HPCSE - I

«MPI Programming Model - Part I»

Panos Hadjidoukas

Schedule and Goals

- 10.11.2017: MPI - part 1
  - study the basic features of MPI
  - able to understand and write MPI programs
- 24.11.2017: MPI - part 2
  - asynchronous communication
  - how MPI works
  - MPI I/O
  - study and discuss more examples
- 01.12.2017: MPI + OpenMP
  - hybrid programming model
  - nested OpenMP parallel

Outline

- Strong and weak scaling
- Introduction to MPI
- Point-to-point communication
- Collective communication

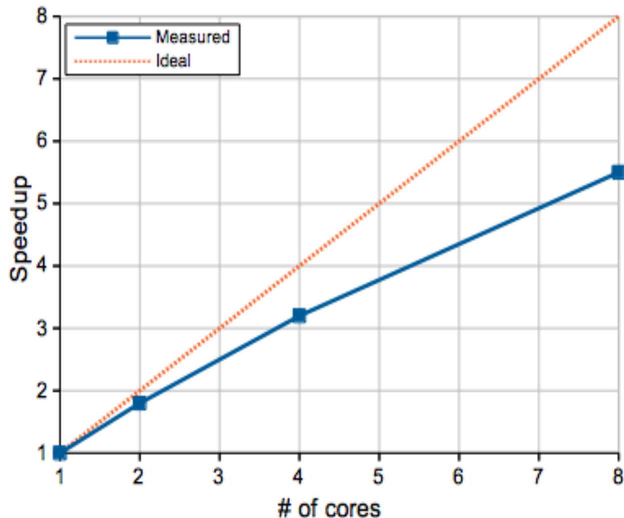
Performance metrics

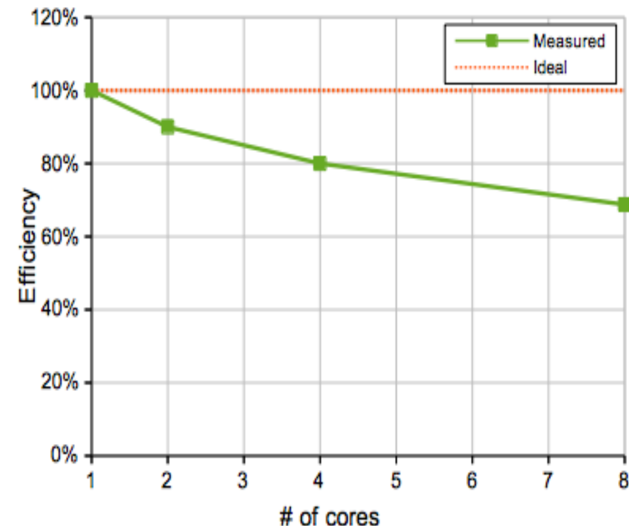
- ACM Gordon Bell Prize (every year at the SC conference): “awarded for **peak performance** or special achievements in **scalability** and **time-to-solution** on important science and engineering problems”

- Time-to-solution: solve a problem as fast as possible
- Peak performance: percentage of the peak (FLOP/s)
- Scalability: strong and weak

Strong scaling

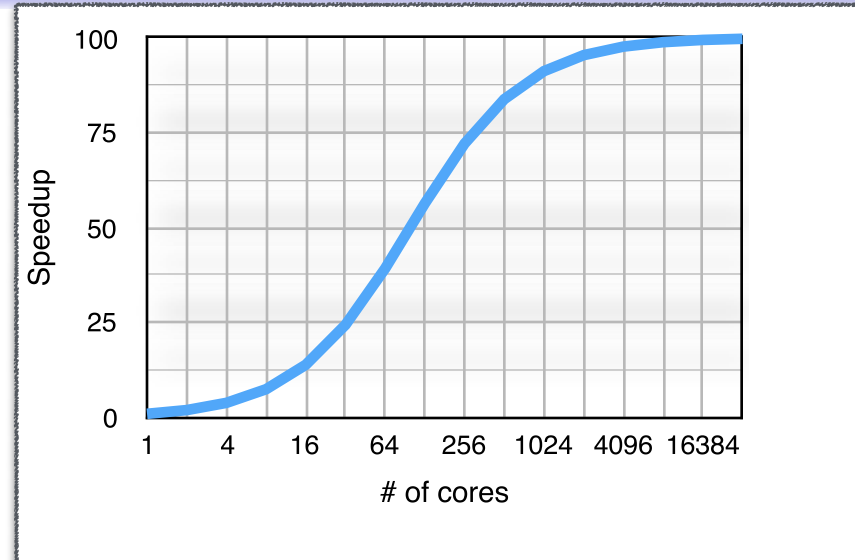
- T(p): execution time on p processors
- Speedup(p) = T(1)/T(p)
- Strong scaling: keep the problem size constant as you increase the number of CPU cores p
- Strong scaling Efficiency(p) = Speedup(p)/p (x100%)

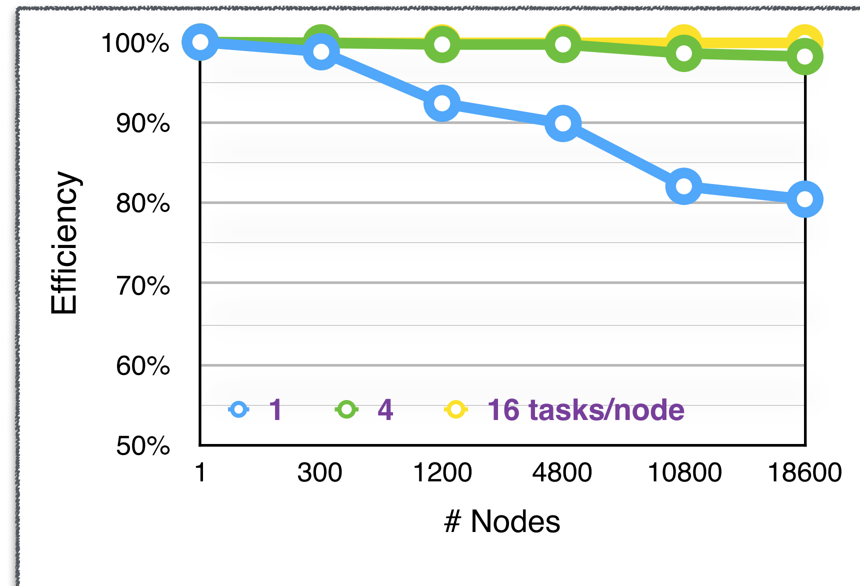




Weak scaling

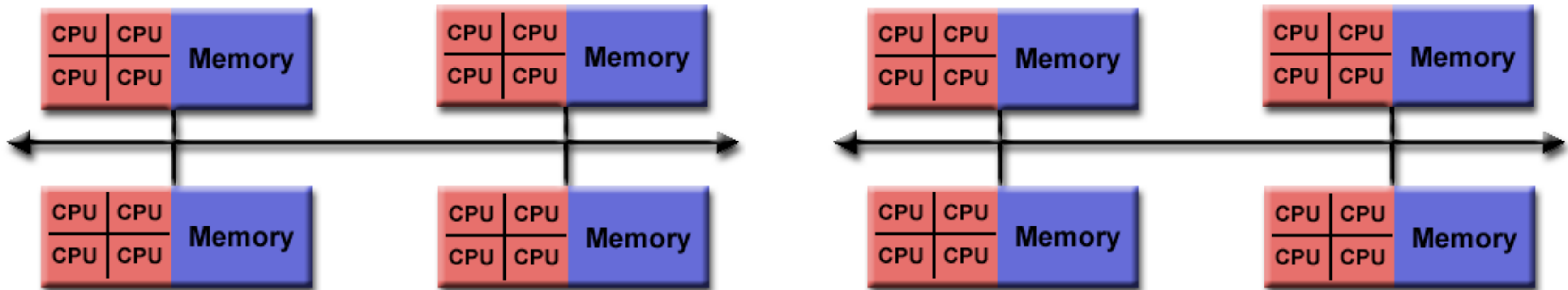
- Problem of strong scaling: speedup is limited by the serial fraction s of the code (Amdahl's Law)
  - s = 1% → max speedup = 100
- Weak scaling: constant work per processing unit
  - increase the problem size with the number of CPU cores p
  - problem size = computational workload
- **Efficiency**(p) = T(1)/T(p) (x100%)
  - How well you can solve bigger problems
  - T(p) is expected to increase due to parallelization (e.g. communication) overheads





Hybrid distributed / shared memory

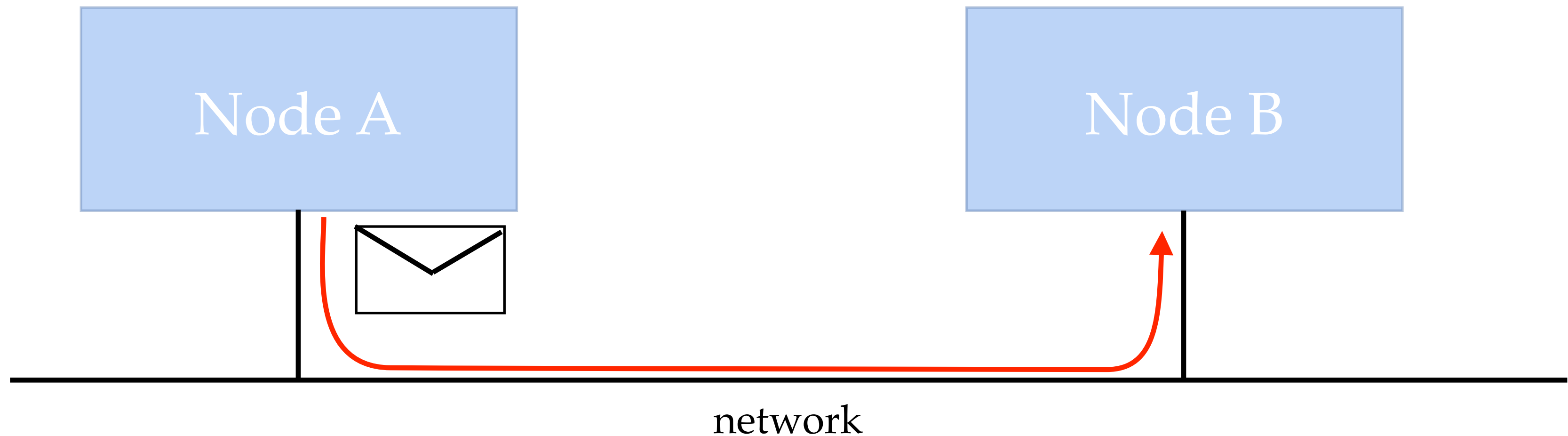
- To go beyond a single shared memory compute node we have to hook together many nodes with a network to form a hybrid architecture using shared and distributed memory.



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.
  - "nodes" = processes running on different compute nodes or even on the same compute node
- 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**: 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**: 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
  - On most supercomputers MPI comes preinstalled
  - Two main implementations: MPICH and OpenMPI
  - Euler modules:
    - module load open_mpi
    - module load mvapich2
- 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-sensitive file systems)
  - Most wrappers have options that inform the user of the

```
$ mpicc -compile_info
clang -Wl,-flat_namespace -Wl,-commons,use_dylibs -I/usr/local/Cellar/mpich/3.2_3/include -L/usr/local/Cellar/mpich/3.2_3/lib -lmpi -lpmpi
```

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`.

```
#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

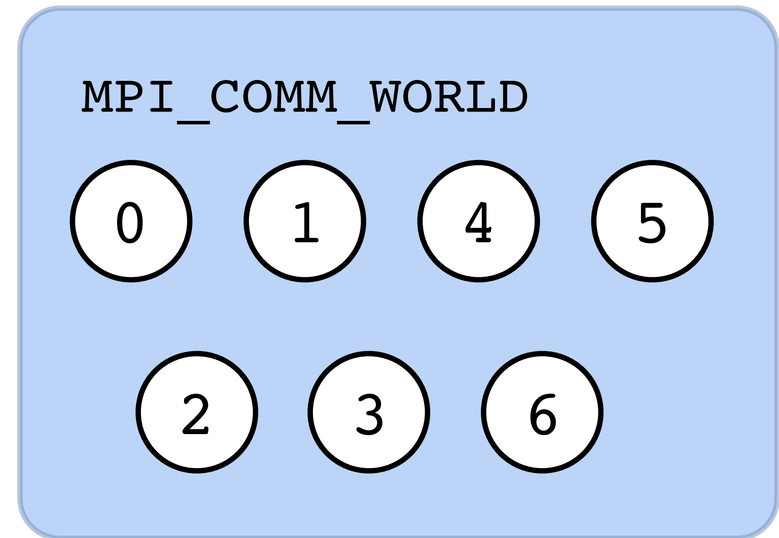
- You've seen two of the five functions connected with setting up the MPI environment.

```
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 <stdio.h>
#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);

  printf("I am rank %d of %d\n", rank, size);

  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 -n number_of_processes executable [options]`

```
$ mpiexec -n 4 ./hello
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 Euler.
- Different processes can in principle run different executables but we will only write SPMD (single program multiple data) programs.

Hands on session: from OpenMP to MPI

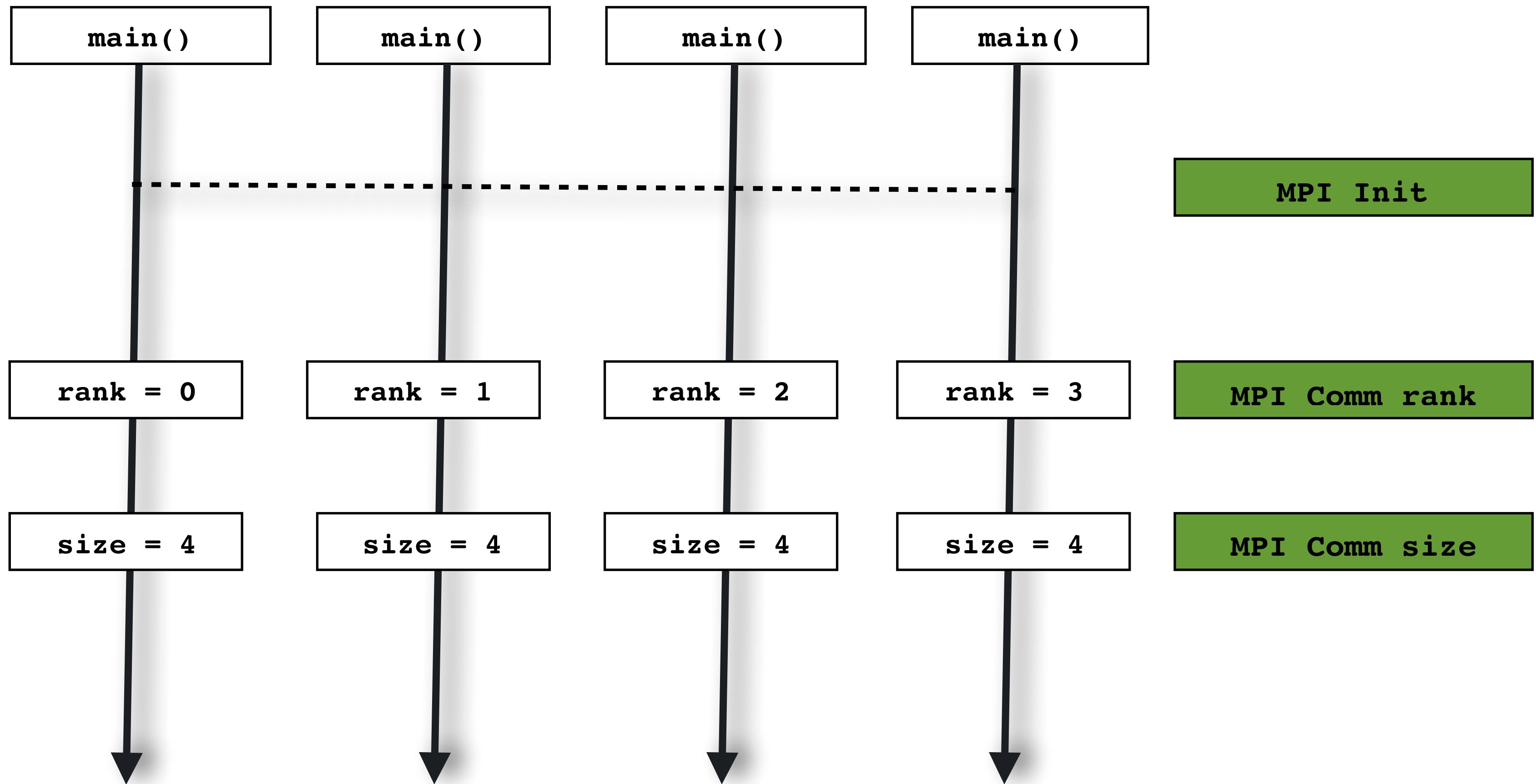
- Goal: parallelize the following code using OpenMP
- Challenge: single parallel region, created at the start of main
  - Not available: shared variables, work sharing
  - Available: synchronization (critical, barrier), library calls

```
int main(int argc, char** argv) {
{
  int rank = 0; // some parametrization, useful for the next steps
  int size = 1; // as above

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

  unsigned long const nterms = 100000000;
  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;
  if (rank == size-1) end = nterms;
  for (unsigned long t = start; t < end; ++t)
     localsum += (1.0 - 2* (t % 2)) / (2*t + 1);

  sum = localsum;

  if (rank==0) // only one prints
     printf("rank %d: pi= %lf\n", rank, 4.*sum);

  return 0;
}
```

Codes: https://gitlab.ethz.ch/hpcse17/hs2017/tree/master/examples/mpi1/openmp

SPMD execution model

- The `mpirun / mpiexec` utility (spawner) starts the executable on the target cores



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. Needed for heterogeneous machines.
  - length of data in the buffer (**count**).
- 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	wchar_t	wchar_t
MPI_SHORT	short	short
MPI_INT	int	int
MPI_LONG	long	long
MPI_LONG_LONG	long long	long long
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 (example: send_recv2.c)

A first example of message passing

- A parallel “Hello World” program
  - rank 0 sends a double with tag 88 to rank 1
  - rank 1 receives a double with tag 88 from rank 1 and prints it

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

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

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank==0) { // "master"
    double x=99;
    MPI_Send(&x, 1, MPI_DOUBLE, 1, 88, MPI_COMM_WORLD);
  }
  else if (rank==1) { // "worker"
    double y=0;

    MPI_Status status;
    MPI_Recv(&y, 1, MPI_DOUBLE, 0, 88, MPI_COMM_WORLD, &status);
    printf("rank %d: y = %lf\n", rank, y);
  }

  MPI_Finalize();

  return 0;
}
```

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

Answer:
size >= 2: all good
size = 1: failure

One more example

- A parallel “Hello World” program
  - rank 0 sends a string with tag 88 to rank 0
  - rank 1 receives a string with tag 88 from rank 1 and prints it

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

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

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank==0) { // "master"
    char text[256];
    strcpy(text, "Hello world!");
    MPI_Send(text, strlen(text), MPI_CHAR, 1, 88, MPI_COMM_WORLD);
  }
  else if (rank==1) { // "worker"
    char text[256];

    MPI_Status status;
    MPI_Recv(text, 256, MPI_CHAR, 0, 88, MPI_COMM_WORLD, &status);
    printf("rank %d: test = %s\n", rank, text);
  }

  MPI_Finalize();

  return 0;
}
```

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: hope that you are lucky

- Hope that for such a small message MPI will always buffer it when using a standard send.
- If this works or not depends on the side of the buffer sent.

```
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;
}
```

More on this in 2 weeks

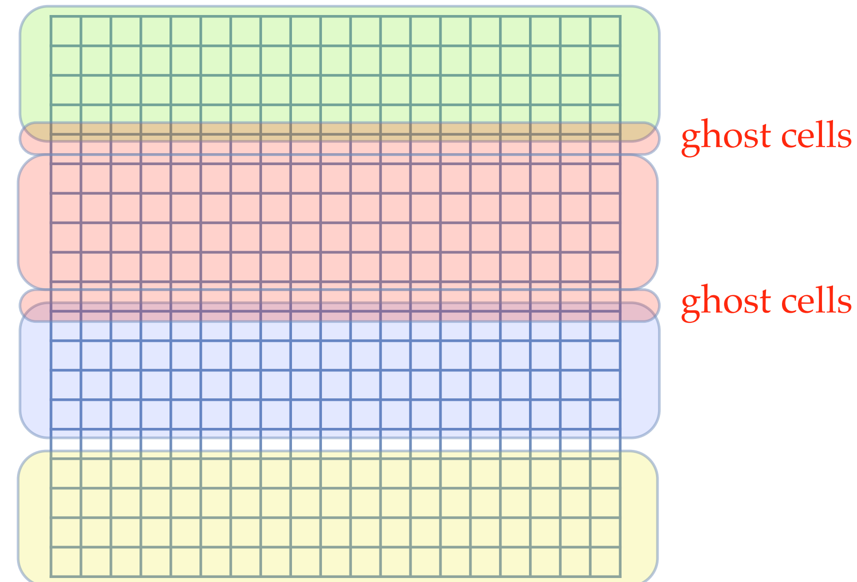
Domain decomposition for PDEs

- 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 in 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);
}
```

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);

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

  unsigned long nterms;
  terms = 100000000;
  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 (unsigned long 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
    printf("rank %d: pi= %lf\n", rank, 4.*sum);

  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);

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

  unsigned long const nterms;
  nterms = 100000000;
  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 (unsigned long 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
    printf("rank %d: pi= %lf\n", rank, 4.*sum);

  MPI_Finalize();
  return 0;
}
```

In-place reduction

- 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);

  double sum=0.;

  unsigned long const nterms = 100000000;
  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 (unsigned long 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
    printf("rank %d: pi= %lf\n", rank, 4.*sum);

  MPI_Finalize();
  return 0;
}
```

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** operation collects data from the other ranks into a big buffer
  - example: gathering pieces of a distributed vector into a big local one

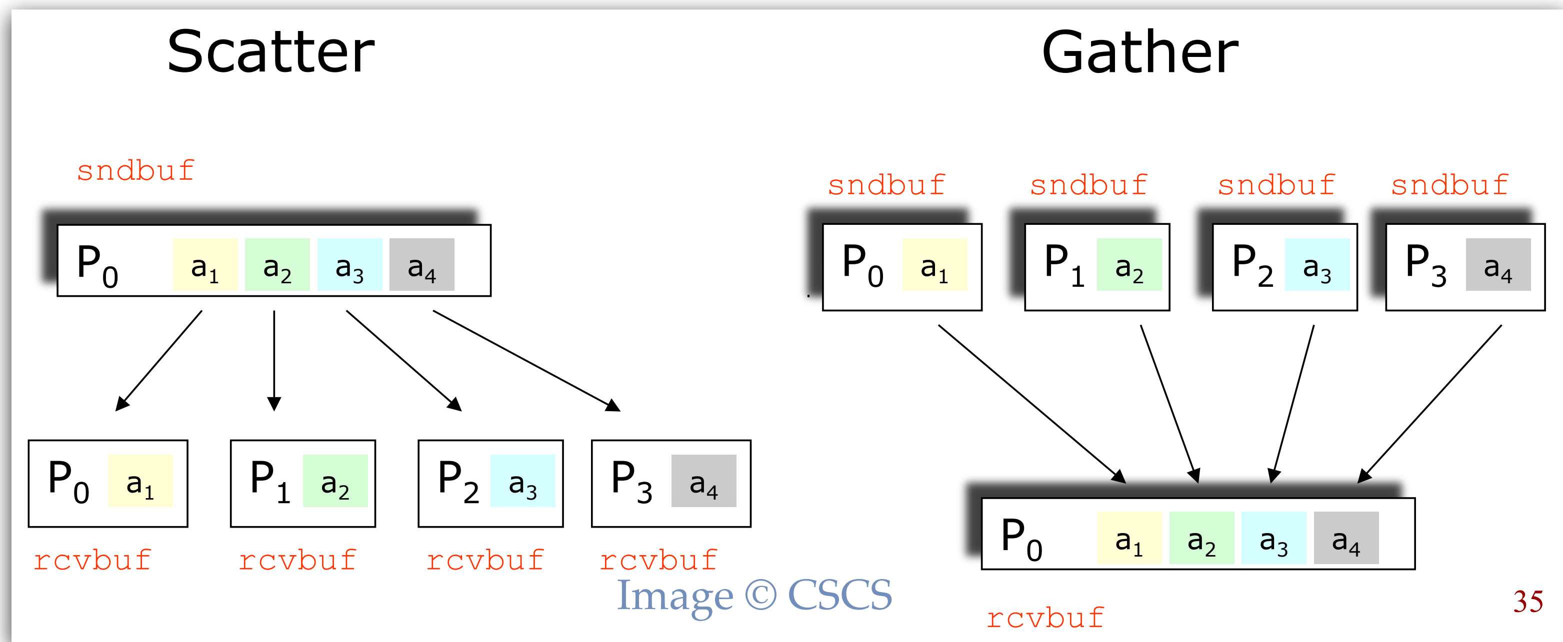


Image © CSCS

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
```

Gathering results

- Rank 0 collects the partial results using `MPI_Recv`

```
// now collect all to the master (rank 0)
if (rank == 0)
{
   double tmp[size];

   tmp[0] = localsum;
   for (int i = 1; i < size; i++)
       MPI_Recv(&tmp[i], 1, MPI_DOUBLE, i, 7777, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

   sum = tmp[0];
   for (int i = 1; i < size; i++) sum += tmp[i];
}
else
   MPI_Send(&localsum, 1, MPI_DOUBLE, 0, 7777, MPI_COMM_WORLD);
```

- Using the collective `MPI_Gather` routine

```
// now collect all to the master (rank 0)
double tmp[size];
MPI_Gather(&localsum, 1, MPI_DOUBLE, tmp, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (rank == 0)
{
   sum = tmp[0];
   for (int i = 1; i < size; i++)
       sum += tmp[i];
}
```

Broadcasting data

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

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

  double sum=0, localsum=0;

  unsigned long const nterms;
  if (rank == 0) nterms = 10000000;

  // we need to share the value of nterms with the other ranks
  ???

  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 (unsigned long 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
    printf("rank %d: pi= %lf\n", rank, 4.*sum);

  MPI_Finalize();
  return 0;
}
```

Broadcast operation

- Rank 0 broadcasts the local value using `MPI_Send`

```
if (rank == 0) nterms = 10000000;

// we need to share the value of nterms with the other ranks
if (rank == 0)
{
  for (int i = 1; i < size; i++)
    MPI_Send(&nterms, 1, MPI_UNSIGNED_LONG, i, 8888, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(&nterms, 1, MPI_UNSIGNED_LONG, 0, 8888, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

double const step = (nterms+0.5l) /  size;
```

- 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
```

- Simpler and more efficient code:

```
if (rank == 0) nterms = 10000000;

// we need to share the value of nterms with the other ranks
MPI_Bcast(&nterms, 1, MPI_UNSIGNED_LONG, 0, MPI_COMM_WORLD);

double const step = (nterms+0.5l) /  size;
```

Broadcast operation

- Rank 0 broadcasts the local value using `MPI_Send`

```
if (rank == 0) nterms = 10000000;

// we need to share the value of nterms with the other ranks
if (rank == 0)
{
  for (int i = 1; i < size; i++)
    MPI_Send(&nterms, 1, MPI_UNSIGNED_LONG, i, 8888, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(&nterms, 1, MPI_UNSIGNED_LONG, 0, 8888, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

double const step = (nterms+0.5l) /  size;
```

- 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
```

- Simpler and more efficient code:

```
if (rank == 0) nterms = 10000000;

// we need to share the value of nterms with the other ranks
MPI_Bcast(&nterms, 1, MPI_UNSIGNED_LONG, 0, MPI_COMM_WORLD);

double const step = (nterms+0.5l) /  size;
```

Barrier

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

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

- Where is the MPI bug?

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

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank % 2 == 0) {
      // do something
      MPI_Barrier(MPI_COMM_WORLD);
    }
    else {
      // do something else
    }

    MPI_Finalize();
    return 0;
}
```

References

- HPCSE Lecture Notes, Prof. M. Troyer
- MPI tutorial at LLNL, Blaise Barney
  - https://computing.llnl.gov/tutorials/mpi/
- MPI Specifications
  - http://mpi-forum.org/docs/
- MPI man pages
  - http://mpi.deino.net/mpi_functions/index.htm
  - http://www.mpich.org/static/docs/latest/
- Implementations
  - MPICH: http://www.mpich.org/
  - OpenMPI: https://www.open-mpi.org/