MPI

Charles Bacon

Framework of an MPI Program

- Initialize the MPI environment
  - MPI_Init(…)
- Run computation / message passing
- Finalize the MPI environment
  - MPI_Finalize()

Hello World fragment

```
#include <mpi.h>
  rc = MPI_Init(&argc, &argv);
  rc = MPI_Comm_rank(MPI_COMM_WORLD,
&rank);
  rc = MPI_Comm_size(MPI_COMM_WORLD,
&size);
  printf(“Hello from %d of %d”, rank, size);
  rc = MPI_Finalize();
```

Things to note about hello world

- When you run many tasks of this program, they are all running the same code
  - Can differentiate behavior by, for instance, conditionals on my rank: if (rank == 0) {…}
- This example does not include message passing
- It’s perfectly fair to have many ranks running on the same compute node

Point to point communication

- Send / receive
- Basic information:
  - Who should I send to / receive from?
  - What am I getting? (ints, floats, structs, … ?)
  - How many of those am I getting?
  - What buffer should I save them into?
  - Did it succeed?
  - What was the purpose of the message?

Send / recv example

MPI_Send(data, 10, MPI_INT, target, 0, MPI_COMM_WORLD);

Send 10 integers starting from my “data” pointer

Send them to rank “target” in MPI_COMM_WORLD

With message tag “0”

Send / recv example

MPI_Recv(&buff, 1, MPI_DOUBLE, sendr, 0, MPI_COMM_WORLD, &stat);

Receive from rank “sendr” in MPI_COMM_WORLD

- Can optionally receive from any source

Receive a single double into my “buff” variable

Message should have the “0” tag

- Can optionally receive from any tag

Store status into the “stat” variable.

- Contains source/tag info, plus error code

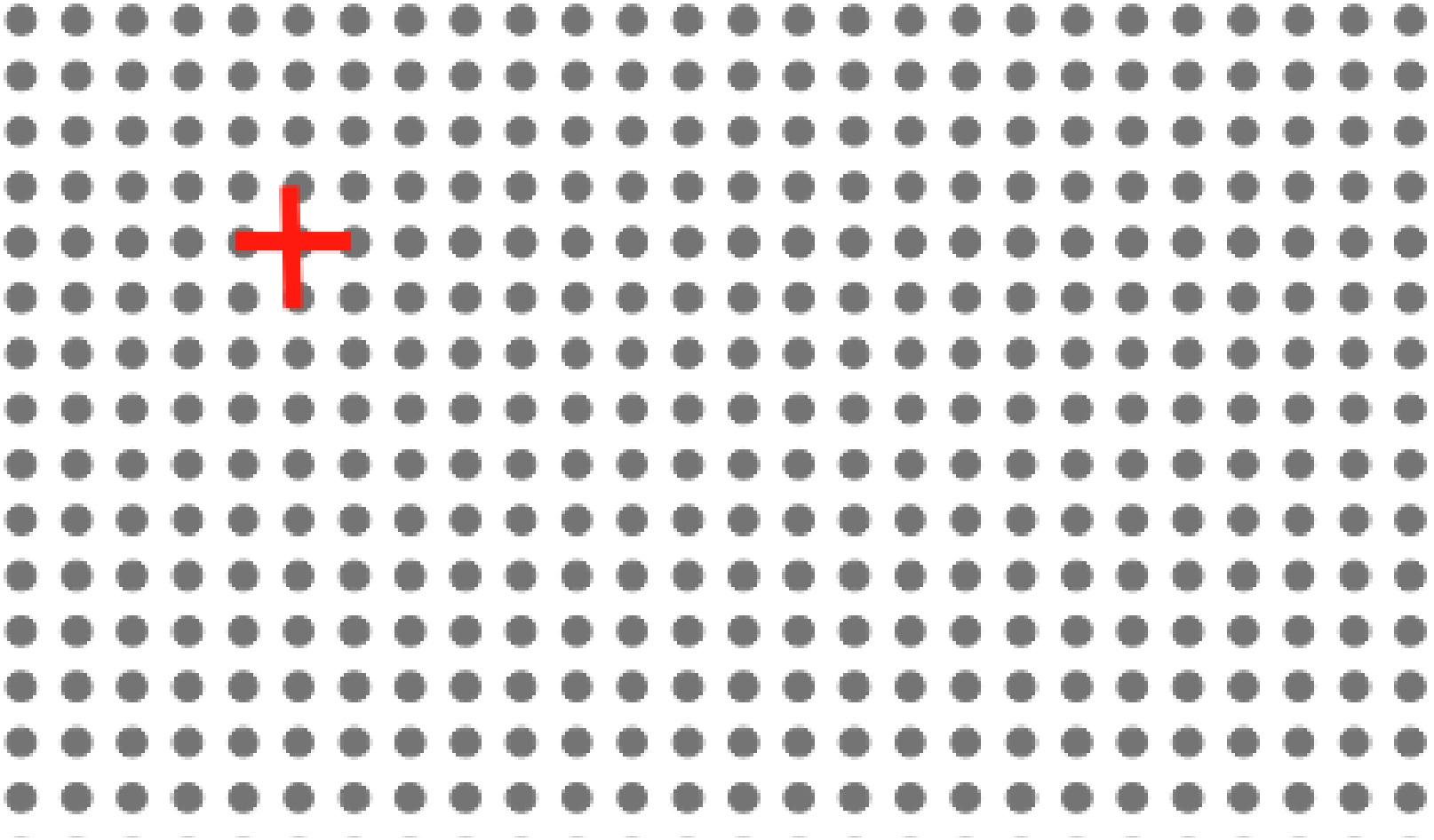
You can now do cool things!

- Point-to-point send and receive are very powerful tools. You can now write most parallel programs you would want to.
- Everything else I’m going to talk about now are to give you finer-grained control over behavior, avoid deadlocks, and get better performance.

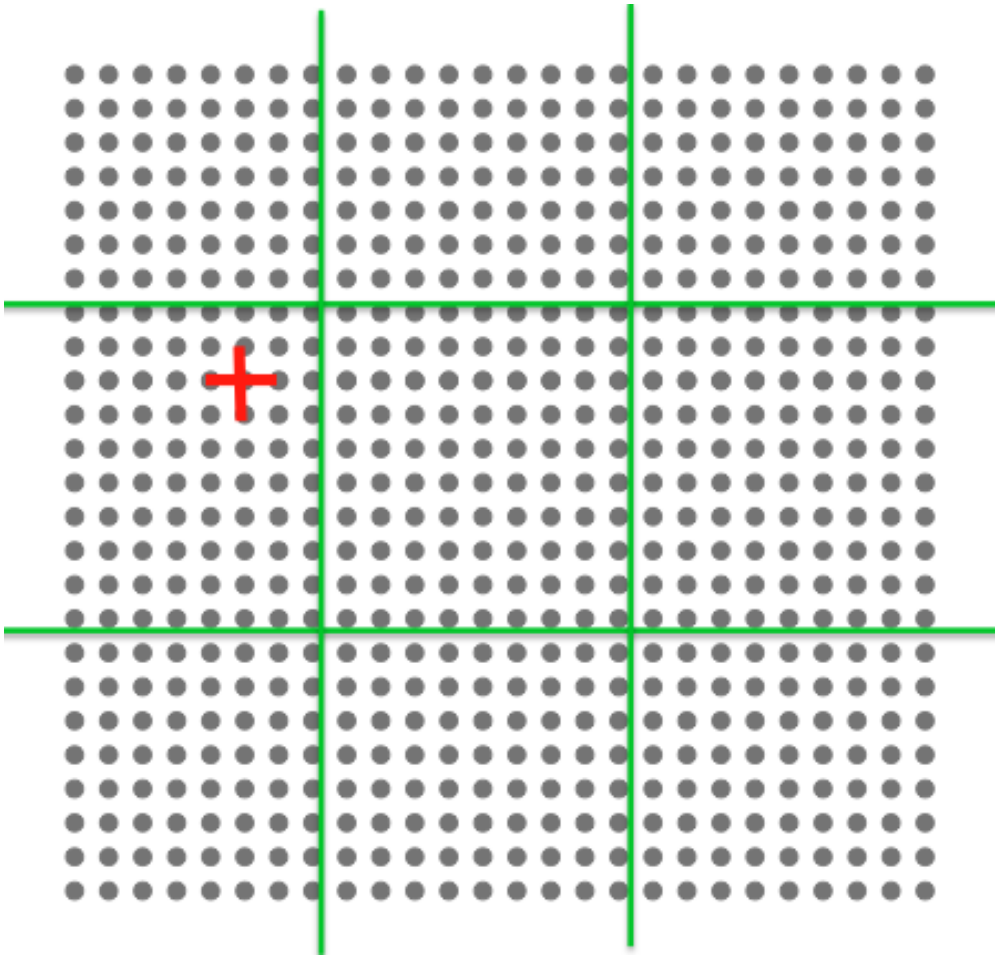
Heat diffusion

- To approximate the solution of the Poisson Problem ∇2u = f on the unit square, with u defined on the boundaries of the domain (Dirichlet boundary conditions), this simple 2nd order difference scheme is often used:
  - (U(x+h,y) - 2U(x,y) + U(x-h,y)) / h_2 + (U(x,y+h) - 2U(x,y) + U(x,y-h)) / h_2 = f(x,y)
  - Where the solution U is approximated on a discrete grid of points x=0, h, 2h, 3h, ... , 1, y=0, h, 2h, 3h, ... 1.
  - To simplify the notation, U(ih,jh) is denoted U_{ij}

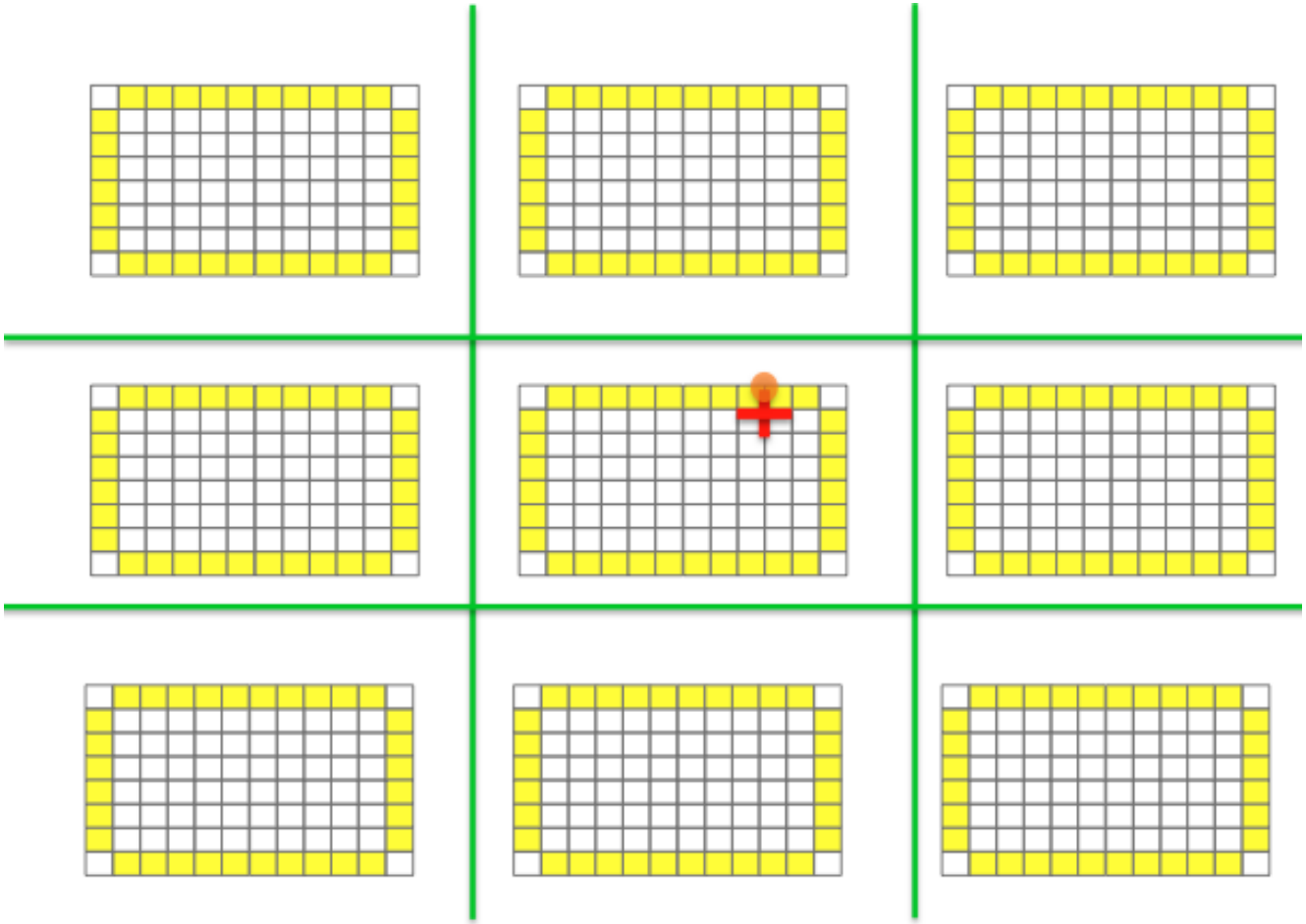
The five-point stencil



Five point stencil, 9 nodes



Ghost cells



Local data structure

- Each node has a local patch of the global array. A “halo” is allocated around the local patch to store the information required from our neighbors
- Each node computes a timestep, then both:
  - Shares its data with its neighbors
  - Receives updated data from its neighbors
- This goes by the name Bulk Synchronous Processing

Heat diffusion outline

```
for(i=0; curr_time < end_time; i++){
    if (i % update_steps == 0) {print_status();}
    local_diffusion(my_data, dx, dt, a);
    ghost_update(my_data, myrank, size);
    curr_time += dt;
}
```

Othing things you can do

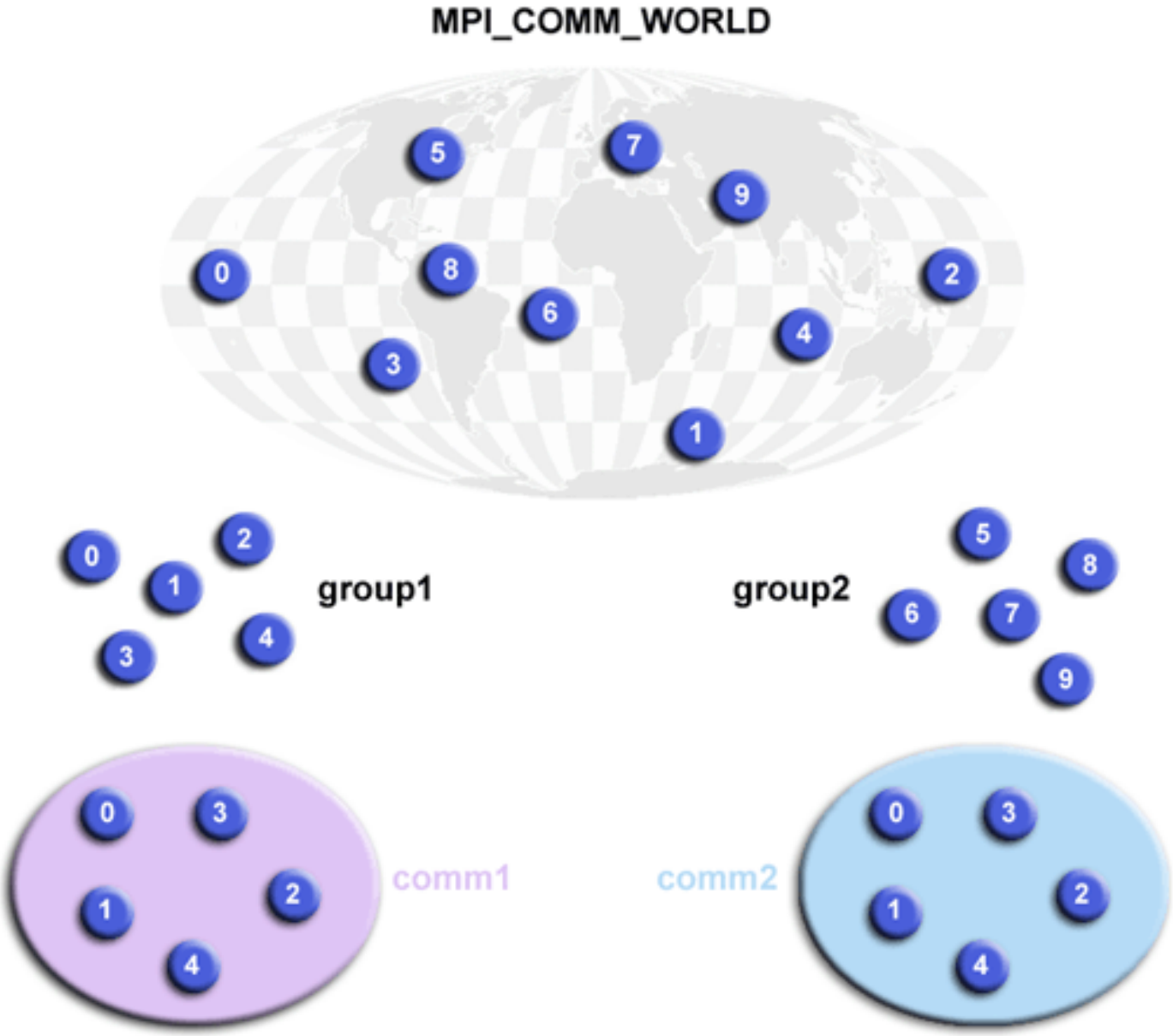
- Calculate PI in a parallel manner
  - Take care for distributed random number generation
- Find prime numbers, find pyramidal numbers
  - Take care for load balancing
- Distribute tasks from rank 0 to worker ranks
  - Take care that rank 0 does not become the bottleneck

Collectives

- Collectives implement communicator-wide communication in a way that allows the implementation to optimize behavior
- Some examples:
  - MPI_Barrier to synchronize
  - MPI_Reduce to get sum/min/max/product of data
  - MPI_Scatter to spread data from one node to many others

Communicators

- Enough dancing around – what are the communicators about?
- Imagine running a coupled simulation in a single code. You might want to run collectives on a subset of the nodes
- Imagine using a matrix-multiply library. How does it know what subset of nodes are participating?


MPI_COMM_WORLD
5
7
9
0
8
2
6
4
3
1
0
2
1
group1
3
4
5
8
group2
6
7
9
0
3
comm1
1
2
4
comm2
0
3
1
2
4

MPI_Bcast and MPI_Scatter

- It’s common in a serial code to read in some startup data, then run
- Please don’t make every rank of your parallel program open and read from a file!
- Instead, let rank 0 read in the file, then either broadcast or scatter the data, as appropriate
  - Your interconnect is way faster than a disk

Parallel I/O and visualization

- I/O gets complicated at scale. Fortunately there are libraries to help
- Look into HDF5, NetCDF, pNetCDF, ADIOS as possible APIs that will save you the headache of architecting scalable parallel I/O
- Also pay attention to visualization via Paraview or VisIt
- MPI-2 does include some parallel file primitives if you want to roll your own

Oh dear: Deadlock

- Before introducing deadlock, I'd like to introduce deadlock.
- Rank 0:

  MPI_Recv(&buff, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &stat);

  MPI_Send(data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
- Rank 1:

  MPI_Recv(&buff, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &stat);

  MPI_Send(data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

Causes of deadlock

- Messages interleaving differently than you expected, causing a message not to be received how you expected
- Not making progress in a send/receive because of blocking semantics

Message ordering guarantees

- If a sender sends two messages to the same destination and both match the same receive, the receive operation will receive the first before the second
- If a receiver posts two receives that match the same message, the earlier receive will match first
- No fairness applies to messages from different senders

Unexpected ordering deadlock

- This can happen, for instance, with MPI_ANY_SOURCE receives.

  MPI_Recv(…, MPI_ANY_SOURCE, …)

  MPI_Recv(…, rank 2, …)

- If rank 0 sends first, then rank 2 sends, no guarantee that you won’t match the “any source” with the rank 2 message, then be stuck forever waiting to hear from rank 2
- This situation can also be resolved with tags

Blocking and Memory semantics

- After MPI_Send completes, what is true?
  - You are free to change the contents of your send buffer
  - The remote side may not have received the data
- After MPI_Recv completes, what is true?
  - The data has landed in your application buffer
- Note that MPI has its own buffers it

You have a lot of control

- You have control over:
  - When your code should be allowed to proceed
  - What the MPI implementation is allowed to assume the receiver of your code has done
  - What buffers are used by the operations
- You have the obligation not to violate the contract associated with the options you choose

First: non-blocking receive

- What if we could have both sides say “I am willing to receive a message, but let me keep going” and then they both performed their sends -> no deadlock
- MPI_Irecv(…) gives you an MPI_Request handle to track the progress of the receive
- MPI_Test() will peek at the status
- MPI_Wait() will block for completion
- You must do one or both of these things to ensure that MPI can make progress!

MPI_Irecv()

- MPI_Irecv (&buf, count, datatype, source, tag, comm, &request);
  - The same as MPI_Recv, but with an MPI_Request at the end instead of an MPI_Status
- MPI_Test: 1 if done, 0 if not
- MPI_Wait: Fills in an MPI_Status after waiting for the MPI_Request to finish
  - Often used: MPI_Waitall() on an array of statuses

Optimization related to irecv

- One nice thing about irecv() is it gives MPI the opportunity to allocate buffer space for the receive before the send posts
- This enables some implementations to do remote direct memory access (RDMA) to place the message into memory from the network device without interrupting the program

MPI_Isend()

- Same kind of thing as MPI_Irecv(), but with a little more onerous contract
- You promise not to modify the buffer being used for the send until the send has completed, because you don’t know when MPI will copy it out
- Again, must either test or wait to ensure progress on the send

Other send/recv variants

- MPI_Sendrecv: combines send and receive into a single call for a pair who are exchanging data
- MPI_Bsend: Tell MPI to use a buffer that you supply, so you have control over memory
- MPI_Ssend: Don’t return until the receive has begun processing
- MPI_Rsend: I promise that the recv posted

MPI Libraries

- One of the real strengths of MPI was that it was well-designed to let third parties write libraries that you can re-use
  - This is a strong motivator for communicators, even if you yourself only use COMM_WORLD
- Now that we have discussed how you can write your own MPI code, let's talk about how you can avoid writing it

Some MPI-friendly software

- PETSc, SUNDIALS: numerical frameworks
- Dense linear algebra: BLAS, ScaLAPACK
- Sparse linear algebra: SuperLU
- Meshing: MOAB, ParMETIS
- Load-balancing: Included in PETSc, ADLB
- Molecular Dynamics: AMBER, CHARMM, NAMD
- Spectral methods: FFTW
- Computational chemsitry: NWChem, GPAW

Other MPI programming models

- At smaller scales, you see a lot of mixing between threaded on-node with OpenMP or CUDA combined with MPI between nodes.
- Another featured introduced later in MPI is one-sided communications; too advanced to cover here
  - Sometimes this will be provided to you by a library interface like Global Arrays (GA)

Debugging and Optimizing

- There are MPI debugging tools available, like DDT, Totalview, and gdb
- For profiling, things like MPI_Wtime() to get timers, gprof, TAU, and HPCToolkit() can collect performance data
- Also, don’t forget to optimize your serial program first
- Pay attention to load balancing

Scalability is not the same as performance

- Performing work in parallel involves book-keeping costs that serial codes don’t have
- Highly performant codes often fail to achieve perfect scaling
- Highly inefficient codes are easier to scale!
- Make sure you measure what matters: time to solution

Things we didn’t cover

- MPI Datatypes:
  - Allows optimizations related to packing/unpacking data for sends and receives
- MPI Topologies:
  - Let MPI assign a logical cartesian structure to your ranks, instead of you coding it
- MPI Communicator routines:
  - Creating new ones, splitting on conditions
- MPI RMA/one-sided
- MPI-IO
- MPI-3 features
  - Non-blocking collectives, neighborhood collectives, new RMA features