LA-UR-17-24553
Approved for public release; distribution is unlimited.

Title: An MPI Tutorial: Collectives and Point-to-Point Communication

Author(s): Garrett, Charles Kristopher

Intended for: Lecture for Parallel Computing Summer Research Internship

Issued: 2017-06-06

An MPI Tutorial

Collectives and Point-to-Point Communication

Kris Garrett

June 2017

Operated by Los Alamos National Security, LLC for the U.S. Department of Energy's NNSA

What is MPI?

- **MPI = Message Passing Interface**

- **Executable is run in multiple processes**
  - Processes may be on multiple nodes of a cluster
  - Multiple processes may be placed on a node to utilize multi-core processors
  - All processes can communicate with each other

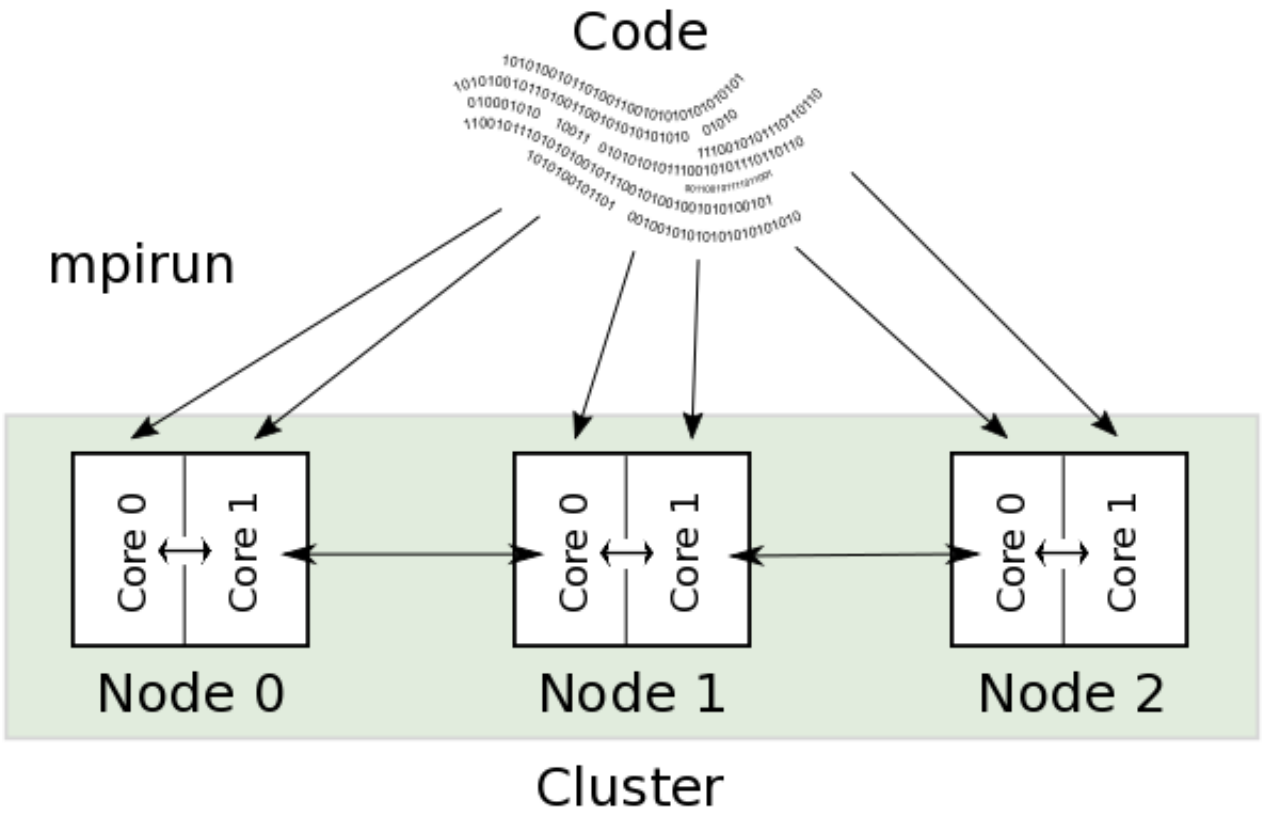
- **C and Fortran library APIs given by the standard**
  - 3rd party bindings exist (Python, C++, etc)
  - Will concentrate on C library bindings here

Process to Build MPI Programs

- **Compile code with an MPI wrapper**
  - `mpicc, mpicxx, mpifort, ...`

- **Run code with an MPI wrapper**
  - `mpirun, mpiexec, aprun, srun`
  - Example: `mpirun –n 4 ./MyProg`
    - This will run the MyProg executable in 4 processes
    - Each process can query:
      - how many total processes are being run
      - a unique index for itself
    - Each process can send/receive data to/from any other process

How MPI Works



First MPI Program

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

int main(int argc, char **argv)
{
    int rank, numRanks;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

    printf("Rank %d of %d\n", rank, numRanks);

    MPI_Finalize();
    return 0;
}
```

```
  --- Output ---
$ mpirun -n 4 ex1.x
Rank 0 of 4
Rank 2 of 4
Rank 3 of 4
Rank 1 of 4
```

First MPI Program

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

int main(int argc, char **argv)
{
    int rank, numRanks;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

    printf("Rank %d of %d\n", rank, numRanks);

    MPI_Finalize();
    return 0;
}
```

Include mpi.h

First MPI Program

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

int main(int argc, char **argv)
{
    int rank, numRanks;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

    printf("Rank %d of %d\n", rank, numRanks);

    MPI_Finalize();
    return 0;
}
```

Init before everything

Finalize after everything

First MPI Program

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

int main(int argc, char **argv)
{
    int rank, numRanks;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

    printf("Rank %d of %d\n", rank, numRanks);

    MPI_Finalize();
    return 0;
}
```

- **Rank** is a unique integer for each process
- Size is the number of processes

First MPI Program

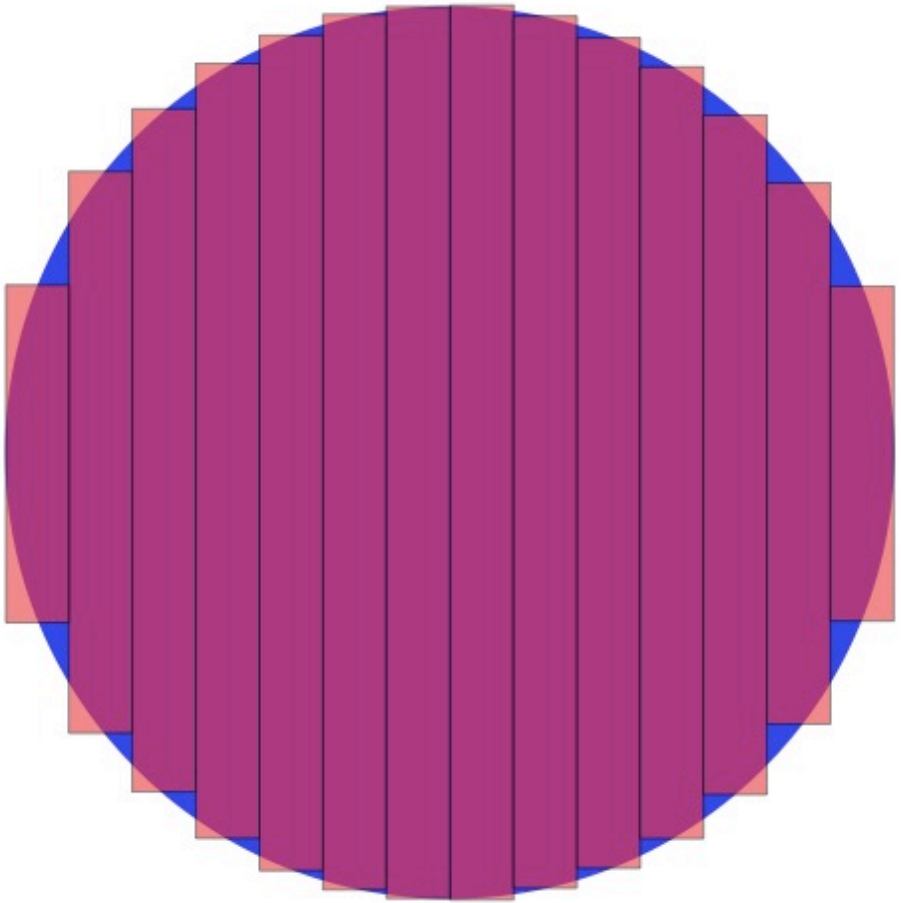
```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    int rank, numRanks;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

    printf("Rank %d of %d\n", rank, numRanks);

    MPI_Finalize();
    return 0;
}
```

- **MPI_COMM_WORLD** is a communicator
  - Represents all processes
- Can create subcommunicators
  - Useful for libraries

MPI Collectives

Let's Calculate Pi

- **Split unit circle into n rectangles**
- **Pi is approximately the sum of rectangles**
- **Each rectangle has**
  - Width: dx = 2 / n
  - Midpoint: x_i = -1 + dx / 2 + dx * i
  - Height: h = 2 * sqrt(1 - x_i^2)
  - Area: area = dx * h



Let's Calculate Pi

```
#include <stdio.h>
#include <math.h>

int main(int argc, char **argv)
{
    const int N = 1000;
    double pi = calcPi(N);
    double error = fabs(pi - M_PI);

    printf("Approximation of pi: %f   Error: %e\n", pi, error);

    return 0;
}
```

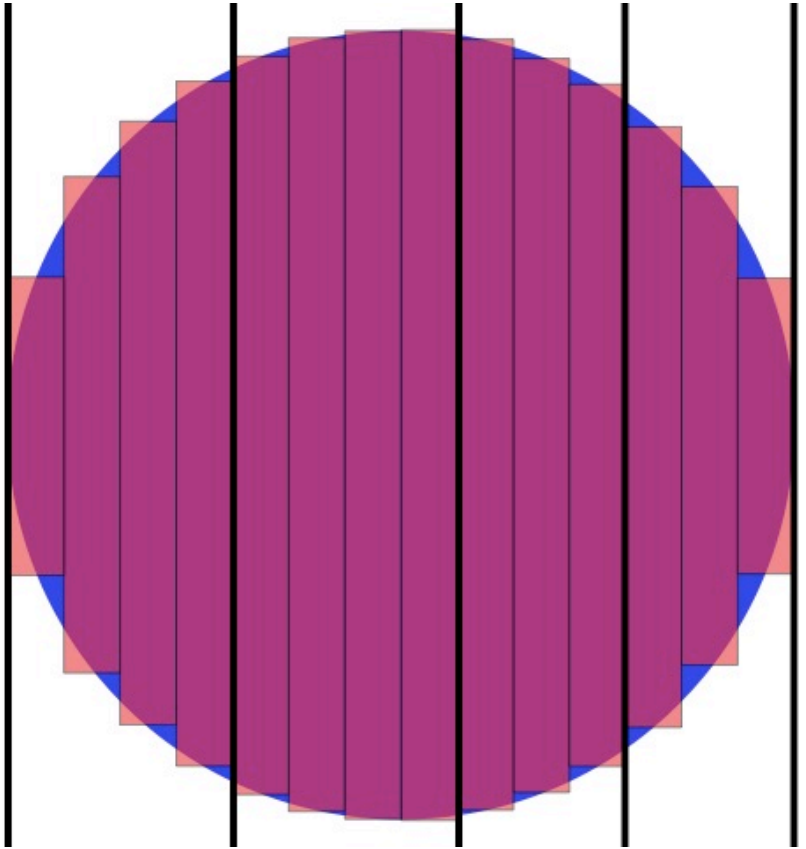
```
--- Output ---
Approximation of pi: 3.141623   Error: 3.080323e-05
```

Let's Calculate Pi

```
double calcPi(int n)
{
    double a = -1.0;    // Left endpoint
    double b = 1.0;     // Right endpoint
    double dx = (b - a) / n;
    double area = 0.0;

    for (int i = 0; i < n; i++) {
        double x = a + dx / 2.0 + i * dx;
        double h = 2.0 * sqrt(1.0 - x * x);
        area = area + dx * h;
    }

    return area;
}
```

Let's Calculate Pi with MPI

- **Sets of rectangles owned by a rank**
  - Each rank calculates a 'slice' of pi
  - Add all the slices in the end
- **Different number of rectangles for each rank is possible if n is not divisible by the number of ranks**
  - Example:
    - 14 total rectangles into 4 ranks
    - Ranks 0 and 1 have 4 rectangles
    - Rank 2 and 3 have 3 rectangles

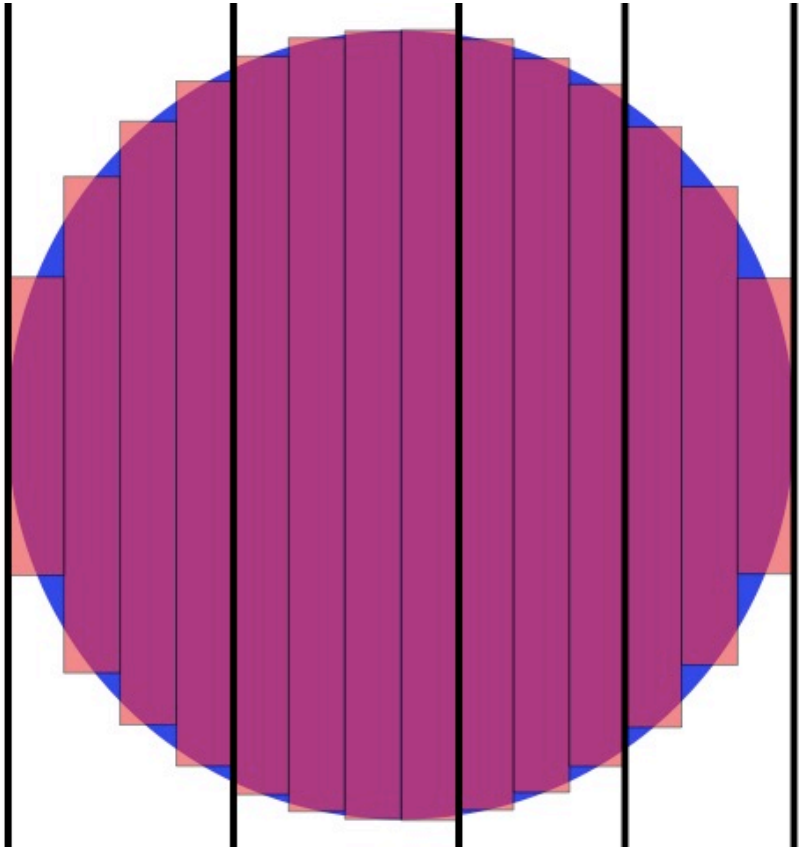


Let's Calculate Pi with MPI

- **Create function to get local problem information**

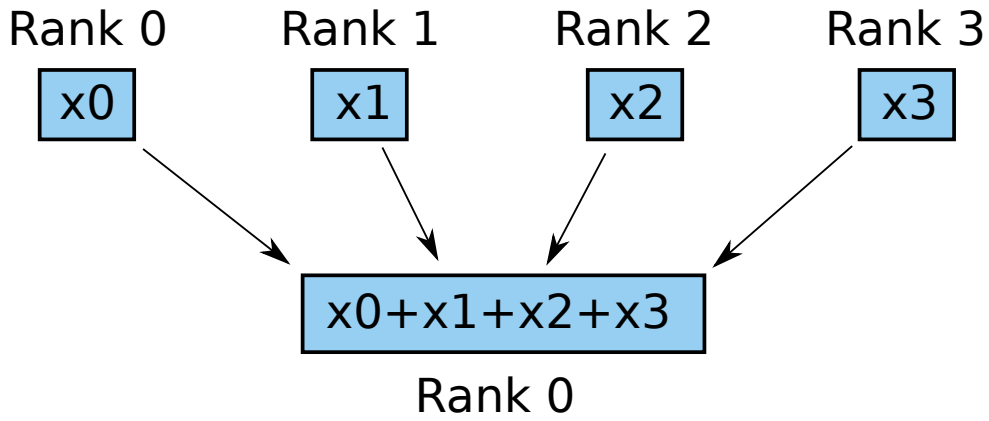
```
void getMyBounds(int globalN,
                 int *begin,
                 int *end,
                 int *localN)
```

Rank	globalN	begin	end	localN
0	14	0	3	4
1	14	4	7	4
2	14	8	10	3
3	14	11	13	3



Let's Calculate Pi with MPI: Reduce

- **To sum across ranks, use MPI_Reduce**
  - Performs a binary operation across all ranks in a communicator
- **Common built-in functions for reduce include**
  - MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD
- **Can create your own binary operation**
- **MPI_Allreduce**
  - All ranks get sum



Anatomy of MPI_Reduce

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

sendbuf	Address of send buffer
recvbuf	Address of receive buffer (significant only at root)
count	Number of elements in send buffer
datatype	Data type of elements of send buffer
op	Reduce operation
root	Rank of root process
comm	Communicator

Returns error.
Should always check for return of MPI_SUCCESS.
Won't do that here for clarity of code and space.

MPI Data Types

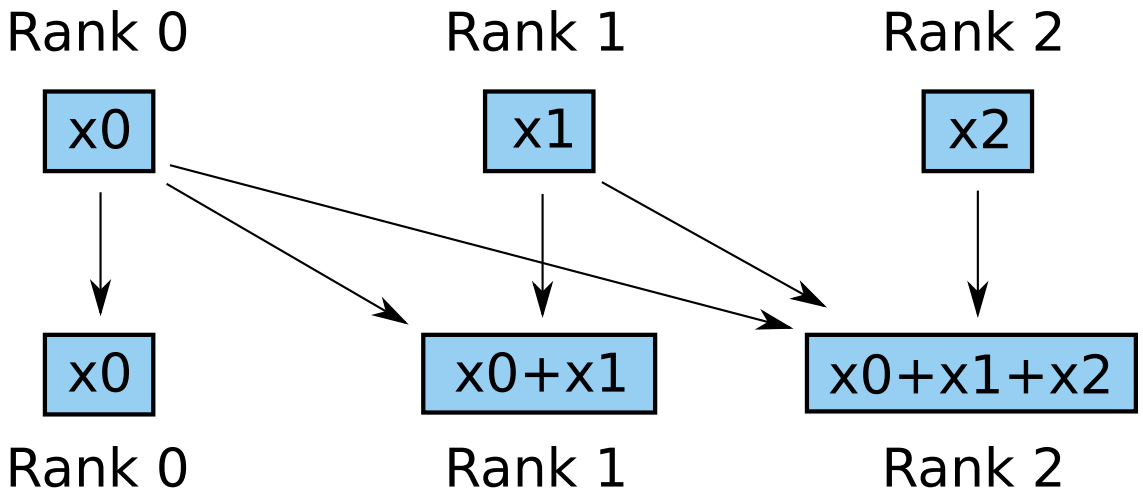
```
--- List of Common C Data Types ---

MPI_CHAR              MPI_SHORT
MPI_INT               MPI_LONG
MPI_LONG_LONG         MPI_FLOAT
MPI_DOUBLE            MPI_LONG_DOUBLE
MPI_INT8_T            MPI_INT16_T
MPI_INT32_T           MPI_INT64_T
MPI_BYTE
```

There are also MPI data types corresponding to Fortran data types

Let's Calculate Pi with MPI: Scan

- **To calculate offsets for ownership of rectangles, use MPI_Scan**
  - Performs an inclusive scan operation across all ranks in a communicator

- **MPI_Exscan**
  - Exclusive scan
  - Rank 0: 0
  - Rank 1: x0
  - Rank 2: x0+x1



Anatomy of MPI_Scan

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

sendbuf      Address of send buffer
recvbuf      Address of receive buffer
count        Number of elements in send buffer
datatype     Data type of elements of send buffer
op           Reduce operation
comm         Communicator
```

Let's Calculate Pi with MPI: main

```
int main(int argc, char **argv)
{
    const int globalN = 1000;
    int localN, begin, end;
    double dx, a, b, piSlice, pi;

    // Init MPI
    MPI_Init(&argc, &argv);

    // Get local problem extents
    getMyBounds(globalN, &begin, &end, &localN);
    dx = 2.0 / globalN;
    a = -1.0 + begin * dx;
    b = a + localN * dx;
```

Let's Calculate Pi with MPI: main

```
    // Calculate local circle area and sum values together
    piSlice = calcPiSlice(a, b, localN);
    MPI_Reduce(&piSlice, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print value of pi and error
    if (begin == 0) {
        double error = fabs(pi - M_PI);
        printf("Approximation of pi: %f   Error: %e\n", pi, error);
    }

    // End
    MPI_Finalize();
    return 0;
}
```

Let's Calculate Pi with MPI: calcPiSlice

```
double calcPiSlice(double a, double b, int n)
{
    double dx = (b - a) / n;
    double area = 0.0;

    for (int i = 0; i < n; i++) {
        double x = a + dx / 2.0 + i * dx;
        double h = 2.0 * sqrt(1.0 - x * x);
        area = area + dx * h;
    }

    return area;
}
```

Let's Calculate Pi with MPI: getMyBounds

```
void getMyBounds(int globalN, int *begin, int *end, int *localN)
{
    int rank, numRanks;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

    *localN = globalN / numRanks;
    if (rank < globalN % numRanks)
        *localN = *localN + 1;

    MPI_Scan(localN, end, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    *begin = *end - *localN;
    *end = *end - 1;
}
```

Let's Calculate Pi with MPI: getMyBounds

```
void getMyBounds(int globalN, int *begin, int *end, int *localN)
{
    int rank, numRanks;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

    *localN = globalN / numRanks;
    if (rank < globalN % numRanks)
        *localN = *localN + 1;

    MPI_Scan(localN, end, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    *begin = *end - *localN;
    *end = *end - 1;
}
```

Trick to get the local problem size when the global problem size is not evenly divisible

Let's Calculate Pi with MPI: getMyBounds

```
void getMyBounds(int globalN, int *begin, int *end, int *localN)
{
    int rank, numRanks;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

    *localN = globalN / numRanks;
    if (rank < globalN % numRanks)
        *localN = *localN + 1;

    MPI_Scan(localN, end, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    *begin = *end - *localN;
    *end = *end - 1;
}
```

Example:

```
globalN = 7   numRanks = 3
    Rank 0: localN = 3
    Rank 1: localN = 2
    Rank 2: localN = 2
```

Let's Calculate Pi with MPI: getMyBounds

```
void getMyBounds(int globalN, int *begin, int *end, int *localN)
{
    int rank, numRanks;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

    *localN = globalN / numRanks;
    if (rank < globalN % numRanks)
        *localN = *localN + 1;

    MPI_Scan(localN, end, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    *begin = *end - *localN;
    *end = *end - 1;
}
```

Example:

```
globalN = 7   numRanks = 3
  Rank 0: end = 3
  Rank 1: end = 5 (3+2)
  Rank 2: end = 7 (3+2+2)
```

Let's Calculate Pi with MPI: getMyBounds

```
void getMyBounds(int globalN, int *begin, int *end, int *localN)
{
    int rank, numRanks;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

    *localN = globalN / numRanks;
    if (rank < globalN % numRanks)
        *localN = *localN + 1;

    MPI_Scan(localN, end, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    *begin = *end - *localN;
    *end = *end - 1;
}
```

Example:

```
globalN = 7   numRanks = 3
Rank 0: begin = 0 end = 2
Rank 1: begin = 3 end = 4
Rank 2: begin = 5 end = 6
```

Common Collectives: MPI_Barrier

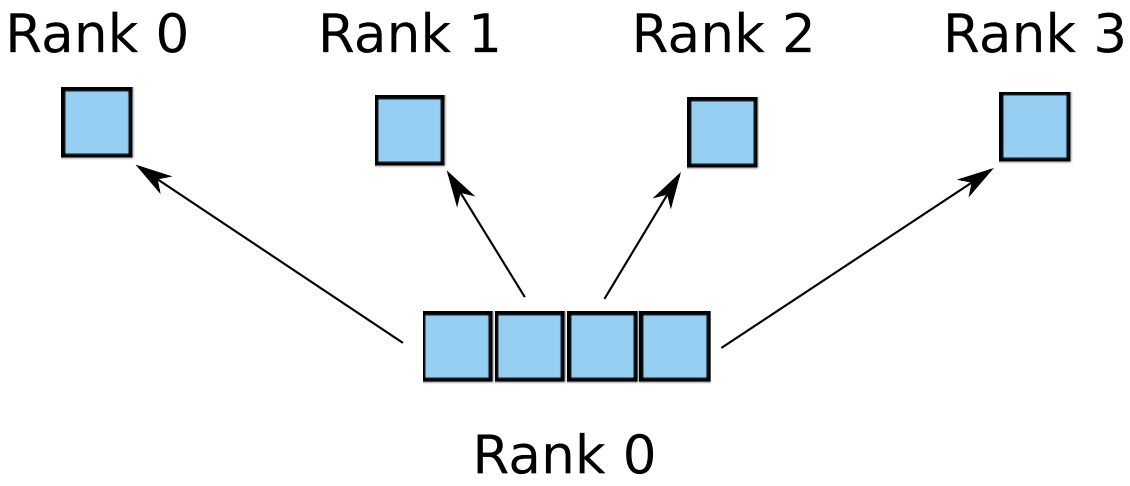
- **Make all ranks wait until they hit the barrier**
  - All ranks in communicator must call this before code moves forward
  - Can be useful for debugging

- **Be careful not to put this in a branching statement**
  - Could cause a process to stall
  - Example: if, for, case, while

```
// Rank 0 will never continue
if (rank == 0)
    MPI_Barrier(MPI_COMM_WORLD);
```

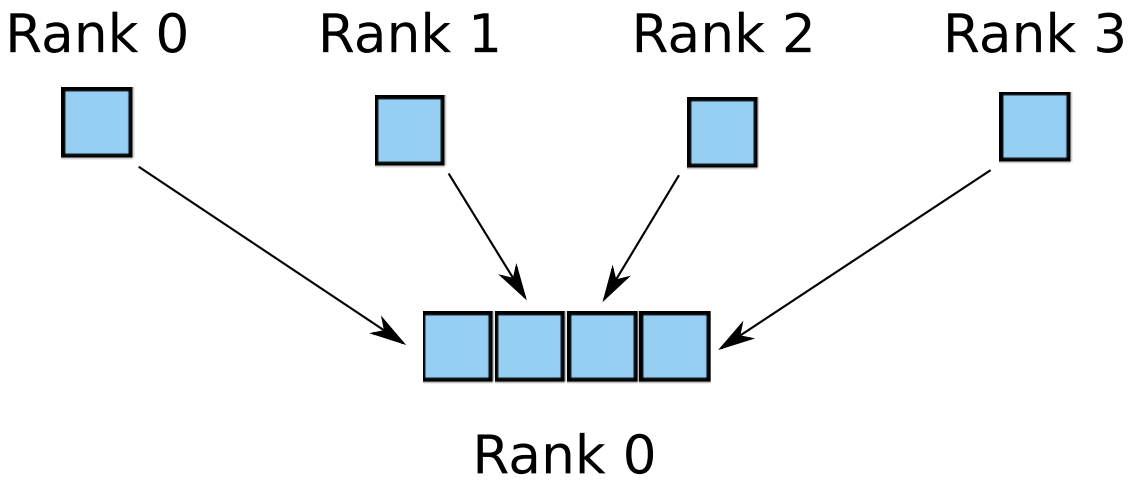
Common Collectives: MPI_Scatter

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



Common Collectives: MPI_Gather

- Elements from each rank goes into one array
- Allgather: every rank gets the whole array



Other Collectives

- **MPI_Alltoall**
  - Every rank sends different data to every other rank
- **MPI_Alltoallw**
  - Most general version of MPI_Alltoall
  - Allows different data sizes
- **Varying Count Versions**
  - MPI_Gatherv, MPI_Scatterv, MPI_Allgatherv, MPI_Alltoallv
  - More general versions than without ending in ‘v’
  - Use if number of elements is different per rank

Other Collectives

- **MPI_Reduce_scatter, MPI_Reduce_scatter_block**
  - Reduce then scatter

- **Immediate Versions**
  - Will discuss immediate MPI calls in next section
  - Add ‘I’ before each name
  - Examples: MPI_Igather, MPI_Iscatter, MPI_Ireduce

MPI Point-to-Point

How to Send Data to 1 Rank

- **4 Types of Sends**
  - `MPI_Send`
    - May buffer message and return quickly (usually for short messages)
    - May wait until message is sent (usually for long messages)
  - `MPI_Bsend (Buffered Send)`
    - Buffers message to return quickly
    - User can add buffer space for this call: `MPI_Buffer_attach`
  - `MPI_Ssend (Synchronous Send)`
    - Will wait until message is sent (no buffering)
  - `MPI_Rsend (Ready Send)`
    - Assumes receive is already posted
    - Reduces handshaking overhead
- **These 4 send types ensure you can use the message buffer after the call is complete**

Anatomy of MPI_Send

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

buf	Initial address of send buffer
count	Number of elements to send
datatype	Datatype of each send buffer element
dest	Rank of destination
tag	Message tag
comm	Communicator

Receiver must match tag and communicator.
Can use `buf` after call has completed.
Can set `dest=MPI_PROC_NULL`.

Anatomy of MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

buf	Initial address of receive buffer
count	Maximum number of elements to receive
datatype	Datatype of each receive buffer entry
source	Rank of source
tag	Message tag
comm	Communicator
status	Status object

Only 1 receive type as compared to the 4 send types.

Can use `buf` after call has completed.

Can receive less than count number of elements.

Can set `tag=MPI_ANY_TAG` and `source=MPI_ANY_SOURCE` or `MPI_PROC_NULL`.

MPI_Status

- **In C, `status` is a structure containing**
  - `MPI_SOURCE`
  - `MPI_TAG`
  - `MPI_ERROR`

- **Call `MPI_Get_count` on `status` to get number of elements received**

- **Can use `MPI_STATUS_IGNORE` for status parameter**
  - For array of statuses: `MPI_STATUSES_IGNORE`

Immediate Versions of Send/Recv

- **There is an immediate (non-blocking) version of each send and receive call**
  - `MPI_Isend, MPI_Ibsend, MPI_Issend, MPI_Irsend, MPI_Irecv`
  - Calls return immediately
  - Returns `MPI_Request`

- **Best Practice: call `MPI_Irecv` before `MPI_Isend` to reduce amount of handshaking**

Immediate Versions of Send/Recv

- **Wait on or test MPI_Request to see if call is done**
  - MPI_Wait: Wait until request is done
  - MPI_Test: Check if request is done
  - Message buffer cannot be used until MPI_Wait returns or MPI_Test returns true
  - MPI_Wait or MPI_Test is required to deallocate memory used by an immediate call

- **Variants of Wait and Test for multiple requests**
  - MPI_Waitany, MPI_Waitall, MPI_Waitsome
  - MPI_Testany, MPI_Testall, MPI_Testsome

A Program That Fails

```
/////  2 MPI rank example that will never finish  /////
If (rank == 0) {
    MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, MPI_STATUS_IGNORE);
    MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
}
If (rank == 1) {
    MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, MPI_STATUS_IGNORE);
    MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
}
```

A Program That May Fail

```
/////  2 MPI rank example that may never finish  /////
// Relies on send buffering the message
If (rank == 0) {
    MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
    MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, MPI_STATUS_IGNORE);
}
If (rank == 1) {
    MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
    MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, MPI_STATUS_IGNORE);
}
```

A Program That Succeeds

```
/////  2 MPI Rank example that will finish  /////
If (rank == 0) {
    MPI_Send(sendbuf, count, MPI_DOUBLE, 1, tag, comm);
    MPI_Recv(recvbuf, count, MPI_DOUBLE, 1, tag, comm, MPI_STATUS_IGNORE);
}
If (rank == 1) {
    MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag, comm, MPI_STATUS_IGNORE);
    MPI_Send(sendbuf, count, MPI_DOUBLE, 0, tag, comm);
}
```

A Program That Succeeds

```
/////  2 MPI Rank example that will finish  /////
If (rank == 0) {
    MPI_Sendrecv(sendbuf, count, MPI_DOUBLE, 1, tag,
                 recvbuf, count, MPI_DOUBLE, 1, tag,
                 comm, MPI_STATUS_IGNORE);
}
If (rank == 1) {
    MPI_Sendrecv(sendbuf, count, MPI_DOUBLE, 0, tag,
                 recvbuf, count, MPI_DOUBLE, 0, tag,
                 comm, MPI_STATUS_IGNORE);
}
```

Isend/Irecv to Eliminate Blocking

```
If (rank == 0) {
    MPI_Irecv(recvbuf, count, MPI_DOUBLE, 1, tag, comm,
              MPI_STATUS_IGNORE, &request[0]);
    MPI_Isend(sendbuf, count, MPI_DOUBLE, 1, tag, comm, &request[1]);
}
If (rank == 1) {
    MPI_Irecv(recvbuf, count, MPI_DOUBLE, 0, tag, comm,
             MPI_STATUS_IGNORE, &request[0]);
    MPI_Isend(sendbuf, count, MPI_DOUBLE, 0, tag, comm, &request[1]);
}
MPI_Waitall(2, request, MPI_STATUSES_IGNORE);
```

MPI_Probe

- **MPI_Probe, MPI_Iprobe**
  - Probe to see if a message exists to receive
  - Typical usage: probe then receive if message exists
  - Useful if you don't know the size of the message
  - Useful if you don't know when you will get messages

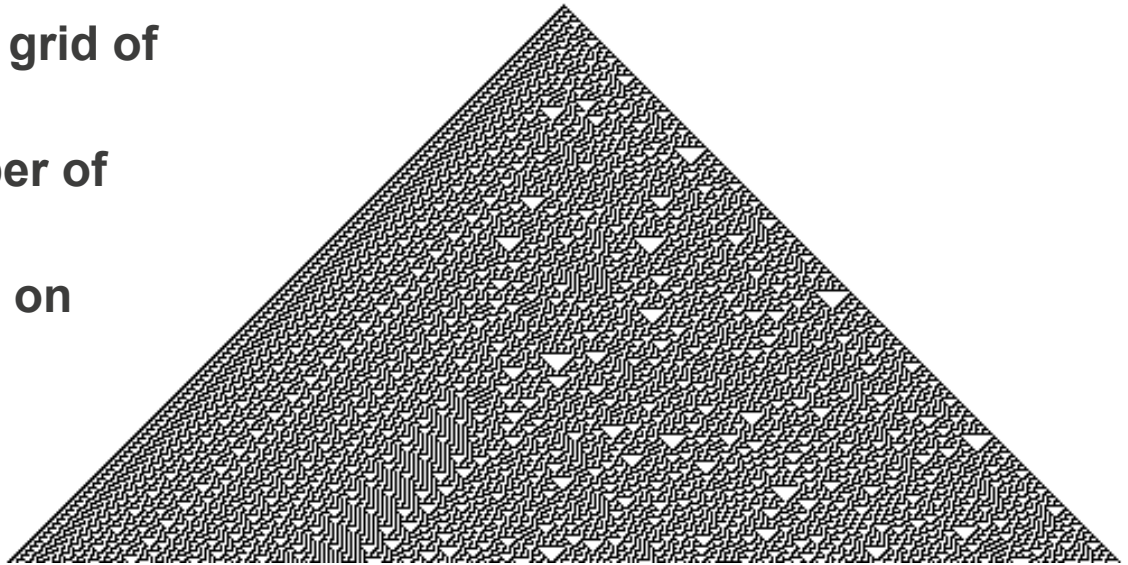
- **MPI_Mprobe, MPI_Improbe, MPI_Mrecv, MPI_Imrecv**
  - "Matching" probe and recv
  - Makes sure message from probe and message received are the same
  - Useful for threaded programs

Persistent Communication Request

- **Setup communication**
  - `MPI_Send_init, MPI_Bsend_init, MPI_Rsend_init, MPI_Ssend_init, MPI_Recv_init`
- **Start the communication**
  - `MPI_Start, MPI_Startall`
  - Returns an `MPI_Request`
- **Wait or Test for communication to complete**

Cellular Automata Example

- Cellular automata live on a grid of cells
- Each cell has a finite number of states
- Each cell is updated based on values of neighbors



Pattern	111	110	101	100	011	010	001	000
Rule	0	0	0	1	1	1	1	0

Cellular Automata: main

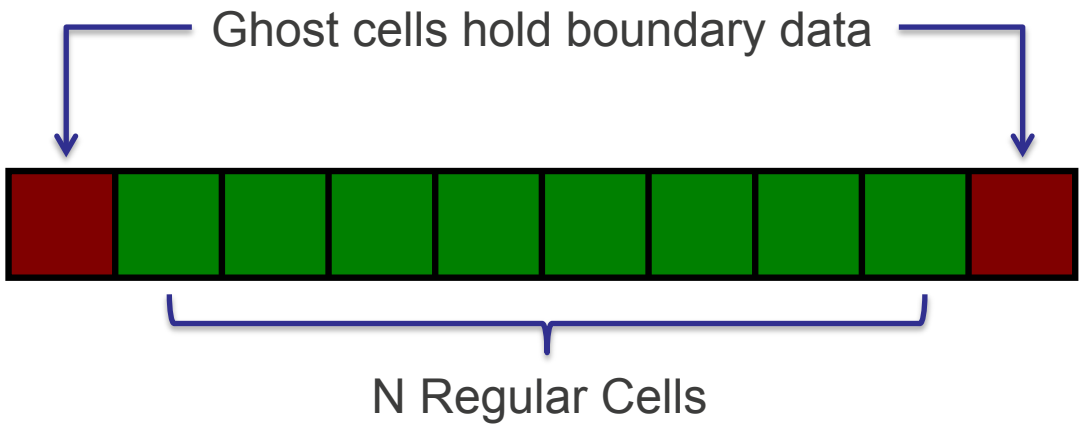
```
int main(int argc, char **argv) {
    int numIters = 200;
    int N = 2 * numIters - 1;
    char *data;

    data = malloc(numIters * (N+2) * sizeof(char));
    memset(data, 0, numIters * (N+2) * sizeof(char));
    data[I2D(numIters, 0)] = 1;

    for (int iter = 1; iter < numIters; iter++)
        applyRule(N, &data[I2D(0, iter-1)], &data[I2D(0, iter)]);

    writeData(N, numIters, data, "ex4.txt");
    return 0;
}
```

+2 for boundary values

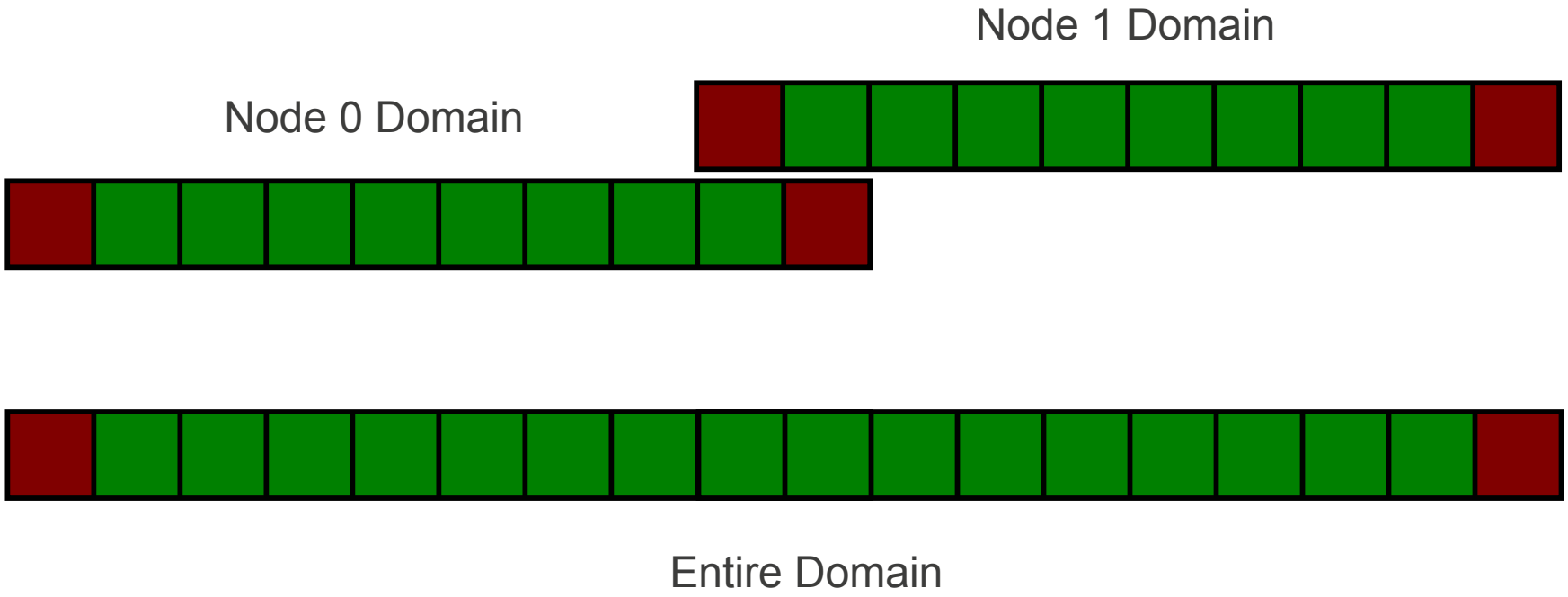
Cellular Automata: I2D Macro

```
// This is a common macro in C for 2D arrays
#define I2D(i,j) ((i) + (j) * (N+2))
```

Cellular Automata: Domain



Cellular Automata (Parallel): Domain



Cellular Automata (Parallel): main

```
int main(int argc, char **argv) {
    int numIters = 200;
    int totalN = 2 * numIters - 1;
    char *data;
    int begin, end, localN, rank;
    char filename[20];

    // Initialize MPI
    MPI_Init(&argc, &argv);

    // Get bounds of local problem
    getMyBounds(totalN, &begin, &end, &localN);
```

Cellular Automata (Parallel): main

```
// Allocate data and set initial condition
data = malloc(numIters * (localN+2) * sizeof(char));
memset(data, 0, numIters * (localN+2) * sizeof(char));
if (numIters >= begin && numIters <= end) {
    data[I2D(numIters - begin + 1, 0)] = 1;
}

// Apply rule
for (int iter = 1; iter < numIters; iter++) {
    commNeighbors(localN, &data[I2D(0, iter-1)]);
    applyRule(localN, &data[I2D(0, iter-1)], &data[I2D(0, iter)]);
}
```

Cellular Automata (Parallel): main

```
// Allocate data and set initial condition
data = malloc(numIters * (localN+2) * sizeof(char));
memset(data, 0, numIters * (localN+2) * sizeof(char));
if (numIters >= begin && numIters <= end) {
    data[I2D(numIters - begin + 1, 0)] = 1;
}

// Apply rule
for (int iter = 1; iter < numIters; iter++) {
    commNeighbors(localN, &data[I2D(0, iter-1)]);
    applyRule(localN, &data[I2D(0, iter-1)], &data[I2D(0, iter)]);
}
```

Initial Condition

Communicate boundary before applying rule at each iteration

Cellular Automata (Parallel): main

```
    // Write out data for each rank
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    sprintf(filename, "ex5_%d.txt", rank);
    writeData(localN, numIters, data, filename);

    // End
    MPI_Finalize();
    return 0;
}
```

Cellular Automata (Parallel): main

```
    // Write out data for each rank
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    sprintf(filename, "ex5_%d.txt", rank);
    writeData(localN, numIters, data, filename);

    // End
    MPI_Finalize();
    return 0;
}
```

Write data for each rank

Might want rank 0 to create an additional file containing the name of all the data files

Cellular Automata: commNeighbors

```
void commNeighbors(int localN, char *data) {
    const int RECV_LEFT_TAG  = 1;
    const int SEND_RIGHT_TAG = 1;
    const int RECV_RIGHT_TAG = 2;
    const int SEND_LEFT_TAG  = 2;
    int rank, numRanks;
    int leftRank, rightRank;
    MPI_Request mpiRequest[4];

    // Get rank info
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);
    leftRank = (rank == 0) ? MPI_PROC_NULL : rank - 1;
    rightRank = (rank == numRanks-1) ? MPI_PROC_NULL : rank + 1;
```

Cellular Automata: commNeighbors

```
void commNeighbors(int localN, char *data) {
    const int RECV_LEFT_TAG  = 1;
    const int SEND_RIGHT_TAG = 1;
    const int RECV_RIGHT_TAG = 2;
    const int SEND_LEFT_TAG  = 2;
    int rank, numRanks;
    int leftRank, rightRank;
    MPI_Request mpiRequest[4];

    // Get rank info
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);
    leftRank = (rank == 0) ? MPI_PROC_NULL : rank - 1;
    rightRank = (rank == numRanks-1) ? MPI_PROC_NULL : rank + 1;
```

Call to send/recv with MPI_PROC_NULL does nothing

Cellular Automata: commNeighbors

```
    // recv and send
    MPI_Irecv(&data[0],        1, MPI_CHAR, leftRank,  RECV_LEFT_TAG,
              MPI_COMM_WORLD, &mpiRequest[0]);
    MPI_Irecv(&data[localN+1], 1, MPI_CHAR, rightRank, RECV_RIGHT_TAG,
              MPI_COMM_WORLD, &mpiRequest[1]);
    MPI_Isend(&data[1],        1, MPI_CHAR, leftRank,  SEND_LEFT_TAG,
              MPI_COMM_WORLD, &mpiRequest[2]);
    MPI_Isend(&data[localN],   1, MPI_CHAR, rightRank, SEND_RIGHT_TAG,
               MPI_COMM_WORLD, &mpiRequest[3]);

    // Wait for communication to complete
    MPI_Waitall(4, mpiRequest, MPI_STATUSES_IGNORE);
}
```

Cellular Automata: commNeighbors

```
    // recv and send
    MPI_Irecv(&data[0],        1, MPI_CHAR, leftRank,  RECV_LEFT_TAG,
              MPI_COMM_WORLD, &mpiRequest[0]);
    MPI_Irecv(&data[localN+1], 1, MPI_CHAR, rightRank, RECV_RIGHT_TAG,
              MPI_COMM_WORLD, &mpiRequest[1]);
    MPI_Isend(&data[1],        1, MPI_CHAR, leftRank,  SEND_LEFT_TAG,
              MPI_COMM_WORLD, &mpiRequest[2]);
    MPI_Isend(&data[localN],   1, MPI_CHAR, rightRank, SEND_RIGHT_TAG,
               MPI_COMM_WORLD, &mpiRequest[3]);

    // Wait for communication to complete
    MPI_Waitall(4, mpiRequest, MPI_STATUSES_IGNORE);
}
```

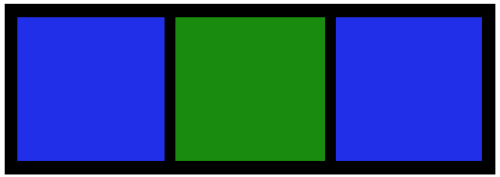
Can ignore status

2D Cellular Automata

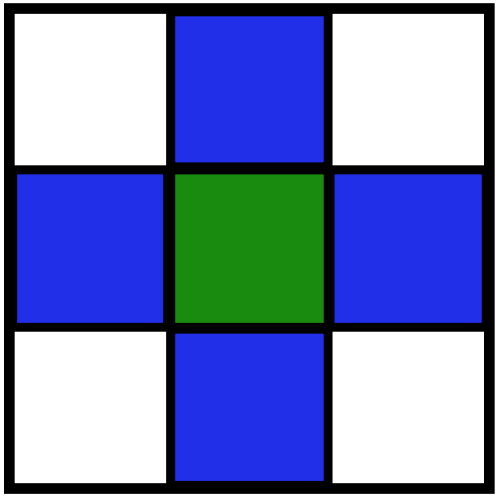
- **In 1D, used a 3 point stencil**
- **In 2D, 5 point and 9 point stencils are common**

Green cell depends on blue cells

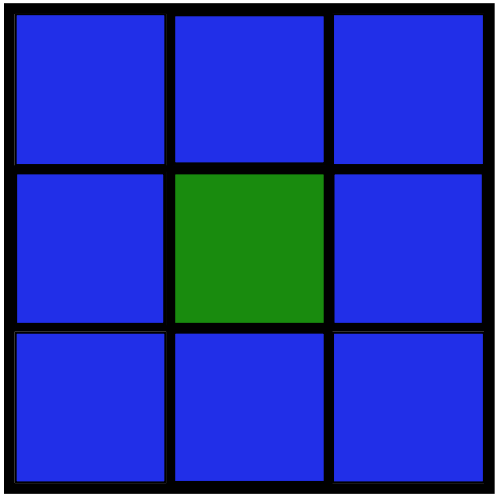
1D 3 Point Stencil



2D 5 Point Stencil

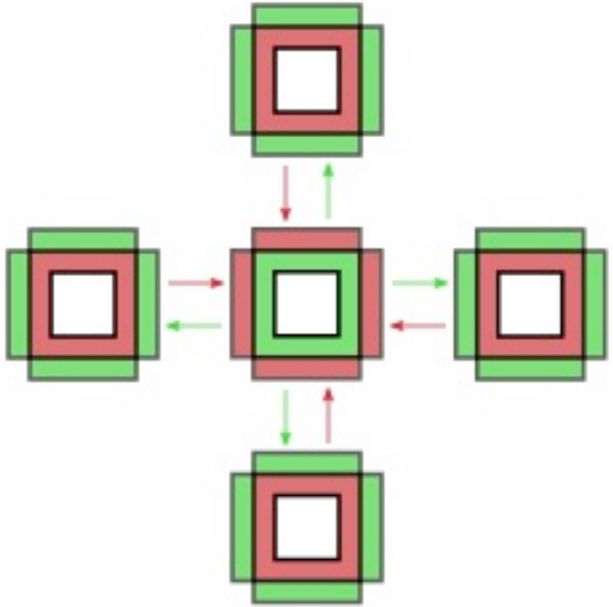


2D 9 Point Stencil

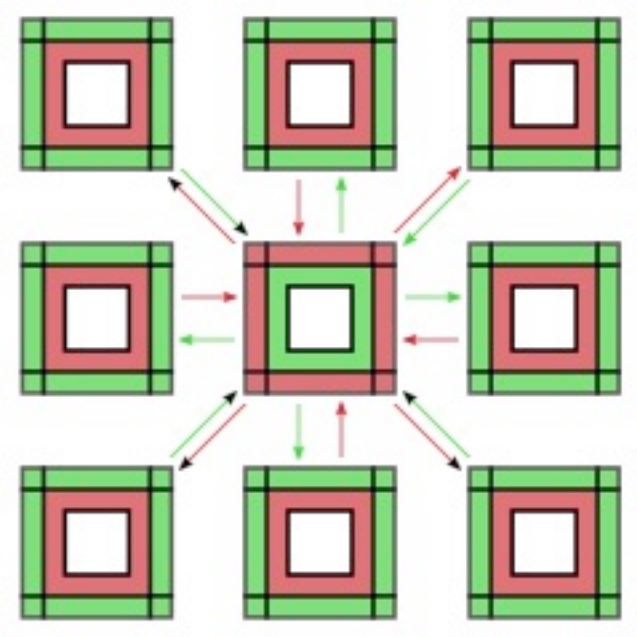


2D Cellular Automata

- **5 point stencil communication**
  - Post 4 Irecv calls
  - Post 4 Isend calls
  - Post a wait all



2D Cellular Automata



- **9 point stencil communication**
  - Option 1
    - Post 8 Irecv calls
    - Post 8 Isend calls
    - Post a wait all
  - Option 2
    - Post 2 Irecv calls (up/down)
    - Post 2 Isend calls (up/down)
    - Post a wait all
    - Post 2 Irecv calls (left/right with ghost cells)
    - Post 2 Isend calls (left/right with ghost cells)
    - Post a wait all

The End