea is to decouple data movement with process synchronization.
  • Each process exposes part of its memory to other processes
  • The other processes can directly read from and write to the exposed memory
  • Data can be moved without requiring the remote process to synchronize

![Diagram of one-sided communication]

- Process 0
- Process 1
- Process 2
- Process 3

Global Address Space

Private Memory

Private Memory

Private Memory

Private Memory
Comparing One- and Two-Sided Communication

Two-Sided Communication

- **Process 0**
  - SEND(data)

- **Process 1**
  - DELAY
  - RECV(data)

Sending process is delayed by receiving process

One-Sided Communication

- **Process 0**
  - PUT(data)
  - GET(data)

- **Process 1**
  - DELAY

Delay in receiving process does not affect sending process
Advantages of One-Sided Communication

- Multiple data transfers can occur with a single synchronization operation.
- No need for tag matching.
- Can provide better performance / scalability in some cases.
  - Some systems provide hardware support for remote memory access.
  - Some communication patterns can be more easily represented.
    - E.g., if the communication pattern is not known a-priori

Weak scaling of MONC weather model [1]

One-Sided Communication General Process

- Create remote accessible memory, called a “window”
  - Window initialization is done collectively
- Start an RMA epoch (synchronization)
- Read, write, and update remote memory
  - MPI_put – Put data into remote window
  - MPI_get – Get data from remote window
- Stop an RMA epoch (synchronization)
- Collectively free the window.

- **Origin**: the process initiating the request (performs the call)
- **Target**: the process whose memory is accessed

**Note**: All access calls are non-blocking
Window Creation

- There are four models of window creation:
  - MPI_WIN_ALLOCATE
    - Create a buffer and make it remotely accessible.
  - MPI_WIN_CREATE
    - Make an already allocated buffer (i.e., local data) remotely accessible.
  - MPI_WIN_CREATE_DYNAMIC
    - You do not have a buffer yet, but will have one in the future.
    - Can dynamically add / remove buffers to / from the window.
      - MPI_WIN_ATTACH
      - MPI_WIN_DETACH
  - MPI_WIN_ALLOCATE_SHARED
    - Multiple processes on the same node share the same buffer.
Window Creation

- `int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)`

- `int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)`

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

- `int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)`
int main(int argc, char *argv[]) {
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);

    // Collectively create remote accessible memory
    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 Example

```c
int main(int argc, char *argv[])
{
    int *a;   MPI_Win win;

    MPI_Init(&argc, &argv);

    // Create private memory
    MPI_Alloc_mem(1000*sizeof(int), MPI_INFO_NULL, &a);
    a[0] = 1;   a[1] = 2;

    // Collectively declare memory as remotely accessible
    MPI_Win_create(a, 1000*sizeof(int), sizeof(int), MPI_INFO_NULL,
                    MPI_COMM_WORLD, &win);

    // Array ‘a’ is now accessible from all processes in MPI_COMM_WORLD

    MPI_Win_free(&win);
    MPI_Free_mem(a);
    MPI_Finalize();
    return 0;
}
```
int 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));
    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 from all processes in MPI_COMM_WORLD

    MPI_Win_detach(win, a);    free(a);
    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
Transferring Data

- MPI provides ability to read, write, and atomically modify data in remotely accessible memory
  - MPI_PUT – Put data in remote memory window
  - MPI_GET – Get data from remote memory window
  - MPI_ACCUMULATE
  - MPI_GET_ACCUMULATE
  - MPICOMPARE_AND_SWAP
  - MPI_FETCH_AND_OP
MPI_Get

• int 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 the data from target to origin.
**MPI_Put**

- int MPI_Put(const 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 the data **from** origin **to** target.
Synchronization Modes

- **RMA synchronization defines:**
  - When a process is allowed to read/write remotely accessible memory.
  - When transferred data is available for use.
- **3 types of synchronization modes provided by MPI:**
  - Fence (active target)
  - Post-start-complete-wait (generalized active target)
  - 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
**MPI_WIN_FENCE**

- Collective synchronization
- Starts *and* ends access and exposure epochs on all processes in the window.
- All processes do MPI_WIN_FENCE to open epoch.
- All processes can issue data transfer operations.
- All processes do MPI_WIN_FENCE to close epoch.
- All operations complete at second fence synchronization

```c
int MPI_Win_fence(int assert, MPI_Win win)
```
PSCW: Generalized Active Target Synchronization

Like MPI_WIN_FENCE, but origin and target specify who they communicate with.

- **Target: Exposure epoch**
  - Opened with MPI_Win_post
  - Closed with MPI_Win_wait

- **Origin: Access Epoch**
  - Opened with MPI_Win_start
  - Closed with MPI_Win_complete

Processes can be both origins and targets.

---

**MPI_Win_post/start**

```
MPI_Win_post/start(MPI_Group grp, int assert, MPI_Win win)
```

**MPI_Win_complete/wait**

```
MPI_Win_complete/wait(MPI_Win win)
```
Lock/Unlock: Passive Target Synchronization

- Passive mode: One-sided, asynchronous communication
  - Target does **not** participate in communication operation
Which synchronization mode should I use?

• RMA communication has lower overhead than send/recv
  • Two-sided: Matching, queueing, buffering, etc.
  • One-sided: No matching, no buffering, always ready to receive

• Active mode: bulk synchronization
  • E.g., ghost cell exchange

• Passive mode: asynchronous data movement
  • Useful when dataset is large, requiring memory of multiple nodes
  • Also, when data access and synchronization pattern is dynamic
  • Common use case: distributed, shared arrays
Advanced MPI: I/O
Non-Parallel I/O
Independent Parallel I/O

- Each process writes to a separate file
- Pro: Parallelism
- Con: lots of small files
Cooperative Parallel I/O

- All processes write to a single file
Independent I/O: The Basics

• Similar process to Posix I/O:
  • Open the file
  • Read / write data
  • Close the file

• In MPI:
  • Open the file: MPI_File_open
  • Write data to file: MPI_File_write
  • Close the file: MPI_File_close

• Independent I/O can perform better than collective I/O in some cases:
  • Synchronization of collective calls is not natural
  • Overhead of collective calls outweighs their benefit
  • For example: very small I/O during header reads
**MPI_File_open**

```c
int MPI_File_open(MPI_Comm comm, const char *filename,
                  int amode, MPI_Info info, MPI_File *fh)
```

- **Collectively opens a file.**
- `amode` = file access mode:
  - MPI_MODE_APPEND
  - MPI_MODE_CREATE -- Create the file if it does not exist.
  - MPI_MODE_DELETE_ON_CLOSE
  - MPI_MODE_EXCL -- Error creating a file that already exists.
  - MPI_MODE_RDONLY -- Read only.
  - MPI_MODE_RDWR -- Reading and writing.
  - MPI_MODE_SEQUENTIAL
  - MPI_MODE_WRONLY -- Write only.
**MPI_File_write**

\[
\text{int MPI\_File\_write(MPI\_File fh, ROMIO\_CONST void *buf, int count, MPI\_Datatype datatype, MPI\_Status *status)}
\]

- *Independently* write data to a file
- Can use custom MPI_Datatypes

**MPI_File_close**

\[
\text{int MPI\_File\_close(MPI\_File * fh)}
\]

- *Collectively* closes a file.
Independent I/O: Example

```c
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    MPI_File f;  // MPI File handle
    int data[1000]; rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_File_open(MPI_COMM_WORLD, "file.out",
                  MPI_MODE_CREATE|MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &f);
    if ( rank == 0 )
        MPI_File_write(f, data, 1000, MPI_INT, MPI_STATUS_IGNORE);
    MPI_File_close(&f);
    MPI_Finalize;
    return 0;
}
```
Ways to Access a Shared File

- MPI_File_seek
- MPI_File_read
- MPI_File_write
- MPI_File_read_at
- MPI_File_write_at
- MPI_File_read_shared
- MPI_File_write_shared

Similar to POSIX IO

Combines seek and IO

Uses shared file pointer
Collective I/O: The Basics

• All processes in the communicator call the collective I/O function
• The basic idea is to build large blocks of data, so that reads and writes will be large.
  • This is handled at the MPI-IO layer
  • Requests from different processes may be merged together
  • Especially effective when accesses of different processes are non-contiguous and interleaved.
Collective I/O Functions

<table>
<thead>
<tr>
<th>Independent I/O</th>
<th>Collective I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_File_write_at</td>
<td>MPI_File_write_at_all</td>
</tr>
<tr>
<td>MPI_File_read_at</td>
<td>MPI_File_read_at_all</td>
</tr>
</tbody>
</table>

- `_all` indicates that all processes in communicator passed to `MPI_File_open` will call this function.
- Each process specifies only its own access information.

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset,
                      ROMIO_CONST void *buf, int count,
                      MPI_Datatype datatype, MPI_Status *status)

int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset,
                          ROMIO_CONST void *buf, int count,
                          MPI_Datatype datatype, MPI_Status *status)
```
Collective I/O Functions

- MPI_File_seek
- MPI_File_read_all
- MPI_File_write_all
- MPI_File_read_at_all
- MPI_File_write_at_all
- MPI_File_read_ordered
- MPI_File_write_ordered

Similar to POSIX IO

Combines seek and IO

Uses shared file pointer
**Independent vs Collective I/O: Performance**

- Collective I/O can provide better performance
- Improvement is more significant at higher core counts.

---

Delivering science and technology to protect our nation and promote world stability

Los Alamos National Laboratory
EST. 1943