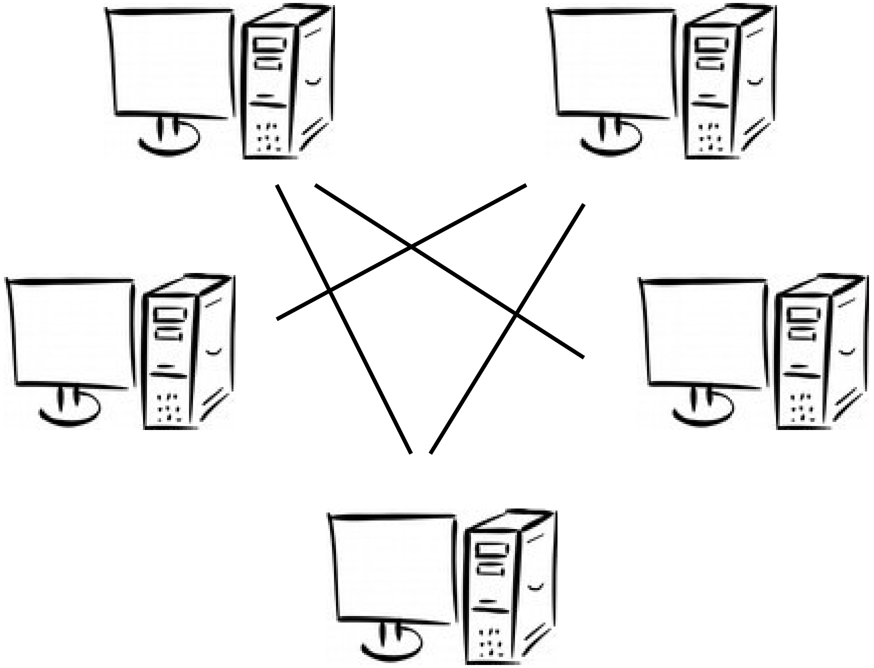
CS 470
Spring 2019

Mike Lam, Professor



Distributed Programming & MPI

MPI paradigm

- Single program, multiple data (SPMD)
  - One program, multiple processes (ranks)
  - Processes communicate via messages
    - An MPI *message* is a collection of fixed-size data elements
    - Underlying mechanism (e.g., sockets) is implementation-dependent
  - Multiple processes may run on the same node
    - They do NOT share an address space!
    - But intra-node communication will be faster than inter-node
  - Processes are grouped into communicators
    - May be in multiple communicators simultaneously
    - Default communicator: MPI_COMM_WORLD (all processes)

Message-Passing Interface (MPI)

- MPI is a standardized software library interface
  - Available online: `http://www.mpi-forum.org/docs/`
  - MPI-1 released in 1994 after Supercomputing ‘93
  - MPI-2 (1996) added one-sided operations and parallel I/O
  - MPI-3 (2012) improved non-blocking and one-sided operations
    - Also added tooling interface
  - Latest version (MPI-3.1) approved June 2015
  - Working groups currently designing MPI-4.0
- Several widely-used implementations
  - OpenMPI and MPICH (on our cluster)
  - MVAPICH / MVAPICH2 (higher performance)

MPI-3.1 support

Status of MPI-3.1 Implementations

	MPICH	MVAPICH	Open MPI	Cray MPI	Tianhe MPI	Intel MPI	IBM BG/Q MPI [1]	IBM PE MPICH [2]	IBM Platform	SGI MPI	Fujitsu MPI	MS MPI	MPC	NEC MPI
NBC	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	(*)	✔	✔
Nbrhood collectives	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✘	✔	✔
RMA	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✘	Q2'17	✔
Shared memory	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	*	✔
Tools Interface	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	*	Q4'16	✔
Comm-creat group	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✘	*	✔
F08 Bindings	✔	✔	✔	✔	✔	✔	✔	✘	✘	✔	✘	✘	Q2'16	✔
New Datatypes	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	✔	✔
Large Counts	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	Q2'16	✔
Matched Probe	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	Q2'16	✔
NBC I/O	✔	Q3'16	✔	✔	✘	✔	✘	✘	✘	✔	✘	✘	Q4'16	✔

Source: `http://mpi-forum.org/mpi31-impl-status-Jun16.pdf`

MPI development

- MPI is more than a library (unlike pthreads)
  - Compiler wrapper (`mpicc` / `mpiCC` / `mpif77`)
    - Still need to `#include <mpi.h>`
  - Program launcher (`mpirun`)
  - Job management integration (`salloc` / `sbatch`)
    - **Do not use `srun`!**
    - SLURM *tasks* = MPI *ranks* / *processes*
- System admins use modules to ease setup
  - Command: `module load mpi`
  - Populates your shell environment w/ MPI paths
    - To use MPICH (needed for P4): `module load mpi/mpich-3.2.1`

MPI conventions

- Identifiers start with "`MPI_`"
  - Also, first letter following underscore is uppercase
- MPI must be initialized and cleaned up
  - `MPI_Init` and `MPI_Finalize`
  - For `MPI_Init`, you can "pass through" `argc` and `argv`
  - No MPI calls before `MPI_Init` or after `MPI_Finalize`!
- Task parallelism is based on rank / process ID
  - `MPI_Comm_rank` and `MPI_Comm_size`
  - Often rank 0 is considered to be special (the "master" process)
- I/O is asymmetrical
  - All ranks may write to `stdout` (or `stderr`)
  - Usually, only rank 0 can read `stdin`

Basic MPI functions

```
int MPI_Init (int *argc, char ***argv)

int MPI_Finalize ()


int MPI_Comm_size (MPI_Comm comm, int *size)

int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

Default communicator: `MPI_COMM_WORLD`

Pointer to **out** parameter

```
double MPI_Wtime ()

int MPI_Barrier (MPI_Comm comm)
```

MPI Hello World

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

int main(int argc, char **argv)
{
    int mpi_rank;
    int mpi_size;

    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

    printf("Hello from process %2d / %d!\n", mpi_rank+1, mpi_size);

    MPI_Finalize();

    return 0;
}
```

MPI “Hello world” example

- Copy `/shared/cs470/mpi-hello` to your home folder
- Build with “`make`”
- Run locally (don’t do this normally!)
  - `mpirun ./hello`
- Run on cluster
  - `salloc mpirun ./hello`
  - `salloc -n 4  mpirun ./hello`
  - `salloc -n 16 mpirun ./hello`
  - `salloc -N 4  mpirun ./hello`

Point-to-point messages

- MPI an **explicit** message-passing paradigm
  - You (the developer) decide how to split up data
  - You manage memory allocation manually
  - You decide how to send data between processes
  - Most direct mechanism: point-to-point messages

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

dest / src: must correspond

tag: must match

count: recv count must be equal to or higher than send count

dtype: must match

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

MPI datatypes

C data type	MPI data type
`char` `unsigned char`	`MPI_CHAR` `MPI_UNSIGNED_CHAR`
`int` `unsigned`	`MPI_INT` `MPI_UNSIGNED`
`long` `unsigned long`	`MPI_LONG` `MPI_UNSIGNED_LONG`
`long long`	`MPI_LONG_LONG`
`float`	`MPI_FLOAT`
`double`	`MPI_DOUBLE`

Generic receiving

- All parameters are required for `MPI_Send`
- `MPI_Recv` allows for some ambiguity
  - `count` is the *maximum* count (actual could be lower)
  - `src` can be `MPI_ANY_SOURCE` and `tag` can be `MPI_ANY_TAG`
- The `status` parameter provides this info
  - Pointer to `MPI_Status` struct that is populated by `MPI_Recv`
  - After receive, access members `MPI_SOURCE` and `MPI_TAG`
  - Use `MPI_Get_count` to calculate true count
  - If you don't need any of these, pass `MPI_IGNORE_STATUS`

Postel's Law: "Be *conservative in what you do;*
be liberal in what you accept from others."

MPI Send/Receive Example

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

int main(int argc, char *argv[])
{
    int my_rank;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) {

        // master process: receive a single integer from any source
        int data = -1;
        MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Received data in rank %d: %d\n", my_rank, data);

    } else {

        // other processes: send our rank to the master
        MPI_Send(&my_rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

    }

    MPI_Finalize();
    return 0;
}
```

Blocking and safety

- Exact blocking behavior is implementation-dependent
  - `MPI_Send` **may** block until the message is sent
    - Sometimes depends on the size of the message
    - `MPI_Ssend` will **always** block until the message is received
  - `MPI_Recv` will **always** block until the message is received
- A program is unsafe if it relies on MPI-provided buffering
  - You can use `MPI_Ssend` to check your code (forces blocking)
  - Use `MPI_SendRecv` if both sending and receiving
  - Or use `MPI_Isend` / `MPI_Recv` pairs

```
int MPI_Sendrecv (void *send_buf, int send_count, MPI_Datatype send_dtype, int dest, int send_tag
                  void *recv_buf, int recv_count, MPI_Datatype recv_dtype, int src,  int recv_tag,
                  MPI_Comm comm, MPI_Status *status)
```

Non-blocking send/receive

- Some operations are guaranteed not to block
  - Point-to-point: `MPI_Isend` and `MPI_Irecv`
  - Includes some collectives (in MPI-3)
- These operations merely “request” some communication
  - `MPI_Request` variables can be used to track these requests
  - `MPI_Wait` blocks until an operation has finished
  - `MPI_Test` sets a flag if the operation has finished

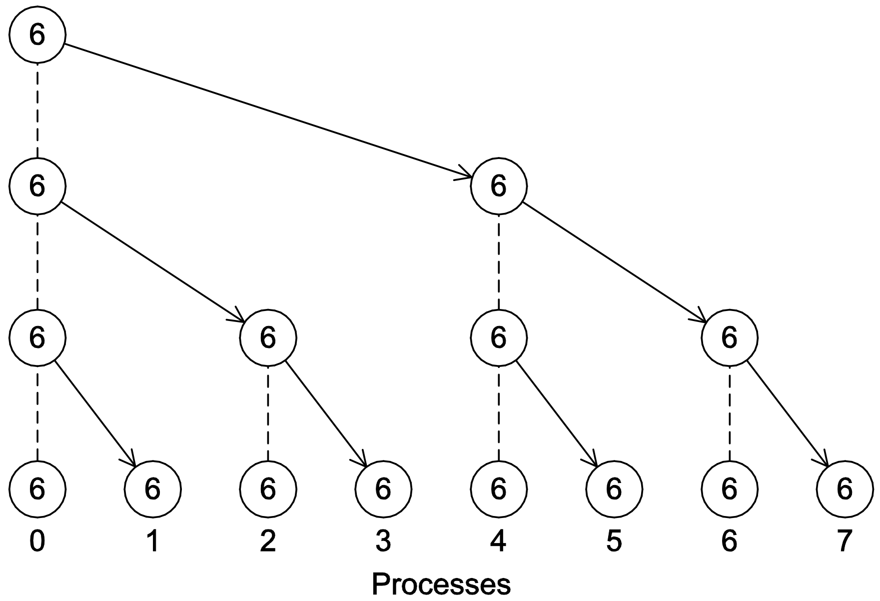
```
int MPI_Isend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv (void *buf, int count, MPI_Datatype dtype, int src,  int tag, MPI_Comm comm, MPI_Request *request,
               MPI_Status *status)

int MPI_Wait (MPI_Request *request,            MPI_Status *status)
int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
```

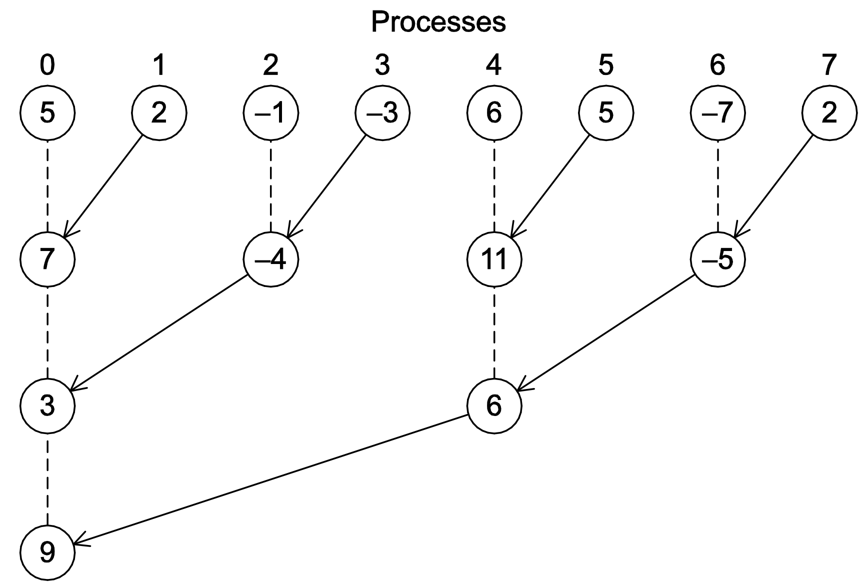
Issues with point-to-point

- No global message order guarantees
  - Between any send/recv pair, messages are nonovertaking
    - If p_1 sends m_1 then m_2 to p_2, then p_2 must receive m_1 first
  - No guarantees about global ordering
  - Communication between **all** processes can be tricky
- Process 0 reads input, distributes data, and collects results
  - Using point-to-point operations does not scale well
  - Need a more efficient method

- Collective operations provide *correct* and *efficient* built-in all-process communication

Tree-structured communication

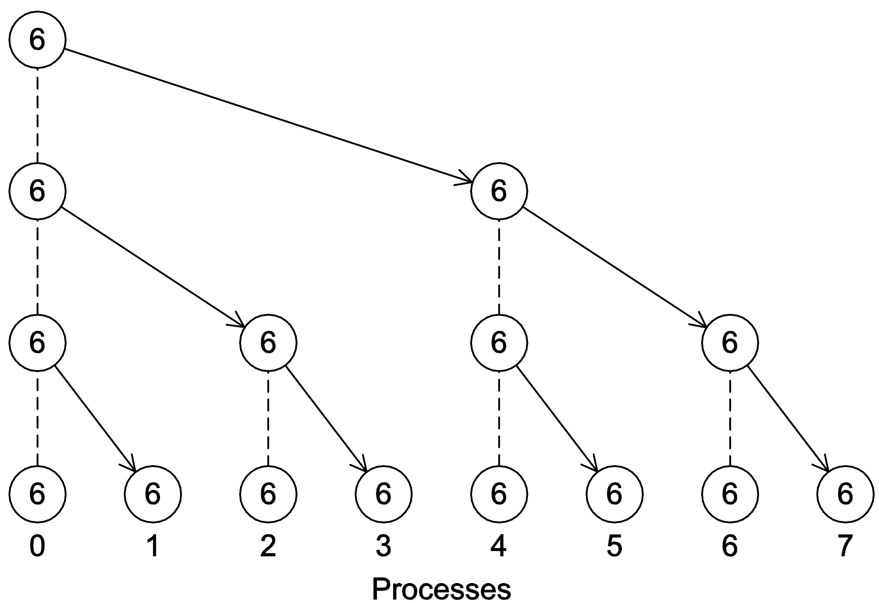




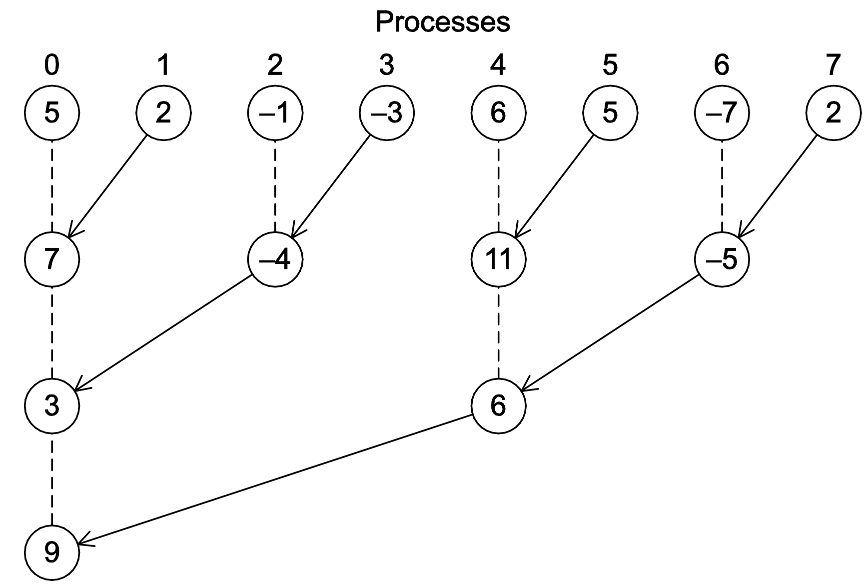
Broadcast **Reduction**

```
int MPI_Bcast     (void       *buf,                    int count,
                   MPI_Datatype dtype,                  int root, MPI_Comm comm)

int MPI_Reduce    (void *send_buf, void *recv_buf, int count,
                   MPI_Datatype dtype, MPI_Op op,  int root, MPI_Comm comm)
```

Tree-structured communication



Broadcast



Reduction

```
int MPI_Bcast     (void        *buf,                          int count,
                   MPI_Datatype dtype,                        int root, MPI_Comm comm)

int MPI_Reduce    (void *send_buf, void *recv_buf,           int count,
                   MPI_Datatype dtype, MPI_Op op,             int root, MPI_Comm comm)
```

cannot be aliases

usually rank 0

MPI Broadcast Example

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

int main(int argc, char *argv[])
{
    int my_rank;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // send rank id from process 0 to all processes
    int data = my_rank;
    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Received data in rank %d: %d\n", my_rank, data);

    MPI_Finalize();
    return 0;
}
```

Collective reductions

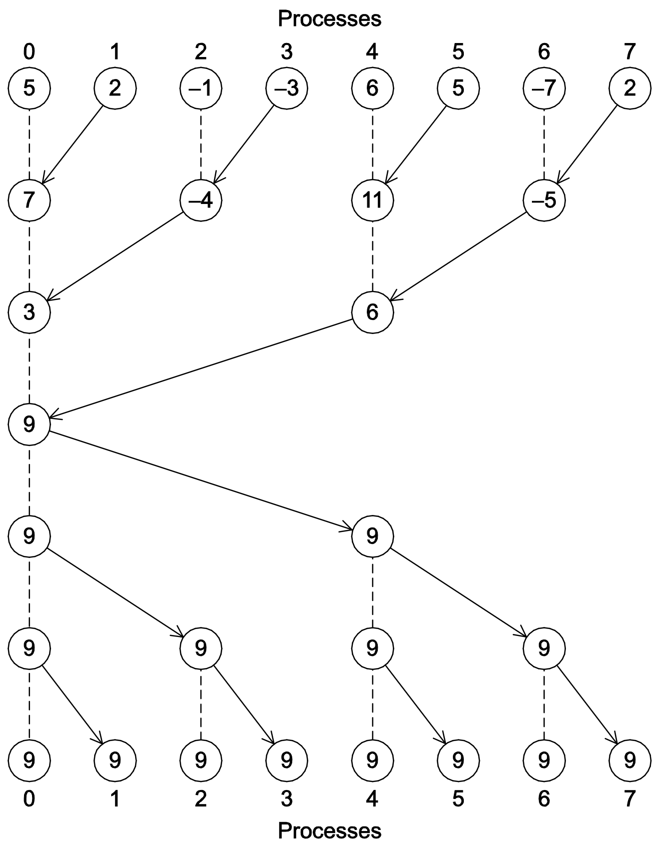
- Reduction operations
  - `MPI_SUM`, `MPI_PROD`, `MPI_MIN`, `MPI_MAX`
- Collective operations are matched based on ordering
  - Not on source / dest or tag
  - Try to keep code paths as simple as possible

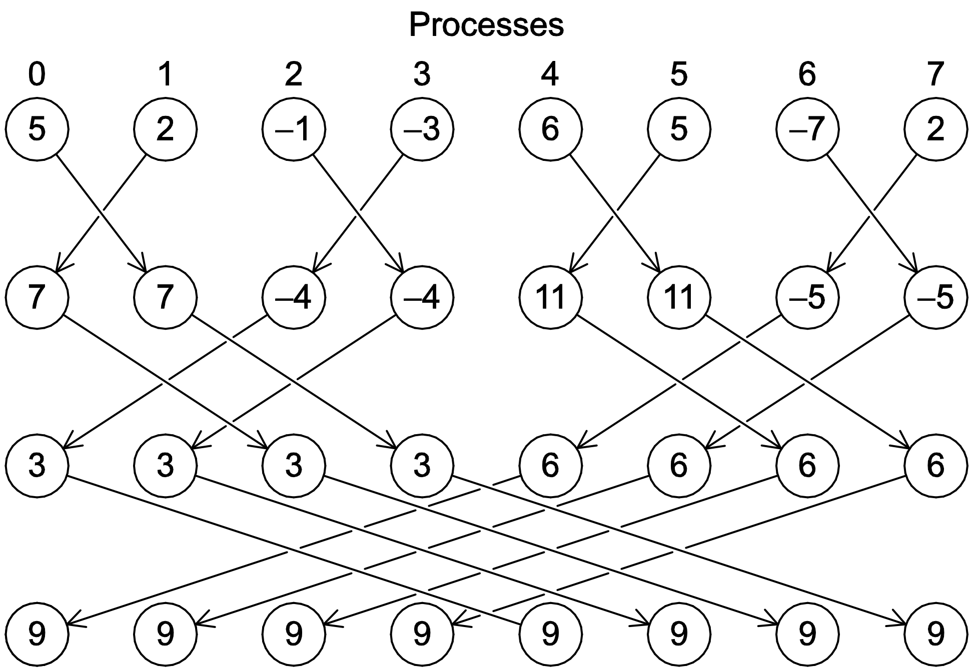
Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

NOTE: Reductions with count > 1 operate on a per-element basis

MPI_Allreduce

- Combination of `MPI_Reduce` and `MPI_Broadcast`
  - “Butterfly” communication pattern





Data distribution

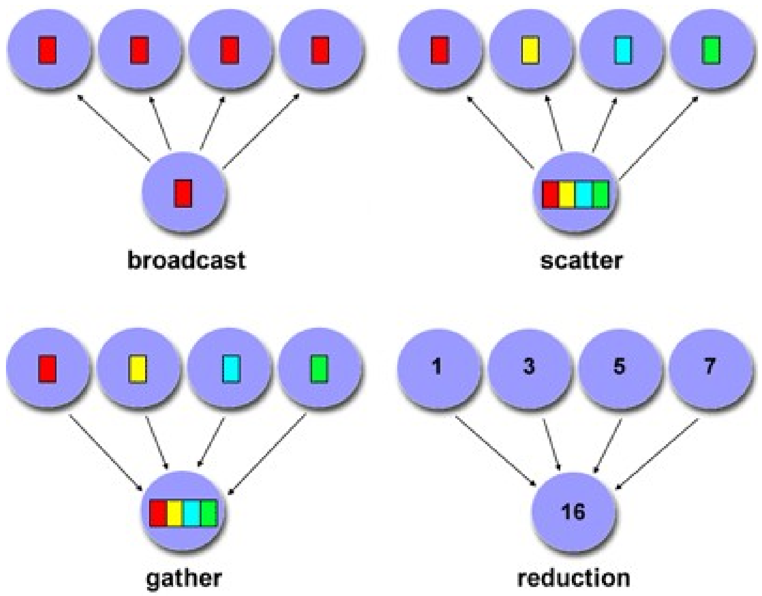
- `MPI_Scatter` and `MPI_Gather`
  - `MPI_Allgather` (gather + broadcast)
  - Provides efficient data movement in common patterns
  - Send and receive buffers must be different (or use `MPI_IN_PLACE`)
- Partitioning: block vs. cyclic
  - Usually application-dependent
  - Block is the default; use `MPI_Type_vector` for cyclic or block-cyclic

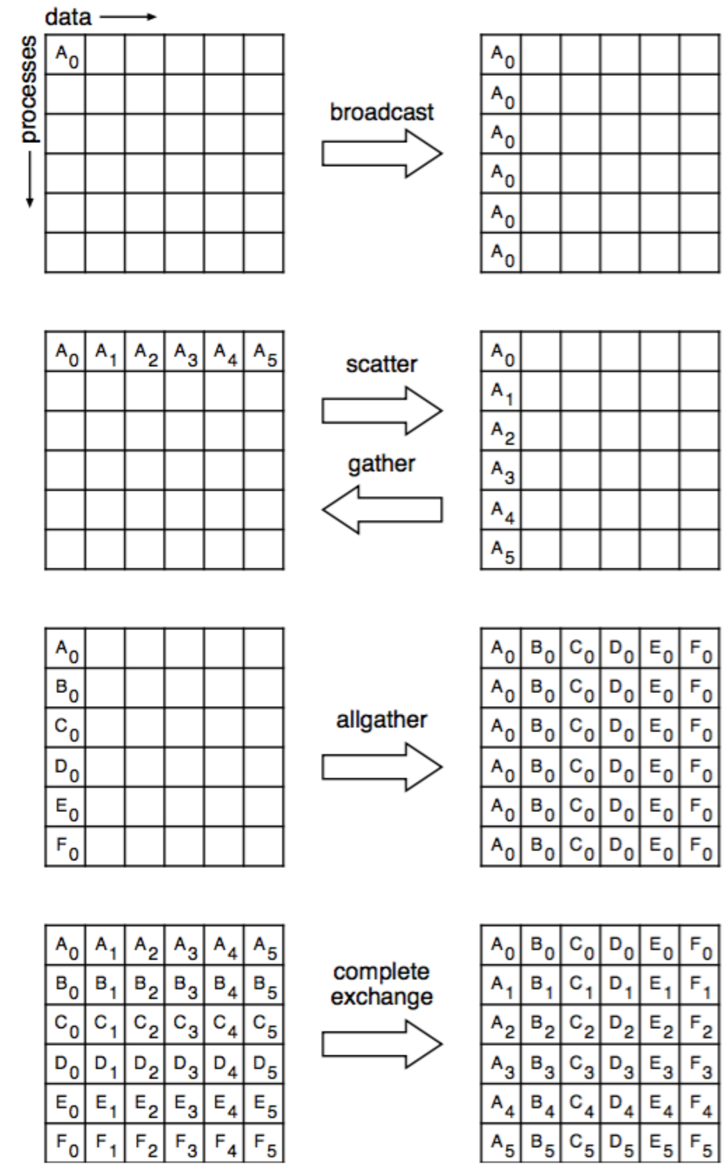
Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

MPI Gather Example

```
int main(int argc, char *argv[])
{
    int my_rank, num_ranks;
    int data[MAX_SIZE];

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);

    // initialize 'data' to dummy values
    for (int i = 0; i < num_ranks; i++) {
        data[i] = -1;
    }

    // send rank id from every process to process 0
    MPI_Gather(&my_rank, 1, MPI_INT,
                   data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // print 'data' at process 0
    if (my_rank == 0) {
        printf("Received data in rank %d: ", my_rank);
        for (int i = 0; i < num_ranks; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

MPI collective summary





MPI_Bcast()	Broadcast (one to all)
MPI_Reduce()	Reduction (all to one)
MPI_Allreduce()	Reduction (all to all)

MPI_Scatter()	Distribute data (one to all)
MPI_Gather()	Collect data (all to one)
MPI_Alltoall()	Distribute data (all to all)
MPI_Allgather()	Collect data (all to all)

*(these four include "*v" variants for variable-sized data)*

MPI reference (PDF on website)

General

```
struct MPI_STATUS {
  int MPI_SOURCE
  int MPI_TAG
  int MPI_ERROR
}
```

```
int MPI_Init (int *argc, char ***argv)        int MPI_Comm_size (MPI_Comm comm, int *size)
int MPI_Finalize ()                           int MPI_Comm_rank (MPI_Comm comm, int *rank)
int MPI_Barrier (MPI_Comm comm)                     Default communicator: MPI_COMM_WORLD
double MPI_Wtime ()
```

Point-to-point Operations

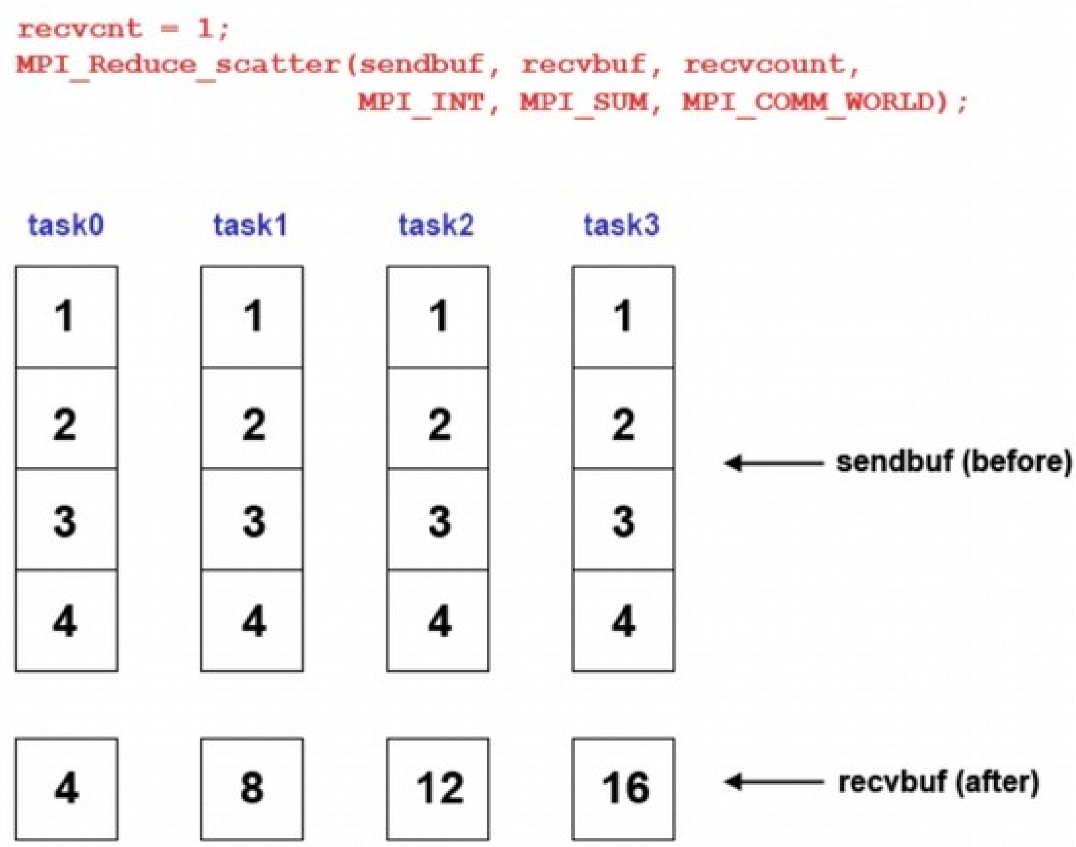
```
int MPI_Send  (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
int MPI_Ssend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
int MPI_Recv  (void *buf, int count, MPI_Datatype dtype, int src,  int tag, MPI_Comm comm, MPI_Status *status)
                    (maximum count)                 (MPI_ANY_SOURCE / MPI_ANY_TAG)              (MPI_STATUS_IGNORE)

int MPI_Sendrecv (void *send_buf, int send_count, MPI_Datatype send_dtype, int dest, int send_tag
                  void *recv_buf, int recv_count, MPI_Datatype recv_dtype, int src,  int recv_tag,
                  MPI_Comm comm, MPI_Status *status)

int MPI_Isend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv (void *buf, int count, MPI_Datatype dtype, int src,  int tag, MPI_Comm comm, MPI_Request *request,
               MPI_Status *status)

int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait (MPI_Request *request,            MPI_Status *status)
int MPI_Get_count (MPI_Status *status, MPI_Datatype dtype, int *count)
```

Collective Operations

```
int MPI_Bcast     (void      *buf,                  int count, MPI_Datatype dtype,            int root, MPI_Comm comm)

int MPI_Reduce    (void *send_buf, void *recv_buf, int count, MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm)
int MPI_Allreduce (void *send_buf, void *recv_buf, int count, MPI_Datatype dtype, MPI_Op op,           MPI_Comm comm)

int MPI_Scatter   (void *send_buf,            int send_count, MPI_Datatype send_dtype,
                   void *recv_buf,            int recv_count, MPI_Datatype recv_dtype,      int root, MPI_Comm comm)

int MPI_Gather    (void *send_buf,            int send_count, MPI_Datatype send_dtype,
                   void *recv_buf,            int recv_count, MPI_Datatype recv_dtype,      int root, MPI_Comm comm)

int MPI_Allgather (void *send_buf,            int send_count, MPI_Datatype send_dtype,
                   void *recv_buf,            int recv_count, MPI_Datatype recv_dtype,                MPI_Comm comm)

int MPI_Alltoall  (void *send_buf,            int send_count, MPI_Datatype send_dtype,
                   void *recv_buf,            int recv_count, MPI_Datatype recv_dtype,                MPI_Comm comm)
```

More collectives

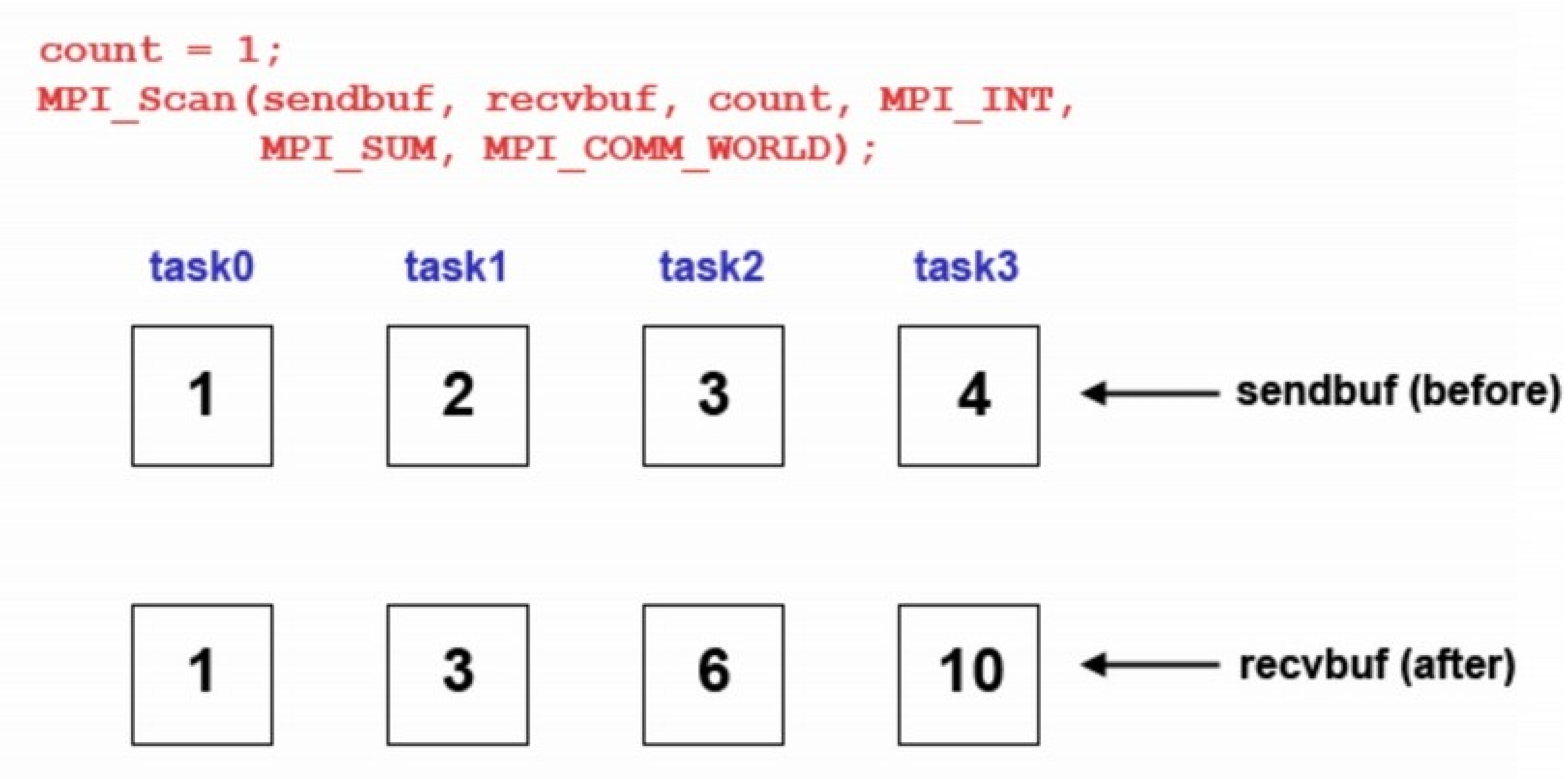
- `MPI_Reduce_scatter`
  - Reduce on a vector, then distribute result



https://computing.llnl.gov/tutorials/mpi/

More collectives

- `MPI_Scan`
  - Compute partial reductions



https://computing.llnl.gov/tutorials/mpi/

MPI datatypes

- MPI provides basic datatypes
  - `MPI_INT`, `MPI_LONG`, `MPI_CHAR`, etc.
- MPI also provides ways to create new datatypes
  - `MPI_Type_contiguous`: simple arrays

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

  - `MPI_Type_vector`: blocked and strided arrays
    - Useful for cyclic or block-cyclic data distributions

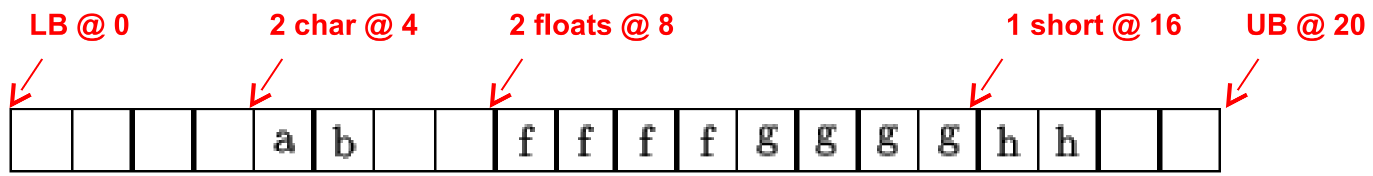
```
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

  - Derived datatypes: records
  - New datatypes must be committed before they are used

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

Derived datatypes

- Goal: Pack related data together to reduce total messages
  - Very similar to C structs, but more detailed
  - Allows MPI to optimize internal representations




```
MPI_Type_create_struct(5, array_of_block_lengths,
                          array_of_displacements,
                          array_of_types,
                          &new_type)

array_of_block_lengths = (1, 2, 2,  1,  1)
array_of_displacements = (0, 4, 8, 16, 20)
array_of_types = (MPI_LB, MPI_CHAR, MPI_FLOAT, MPI_SHORT, MPI_UB)
```

Virtual topologies

- It is often convenient for MPI to be aware of data decomposition details
- MPI provides built-in Cartesian system support.
  - `MPI_Dims_create()`
  - `MPI_Cart_create()`
  - `MPI_Cart_get()`
  - `MPI_Cart_coords()`
  - `MPI_Cart_shift()`

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Parallel file I/O (MPI-2)

- MPI provides a parallel file I/O interface
  - Uses derived data types to create per-process views of a file on disk
  - `MPI_File_open()`
  - `MPI_File_set_view()`
  - `MPI_File_read_at()`
  - `MPI_File_read()`
  - `MPI_File_read_shared()`
  - `MPI_File_write_at()`
  - `MPI_File_write()`
  - `MPI_File_write_shared()`
  - `MPI_File_close()`

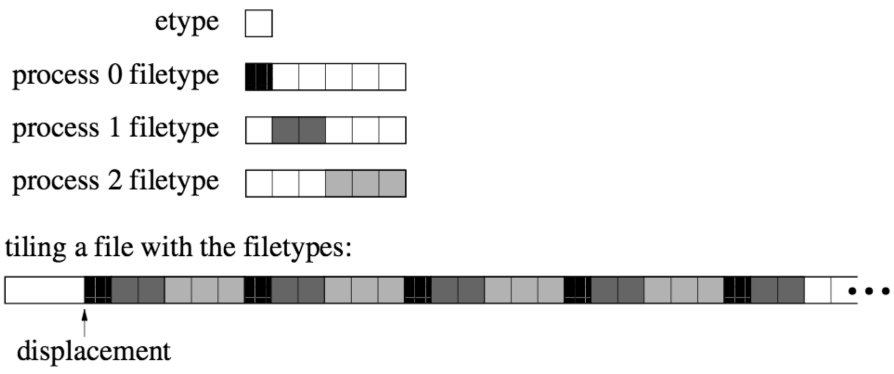


Figure 13.2: Partitioning a file among parallel processes

One-sided communication (MPI-2)

- MPI provides remote memory access (RMA)
  - This allows programmers to take advantage of hardware-specific direct memory access features like DMA
  - `MPI_Win_create()`
  - `MPI_Win_allocate()`
  - `MPI_Put()`
  - `MPI_Get()`
  - `MPI_Accumulate()`
  - `MPI_Win_free()`

Non-blocking collectives (MPI-3)

- MPI now provides non-blocking forms of major collective operations
- Like `MPI_Irecv()`, these calls begin the communication and should be concluded with a call to `MPI_Wait()`
  - `MPI_Ibarrier()`
  - `MPI_Ibcast()`
  - `MPI_Igather()`
  - `MPI_Iscatter()`
  - `MPI_Iallgather()`
  - `MPI_Ialltoall()`
  - `MPI_Ireduce()`
  - `MPI_Iallreduce()`
  - `MPI_Ireduce_scatter()`
  - `MPI_Iscan()`

Why MPI_Ibarrier?

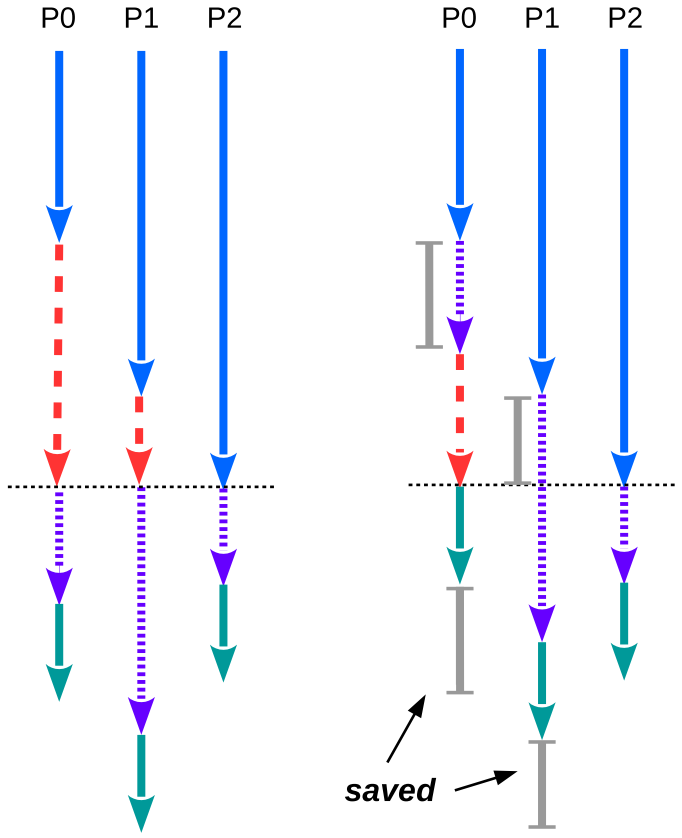
- Why would you want a *non-blocking* barrier?

```
work1();
MPI_Barrier(MPI_COMM_WORLD);
work2();         // independent
work3();         // dependent on work1()
```

Version 1

```
work1();
MPI_Request rq;
MPI_Ibarrier(MPI_COMM_WORLD, &rq);
work2();         // independent
MPI_Wait(&rq, MPI_STATUS_IGNORE);
work3();         // dependent on work1()
```

Version 2



Version 1 **Version 2**

Tools interface (MPI-3)

- MPI now provides a way to tweak parameters and access monitoring information in a cross-platform manner
- Control variables (`cvar`)
  - Startup options
  - Buffer sizes
- Performance variables (`pvar`)
  - Packets sent
  - Time spent blocking
  - Memory allocated

```
MPI_T_cvar_get_info()
MPI_T_cvar_handle_alloc()
MPI_T_cvar_read()
MPI_T_cvar_write()
```

```
MPI_T_pvar_get_info()
MPI_T_pvar_session_create()
MPI_T_pvar_start() / stop()
MPI_T_pvar_handle_alloc()
MPI_T_pvar_read()
MPI_T_pvar_reset()
```

Distributed memory summary

- Distributed systems can scale massively
  - Hundreds or thousands of nodes, petabytes of memory
  - Millions/billions of cores, petaflops of computation capacity
- They also have significant issues
  - Non-uniform memory access (NUMA) costs
  - Requires explicit data movement between nodes
  - More difficult debugging and optimization
- Core design tradeoff: data distribution
  - How to partition, and what to send where (duplication?)
  - Goal: minimize data movement
  - Paradigm: computation is “free” but communication is not