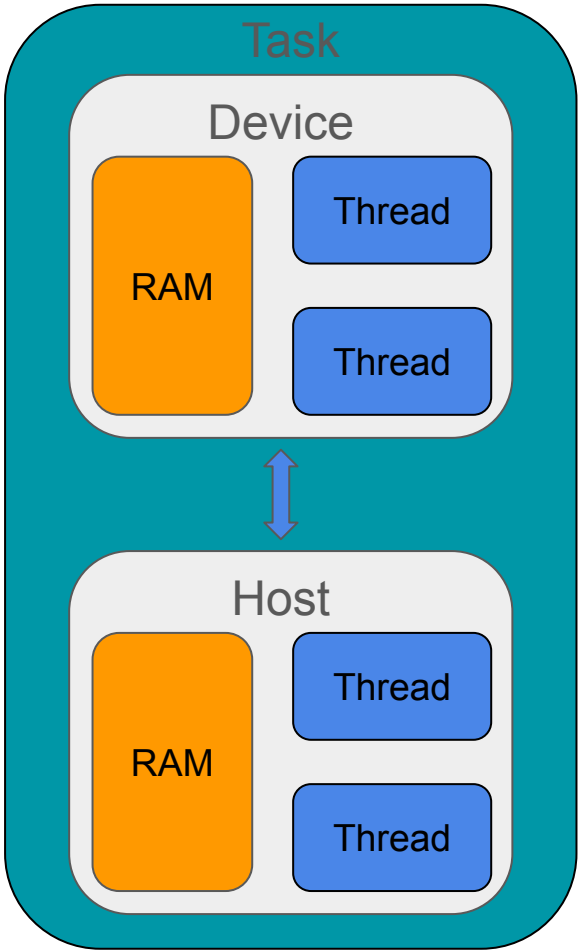
Session 3: MPI

Outline

- General description
- Point-to-point communication
- Collective communication
- Closing remarks

The Message Passing Interface (MPI) is the distributed-memory component to hPIC2's hybrid parallelism

- Unlike with shared-memory, MPI dominates distributed-memory parallelism
- Literally, multiple copies (**processes** or **tasks**) of program are launched and connected
- Each task maintains its own memory
  - Each task can also have its own shared-memory instance
- With MPI+Kokkos, we can now run on any supercomputer
  - With GPUs, use 1 GPU per MPI task
  - Without GPUs, generally use one CPU socket per MPI task



MPI is a standard, NOT a library in and of itself

- Similar to how C++ is a standard, but with many compilers that actually do the work
- The standard describes function names, call signatures, and functionality
  - The standard describes C and FORTRAN functions and requires the C functions to work with C++
- How these functions are coded is left up to **implementations**
- Many implementations available from many vendors
  - OpenMPI (not to be confused with OpenMP) is probably the most common today
  - MPICH is an early (but still around) implementation from Argonne
  - MVAPICH2 another popular open-source option
  - Intel, IBM, Cray all have their own closed-source implementations
- The point: standard-compliant code will work with any up-to-date implementation

Like Kokkos, MPI must be initialized and finalized

- Include the mpi.h header
- Passing arguments allows MPI to “cut out” its own command-line options
- MPI is C code, so no fancy scope guard object. Just gotta remember to finalize

```
#include <Kokkos_Core.hpp>
#include <mpi.h>

int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  // Can alos use MPI_Init(NULL, NULL) with no arguments.
  // If using Kokkos, initialize it after MPI.
  Kokkos::initialize(argc, argv);
  {
    // Do stuff.
  }
  Kokkos::finalize();
  MPI_Finalize();
  return 0;
}
```

A **communicator** is an ordered set of tasks, all connected

- A **group** is an unordered set of tasks
  - So a communicator is a group imbued with an ordering
- Each task's order is called its **rank**
- One communicator is always available, MPI_COMM_WORLD
  - This consists of all tasks that the program starts with
  - This is a macro that can be used in code, we'll see examples later
- A communicator is necessary for almost all MPI functions so that the implementation knows how the available tasks are connected
- New communicators can be created by splitting or reordering old communicators, or from scratch
  - hPIC2 doesn't do this however, we pretty much just stick with the original MPI_COMM_WORLD for simplicity

Communicator size and task's rank found with simple utils

- Since MPI is C code, get function output by pre-allocating the output variable and passing a pointer to it
  - They couldn't just return things, could they…
- When you run this with multiple tasks, each task will fill their local rank variable with a different value
- So each task runs the same code, but with different data

```c
#include <mpi.h>

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

    int comm_size;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Finalize();
    return 0;
}
```

Data arrays can be sent directly to other ranks

```
void MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator
);
```

Pointer to data. Implicitly cast

Number of objects in array

Datatype of array. See next slide

Rank to which to send data

Unique tag for this message

Communicator in which to do send

MPI has many pre-defined datatypes

- MPI uses these to determine size of sent array (**buffer**) in bytes
- Matching these datatypes is important
- Datatype sizes differ based on architecture
  - On some system, a long int is the same size as an int, but on others, it's the same size as a long long int
  - Matching MPI datatype ensures the sizes are consistent

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

Message must be received on the other end

```
void MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status *status
);
```

Pointer to pre-alloc'd data. Implicitly cast

Length of received array

Received datatype

Source rank. Can be MPI_ANY_SOURCE

Tag matching desired MPI_Send tag.
Can be MPI_ANY_TAG

Communicator in which to receive

Object with info about how the receive went. Typically, MPI_STATUS_IGNORE to do nothing and save cycles

Example: Send/Recv short array

- Both ranks allocate a length-3 array
- Rank 0 fills it with data and sends it to rank 1
- Rank 1 receives data and prints it to stdout
- If length is not known by destination, first do a Send/Recv for the length of the array
- Aside: code requires >1 task to run
  - Not good design!
  - Code should be designed to work with any number of tasks

```cpp
#include <mpi.h>
#include <iostream>

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

    double data[3];
    if (rank == 0) {
        data[0] = 0.0; data[1] = 1.0; data[2] = 2.0;
        MPI_Send(
            data,
            3,
            MPI_DOUBLE,
            1,
            0,
            MPI_COMM_WORLD
        );
    }
    else if (rank == 1) {
        MPI_Recv(
            data,
            3,
            MPI_DOUBLE,
            0,
            0,
            MPI_COMM_WORLD,
            MPI_STATUS_IGNORE
        );
        std::cout << "Received data: " << data[0] << ", "
                  << data[1] << ", " << data[2] << "\n";
    }

    MPI_Finalize();
    return 0;
}
```

std::vectors can be used as the arrays

- vectors are guaranteed to be contiguous (data is stored in an unbroken line in memory) by the C++ standard
- The “data” method returns a raw pointer to the data stored by a vector
- The “size” method returns length
- Pass this directly to MPI
- vectors are more C++-like and allow for easy dynamic sizing

```
std::vector<double> array(3);
if (rank == 0) {
    array[0] = 1.0; array[1] = 2.0; array[2] = 3.0;
    MPI_Send(
        array.data(),
        array.size(),
        MPI_DOUBLE,
        1,
        0,
        MPI_COMM_WORLD
    );
}
else if (rank == 1) {
    MPI_Recv(
        array.data(),
        array.size(),
        MPI_DOUBLE,
        0,
        0,
        MPI_COMM_WORLD,
        MPI_STATUS_IGNORE
    );
    std::cout << "Received data: " << array[0] << ", "
              << array[1] << ", " << array[2] << std::endl;
}
```

Views can sometimes be used too

- MPI can be CUDA-aware, where it'll accept pointers to device memory
- If not, HostMirrors usually work
- Can tell at compile-time with some macro definitions
- Only works if View is contiguous, which is usually true unless the View is actually a SubView
  - Can use the .span_is_contiguous() method to check at runtime

```
Kokkos::View<double*> array("array", 3);
if (rank == 0) Kokkos::deep_copy(array, 1.0);
#if MPIX_CUDA_AWARE_SUPPORT && defined(KOKKOS_ENABLE_CUDA)
Kokkos::View<double*> mpi_array = array;
#else
Kokkos::View<double*>::HostMirror mpi_array = Kokkos::create_mirror_view(array);
Kokkos::deep_copy(mpi_array, array);
#endif
if (rank == 0) {
    MPI_Send(mpi_array.data(), 3, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
}
else if (rank == 1) {
    MPI_Recv(mpi_array.data(), 3, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
Kokkos::deep_copy(array, mpi_array);
```

Send/Recv are **blocking** functions

- Send doesn't return until communication is complete
- Very easy to accidentally hang
- In example on right, both ranks want to Send an array to the other, then receive
- But both just wait for the other to Recv forever!

(Note: modern compilers are smart enough that this might not actually hang, but leave nothing to chance)

```
std::vector<double> array_A(3);
std::vector<double> array_B(3);
if (rank == 0) {
    array_A[0] = 1.0; array_A[1] = 2.0; array_A[2] = 3.0;
    MPI_Send(array_A.data(), 3, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
    MPI_Recv(array_B.data(), 3, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else if (rank == 1) {
    array_B[0] = 4.0; array_B[1] = 5.0; array_B[2] = 6.0;
    MPI_Send(array_B.data(), 3, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(array_A.data(), 3, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Non-blocking communications can be used instead

- Functions preceded with “I” for “immediate return”
- Task continues after call, but data may still be in use, so shouldn’t touch it until we are sure data is no longer in use by communicator
- Call comes with request handle that can be used to verify status of communication
- MPI_Wait will wait until request is finished; safe to modify data afterward
- Non-blocking calls can be used for latency hiding
- Most MPI calls have a non-blocking version

```
std::vector<double> array_A(3);
std::vector<double> array_B(3);
MPI_Request request;
if (rank == 0) {
    array_A[0] = 1.0; array_A[1] = 2.0; array_A[2] = 3.0;
    MPI_Isend(array_A.data(), 3, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);
    MPI_Recv(array_B.data(), 3, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // Don't touch array_A until the send is complete!
    MPI_Wait(&request, MPI_STATUS_IGNORE);
    // Now okay to modify array_A.
    array_A[0] = 7.0;
}
else if (rank == 1) {
    array_B[0] = 4.0; array_B[1] = 5.0; array_B[2] = 6.0;
    MPI_Isend(array_B.data(), 3, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &request);
    MPI_Recv(array_A.data(), 3, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
}
```

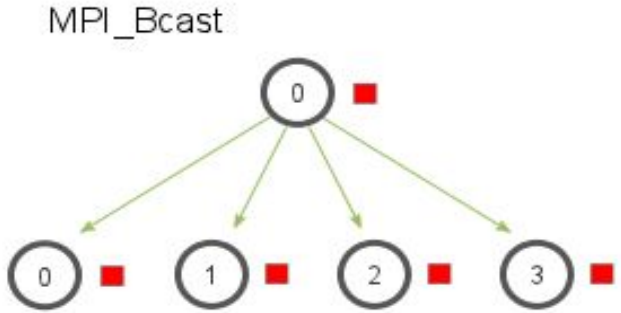
Collective functions must be called by all tasks in a communicator

- Definition: tasks will wait at collective function call until EVERY task has reached it
- Simplest is MPI_Barrier, which does nothing except effectively wait until all tasks reach it
- If one task is stuck, ALL tasks hang!
- Practically, it makes little sense to do P2P communications unless code is tailored to a certain # of tasks, so we mostly use collective communications

```
MPI_Barrier(MPI_COMM_WORLD);
```

Bcast broadcasts

- Copies data from an array on one rank to all other ranks
- In example, all ranks alloc, but only rank 0 fills data. Then Bcast copies it to all other ranks
- Generally wanna choose rank 0 as the root
  - It's the only rank guaranteed to appear regardless of the number of tasks
  - Scales up to arbitrarily many tasks

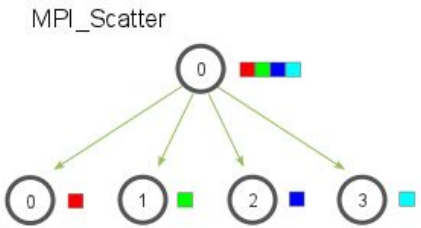


https://mpitutorial.com/

```
std::vector<double> array(3);
if (rank == 0) {
    array[0] = 1.0; array[1] = 2.0; array[2] = 3.0;
}
MPI_Bcast(
    array.data(),  // Pointer to array
    3,             // Length of array
    MPI_DOUBLE,    // Array datatype
    0,             // Root rank
    MPI_COMM_WORLD // Communicator
);
```

Scatter splits an array into chunks and sends the chunks out to all other tasks

- Rank 0 fills an array with 2 entries per task
- All tasks participate in the Scatter, including the root rank
- The receive buffer from each task is filled with its corresponding chunk from the root task's send buffer



https://mpitutorial.com/

```cpp
int comm_size;
MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
std::vector<double> received(2);
std::vector<double> sent(2*comm_size);
if (rank == 0) {
    for (int i = 0; i < sent.size(); i++) {
        sent[i] = i;
    }
}
MPI_Scatter(
    sent.data(),     // send buffer. only needs to be valid on root rank
    2,               // Number of elements to send to each rank
    MPI_DOUBLE,      // Send buffer datatype
    received.data(), // Buffer into which to put received data
    2,               // Length of receive buffer
    MPI_DOUBLE,      // Receive buffer datatype
    0,               // Root rank
    MPI_COMM_WORLD   // Communicator
);
std::cout << "Rank " << rank << " received: "
          << received[0] << " " << received[1] << std::endl;
```

Example output with 4 tasks:

```
Rank 0 received: 0 1
Rank 2 received: 4 5
Rank 1 received: 2 3
Rank 3 received: 6 7
```

What if we want each rank to receive a different size?

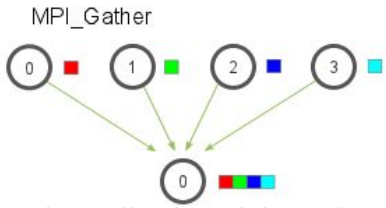
- Use Scatterv (v for variable-length)
- Root rank puts all data into one big buffer
- Root also needs an array to tell MPI how many things to send to each task, and where that task's data starts in the big send buffer
- Receiving ranks only need to know how many things to receive

```cpp
int comm_size;
MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
std::vector<double> received(rank);
std::vector<double> sent;
std::vector<int> counts;
std::vector<int> displacements;
if (rank == 0) {
    int total_num_to_send = comm_size * (comm_size - 1) / 2;
    sent.resize(total_num_to_send);
    counts.resize(comm_size);
    displacements.resize(comm_size);
    for (int i=0; i<total_num_to_send; i++) {
        sent[i] = i;
    }
    for (int i=0; i<comm_size; i++) {
        counts[i] = i;
        displacements[i] = i * (i - 1) / 2;
    }
}
MPI_Scatterv(
    sent.data(),          // send buffer. only needs to be valid on root rank
    counts.data(),        // Number of entries to send to each rank.
    displacements.data(), // Offset in send buffer for each rank's data
    MPI_DOUBLE,           // Send buffer datatype
    received.data(),      // Buffer into which to put received data
    rank,                 // Length of receive buffer
    MPI_DOUBLE,           // Receive buffer datatype
    0,                    // Root rank
    MPI_COMM_WORLD        // Communicator
);
std::cout << "Rank " << rank << " received: ";
for (int i=0; i<rank; i++) {
    std::cout << received[i] << " ";
}
std::cout << std::endl;
```

Output with 4 tasks:

```
Rank 0 received:
Rank 2 received: 1 2
Rank 3 received: 3 4 5
Rank 1 received: 0
```

Gather is like the opposite of Scatter



https://mpitutorial.com/

- All tasks combine chunks into one big array and send it to a root rank
- Like Scatterv, there is a Gatherv where number of elements received from each task is variable

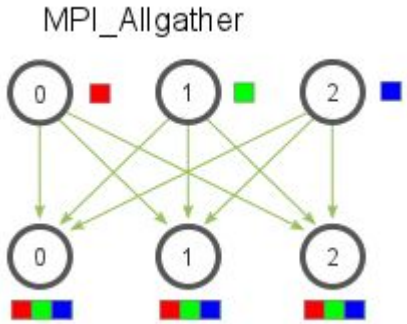
```cpp
int comm_size;
MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
std::vector<double> to_send(2);
to_send[0] = rank; to_send[1] = rank*2.0;
std::vector<double> to_recv;
if (rank == 0) {
    to_recv.resize(2*comm_size);
}
MPI_Gather(
    to_send.data(), // Send buffer
    2,              // Number of elements to send
    MPI_DOUBLE,     // Type of elements to send
    to_recv.data(), // Receive buffer
    2,              // Number of elements received per task
    MPI_DOUBLE,     // Type of elements received
    0,              // Rank of root
    MPI_COMM_WORLD  // Communicator
);
if (rank == 0) {
    std::cout << "Received: ";
    for (int i=0; i<to_recv.size(); i++) {
        std::cout << to_recv[i] << " ";
    }
    std::cout << std::endl;
}
```

Output with 4 tasks

```
Received: 0 0 1 2 2 4 3 6
```

Allgather to get result of gather on every task

- Just a Gather but without a root, every task must be prepared to receive the data
- Yes, there is an Allgatherv!



https://mpitutorial.com/

```
int comm_size;
MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
std::vector<double> to_send(2);
to_send[0] = rank; to_send[1] = rank*2.0;
std::vector<double> to_recv(2*comm_size);
MPI_Allgather(
    to_send.data(), // Send buffer
    2,              // Number of elements to send
    MPI_DOUBLE,     // Type of elements to send
    to_recv.data(), // Receive buffer
    2,              // Number of elements received per task
    MPI_DOUBLE,     // Type of elements received
    MPI_COMM_WORLD  // Communicator
);
std::cout << "Rank " << rank << " received: ";
for (int i=0; i<to_recv.size(); i++) {
    std::cout << to_recv[i] << " ";
}
std::cout << std::endl;
```

Output with 4 tasks

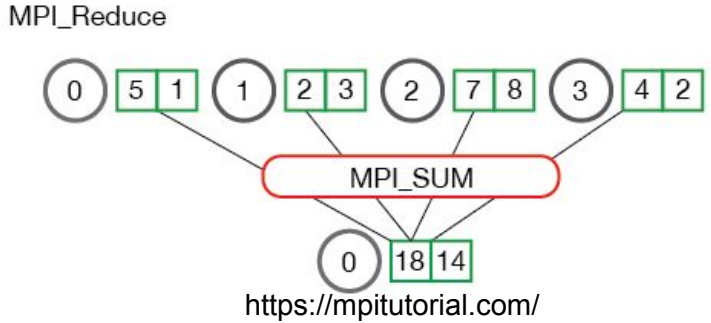
```
Rank 2 received: 0 0 1 2 2 4 3 6
Rank 3 received: 0 0 1 2 2 4 3 6
Rank 0 received: 0 0 1 2 2 4 3 6
Rank 1 received: 0 0 1 2 2 4 3 6
```

Reduce to perform a “sum”

- Essentially sums each entry of an array over the tasks
- But it doesn’t just have to be a sum!
  - Can also find the min or max, a product, or logical OR and AND
- Allreduce sends reduction result to all tasks, like Allgather
- There is no Reducev; all tasks must submit array of the same length!

Output with 4 tasks

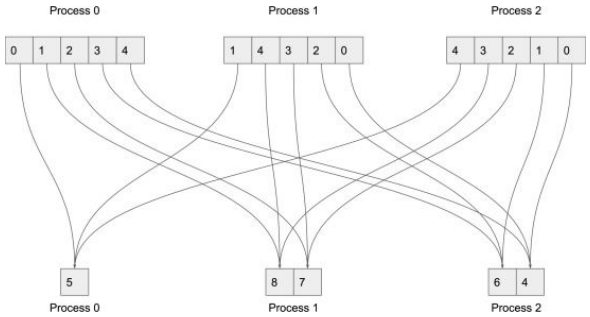
```
Received: 6 12
```



https://mpitutorial.com/

```cpp
int comm_size;
MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
std::vector<double> to_reduce(2);
to_reduce[0] = rank; to_reduce[1] = rank*2.0;
std::vector<double> reduced;
if (rank == 0) {
    reduced.resize(2);
}
MPI_Reduce(
    to_reduce.data(), // Send buffer
    reduced.data(),   // Receive buffer
    2,                // Number of elements to reduce
    MPI_DOUBLE,       // Type of elements to reduce
    MPI_SUM,          // Reduction operator
    0,                // Rank of root
    MPI_COMM_WORLD    // Communicator
);
if (rank == 0) {
    std::cout << "Received: ";
    for (int i=0; i<reduced.size(); i++) {
        std::cout << reduced[i] << " ";
    }
    std::cout << std::endl;
}
```

Reduce_scatter is like Reduce + Scatter



- Performs a reduction over all tasks on an array
- Then split up the result into chunks and send each chunk to a different task
- Useful for setting up potential solve

Output with 4 tasks

```
Rank 0 received:
Rank 3 received: 18 24 30
Rank 1 received: 0
Rank 2 received: 6 12
```

```cpp
int comm_size;
MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int total_size = comm_size * (comm_size - 1) / 2;
std::vector<double> to_reduce(total_size);
std::vector<double> reduced(rank);
std::vector<int> counts(comm_size);
for (int i=0; i<total_size; i++) {
    to_reduce[i] = i * rank;
}
for (int i=0; i<comm_size; i++) {
    counts[i] = i;
}
MPI_Reduce_scatter(
    to_reduce.data(), // Send buffer
    reduced.data(),   // Receive buffer
    counts.data(),    // Number of elements to scatter to each task
    MPI_DOUBLE,       // Type of elements to reduce
    MPI_SUM,          // Reduction operator
    MPI_COMM_WORLD    // Communicator
);
std::cout << "Rank " << rank <<  " received: ";
for (int i=0; i<reduced.size(); i++) {
    std::cout << reduced[i] << " ";
}
std::cout << std::endl;
```

To run MPI code, use mpirun or mpiexec

- E.g., rather than just `./hpic2`, use `mpiexec -np 4 ./hpic2` to run with 4 tasks
- There are a million different command-line options for mpiexec
  - Look up the command-line options for your implementation on its man pages
- Important to note that number of tasks is decided at runtime, NOT when writing code
- We could check to make sure the user is running with a valid number of tasks, but better practice to write code that works no matter how many tasks the user throws at it

The “MPI Mindset”

- With Kokkos, explicitly define a work load for each thread and launch it
- With MPI, when you write code, you must realize that every task will be executing that code, perhaps with different data
- Code should be designed to work with arbitrarily many tasks
- Communication is the enemy
  - Avoid communication if at all possible. Usually having every task perform some computation is cheaper than having one task do it and broadcast results
  - Reduce size of communications. Instead of sending a whole array, consider sending only the portion needed by other tasks
  - Hide latency. The actual sending/receiving is done by a handler outside of your program. Set up communication as early as possible, do task-local calculations while waiting for data to be received.
- I find the man pages at https://www.open-mpi.org/doc/v4.1/ to be super useful in reminding myself of syntax and functionality

Exercise: distribute your 1D particle pusher!

We'll reference the exercise from the Kokkos session, so have that finished and handy. We're going to distribute it in the same way that hPIC2's finite-difference mesh pusher is distributed: particles are pushed throughout the entire domain, but the domain is split up for the field solve. Don't worry about the field solve just yet. What it means is that we will reduce the accumulated charge density over the mesh, but split up the reduction result and send chunks to the MPI ranks that will be responsible for the field on that chunk (sound familiar?)

1. Do the exercise from Session 2 if you haven't already.
2. Add MPI_Init and MPI_Finalize.
3. Split the particles as evenly as possible over the tasks, ensuring that the total number of particles is as desired even when it is not divisible by the number of tasks.
4. Do the same for the nodes.
5. Initialize and push only the local number of particles.
6. After every push, use MPI_Reduce_scatter to reduce the charge density over all tasks and distribute chunks to local arrays having the length that you determined in step 4.