

Parallelism beyond the node

Felice Pantaleo

CERN Experimental Physics Department

felice@cern.ch

Real-time feedback

- [click here](#)
- Typos, confused explanations, bad examples
- This is very important to ensure the best teaching standards!

Supercomputers

Sometimes:

- You are willing to sacrifice some efficiency for a faster solution
- After a certain amount of time, your solution becomes useless (e.g. climatology)
- The amount of data and parallelism does not fit in the memory of a single machine (cosmology, oil reservoir)
- It is too dangerous or too expensive to run an experiment, and simulating it requires huge amount of resources (weapons simulation for defense, fault simulations)

Efficiency loss? What are you talking about?

- The latency of the DRAM can be measured in tens of nanoseconds
- Sending a byte to a directly connected computer can take 2-3 orders of magnitude longer than DRAM, depending on the interconnect technology
- If you have to use Message Passing, try hard to minimize communication

MPI Basics

MPI

- MPI is a standard : <http://www.mpi-forum.org/>
 - Defines API for C, C++, Fortran77, Fortran90
- library with diverse functionalities:
 - Communication primitives (blocking, non-blocking)
 - Parallel I/O
 - RMA
 - neighborhood collectives

MPI

- A single program is executed with multiple instances, processes, on the same or different nodes
- These instances communicate via library calls for:
 - initialize, finalize, manage working groups/identifiers
 - direct point-to-point communication between two processes
 - collective communication

Processes

- Each process running its own instance of the program has access exclusively to its own data
- Two processes communicate by exchanging messages
- Processes have identifiers
- Function calls are used to send data from one process to another

Processes

Process 1

a=5

Send(a, 2)

Process 2

Processes

Process 1

a=5

Send(a, 2)

Process 2

Recv(b, 1)

b++

Processes

Process 1

a=5

Send(a, 2)

Process 2

Recv(b, 1)

b++

b is now 6

Single Program on Multiple Data

Process 1

```
if pid==1:  
    a=5  
    Send(a,2)  
else:  
    Recv(b,1)  
    b++
```

Process 2

```
if pid==1:  
    a=5  
    Send(a,2)  
else:  
    Recv(b,1)  
    b++
```

SPMD

- Every process runs the same program
- Each process has a unique identifier and runs the version of the program with that particular identifier
- Private data
- You usually run one process per socket/core depending on the parallelization strategy

Communicators

- Communicators provides a separate communication space.
- It is possible to treat a subset of processes as a communication universe: `MPI_COMM_WORLD`
- Two types of communicators:
 - Intra-communicator : a collection of processes that can send messages to each other and engage in collective communication operations.
 - Inter-communicator: are used for sending messages between processes belonging to disjoint intra-communicators.

Hello World

```
#include <mpi.h>
#include <iostream>
int main(int argc, char** argv) {
    MPI_Init(nullptr, nullptr);
    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // Get the rank of the process
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    std::cout << "Hello world from processor " << processor_name << " rank " << rank <<
    " of " << world_size << std::endl;
    MPI_Finalize();
}
```

Hello World

```
~ mpic++ mpi_hello_world.cpp -o mpi_hello_world
```

```
~ mpirun -np 16 ./mpi_hello_world
```

```
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 15 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 0 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 6 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 1 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 4 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 5 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 7 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 10 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 11 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 12 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 13 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 14 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 2 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 3 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 8 of 16
Hello world from processor hpc-200-06-07.cr.cnaf.infn.it rank 9 of 16
```


Exercise MPI Hello World

- Try it

Point-to-Point Communication

Messages

- In general, in order to be able to communicate using messages you need to fill in a header and a payload
- In MPI the header includes:
 - the id of the sender and receiver
 - the tag: the "subject" of the message
 - the datatype of the content
 - the number of elements of that datatype
 - the position of the first element to send/receive

Messages

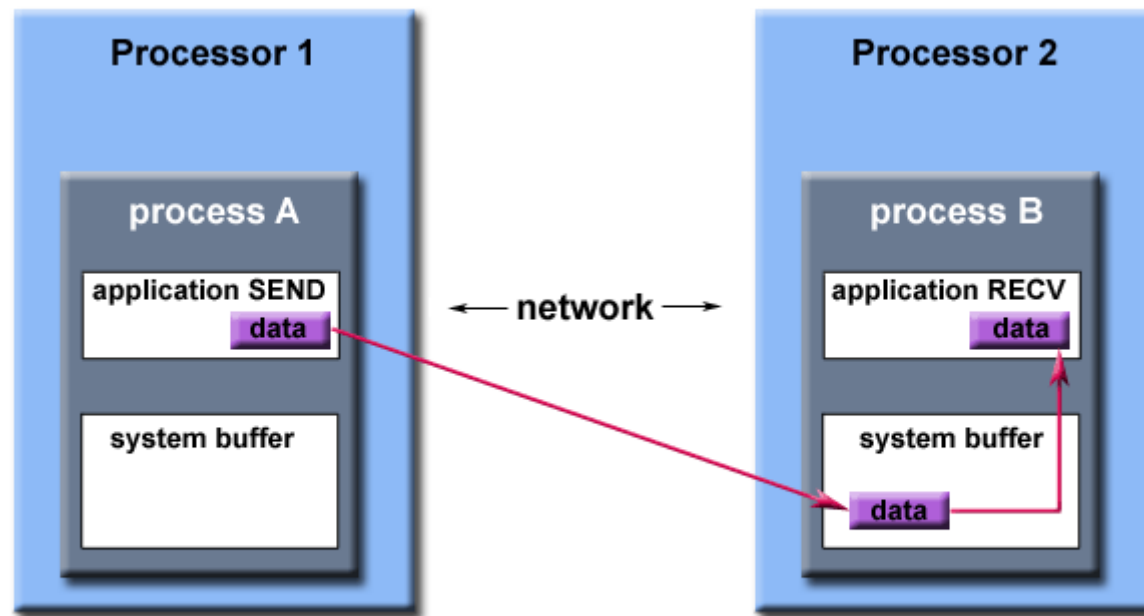
- If the sender waits for the message to be received, the communication is synchronous
- An asynchronous send returns immediately after the message has been sent
- Receiving is usually synchronous
- Messages have to match, otherwise deadlocks can occur

Data types

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

System buffer

- A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
- Opaque to the programmer and managed entirely by the MPI library
- Able to exist on the sending side, the receiving side, or both
- Allows asynchronous operations



Path of a message buffered at the receiving process

Blocking and non blocking communication

```
x = 0
```

```
MPI_Ssend(&x...)
```

```
..other work to do..
```

```
x = 0
```

```
MPI_Isend(&x..., req)
```

```
..other work to do..
```

```
MPI_Wait(..., req)
```

What's the difference?

Send a message! Example

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

int main(int argc, char *argv[]) {
    int numtasks, rank, dest, source, rc, count, tag=1;
    char inmsg, outmsg='x';
    MPI_Status Stat;    // required variable for receive routines
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```


Send a message! Example

```
// rank 0 sends to rank 1 and waits to receive a return message
if (rank == 0) {
    dest = 1;
    MPI_Ssend(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}

// rank 1 waits for rank 0 message then returns a message
else if (rank == 1) {
    source = 0;
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
}

MPI_Finalize();
}
```

Your turn now: Ping Pong

- Modify the previous example to send and receive a message:
 - rank 0 sends a message to rank 1.
 - once received, rank 1 sends the same message to rank 0
- Measure time between a send and receive (ping)
- Try to run it on many iterations such that the total time is between 1s and 10s
- Measure bandwidth and investigate how it changes with a varying message size
- time can be measured with:

```
double MPI_Wtime( )
```

Blocking ping pong exercise

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

Int main() {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;    // required variable for receive routines

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// task 0 sends to task 1 and waits to receive a return message
if (rank == 0) {
    dest = 1;
    source = 1;
    MPI_Ssend(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
}
```

Blocking ping pong exercise

```
// task 1 waits for task 0 message then returns a message
else if (rank == 1) {
    dest = 0;
    source = 0;
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    MPI_Ssend(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}

// query receive Stat variable and print message details
MPI_Get_count(&Stat, MPI_CHAR, &count);
std::cout << "Rank " << rank << " Received " << count << " char(s) from
rank " << Stat.MPI_SOURCE << " with tag " << Stat.MPI_TAG << std::endl;

MPI_Finalize();
}
```

The blocking ring exercise

- Write an MPI program in which each process sends its `rankId` to its neighbors `rankId+1` and `rankId-1`
- Close the ring by making the last `rankId` communicate with the `rankId=0`
- Measure the time for 1000 iterations and a variable number of processes

Non-Blocking ring exercise

```
#include "mpi.h"

int main() {
    int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
    constexpr int nRequests = 4;
    MPI_Request reqs[nRequests]; // required variable for non-blocking calls
    MPI_Status stats[nRequests]; // required variable for Waitall routine

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // determine left and right neighbors
    prev = rank-1;
    next = rank+1;
    if (rank == 0) prev = numtasks - 1;
    if (rank == (numtasks - 1)) next = 0;
```

Non-Blocking ring exercise

```
// post non-blocking receives and sends for neighbors
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

// do some work while sends/receives progress in background

// wait for all non-blocking operations to complete
// MPI_Waitall (count,&array_of_requests,&array_of_statuses)
MPI_Waitall(nRequests, reqs, stats);

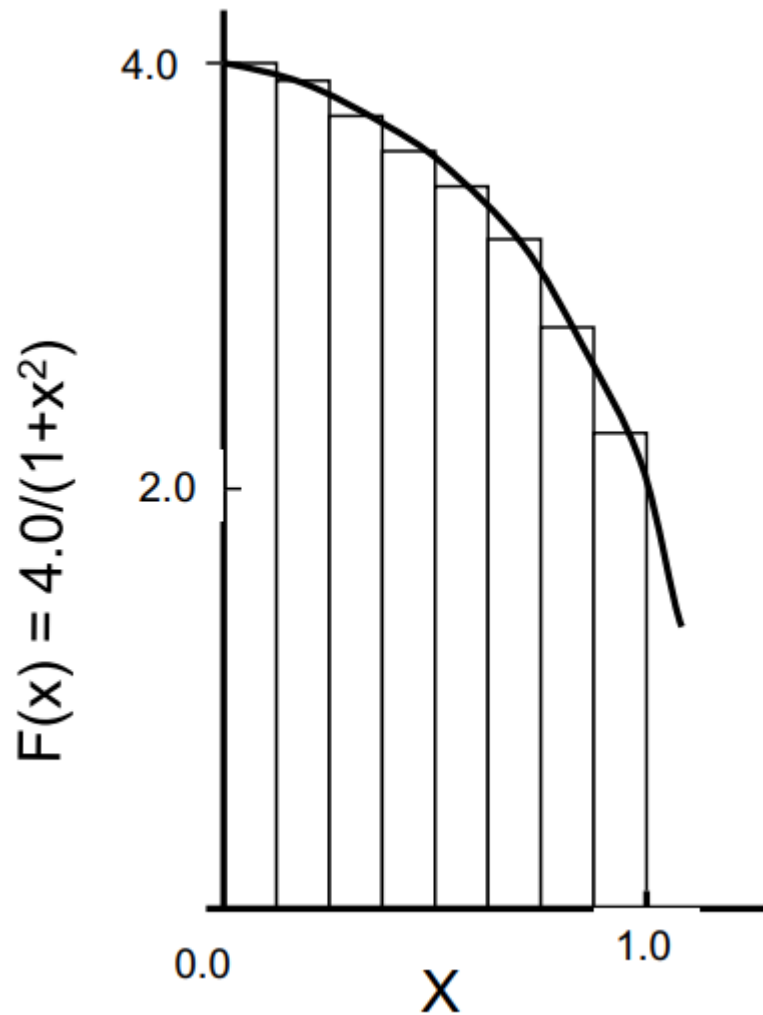
// continue - do more work

MPI_Finalize();
}
```

The non-blocking ring exercise

- Modify the previous program in order to use non-blocking communication
- Measure the time for 1000 iterations and a variable number of processes
- Do you notice any speed-up?

Pi



We know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

- The integral can be approximated as the sum of rectangles:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Numerical integration

```
constexpr int num_steps = 1<<20;
double pi = 0.;
constexpr double step = 1.0/(double) num_steps;
double sum = 0.;

for (int i=0; i< num_steps; i++){
    auto x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;

std::cout << "result: " << std::setprecision (15) <<
pi << std::endl;
```

Numerical integration

- Modify your Hello World program so that each process independently computes the value of π and prints it to the screen.
- Choose a number of steps per process and try to parallelize it using MPI
- Every process sends its partial result to rank 0
- rank 0 executes the final sum
- Make sure everything works even if the number of steps is not multiple of the number of processes
- Compare timing with same number of threads as processes in `tbb/std::threads`

Probe before receiving

If you don't want to allocate the maximum possible amount of memory for the receiving buffer you can use `MPI_Probe`

```
MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status* status)
```

It will ask for the properties of the incoming message without receiving it:

```
MPI_Probe(0, 0, MPI_COMM_WORLD, &status);  
MPI_Get_count(&status, MPI_INT, &number_amount);  
std::vector<int> number_buf (number_amount);  
MPI_Recv(number_buf.data(), number_amount, MPI_INT, 0, 0,  
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Collective Communication

Collective communication/synchronization

- A message can be sent to/received from a group of processes
 - Broadcast, scatter, gather, reduce
- A group of processes can synchronize
 - Achieved by means of barriers
 - A process in the group has to wait for **all** the other processes in the group before it can start executing the next line of code
 - Usually needed for timing, not for correctness
- Use collective communication when possible
 - they are implemented more efficiently than the sum of their point-to-point equivalent calls

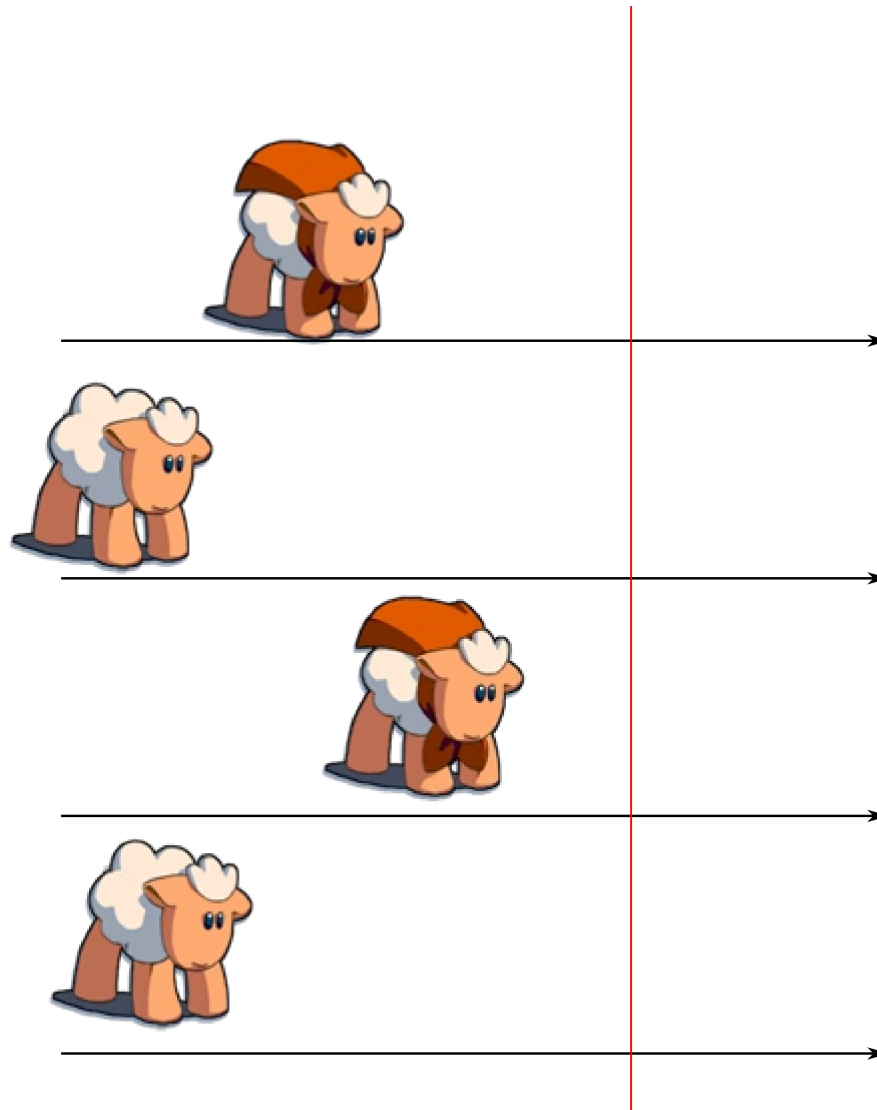
Barrier

`MPI_Barrier(MPI_Comm communicator)`



Barrier

`MPI_Barrier(MPI_Comm communicator)`



Barrier

`MPI_Barrier(MPI_Comm communicator)`



Barrier

`MPI_Barrier(MPI_Comm communicator)`

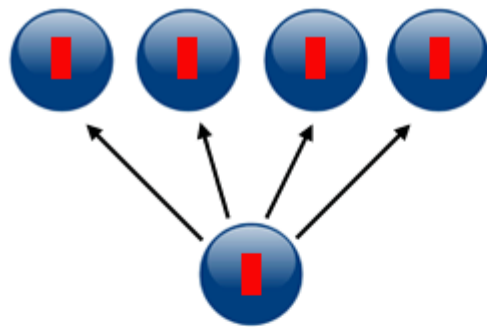


Barrier

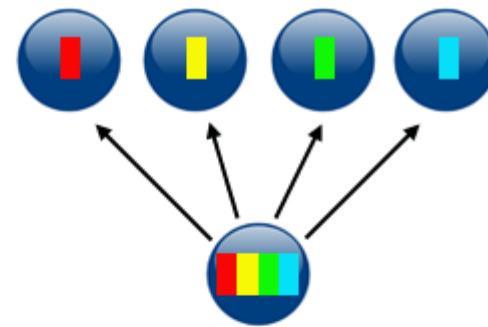
`MPI_Barrier(MPI_Comm communicator)`



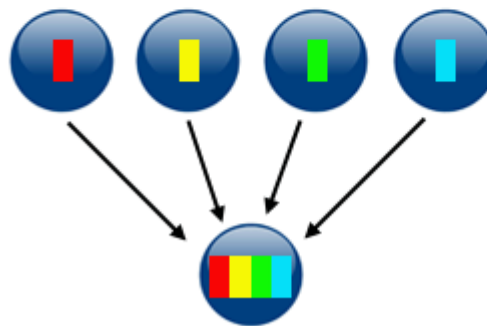
Collective communication



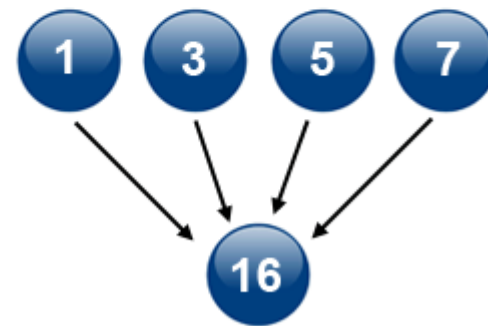
broadcast



scatter



gather



reduction

Collective communication

```
MPI_Bcast(  
    void* data,  
    int count,  
    MPI_Datatype datatype,  
    int root,  
    MPI_Comm communicator)
```

```
MPI_Gather(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    int root,  
    MPI_Comm communicator)
```

```
MPI_Scatter(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    int root,  
    MPI_Comm communicator)
```

```
MPI_Reduce(  
    void* send_data,  
    void* recv_data,  
    int count,  
    MPI_Datatype datatype,  
    MPI_Op op,  
    int root,  
    MPI_Comm communicator)
```

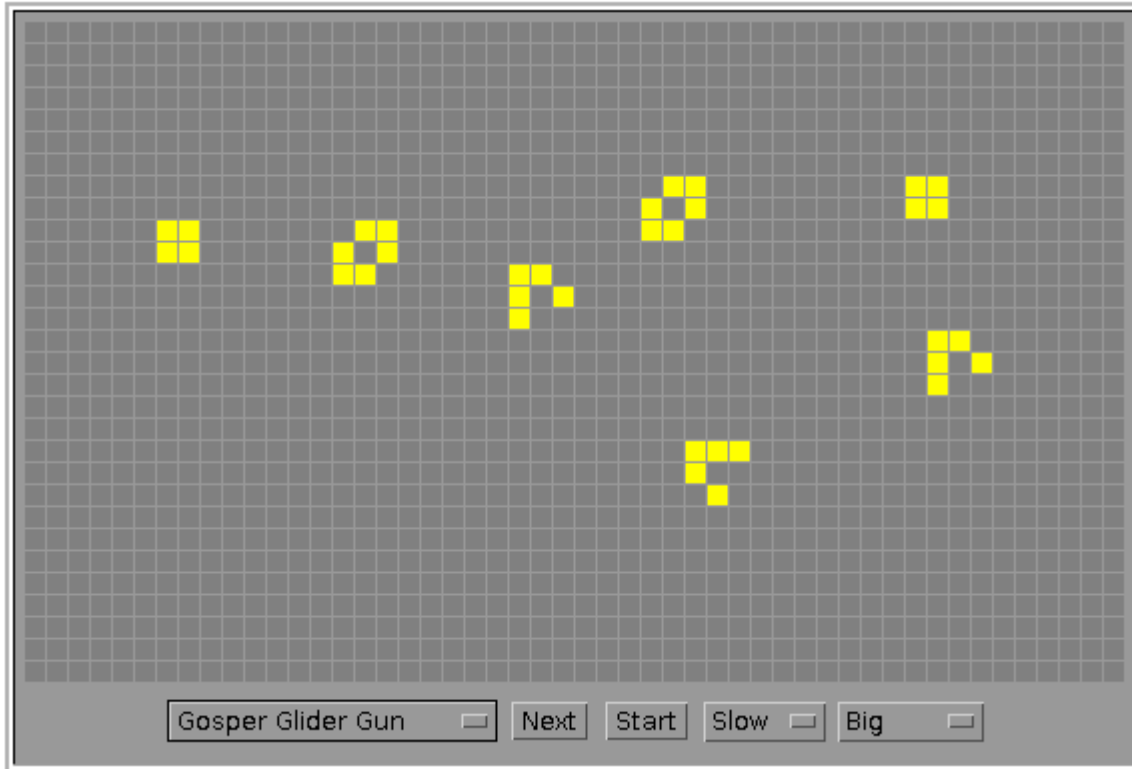
Reduce operations

- `MPI_MAX` - Returns the maximum element.
- `MPI_MIN` - Returns the minimum element.
- `MPI_SUM` - Sums the elements.
- `MPI_PROD` - Multiplies all elements.
- `MPI_LAND` - Performs a logical and across the elements.
- `MPI_LOR` - Performs a logical or across the elements.
- `MPI_BAND` - Performs a bitwise and across the bits of the elements.
- `MPI_BOR` - Performs a bitwise or across the bits of the elements.
- `MPI_MAXLOC` - Returns the maximum value and the rank of the process that owns it.
- `MPI_MINLOC` - Returns the minimum value and the rank of the process that owns it.

Exercise Pi

- Modify the numerical integration exercise to use the collective reduction

Exercise Game of Life



- Cellular Automaton
- Any live cell with fewer than two live neighbours dies
- Any live cell with more than three live neighbours dies
- Any live cell with two or three live neighbours lives, unchanged, to the next generation.
- Any dead cell with exactly three live neighbours will come to life.
- Borders should be treated as portals

Final MPI exercise - Game of Life

- p processors
- board $N \times M$ booleans (x and o)
- initially the master sends a piece of the board to each processor
- each processor computes its CA and exchanges borders information with neighboring processors
- at each m steps, the master gathers the entire board and prints it on screen (x and o)