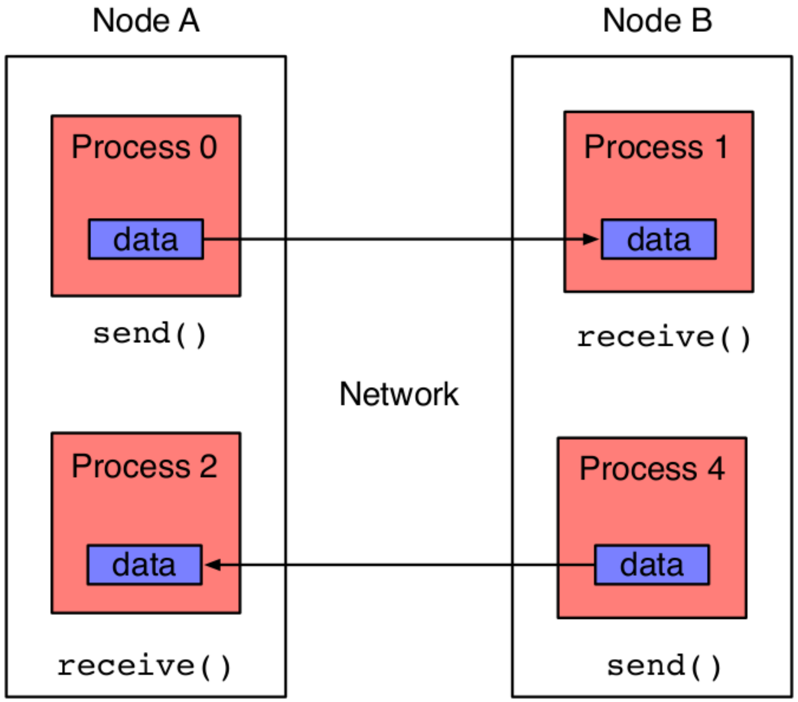
Overview: Message Passing

- message passing in a nutshell
- a bit of history (the advent of MPI-1)
- MPI basics
  - what is MPI?
  - motivation and history
  - “hello world” MPI program
  - code compilation and execution
- MPI point-to-point communication and transfer semantics
  - blocking semantics point-to-point communication
  - non-blocking semantics point-to-point communication
- MPI collectives, datatypes, and communicators

Message Passing in a Nutshell

- parallelism realized by multiple processes (aka tasks) each with their own local memory address space
- data is moved from address space of one process to that of another by sending/receiving messages
- processes may run on separate compute nodes, different cores within a node, or even on same processor core
- all variables in a process are local to this process. No concept of shared-memory
- strictly required if target parallel computer is distributed-memory.
- “de facto” standard is MPI



A bit of history (the advent of MPI-1)

- parallel computer vendors initially developed own message-passing APIs
  - e.g. Fujitsu's APLib for the AP1000 series (1991–1998)
  - big issue: portability across machines was difficult (if not impossible)
  - one typically ended with a different version of the parallel code for each different machine vendor !!!
- early work on a standard started in 1992 at Oak Ridge National Lab and Rice Uni
- at that stage, there was a plethora of different message passing environments
- target was C and FORTRAN applications
- MPI-1 released in May 94 (over 40 academic and government participants)
  - contains: point-to-point communications, collective operations, process topologies
- minor clarifications: MPI 1.1 (June 95), MPI 1.2 (July 97)

What is MPI?

The Message Passing Interface (MPI) is a standardized specification of a set of library subroutines for the portable and flexible development of efficient message-passing parallel programs

- MPI Forum in charge of standardization (40 participating organizations, including vendors, researchers, software library developers, and users)
- revised several times, with the most recent being MPI-4. Actual implementations differ in the version/features of the standard they support
- supported on virtually all HPC platforms. Several free (e.g., OpenMPI, MPICH) and commercial implementations (Intel MPI) available
- provides FORTRAN, C (this course), and C++ bindings
- very broad standard with a huge # of library subroutines (over 440 in MPI-3). Fortunately, most applications merely require less than a dozen of those
- documentation for all versions of the MPI standard available here

How does MPI work?

MPI conforms with the following rules:

- Single Program Multiple Data (SPMD) model: the same program runs on all processes. All processes taking part in a parallel calculation can be distinguished by a unique identifier called rank
- The program is written in a sequential language like Fortran, C, or C++. Data exchange is carried out via calls to MPI library subroutines
- All variables in a process are local to this process. There is no concept of shared-memory

“Hello world” MPI program (I)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  int np, me, ierr;
  ierr=MPI_Init(&argc, &argv);
  ierr=MPI_Comm_size(MPI_COMM_WORLD, &np);
  ierr=MPI_Comm_rank(MPI_COMM_WORLD, &me);
  printf("Hello world I am %d out of %d\n", me, np);
  ierr=MPI_Finalize();
}
```

- All MPI calls return an error code (here `ierr`) which tells the user program whether MPI operation succeeded or not (`MPI_SUCCESS` means no error)
- `MPI_Init` initializes parallel environment. MUST precede any other MPI library call
- Upon initialization, MPI sets up the world communicator (`MPI_COMM_WORLD`)
  - → A communicator defines a group of processes referred to by a handler
  - → `MPI_COMM_WORLD` handler describes all processes started with parallel program
  - → If required, other communicators can be defined as subsets of `MPI_COMM_WORLD`
  - → Almost all MPI calls require a communicator handler as an argument

“Hello world” MPI program (II)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  int np, me, ierr;
  ierr=MPI_Init(&argc, &argv);
  ierr=MPI_Comm_size(MPI_COMM_WORLD, &np);
  ierr=MPI_Comm_rank(MPI_COMM_WORLD, &me);
  printf("Hello World, I am %d out of %d\n", me, np);
  ierr=MPI_Finalize();
}
```

- The calls to `MPI_Comm_size` and `MPI_Comm_rank` determine the number of processes running the parallel code, and the unique identifier (called *rank*) of the calling process, respectively

  → The ranks in a communicator are consecutive, starting from zero
- The call to `MPI_Finalize` shuts down the parallel program

  → No process except 0 is guaranteed to execute any code after `MPI_Finalize`

Code compilation and execution

The way MPI programs are compiled and started is NOT fixed by the standard

- Compiler and linker need special options that specify where modules and libraries, resp., can be found. Considerable variation in those locations among installations
- Most MPI implementations provide compiler wrapper scripts (e.g., `mpicc`) that automatically supply the required options to the underlying native compiler
- Typically a script called `mpirun` is provided to start a message-passing program
  → Processor cores may have to be allocated from batch system in advance
  → How exactly processes are created is entirely up to the implementation
  → Typically `mpirun` uses the batch system's infrastructure to launch processes
- For our example, a "common" implementation may require the following steps:

```
$ mpicc -O3  hello.c -o hello
$ mpirun -np 4 ./hello
Hello World, I am 0 out of 4
Hello World, I am 2 out of 4
Hello World, I am 1 out of 4
Hello World, I am 3 out of 4
```

MPI messages

- A MPI message is defined as a 1D array of elements of a particular MPI data type
- MPI data types can be either basic (see table below) or derived

MPI data type	**C data type**
`MPI_CHAR`	`char`
`MPI_INT`	`int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_BYTE`	`unsigned char`
⋮	⋮

- MPI derived types created by calling appropriate MPI calls (later in the lecture)
- MPI needs to know the data type of messages as it supports heterogeneous environments where it may be necessary to perform on-the-fly data conversions
- MPI data types on sender and receiver MUST MATCH for messages to proceed

Point-to-point communication (I)

- data exchange that involves exactly one sender and one receiver
- both ends are identified uniquely by their ranks
- each message carries an extra integer, called tag, that MUST MATCH on both ends
- tag is programmer-defined and can be used to create classes of messages; may just be set to some constant value if not needed
- the basic (but not unique!) call to send data from one process to another is `MPI_Send`:

```
int MPI_Send(void *buf,                     // message buffer
             int count,                     // # of items
             MPI_Datatype datatype,         // MPI data type
             int dest,                      // destination rank
             int tag,                       // message tag
             MPI_Comm comm);                // MPI communicator handler
```

Point-to-point communication (II)

- the basic (but not unique!) call to receive a message is `MPI_Recv`:

```
int MPI_Recv(void *buf,                  // message buffer
             int count,                  // maximum # of items
             MPI_Datatype datatype,// MPI data type
             int source,                 // source rank
             int tag,                    // message tag
             MPI_Comm comm,              // MPI communicator handler
             MPI_Status *status);        // pointer to status object
```

- `status` is an output argument which may be used to guess parameters that have not been fixed by the `MPI_Recv` arguments. In particular:
  - Actual message size (`count` is only a maximum value at receiver side)
  - Sender's rank if receive not tailored to particular sender (source=`MPI_ANY_SOURCE`)
  - Message tag if receive not tailored to particular tag (tag=`MPI_ANY_TAG`)

Blocking semantics (crucial slide)

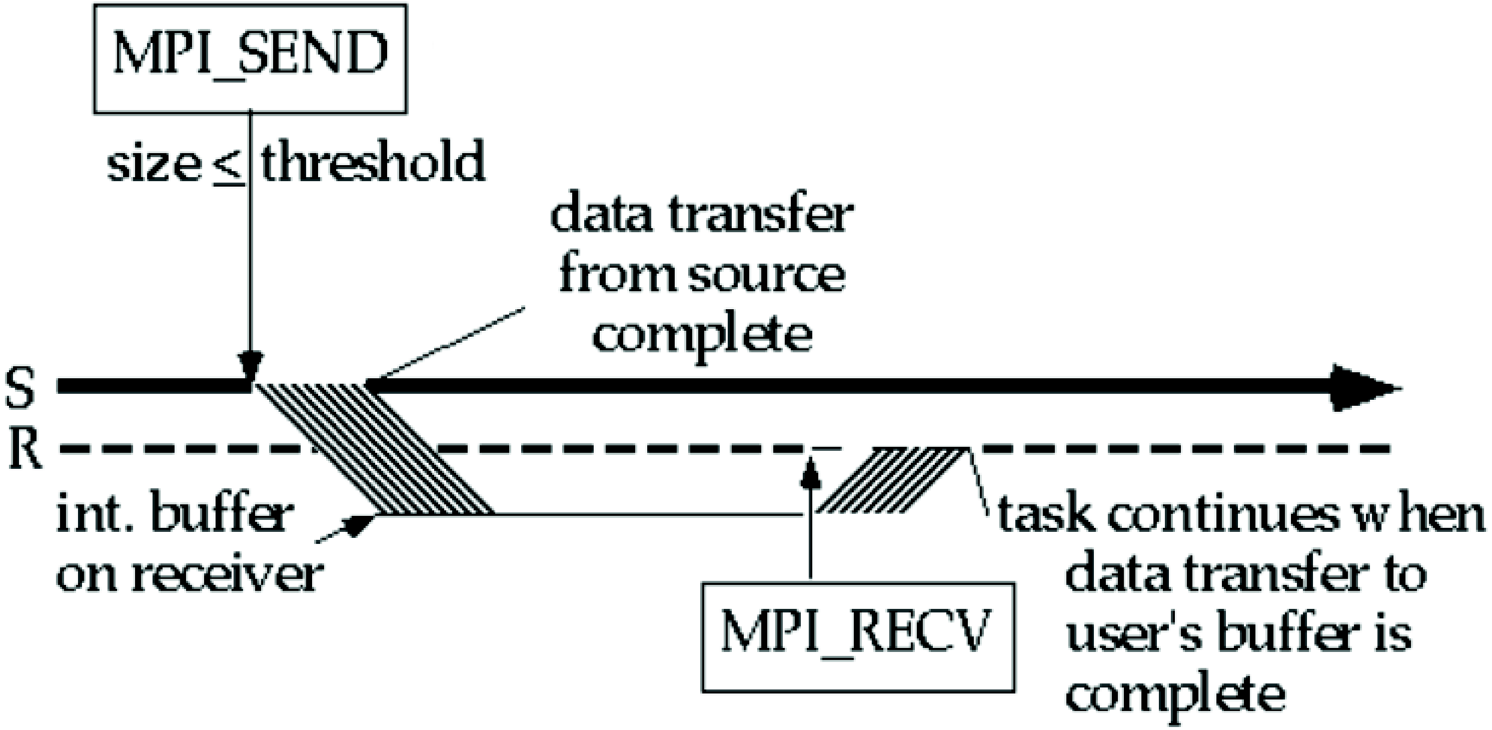
`MPI_Send` and `MPI_Recv` have blocking semantics, meaning that:

1. buffer can safely be written upon `MPI_Send` return without altering on-going comm
2. one can be sure that the message has been received upon `MPI_Recv` return

- this provides high freedom in the actual implementation of `MPI_Send`, i.e., it JUST specifies that it MUST comply with blocking semantics
- internally, it may work synchronously (e.g., it may return once message transfer has at least started after a handshake with the receiver process)
- however, it may also copy the message to an internal buffer and return immediately, allowing handshake and transmission progress to occur in the background
- it may even switch its behaviour depending on any explicit or hidden parameters
- e.g., most MPI implementations provide a (small) internal buffer for short messages, and switch to synchronous mode when internal buffer is full or too small
- this has to be taken into account when writing parallel programs to avoid so-called deadlocks

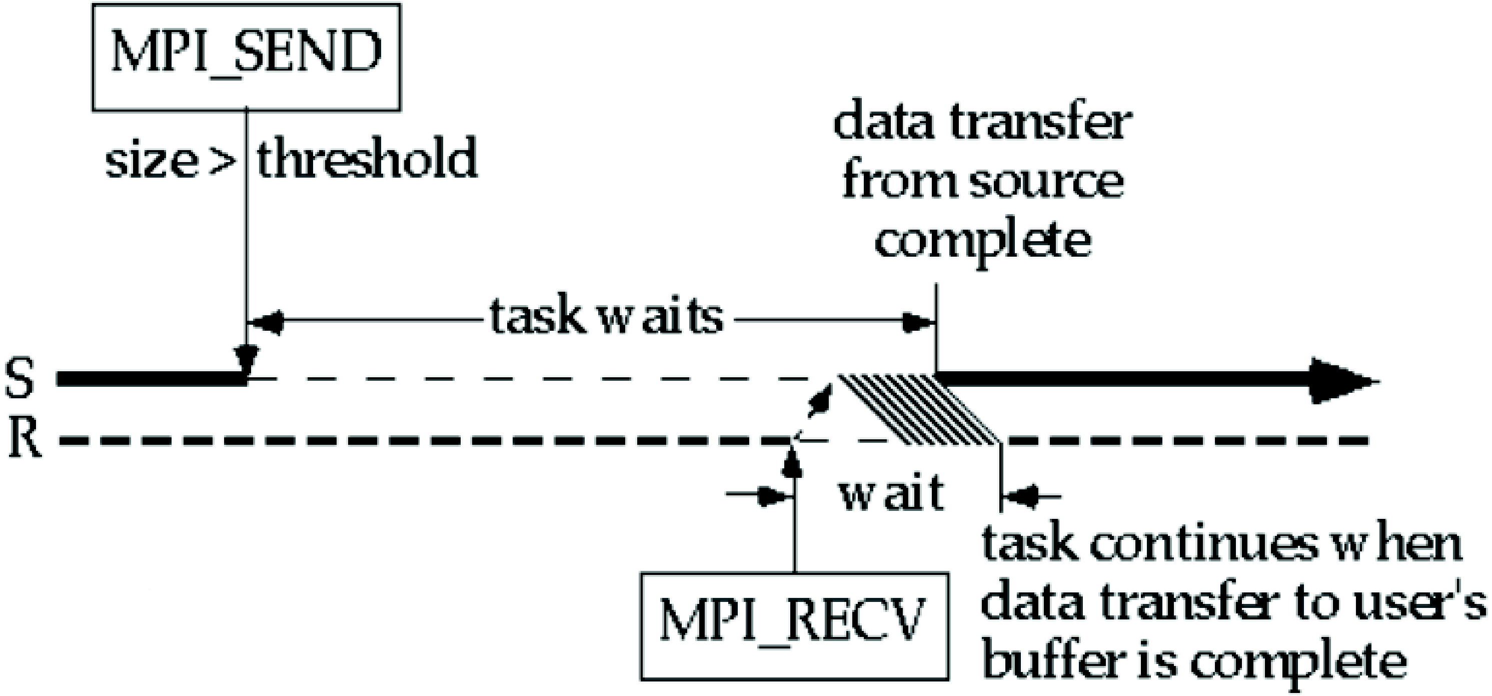
Possible implementation of `MPI_Send`

One possible implementation of `MPI_Send` with “small” message sizes



Another possible implementation of `MPI_Send`

Another possible implementation of `MPI_Send` with “large” message sizes



Let us think ...

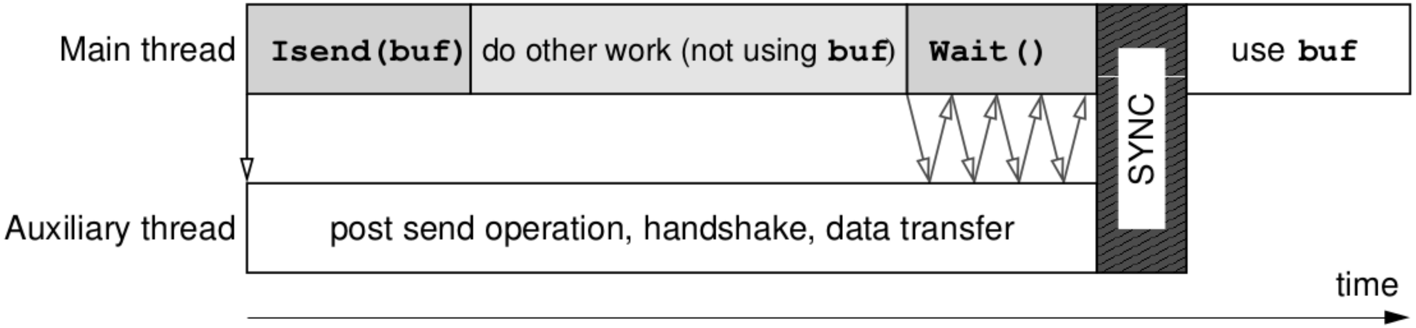
Consider the execution of the following MPI program on two processes, attempting to send each other's `a` array:

```
char a[N]; int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// code to initialize a goes here ...
MPI_Send(a, N, MPI_CHAR, 1-rank, 0, MPI_COMM_WORLD);
MPI_Recv(a, N, MPI_CHAR, 1-rank, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

Do you anticipate any issue with this MPI program? If yes, how would you solve it?

Non-blocking point-to-point communication (concept)

- MPI has support for non-blocking sends (`MPI_Isend`) and receives (`MPI_Irecv`)
- Merely initiate message transmission and return **very quickly** to the user code
- The message buffer must not be used as long as user code has not been notified that it is safe to do so
- **If MPI implemented efficiently**, sync and data transfer can occur in the **background**, leaving CPU free for useful computations (**comm/comp overlap**)



- Many non-blocking sends/receives can be pending at any time on a given process
- Non-blocking/blocking calls are **mutually compatible**
  → `MPI_Send` matches `MPI_Irecv`, `MPI_Isend` matches `MPI_Recv`, . . .

Non-blocking point-to-point communication (`Isend` and `Irecv`)

- `MPI_Isend` initiates a non-blocking send

```
int MPI_Isend(const void *buf,          // message buffer
              int count,                // # of items
              MPI_Datatype datatype,    // MPI data type
              int dest,                 // destination rank
              int tag,                  // message tag
              MPI_Comm comm,            // MPI communicator
              MPI_Request *request)     // request handle
```

- Compared to `MPI_Send`, and additional output argument, *request handle*
- Serves as an identifier to later refer to "pending" communication request
- Correspondingly, `MPI_Irecv` initiates a non-blocking receive

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

- Compared to `MPI_Recv`, no `status` provided as output
- No actual communication has taken place when the call returns to user code!

Non-blocking point-to-point communication (`Test` or `Wait`)

- Check a pending comm for completion can be done with `MPI_Test` or `MPI_Wait`:

```
int MPI_Test(MPI_Request *request, // pending request
             int *flag,            // true if request complete
             MPI_Status *status)   // status object

int MPI_Wait(MPI_Request *request, // pending request
             MPI_Status *status)   // status object
```

- `MPI_Test` tests for completion, returns true if buffer can be safely used
- `MPI_Wait` **blocks** until message buffer can be safely used
- `status` only contains useful information only if pending communication is a completed receive (i.e., `flag` must be true in case of `MPI_Test`)
- Checking multiple pending comms for completion can be done with `MPI_Waitall` (homework: to investigate this function)

MPI Collective Operations (brief coverage)

- barrier: synchronizes all members in a communicator
  - it should not be used in general, only for debugging or profiling purposes
- broadcast: send same message to many processors
  - must define processors in the group (specified by a communicator)
  - must define who sends and who receives information
  - has blocking semantics; may or may not synchronize processors (implementation dependent)

  e.g. `MPI_Bcast(A, n, MPI_DOUBLE, 0/*root*/, MPI_COMM_WORLD);`
- scatter: 1 process sends unique data to every other in group
- gather: reverse of above
- reduction: gather + an arithmetic/logical operation
  - result goes to just one process, or goes to all processes (`All` variants)

All the above can be constructed from simple sends and receives ... BUT MPI provides (usually highly optimized, underlying network tailored) calls to perform all of these. Use them!

MPI Derived Datatypes

- often, we want to send or receive m items of data with a stride $s > 1$ (e.g. a column in a row-major matrix)
- e.g. for double precision, if s represents the number of elements between the start of each block, we can create a datatype with an implicit stride:

```
MPI_Datatype sVec;
MPI_Type_vector(1 /*number of blocks*/, 1 /*block length*/, s,
                MPI_DOUBLE, &sVec);
MPI_Type_commit(&sVec);
...
MPI_Send(A, m, sVec, ...)
```

- alternatively, we could do:

```
MPI_Type_vector(m, 1, s, MPI_DOUBLE, &matCol);
MPI_Type_commit(&matCol);
..
MPI_Send(A, 1, matCol, ...);
```

- this allows MPI to handle the allocation, copying to/from and de-allocation of temporary buffers

A note of communicators

- MPI allows to create new communicators by duplicating or splitting other communicators (e.g., `MPI_COMM_WORLD`)
- using `MPI_COMM_WORLD` all the way through in MPI programs is in general dangerous, as there might be message mismatches among those that are internally generated by a library and those generated by the application program
- Solution: define a different communicator for user application program and library: