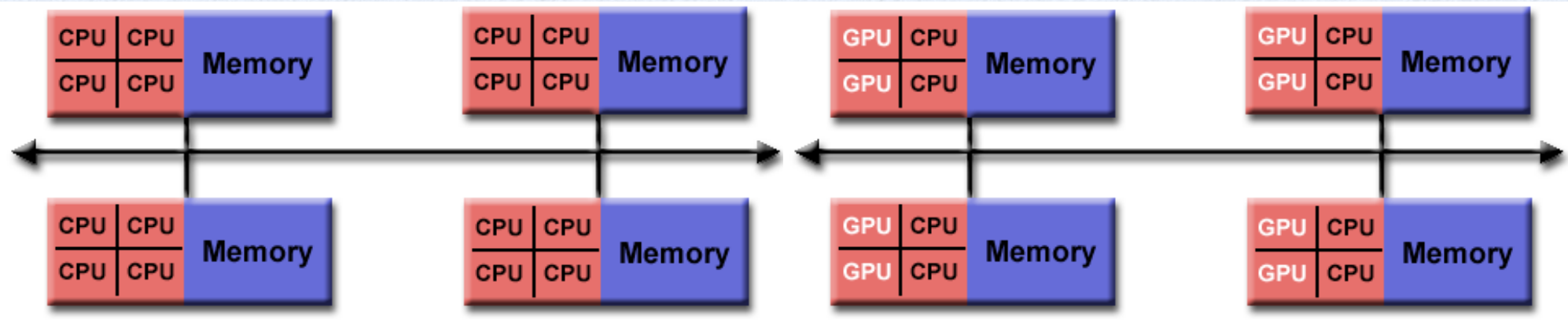
HPCSE II

Message Passing with MPI

hybrid distributed/shared memory

- To go beyond a single shared memory compute node we have to hook together many nodes with a network to form a hybrid architecture using shared and distributed memory.



Some of the fastest computers

#1 Tianhe-2 (China)
Intel Xeon + Xeon-Phi
3'120'000 cores
34 petaflop

10'000 - 100'000 nodes

#2 Titan (USA)
Cray XK-7
560'640 cores
17 petaflop

#3 Sequoia (USA)
IBM BlueGene/Q
98'304 nodes
1'572'864 cores
17 petaflop

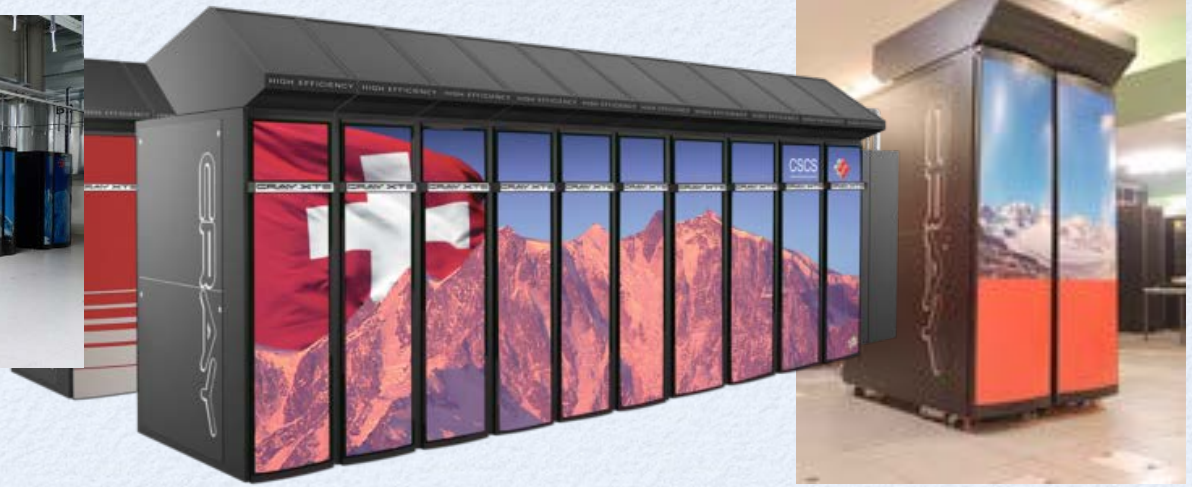
Machines at CSCS

- Switzerland has decided to not spend all the money on the fastest machines but to invest in software as well and develop the fastest software – that will then run all over the world.

#6 Cray XC30 "Piz Daint"
115,984 cores
6 petaflop
Aries interconnect

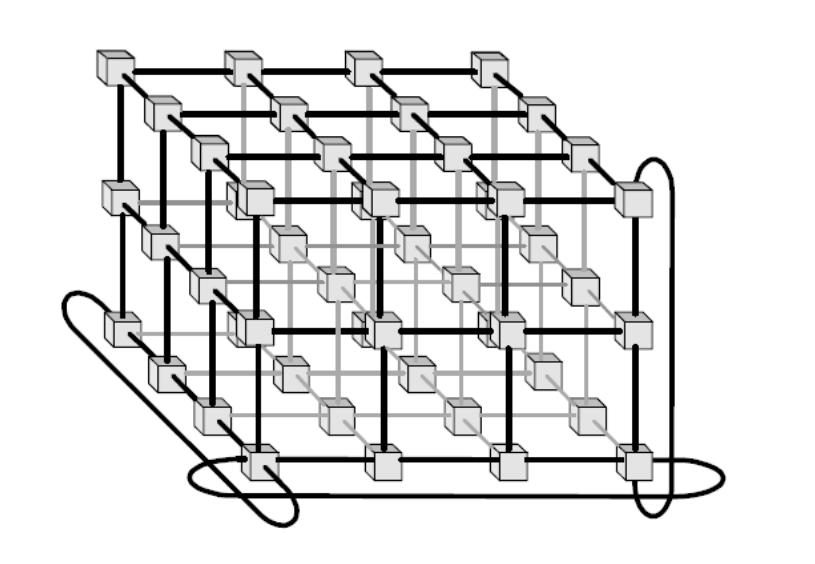
#112 Cray XE6 "Monte Rosa"
1'496 nodes
47'840 cores
316 teraflop
Gemini 3D torus interconnect

#139 Cray XK7 "Tödi"
272 nodes
8160 cores + 272 GPUs
273 teraflop
Gemini 3D torus interconnect



Network interconnects

- A 3D torus in the Cray Gemini interconnect
  - connections to six neighbors
  - latency about 1µs
  - 9.4 Gb/s bandwidth per link



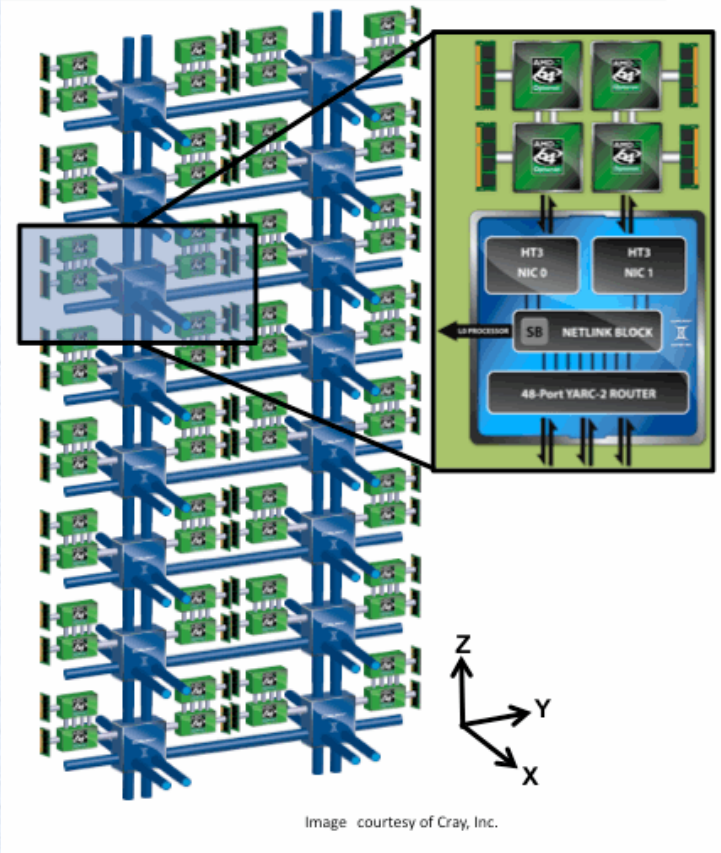
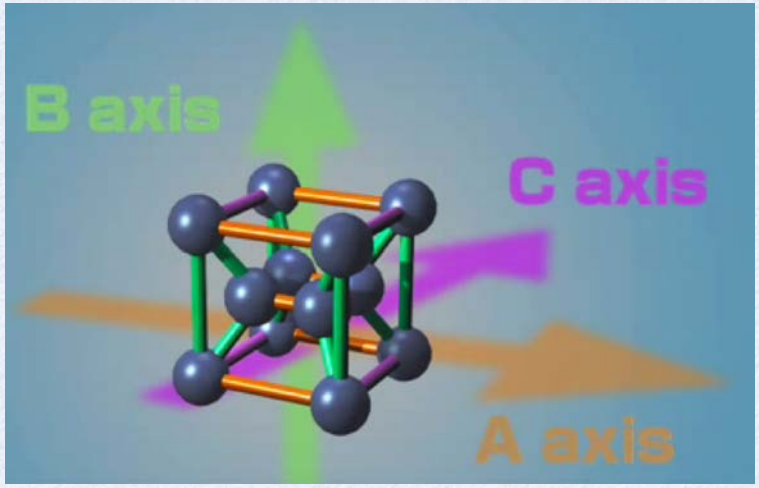


Image courtesy of Cray, Inc.

- IBM BlueGene/Q implements 5D torus
- K computer implements 6D torus

How does one create a 5/6-D torus?

- Space is only three-dimensional!
- The trick is using small extra dimensions:
  - **Cray Gemini:** L x W x H torus with one node at each site
  - **Fujitsu 6-D torus**: put a 2 x 3 x 2 "cube" at each site of a 3D torus. This gives a L x W x H x 2 x 3 x 2 => a 6-dimensional slab

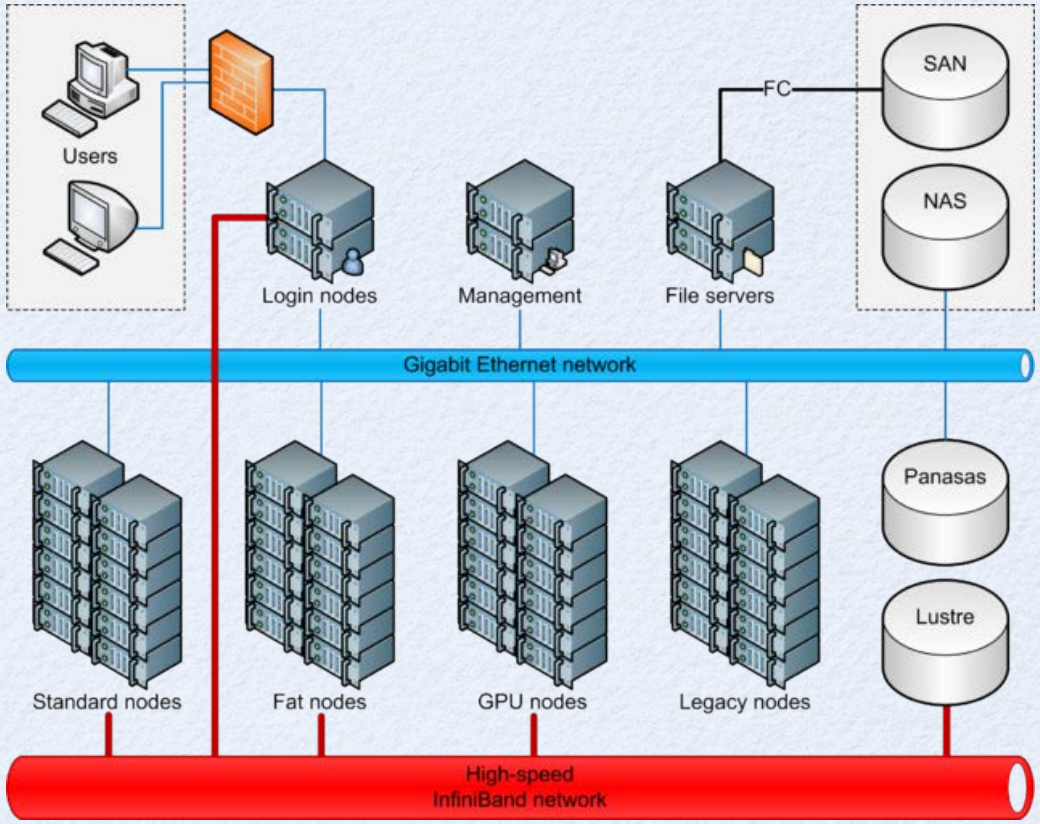


http://www.fujitsu.com/global/about/tech/k/whatis/network/

Beowulf clusters

- Clusters, such as Brutus, nowadays have similar nodes as the high-end machines but a cheaper network

983 nodes
18'400 cores + 48 GPUs
190 teraflop peak
Infiniband network



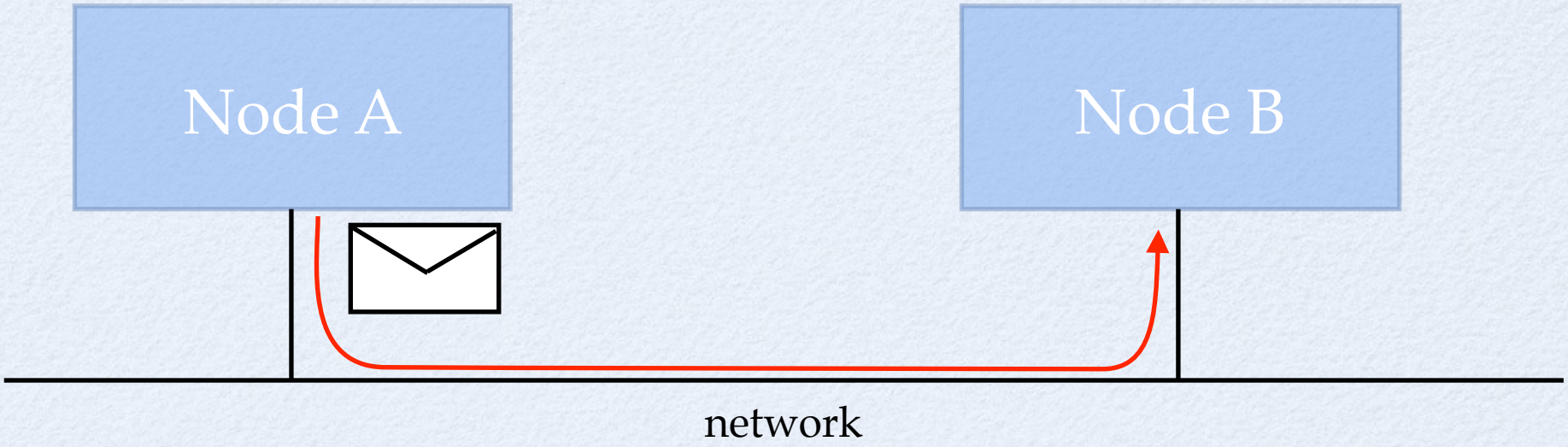
Distributed memory programming

- The number of processes is usually static, e.g. one process launched per core. The p processes are numbered by integer "ranks" 0 to p-1.
- All data is local to some processor, and in the protected memory space of a process. No race conditions!
- Access to the data of other processes needs to be explicitly managed by **message passing**.

- Disadvantages:
  - explicit management of communication is cumbersome
  - harder to program than OpenMP parallelization

- Advantages:
  - explicit manual management of communication allows optimization of the time-consuming communication
  - portable to many different types of machines

Message Passing



- Communication is done by sending messages between nodes.
- All you need to know to get started is how to send e-mails. Sending e-mails is “message passing”.
- The MPI (Message Passing Interface) standard is a standardized API provided by all vendors to implement message passing.

MPI

- MPI is the standard API for message passing libraries
  http://www.mpi-forum.org
- Goals of the MPI standard:
  - portable, efficient, easy to use
  - works on distributed memory, shared memory and hybrid systems
- Versions of the MPI standard:
  - **MPI-1** was first finished in 1992, minor updates over the years (1.1, 1.2, 1.3)
  - **MPI-2** was first proposed 1998 and adds one-sided communication, I/O, and creation of processes
  - **MPI-3** was finalized September 2012 and adds more features, in particular non-blocking collective communication
- We will cover mainly MPI-1 since that is what is needed for most codes

Obtaining MPI and compiling codes

- Install MPI
  - Most supercomputers MPI comes preinstalled.
  - On Brutus type '`module load open_mpi`'
  - Many Linux package managers provide an openmpi package
  - Otherwise get OpenMPI from http://www.open-mpi.org .
- Compiling MPI codes
  - You need to specify the right include path, library path, and libraries for MPI
  - Most MPI distributions come with a wrapper compiler that sets the paths and is typically called **mpicc**, **mpic++**, **mpicxx**, or **mpiCC** (only on systems with case-senstive file systems)
  - Most wrappers have options that inform the user of the compiler options added:

```
$ mpicc --showme:compile
-I/usr/local/include

$ mpicc --showme:link
-L/usr/local/lib -lmpi -lopen-rte -lopen-pal -lutil
```

The structure of an MPI program

- Include the header <mpi.h>
- You need to initialize and terminate the MPI environment in your code.
- Note that you need to pass pointers to argc and argv. The MPI environment might grab some command line options and return a modified list of options.

```
#include <mpi.h>

int main(int argc, char** argv) {

  MPI_Init(&argc, &argv); // initialize the environment

  ... // do something

  MPI_Finalize();         // clean up at the end

  return 0;
}
```

Initialization and termination functions

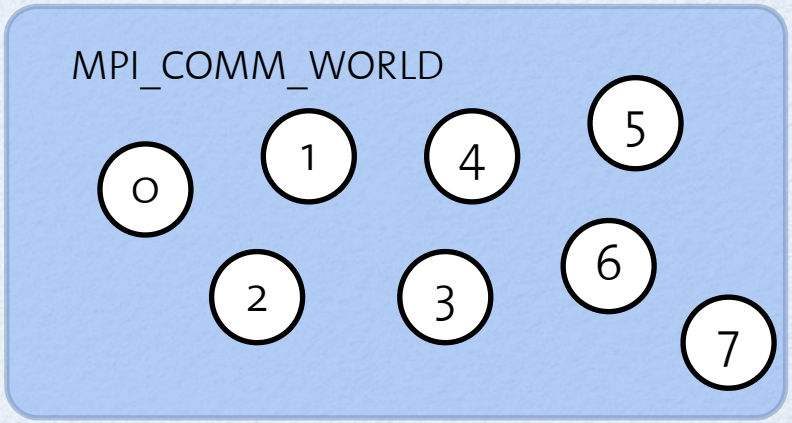
- You've seen two of the five functions connected with setting up the MPI environment.

```
int MPI_Init(int*argc, char***argv);
// initializes the environment

int MPI_Finalize()
// terminates the environment

int MPI_Abort( MPI_Comm comm, int errorcode );
// terminates all processes with the given error code

int MPI_Initialized( int *flag )
// sets the flag to true if MPI has been initialized

int MPI_Finalized( int *flag )
// sets the flag to true if MPI has been finalized
```

Obtaining the rank and size

- MPI numbers the processes inside **communicators**
- By default one communicator, **MPI_COMM_WORLD** is created containing all processes. We will learn later how to create additional communicators.

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

int main(int argc, char** argv) {

  MPI_Init(&argc, &argv);

  int rank;
  int size;

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

  std::cout << "I am rank " << rank <<
               " of " << size << "." << std::endl;

  MPI_Finalize();

  return 0;
}
```



Running the MPI program

- MPI programs need to be launched in multiple copies, on (usually) multiple machines. All implementations provide at least one common way of launching the program:

  mpiexec -np *number_of_processes executable [options]*

```
$ mpiexec -np 4 ./a.out
I am rank 1 of 4.
I am rank 2 of 4.
I am rank 0 of 4.
I am rank 3 of 4.
$
```

- Other options allow to specify the machines on which to run. Use the man pages to find out for your supercomputers or clusters.
- In the exercises you will learn how to launch MPI batch jobs on Brutus.
- Different processes can in principle run different executables but we will only write SPMD (single program multiple data) programs.

What is a message?

- Messages, like letters, consist of an envelope and the message body
- The message **body** is the data to be sent, characterized by
  - **pointer** to a memory **buffer** containing the data
  - the **type** of data in the buffer. This is needed for heterogeneous machines.
  - **length** of data in the buffer.
- The **envelope** contains the addressing information
  - a message **tag**, usually an integer identifying the type of message, like the subject line in an e-mail.
  - rank (id number) of the source and destination nodes
  - the communicator

envelope				body		
source	destination	communicator	tag	buffer	count	datatype

Builtin MPI datatypes

MPI datatype	C datatype	C++ datatype
MPI_CHAR	`char`	`char`
MPI_SIGNED_CHAR	`signed char`	`signed char`
MPI_UNSIGNED_CHAR	`unsigned char`	`unsigned char`
MPI_WCHAR	`int`	`int`
MPI_SHORT	`long`	`long`
MPI_INT	`long long`	`long long`
MPI_LONG	`wchar_t`	`wchar_t`
MPI_LONG_LONG	`short`	`short`
MPI_UNSIGNED_SHORT	`unsigned short`	`unsigned short`
MPI_UNSIGNED	`unsigned int`	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long`	`unsigned long int`
MPI_UNSIGNED_LONG_LONG	`unsigned long long`	`unsigned long long`
MPI_FLOAT	`float`	`float`
MPI_DOUBLE	`double`	`double`
MPI_LONG_DOUBLE	`long double`	`long double`
MPI_BOOL		`bool`
MPI_BYTE		
MPI_PACKED		

Sending and receiving a message

- Messages are sent and received through MPI_Send and MPI_Recv calls

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

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

- An MPI_Recv matches a message sent by MPI_Send if tag, source, and dest match.
  - the **tag** has to be the same. MPI_ANY_TAG can be used as wildcard for MPI_Recv
  - it only matches on the rank specified by **dest**.
  - **source** has to be the rank of the sending process. MPI_ANY_SOURCE can be used as wildcard.
  - The buffer size on the receiving side is the allocated memory, and thus the maximum message size that can be received, and not necessarily the actual size.

A first example of message passing

- A parallel “Hello World” program
  - rank 1 sends a string with tag 42 to rank 0
  - rank 0 receives a string with tag 42 from rank 1 and prints it

```
int main(int argc, char** argv) {

  MPI_Init(&argc, &argv);
  int num;

  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  if(num==0) { // "master"
    MPI_Status status;
    char txt[100];
    MPI_Recv(txt, 100, MPI_CHAR,
            1, 42, MPI_COMM_WORLD, &status);
    std::cout << txt << "\n";
  }
  else { // "worker"
    std::string text="Hello world!";
    MPI_Send(const_cast<char*>(text.c_str()), text.size()+1, MPI_CHAR,
             0, 42, MPI_COMM_WORLD);
  }

  MPI_Finalize();

  return 0;
}
```

What happens if we run it with too few or too many processes?

Error handling in MPI functions

- All MPI functions return an integer argument indicating an error if the return value is not equal to MPI_SUCCESS.
- By default all functions abort in case of an error. This can be changed setting the error handling policy and then dealing with the error yourself.

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

  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  // tell MPI to return an error code instead of aborting
  MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);

  if(num==0) { // "master"
    MPI_Status status;
    char txt[100];
    int err = MPI_Recv(txt, 100, MPI_CHAR,
            1, 42, MPI_COMM_WORLD, &status);
    // now check for an error
    if (err != MPI_SUCCESS) {
      // get the error text for the error code
      int len = MPI_MAX_ERROR_STRING;
      char txt[MPI_MAX_ERROR_STRING];
      MPI_Error_string(err, txt, &len);
      std::cerr << "Runtime error: " << txt << std::endl;
      MPI_Abort(MPI_COMM_WORLD,-1);
    }
    ...
```

Look at errors.cpp for a nicer way to deal with an error

Probing for messages

- Instead of directly receiving you can probe whether a message has arrived:

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
// wait for a matching message to arrive

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
// check if a message has arrived.
// flag is nonzero if there is a message waiting

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int* count)
// gets the number of elements in the message waiting to be received
```

- The MPI_Status object can be queried for information about the message:

```
MPI_Status status;
int count;

// wait for a message
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, &status);
std::cout << "A message is waiting from " << status->MPI_SOURCE
          << "with tag "                  << status->MPI_TAG;

// get the element count
MPI_Get_count(&status, MPI_INT, &count)
std::cout << "and assuming it contains ints there are " << count << "elements";
```

Sending and receiving

- Blocking sends return only when the buffer is ready to be reused. The destination might or might not have received the message yet:

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// synchronous send: returns when the destination has started to receive the message

int MPI_Bsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// buffered send: returns after making a copy of the buffer. The destination might not yet
//                have started to receive the message

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// standard send: can be synchronous or buffered, depending on message size

int MPI_Rsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// ready send: an optimized send if the user can guarantee that the destination has already
//             posted the matching receive
```

- Blocking receive returns once the message has been received

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)
// blocking receive: returns once the message has been received.
//                   the status object can be queried for more information about the message
```

Watch out for deadlocks

- Both ranks wait for the other one to receive the message. We hang forever in a deadlock

```
int main(int argc, char** argv) {
  MPI_Status status;
  int num;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  double d=3.1415927;
  int tag=99;

  if(num==0) {
    MPI_Ssend(&d,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    MPI_Recv (&d,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Ssend(&d,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD);
    MPI_Recv (&d,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,&status);
  }

  MPI_Finalize();
  return 0;
}
```

Attempt 2: be careful about ordering

- It works if we swap the order for one of the ranks, but this might be tough to figure out in general

```
int main(int argc, char** argv) {
  MPI_Status status;
  int num;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  double ds=3.1415927; // to send
  double dr;           // to receive
  int tag=99;

  if(num==0) {
    MPI_Ssend(&ds,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    MPI_Recv (&dr,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Recv (&dr,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,&status);
    MPI_Ssend(&ds,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```

Attempt 3: use MPI_Sendrecv

- MPI_Sendrecv is an optimized implementation for such a swap

```
int main(int argc, char** argv) {
  MPI_Status status;
  int num;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  double ds=3.1415927; // to send
  double dr;           // to receive
  int tag=99;

  if(num==0) {
    MPI_Sendrecv(&ds,1,MPI_DOUBLE,1,tag,
                 &dr,1,MPI_DOUBLE,1,tag,
                 MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Sendrecv(&ds,1,MPI_DOUBLE,0,tag,
                 &dr,1,MPI_DOUBLE,0,tag,
                 MPI_COMM_WORLD,&status);
  }

  MPI_Finalize();
  return 0;
}
```

- But it does not guarantee that there might not be deadlocks with other communications happening at the same time

Attempt 4: buffering

- We can provide a large enough buffer and force a buffered send

```
int main(int argc, char** argv) {
  MPI_Status status;
  int num;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  double ds=3.1415927; // to send
  double dr;           // to receive
  int tag=99;

  // allocate a buffer and attach it to MPI
  int buffer_size = sizeof(double) + MPI_BSEND_OVERHEAD;
  char* buffer = new char[buffer_size];
  MPI_Buffer_attach(buffer,buffer_size);

  if(num==0) {
    MPI_Bsend(&ds,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    MPI_Recv (&dr,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Bsend(&ds,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD);
    MPI_Recv (&dr,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,&status);
  }

  // detach the buffer, making sure all sends are done
  MPI_Buffer_detach(buffer,&buffer_size);
  delete[] buffer;

  MPI_Finalize();
  return 0;
}
```

Attempt 5: hope that you're lucky

- Hope that for such a small message MPI will always buffer it when using a standard send.
- This works on my laptop but might fail elsewhere and is not a good strategy.

```
int main(int argc, char** argv) {
  MPI_Status status;
  int num;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  double ds=3.1415927; // to send
  double dr;           // to receive
  int tag=99;

  if(num==0) {
    MPI_Send(&ds,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    MPI_Recv (&dr,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Send(&ds,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD);
    MPI_Recv (&dr,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,&status);
  }

  MPI_Finalize();
  return 0;
}
```

Domain decomposition for PDEs

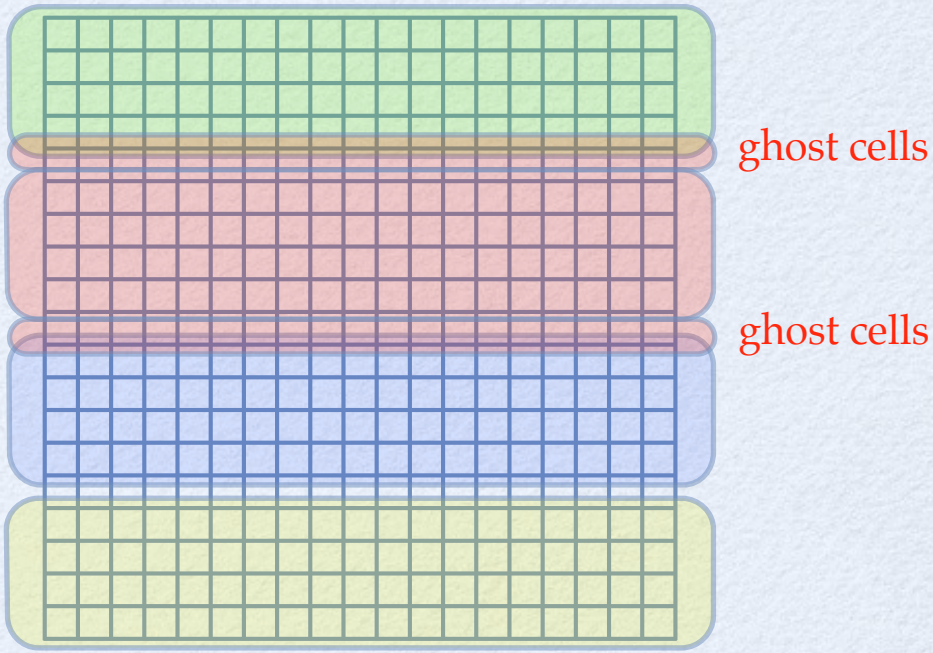
- Simple example: finite difference solution of a diffusion equation

$$\frac{\partial \phi(\vec{r},t)}{\partial t} = D\Delta\phi(\vec{r},t)$$

- Domain decomposition: split the mesh over the nodes of the parallel computer

- The finite difference stencil needs information from the neighboring domains: stored in “ghost cells”

- Message passing is needed to update the ghost cells after each time step



1d diffusion equation by MPI

- We need to exchange the ghost cell values in a deadlock-free way before each iteration

```
for (int t=0; t<iterations; ++t) {
  // first get the ghost cells and send our boundary values to
  // the neighbor for their ghost cells

  // avoid deadlocks by a clear ordering who sends and receives first
  // make sure we have an even number of ranks for this to work
  assert(size %2 == 0);

  if (rank % 2 == 0) {
    MPI_Send(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD);
    MPI_Recv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&status);
    MPI_Send(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD);
    MPI_Recv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Recv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&status);
    MPI_Send(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD);
    MPI_Recv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&status);
    MPI_Send(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD);
  }

  // do calculation
  for (int i=1; i<local_N-1;++i)
    newdensity[i] = density[i] + coefficient * (density[i+1]+density[i-1]-2.*density[i]);

  // and swap
  density.swap(newdensity);
}
```

Overlaying communication and computation

1. This code will not scale well since we waste time in waiting for the ghost cells to arrive. A better strategy is to overlay computation and communication:

    1. start the communication for the ghost cells
    2. update the interior of the local segment
    3. wait for communication to finish
    4. update the boundary values using the ghost cells

- We now use the wait time for communication to perform most of the calculations. Ideally the ghost cells will have arrived before we finish the computation and communication will then be essentially free.

- This needs non-blocking, asynchronous communication

Nonblocking send and receive

- These functions return immediately, while communication is still ongoing.

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

int MPI_Ibsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request)


int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm, MPI_Request *request)

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

- They behave the same way as the corresponding blocking versions but perform the communication asynchronously.
- They fill in an MPI_Request object that can be used to test for completion.

Waiting for completion

- We can wait for one, some, or all communication requests to finish

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
// waits for the communication to finish and fills in the status

int MPI_Waitall(int count, MPI_Request array_of_requests[],
                MPI_Status array_of_statuses[])
// waits for all given communications to finish and fills in the statuses

int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index,
                MPI_Status *status)
// waits for one of the given communications to finish, sets the index to indicate
// which one and fills in the status

int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[], MPI_Status array_of_statuses[])
// waits for at least one of the given communications to finish, sets the number
// of communication requests that have finished, their indices and status
```

Testing for completion and cancellation

- Instead of waiting we can just test whether they have finished

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
// tests if the communication is finished. Sets flag to 1 and fills in the status if
// finished or sets the flag to 0 if not finished.

int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag,
                MPI_Status array_of_statuses[])
// test whether all given communications are finished. Sts flag to 1 and fills in
// the status aray if all are finished or sets the flag to 0 if not all are finished.

int MPI_Testany(int count, MPI_Request array_of_requests[], int *index,
                int *flag, MPI_Status *status)
// test whether one of the given communications is finished. Sets flag to 1 and fills
// in the index and status if one  finished or sets the flag to 0 if none is finished.

int MPI_Testsome(int incount, MPI_Request array_of_requests[], int *outcount,
                 int array_of_indices[], MPI_Status array_of_statuses[])
// tests whether some of the given communications is finished, sets the number
// of communication requests that have finished, their indices and statuses.
```

- We can cancel a request if we don't want to wait any longer

```
int MPI_Cancel(MPI_Request *request)
```

Overlaying communication and computation

- Exchange ghost cells while we compute the interior

```
for (int t=0; t<iterations; ++t) {
  // first start the communications

  if (rank % 2 == 0) {
    MPI_Isend(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[0]);
    MPI_Irecv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[1]);
    MPI_Isend(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[2]);
    MPI_Irecv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[3]);
  }
  else {
    MPI_Irecv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[0]);
    MPI_Isend(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[1]);
    MPI_Irecv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[2]);
    MPI_Isend(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[3]);
  }

  // do calculation  of the interior
  for (int i=2; i<local_N-2;++i)
    newdensity[i] = density[i] + coefficient * (density[i+1]+density[i-1]-2.*density[i]);

  // wait for the ghost cells to arrive
  MPI_Waitall(4,reqs,status);

  // do the boundaries
  newdensity[1] = density[1] + coefficient * (density[2]+density[0]-2.*density[1]);
  newdensity[local_N-2] = density[local_N-2] + coefficient * (
                            density[local_N-1]+density[local_N-3]-2.*density[local_N]);

  // and swap
  density.swap(newdensity);
}
```

Parallelizing the sum for π

- Similar to multithreading but how do we collect the result?

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

  int size;
  int rank;
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  long double sum=0.;
  long double localsum=0.;

  unsigned long const nterms = 100000000;
  long double const step = (nterms+0.5l) /  size;

  // do just one piece on each rank
  unsigned long start = rank * step;
  unsigned long end = (rank+1) * step;
  for (std::size_t t = start; t < end; ++t)
    localsum += (1.0 - 2* (t % 2)) / (2*t + 1);

  // now collect all to the master (rank 0)
  ???????

  if (rank==0) // only one prints
    std::cout << "pi=" << std::setprecision(18) << 4.*sum << std::endl;

  MPI_Finalize();
  return 0;
}
```

Collective Communication

- The naïve reduction takes time O(*N*), which is disaster: the time increases rapidly with *N* for large *N*:

```
// now collect all to the master (rank 0)
if (rank==0) {
  sum = localsum;
  // Master receives from all other ranks
  for (int i=1; i<size;++i) {
    MPI_Recv(&localsum, 1, MPI_LONG_DOUBLE, i, 42, MPI_COMM_WORLD,&status);
    sum += localsum;
  }
}
else
  MPI_Send(&localsum, 1, MPI_LONG_DOUBLE, 0, 42,MPI_COMM_WORLD);
```

- Collective communication between many processes can be optimized by a tree-like communication pattern and finish in $\log_2(N)$ communications per rank instead of the naive *N*.
- Some machines even have additional tree-like networks for collective communication
- Here we need a collective reduction operation.

Collective reductions

- MPI provides two collective reduction operations

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm);
// performs a reduction using the operation op on the data in sendbuf and places the
// results in recvbuf on the root rank.
// if MPI_IN_PLACE is specified as sendbuf then the data to be reduced is assumed to
// be in the recvbuf and will be replaced on the root rank

int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm);
// performs a reduction using the operation op on the data in sendbuf and places the
// results in recvbuf on all ranks
// if MPI_IN_PLACE is specified as sendbuf then the data to be reduced is assumed to
// be in the recvbuf and will be replaced by the reduction
```

- where the following operations are built in and others can be defined

op	description
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

op	description
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR

Parallelizing the sum for π

- Now use MPI_Reduce: the code is simpler and faster

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

  int size;
  int rank;
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  long double sum=0.;
  long double localsum=0.;

  unsigned long const nterms = 100000000;
  long double const step = (nterms+0.5l) /  size;

  // do just one piece on each rank
  unsigned long start = rank * step;
  unsigned long end = (rank+1) * step;
  for (std::size_t t = start; t < end; ++t)
    localsum += (1.0 - 2* (t % 2)) / (2*t + 1);

  // now collect all to the master (rank 0)
  MPI_Reduce(&localsum, &sum, 1, MPI_LONG_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

  if (rank==0) // only one prints
    std::cout << "pi=" << std::setprecision(18) << 4.*sum << std::endl;

  MPI_Finalize();
  return 0;
}
```

In-place reductions

- Use MPI_IN_PLACE to avoid separate local and global sums:

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

  int size;
  int rank;
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  long double sum=0.;

  unsigned long const nterms = 100000000;
  long double const step = (nterms+0.5l) /  size;

  // do just one piece on each rank
  unsigned long start = rank * step;
  unsigned long end = (rank+1) * step;
  for (std::size_t t = start; t < end; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);

  // now collect all to the master (rank 0)
  MPI_Reduce(rank == 0 ? MPI_IN_PLACE : &sum, &sum, 1, MPI_LONG_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

  if (rank==0) // only one prints
    std::cout << "pi=" << std::setprecision(18) << 4.*sum << std::endl;

  MPI_Finalize();
  return 0;
}
```

Parallelizing Simpson integration

- Only the master rank (0) reads the input data. How do we share it?

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

  int size;
  int rank;
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  double a;            // lower bound of integration
  double b;            // upper bound of integration
  int nsteps; // number of subintervals for integration

  // read the parameters on the master rank
  if (rank==0);
    std::cin >> a >> b >> nsteps;

  // we need to share the parameters with the other ranks
  ???

  // integrate just one part on each thread
  double delta = (b-a)/size;
  double result = simpson(func,a+rank*delta,a+(rank+1)*delta,nsteps/size);

  //  collect all to the master (rank 0)
  MPI_Reduce(rank == 0 ? MPI_IN_PLACE : &result, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   // the master prints
  if (rank==0)
    std::cout << result << std::endl;

  MPI_Finalize();
  return 0;
}
```

Broadcast

- MPI provides a collective broadcast operation

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm )
// broadcast the data from the root rank to all others
```

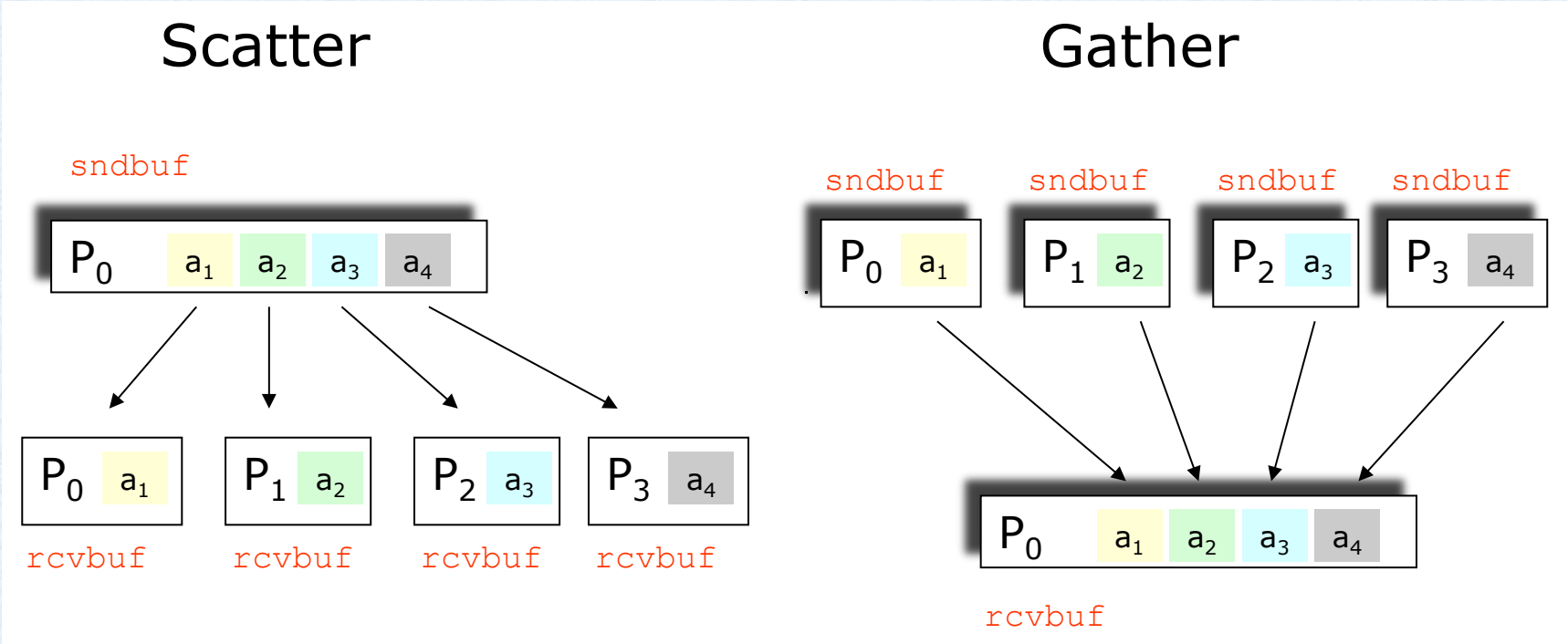
- We can use this to broadcast the data

```
// and then broadcast the parameters to the other ranks
 MPI_Bcast(&a, 1, MPI_DOUBLE,0, MPI_COMM_WORLD);
 MPI_Bcast(&b, 1, MPI_DOUBLE,0, MPI_COMM_WORLD);
 MPI_Bcast(&nsteps, 1, MPI_INT,0, MPI_COMM_WORLD);
```

- This is inefficient since we use three broadcasts.
- We will later pack all parameters into one buffer and broadcast that buffer.

Scatter and gather

- The **scatter** operation sends a different piece of data to each of the ranks
  - Example: take a vector and split it over the other ranks
- The **gather** operations collects data from the other ranks into a big buffer
  - Example: gathering pieces of a distributed vector into a big local one



Gather operations

- There are four versions of gather operations
  - either just one root rank gathers the data or all ranks gather
  - the sizes on each rank can be the same or different
- MPI_IN_PLACE can again be used for the sendbuf

```
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
               void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
// gathers data from the sendbuf buffers into a recvbuf buffer on the root rank
// recvbuf, recvcnt and recvtype are significant only on the root rank
// Note: the sendcnt needs to be the same on all ranks

int MPI_Gatherv(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int *recvcnts, int *displs,
                MPI_Datatype recvtype, int root, MPI_Comm comm)
// similar to MPI_Gather but the sendcnt values can differ from rank to rank
// the root node thus gets an array of recvcnts and of displacements displs
// The displacements specify where the data from each rank starts in the buffer

int MPI_Allgather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
               void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)
// similar to MPI_Gather, but the data is gathered at all ranks and not just a root
// it is semantically the same as an MPI_Gather followed by MPI_Bcast

int MPI_Allgatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                   void *recvbuf, int *recvcounts, int *displs,
                   MPI_Datatype recvtype, MPI_Comm comm)
// similar to MPI_Gatherv, but the data is gathered at all ranks and not just a root
// it is semantically the same as an MPI_Gatherv followed by MPI_Bcast
```

Scatter operations

- There are two versions of scatter operations
  - the sizes on each rank can be the same or different
  - MPI_IN_PLACE can be used for the recvbuf

```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
// scatters data from the sendbuf buffer on the root rank into recvbuf buffers on the
// other ranks. Each rank gets a corresponding junk of the data
// sendbuf, sendcnt and sendtype are significant only on the root rank
// Note: recvcnt needs to be the same on all ranks

int MPI_Scatterv( void *sendbuf, int *sendcnts, int *displs,
                  MPI_Datatype sendtype, void *recvbuf, int recvcnt,
                  MPI_Datatype recvtype, int root, MPI_Comm comm)
// similar to MPI_Scatter but the sendcnt values can differ from rank to rank
// the root node thus spcifies an array of recvcnts and of displacements displs
// The displacements specify where the data for each rank starts in the buffer
```

- And there is a combined reduction plus scatter
  - MPI_IN_PLACE can be used for the sendbuf

```
int MPI_Reduce_scatter(void *sendbuf, void *recvbuf, int *recvcnts,
                       MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
// optimized version of an MPI_Reduce followed by an MPI_Scatter
```

All-to-all and barrier

- **MPI_Alltoall**: *n*-th rank sends *k*-th portion of its data to rank *k* and receives *n*-th portion from node *k*.
  - Everyone scatters and gather at the same time
  - like a matrix transpose. **Attention: slow!**

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

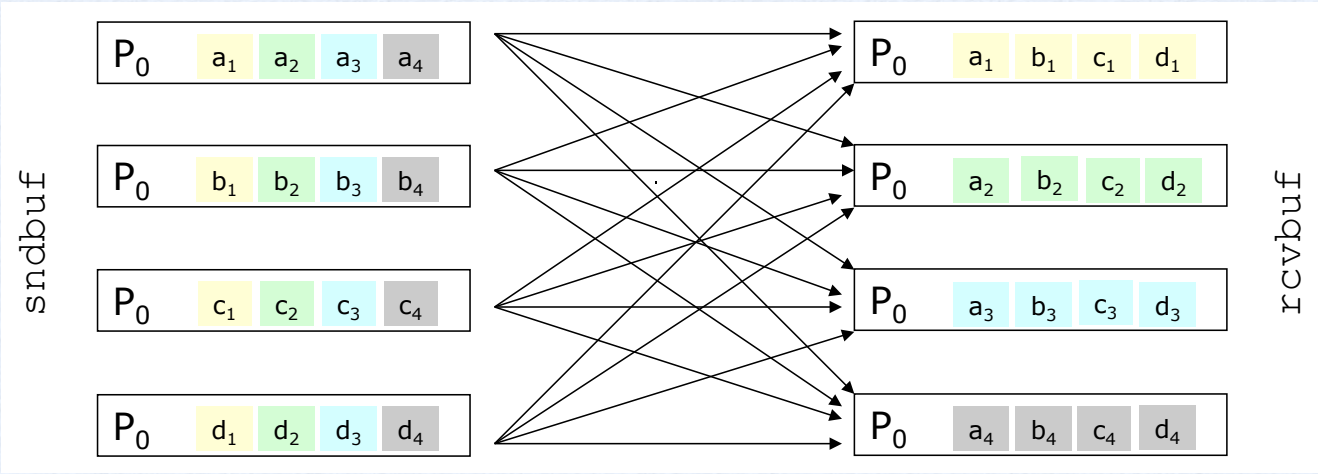


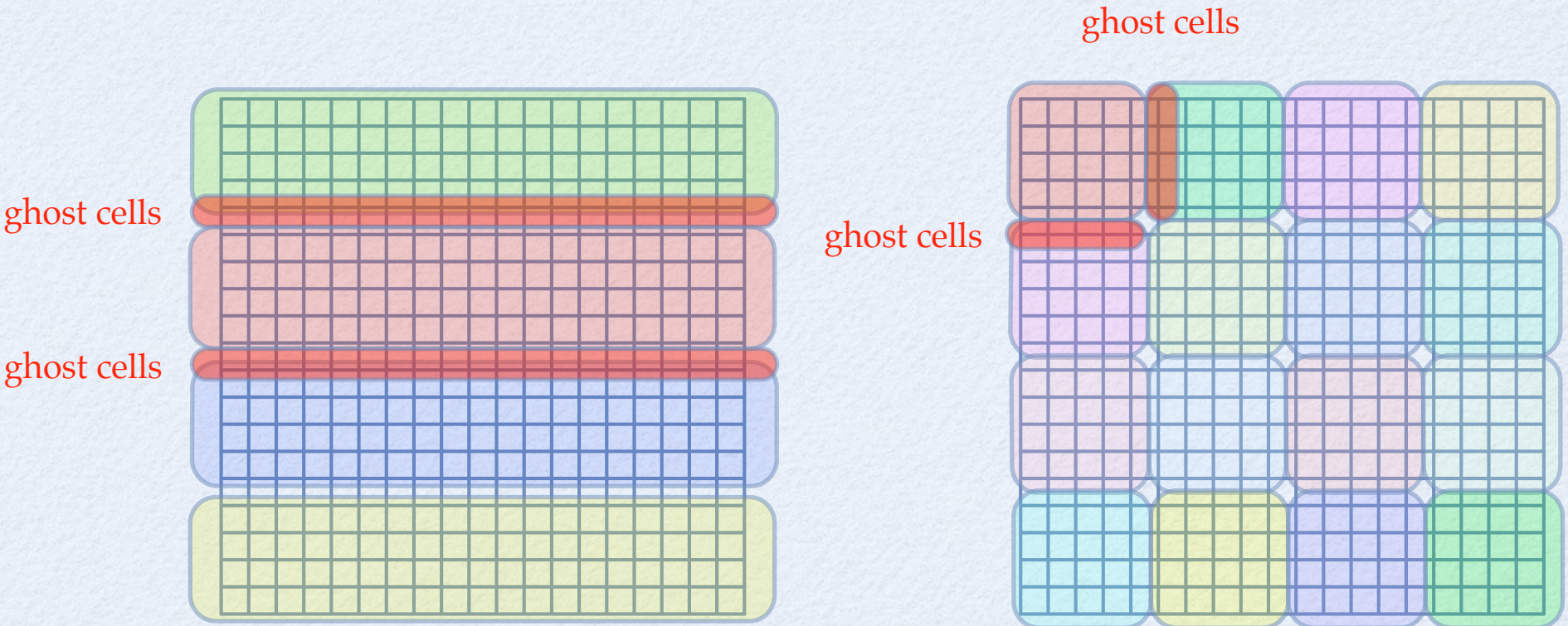
Image © CSCS

- The **MPI_Barrier** waits for all ranks to call it; used for synchronization

```
int MPI_Barrier( MPI_Comm comm )
```

Recall domain decomposition and ghost cells

- How do we best exchange boundary values with the neighboring ranks?
  - In 1D it was just a single number and was easy
  - Sometimes we might be lucky and they could be contiguous arrays
- What shall we do in the general case?
  - pack them into buffers?
  - or just describe to MPI where they are in memory?



Packing and unpacking

- Allocate a sufficiently large buffer and then pack the data into it
- Send/receive the packed buffer with type **MPI_PACKED**
- Finally unpack it on the receiving side

```
int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype,
             void *outbuf, int outcount, int *position, MPI_Comm comm)
// packs the data given as input into the outbuf buffer starting at a given position.
// outcount is the size of the buffer and position gets updated to point to the first
// free byte after packing in the data.
// An error is returned if the buffer is too small.

int MPI_Unpack(void *inbuf, int insize, int *position,
               void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)
// unpack data from the buffer starting at given position into the buffer outbuf.
// position is updated to point to the location after the last byte read

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)
// returns in size an upper bound for the number of bytes needed to pack incount
// values of type datatype. This can be used to determine the required buffer size
```

Packing data into a buffer

- Pack the input data, broadcast it and unpack

```
// create a buffer and pack the values.
// first get the size for the buffer and allocate a buffer
int size_double, size_int;
MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD,&size_double);
MPI_Pack_size(1, MPI_INT,    MPI_COMM_WORLD,&size_int);
int buffer_size = 2*size_double+size_int;
char* buffer = new char[buffer_size];

// pack the values into the buffer on the master
if (rank==0) {
  int pos=0;
  MPI_Pack(&a, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  MPI_Pack(&b, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  MPI_Pack(&nsteps, 1, MPI_INT, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  assert ( pos <= buffer_size );
}

// broadcast the buffer
MPI_Bcast(buffer, buffer_size, MPI_PACKED, 0, MPI_COMM_WORLD);

// and unpack on the receiving side
int pos=0;
MPI_Unpack(buffer, buffer_size, &pos, &a, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &b, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &nsteps, 1, MPI_INT, MPI_COMM_WORLD);
assert ( pos <= buffer_size );

// and finally delete the buffer
delete[] buffer;
```

Sending it bitwise

- The dangerous solution: pack it all into a struct and send it bitwise
- This assumes a homogeneous machine with identical integer and floating point formats.

```
// define a struct for the parameters
struct parms {
  double a;            // lower bound of integration
  double b;            // upper bound of integration
  int nsteps; // number of subintervals for integration
};

parms p;

// read the parameters on the master rank
if (rank==0);
  std::cin >> p.a >> p.b >> p.nsteps;

// broadcast the parms as bytes - warning, not portable on heterogeneous machines
MPI_Bcast(&p, sizeof(parms), MPI_BYTE, 0, MPI_COMM_WORLD);
```