

A Quick Introduction to Parallel computing with MPI



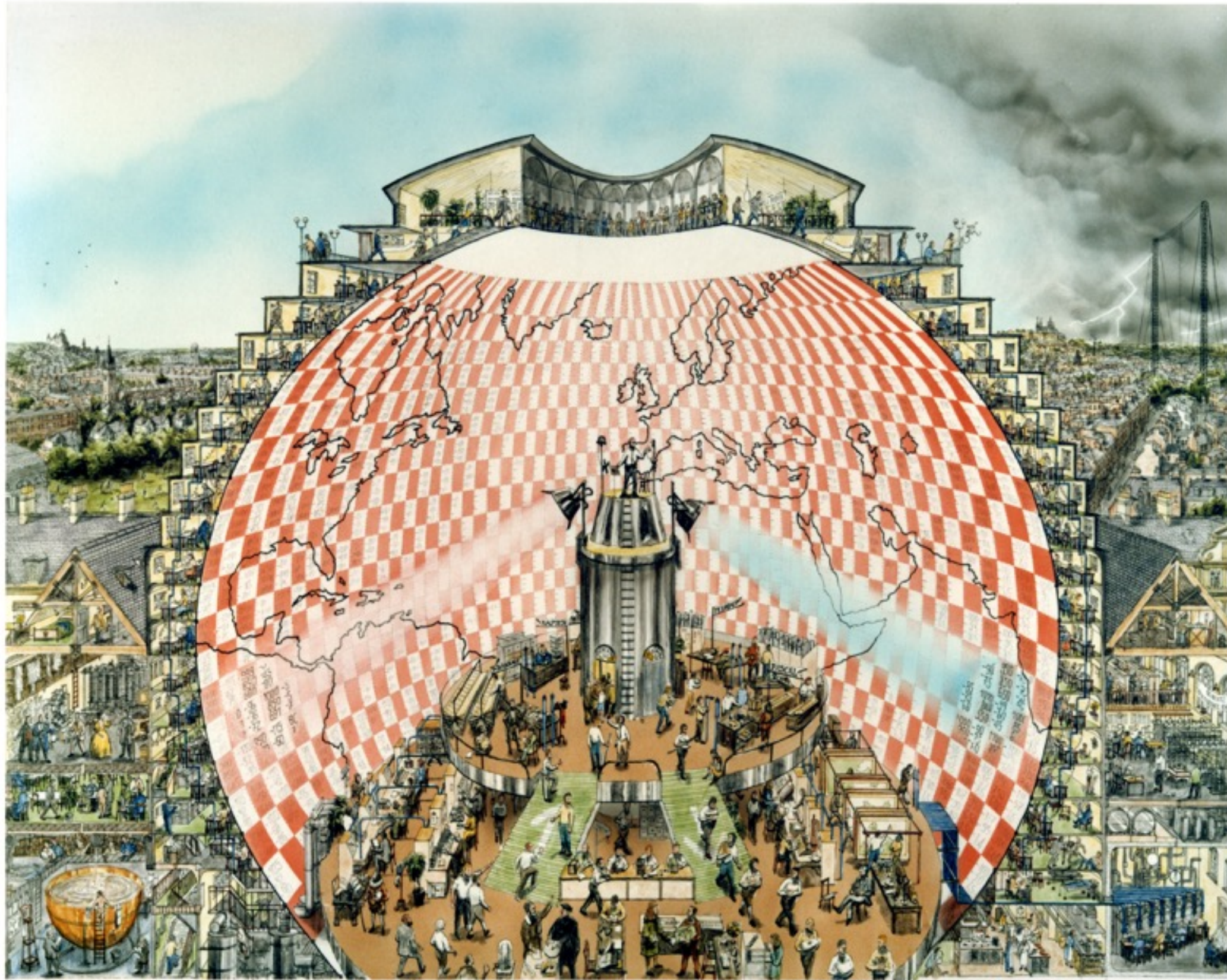
Anthony Yeates

Computing seminar, 15-Mar-2017

Why go parallel?

- Save time (faster rate of computation).
 - Solve larger problems that don't fit on memory of single machine.
 - Do different things at the same time.
 - Make better use of computing resources (e.g. parallel hardware).
-
- Increasingly used outside of traditional physics / chemistry simulations (e.g. big data).

An early vision of parallel computing



artist:
Stephen
Conlon

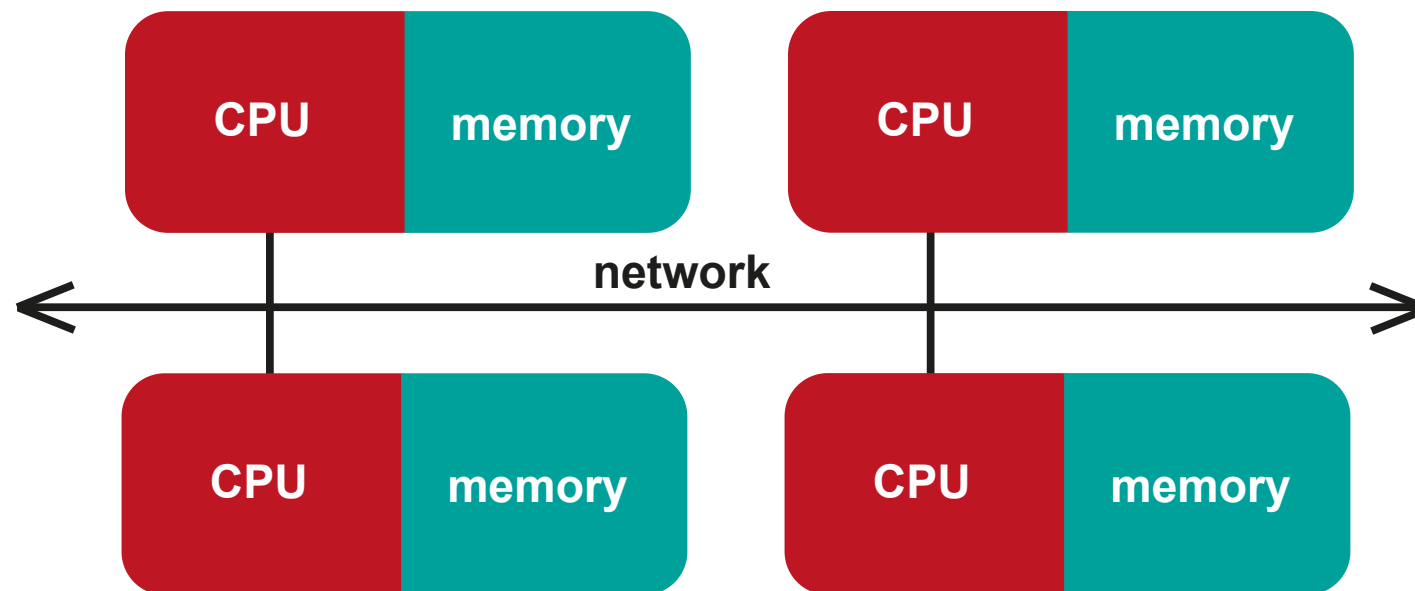
Modern parallel computing



**Blue Gene
(Argonne
National Lab)**

What is MPI?

- **Message Passing Interface** - the standard library specification for message passing programs.
- Distributed memory programming model:



- Common implementations: **OpenMPI**, MPICH, Intel MPI, ...
- Usually used with C, C++ or Fortran.
- Download from: <http://www.open-mpi.org/>

Fundamentals

- All processes execute *exactly the same code*.
- Each process can access only its own memory / variables.
- Code must be specially written with this in mind.

Demo problem: 1D heat equation

- Solve

$$u_t = k u_{xx}, \quad u(x, 0) = g(x) = e^{-100x^2}$$

on $D = \{ -1 < x < 1, t > 0 \}$.

- Discretise D with a grid

$$x_j = -1 + j\Delta x, \quad t_n = n\Delta t, \quad U^{j,n} = u(x_j, t_n)$$

- Use a simple explicit finite-difference scheme

$$\frac{U^{j,n+1} - U^{j,n}}{\Delta t} = k \frac{U^{j-1,n} - 2U^{j,n} + U^{j+1,n}}{(\Delta x)^2}$$

$$\Rightarrow U^{j,n+1} = U^{j,n} + \underbrace{\frac{k\Delta t}{(\Delta x)^2}}_{\mu} \left(U^{j-1,n} - 2U^{j,n} + U^{j+1,n} \right)$$

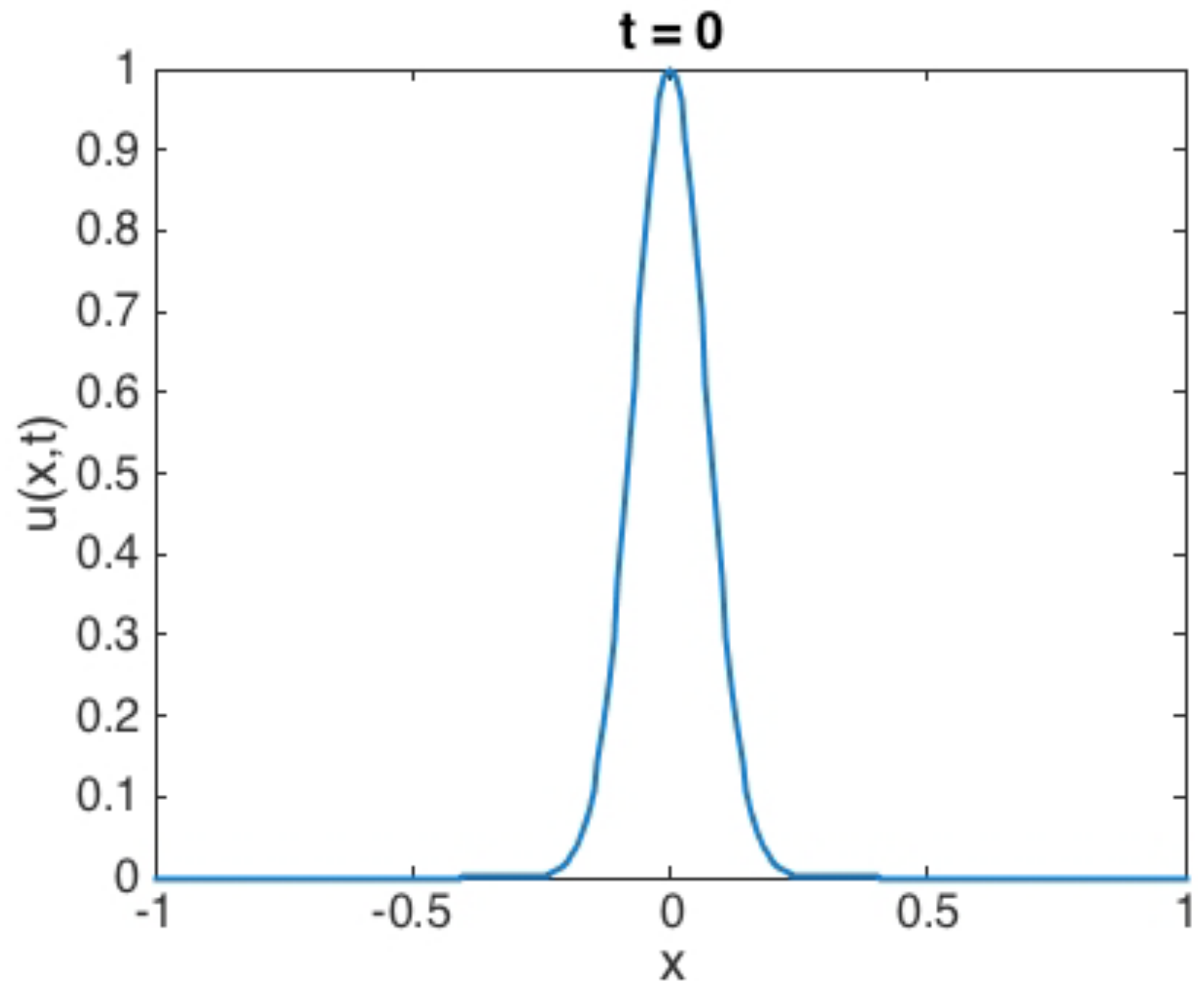
- Boundary conditions

$$U^{0,n} = U^{2,n}, \quad U^{nx+1,n} = U^{nx-1,n}$$

Exact solution

$$u(x, t) = \sqrt{\frac{a}{a + 4kt}} \exp\left(-\frac{x^2}{a + 4kt}\right)$$

$$a = \frac{1}{100}, \quad k = 1$$



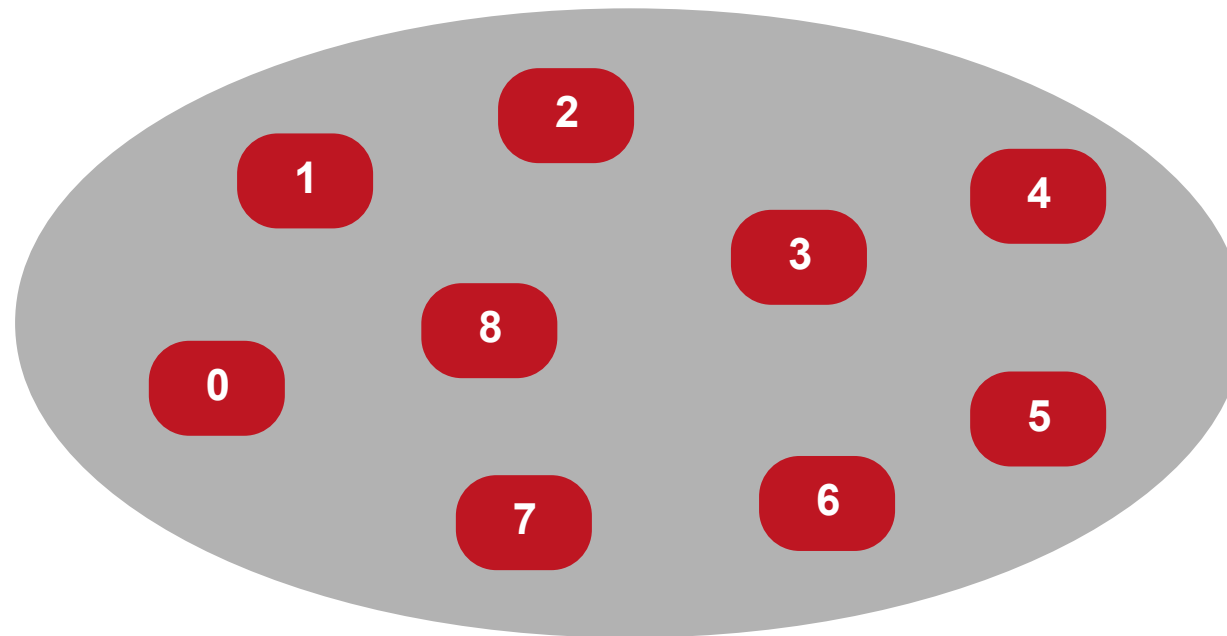
Demo code

- **heat.cpp** (C++) or **heat.f90** (Fortran 90)
- To run from a terminal:

<pre>> mpic++ heat.cpp</pre>	<pre>> mpif90 heat.f90</pre>
<pre>> mpirun -np 2 ./a.out</pre>	<pre>> mpirun -np 2 ./a.out</pre>
- On the maths network, the path is `/usr/lib64/openmpi/bin/`
- Main differences between MPI in C++ and Fortran:
 1. C++ is case sensitive (e.g. `MPI_Comm_size`).
 2. Fortran subroutines all have an extra output argument `ierr`.
 3. Special types like `MPI_Comm` are all just `integer` in Fortran.
 4. Some arguments need to be *references* in C++.
 5. Some defined constants like `MPI_DOUBLE` are different.

Initialising MPI

- **#include <mpi.h>** - include file, always required.
- **MPI_Init(NULL, NULL)** - always required, call first and only once.
- **MPI_Finalize()** - must be last MPI call.



- **MPI_Comm_size(MPI_COMM_WORLD, &nProcs)** - returns the number of processes in the global “communicator”.
- **MPI_Comm_rank(MPI_COMM_WORLD, &myRank)** - returns the “rank” of the current process. - *this tells you your identity.*

Creating a virtual topology

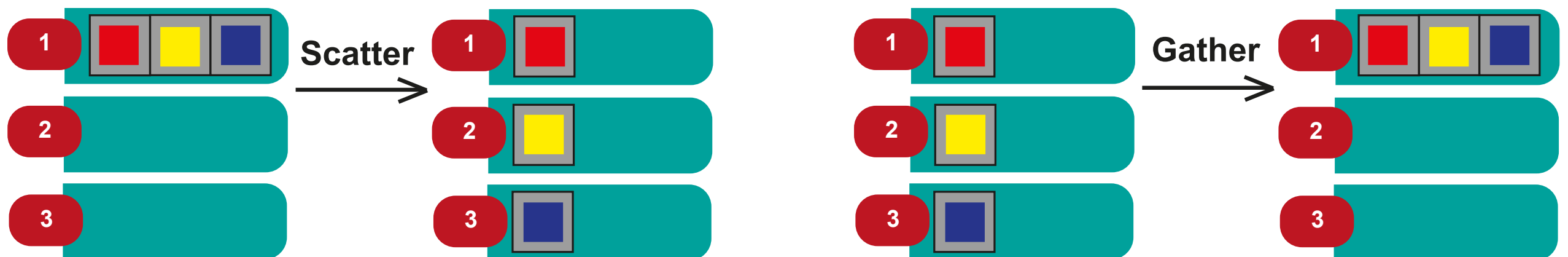
- A “communicator” is a group of processes that can communicate with each other. We use it to define a “virtual topology”.



- **`MPI_Cart_create(MPI_COMM_WORLD, n, dims, periods, reorder, &comm)`**
 - make a *Cartesian* communicator `comm` with `n` directions, where:
 - `dims` - number of processes in each direction;
 - `periods` - whether periodic boundaries in each direction;
 - `reorder` - whether ranking may be reordered.
- **`MPI_Comm_rank(comm, &myRank)`** - get rank *in this communicator*.
- **`MPI_Cart_shift(comm, dirn, displ, &prvRank, &nxtRank)`**
 - return the ranks of nearby processes in the coordinate direction `dirn`, at distance `displ` away.

Collective communications

- Each process needs different initial conditions: we create a global xGlob array in process 0, then distribute chunks to the others.
- **`MPI_Scatter`**(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)
 - root is the rank of the sender, sendbuf is the global array, recvbuf is the local one.



- **`MPI_Gather`**(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)
 - the inverse of `MPI_Scatter`: we use it at the end to output u.

Point-to-point communications

- **`MPI_Send(&buf, count, datatype, dest, tag, comm)`** - send from the buffer `buf` (with `count` values) to rank `dest`.
- **`MPI_Recv(&buf, count, datatype, source, tag, comm, &stat)`** - receive `count` values in the buffer `buf` from rank `source`.
- These are “blocking” operations:
 - `MPI_Send` returns when `buf` is ready for re-use. (This doesn't mean that the message has been received; it may be stored in a “system buffer”.)
 - `MPI_Recv` returns when the requested data has reached the receiving `buf` and can be used.

Point-to-point communications

- **`MPI_Send(&buf, count, datatype, dest, tag, comm)`** - send from the buffer `buf` (with `count` values) to rank `dest`.
- **`MPI_Recv(&buf, count, datatype, source, tag, comm, &stat)`** - receive `count` values in the buffer `buf` from rank `source`.
- a message can only be received if its `tag` (an integer) matches that in the `MPI_Recv` statement (or if it is set to `MPI_ANY_TAG`).
- the structure `stat` contains the source of the message and the tag: `stat.MPI_SOURCE` and `stat.MPI_TAG`.

Things to be aware of

- There is no requirement that each `MPI_Send` is matched by an equivalent `MPI_Recv` (or alternative command).
- There is *no overtaking*: if two messages are send from the same sender to the same destination, and both match an `MPI_Recv` statement, then the first message to be sent will be received first.
- If two processes A and B both send messages to process C that match a single receive statement, then *only one* of the sends will complete.
- A process can get *stuck* on an `MPI_Recv` if no corresponding message is sent.

Deadlock!

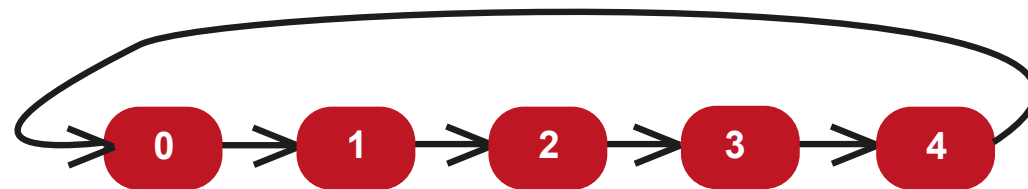
- It is possible to reach “deadlock” where two (or more) processes are each waiting for the other to send or receive.
- For example, suppose there is a *circular* topology and we call:

...

```
MPI_Send( &a, 1, MPI_INT, nxtRank, comm );
```

```
MPI_Recv( &b, 1, MPI_INT, prvRank, comm, &stat );
```

...

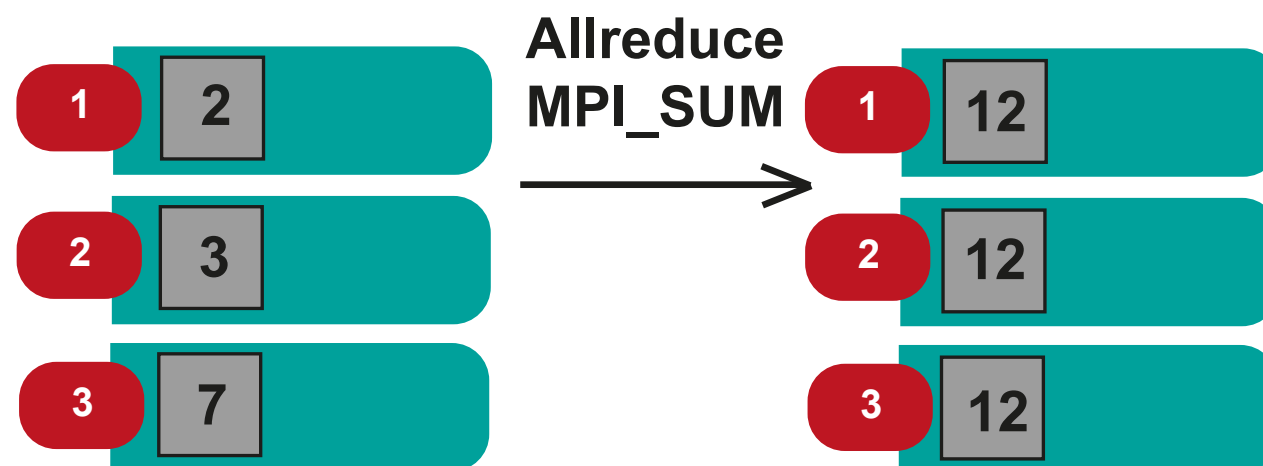


- This is unsafe; if the system buffer runs out of memory (not under our control), then all of the processes could be stuck waiting to send.
- We can see this if we use the “synchronous send” `MPI_Ssend`, which waits until the destination process has started to receive the message before returning.

Collective computations

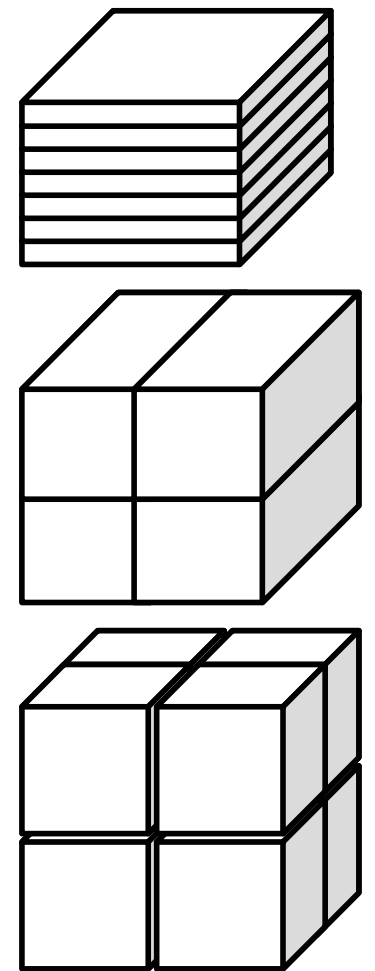
- MPI has routines for collective computation, or “reduction”.
- **`MPI_Allreduce`**(`&sendbuf`,`&recvbuf`,`count`,`datatype`,
`op`,`comm`)
 - perform the global operation `op` using `sendbuf` from each process, and broadcast the overall result to every `recvbuf`.
- Operations include `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, `MPI_MIN`, ...

$$\frac{d}{dt} \int_{-1}^1 u \, dx = \int_{-1}^1 k u_{xx} \, dx = k [u_x]_{-1}^1 = 0$$



Timing

- **MPI_Wtime()** - returns wall-clock time on the individual processor.
- **Improving speed/scaling with number of processors:**
 - increase fraction of program that can be parallelized;
 - balance the workload;
 - minimize communication time (decrease both amount of data and number of calls);
 - use collective communications wherever possible.



Further reading

- A useful tutorial and reference for other commands:
<https://computing.llnl.gov/tutorials/mpi/>
- Another tutorial (more advanced topics) with examples:
<http://mpitutorial.com>
- The official standard:
<http://mpi-forum.org/docs/docs.html>