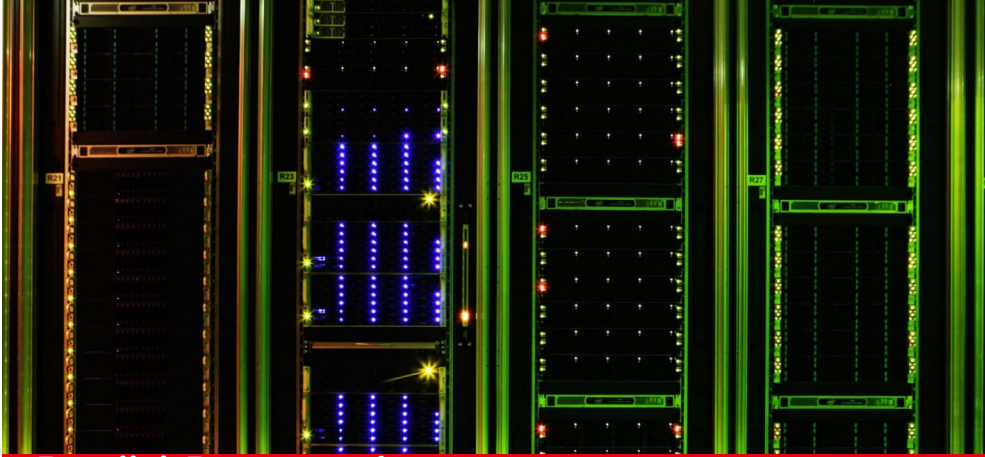


Parallel Programming

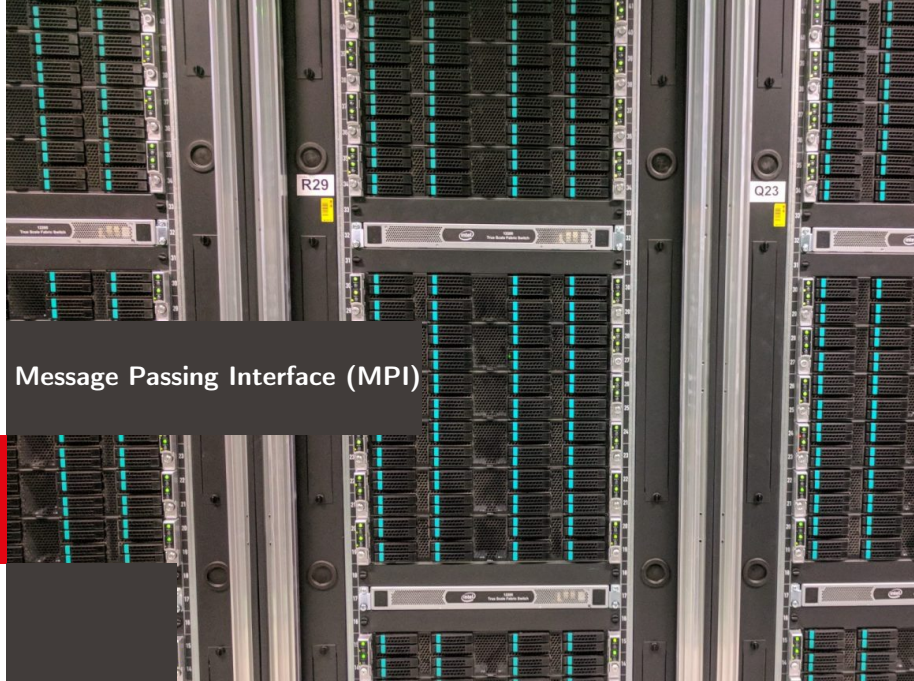
Single-core optimization, MPI, OpenMP, and hybrid programming

Nicolas Richart
Emmanuel Lanti

Course based on V. Keller's lecture notes

27th of October - 1st of November 2024

Message Passing Interface (MPI)



- Introduce distributed memory programming paradigm
- Point-to-point communications
- Collective communications

- MPI is a *Message-Passing Interface* specification
- There are many implementations (MPICH, MVAPICH, Intel MPI, OpenMPI, etc)
- Library interface, not a programming language
- It is standardized
  - Defined by the MPI forum
  - Current version is MPI 4.0
- As such, it is portable, flexible and efficient
- Interface to C and Fortran in standard

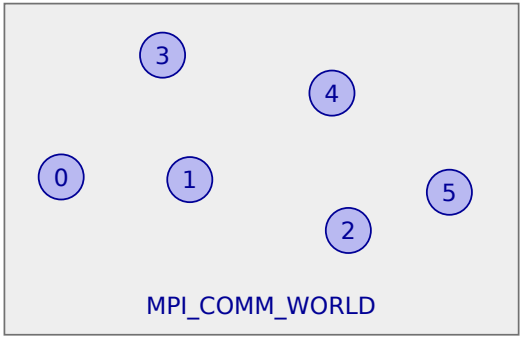
MPI

A simple hello world example

mpi/hello_mpi.cc

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

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);

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

  MPI_Finalize();
  return 0;
}
```

Environment



- MPI code is bordered by a `MPI_Init` and a `MPI_Finalize`
- MPI starts N processes numbered $0, 1, ..., N-1$. It is the process *rank*
- They are grouped in a *communicator* of *size* N
- After init, MPI provides a default communicator called **MPI_COMM_WORLD**

Hello world π

- In the *pi* code initialize/finalize properly MPI
- Print out the number of processes and the rank of each process
- Modify the makefile to use **mpicxx** instead of **g++**
- Write a batch script to run your parallel code

```
#!/bin/bash
#SBATCH -n <ntasks>
module purge
module load <compiler> <mpi library>
srun <my_mpi_executable>
```

Note : *To use MPI on the cluster you first have to load a MPI implementation through the module* **openmpi** *or* **intel-oneapi-mpi**.

Types of communications in MPI

- Point-to-Point (One-to-One)
- Collectives (One-to-All, All-to-One, All-to-All)
- One-sided/Shared memory (One-to...)
- Blocking and Non-Blocking of all types

- Let's derive a minimal message-passing interface
- The goal is to define an interface for **send** and **recv**

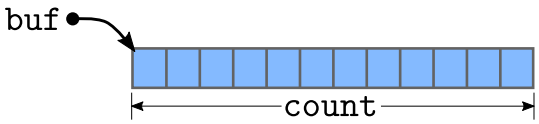
Send/Receive

Syntax

```
1 int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest,
2 int tag, MPI_Comm comm);
3
4 int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
5 int tag, MPI_Comm comm, MPI_Status *status);
```

- **buf** pointer to the data to send/receive
- **count** number of element to send/receive
- **datatype** datatype of the data to send/receive
- **dest**, **source** the rank of the destination/source of the communication
- **tag** a message tag to differentiate the communications
- **comm** communicator in which to communication happens
- **status** object containing information on the communication

- Buffer is a pointer to the first data (`buf`), a size (`count`) and a `datatype`
- Datatypes (extract):
  - `MPI_INT`
  - `MPI_UNSIGNED`
  - `MPI_FLOAT`
  - `MPI_DOUBLE`



- For `std::vector<double> vect` :
  - `buf = vect.data()`
  - `count = vect.size()`
  - `datatype = MPI_DOUBLE`

Send/Receive

Useful constants and status

Constants:

- `MPI_STATUS_IGNORE` to state that the status is ignored
- `MPI_PROC_NULL` placeholder for the source or destination
- `MPI_ANY_SOURCE` is a wildcard for the source of a receive
- `MPI_ANY_TAG` is a wildcard for the tag of a receive

Status:

- Structure containing `tag` and `source`

```
1 MPI_Status status;
2 std::cout << "Tag: " << status.tag << " - "
3 << "Source: " << status.source << std::endl;
```

- Size of the message can be asked using the status

```
1 int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype,
2 int *count);
```

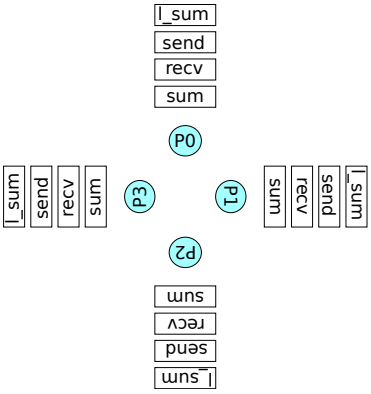
Send/Receive

Example

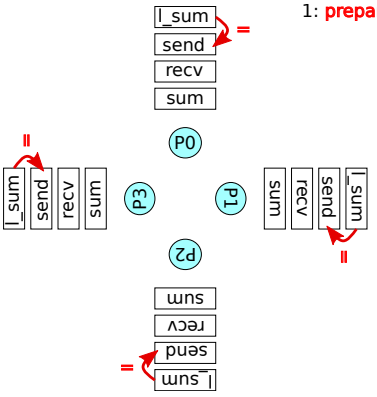
mpi/send_recv.cc

```
16 MPI_Init(NULL, NULL);
17 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18 MPI_Comm_size(MPI_COMM_WORLD, &size);
19
20 assert(size == 2 && "Works only with 2 procs");
21
22 if (rank == 0) {
23   fill_buffer(buf);
24   MPI_Ssend(buf.data(), buf.size(), MPI_INT, 1, 0, MPI_COMM_WORLD);
25 } else if (rank == 1) {
26   MPI_Recv(buf.data(), buf.size(), MPI_INT, 0, 0, MPI_COMM_WORLD,
27            MPI_STATUS_IGNORE);
28 }
29
```

Ring reduction of π



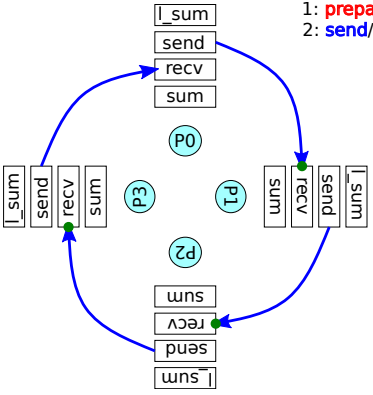
- `l_sum = local_computation();`
  `sum += l_sum;`



1: prepare comms

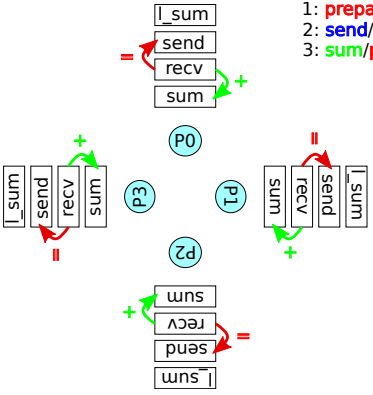
- `send_buf = l_sum;`

Ring reduction of π



- `send(send_buf);`
  `receive(recv_buf);`

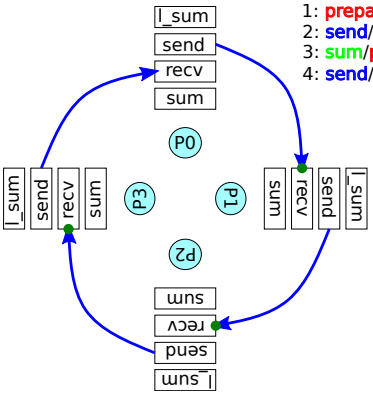
Ring reduction of π



1: prepare comms
2: send/recv
3: sum/prepare

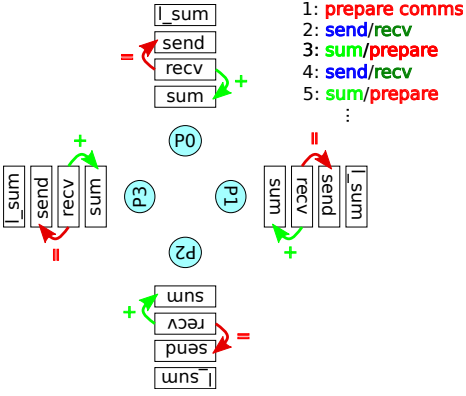
- `send_buf = recv_buf;`
  `sum += recv_buf;`

Ring reduction of π




```
send(send_buf);
receive(recv_buf);
```

Ring reduction of π



- Split the sum space between the processes
- Implement a ring to communicate the partial sum between the processes. using `MPI_Ssend` and `MPI_Recv`

  Remember : *each MPI process runs the same code!*

  Note : *in a loop the next process is* `(prank + 1) % psize` *and the previous is* `(prank - 1 + psize) % psize`

- `MPI_Ssend` : (S for Synchronous) function returns when other end posted matching recv and the buffer can be safely reused
- `MPI_Bsend` : (B for Buffer) function returns immediately, send buffer can be reused immediately
- `MPI_Rsend` : (R for Ready) can be used only when a receive is already posted
- `MPI_Send` : acts like `MPI_Bsend` on small arrays, and like `MPI_Ssend` on bigger ones

Send/Receive

Particularity of `MPI_Send`

mpi/ping_ping.cc

```
31 auto size = 1 << n;
32 fill_buffer(buf, size);
33
34 auto t_start = clk::now();
35 for (size_t repetition = 0; repetition < REP; ++repetition) {
36   MPI_Send(buf.data(), buf.size(), MPI_INT, partner, 0, MPI_COMM_WORLD);
37   MPI_Recv(buf.data(), buf.size(), MPI_INT, partner, 0, MPI_COMM_WORLD,
38            MPI_STATUS_IGNORE);
39 }
40 auto time_s = (second{clk::now() - t_start}).count() / REP;
```

Send/Receive

Particularity of `MPI_Send`

mpi/ping_ping.cc

```
31 auto size = 1 << n;
32 fill_buffer(buf, size);
33
34 auto t_start = clk::now();
35 for (size_t repetition = 0; repetition < REP; ++repetition) {
36   MPI_Send(buf.data(), buf.size(), MPI_INT, partner, 0, MPI_COMM_WORLD);
37   MPI_Recv(buf.data(), buf.size(), MPI_INT, partner, 0, MPI_COMM_WORLD,
38            MPI_STATUS_IGNORE);
39 }
40 auto time_s = (second{clk::now() - t_start}).count() / REP;
```

```
$ mpirun -np 2 ./mpi/ping_ping
PingPing size:       4.0B time:    303.5ns bandwidth:  12.6MiB/s
PingPing size:      16.0B time:    291.0ns bandwidth:  52.4MiB/s
PingPing size:      64.0B time:    289.7ns bandwidth: 210.7MiB/s
PingPing size:     256.0B time:    327.9ns bandwidth: 744.5MiB/s
PingPing size:     1.0KiB time:    686.9ns bandwidth:   1.4GiB/s
```

Syntax

```
1 int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
2 int dest, int sendtag,
3 void *recvbuf, int recvcount, MPI_Datatype recvtype,
4 int source, int recvtag,
5 MPI_Comm comm, MPI_Status *status);
```

- Combines a send and a receive, to help mitigate deadlocks
- Has a in-place variant `MPI_Sendrecv_replace`

Ring reduction of π

Using MPI_Sendrecv

- Modify the previous exercise to use `MPI_Sendrecv`

Non-blocking send/receive

Syntax

```
1 int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,
2 int tag, MPI_Comm comm, MPI_Request *request);
3
4 int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,
5 int tag, MPI_Comm comm, MPI_Request *request);
```

- **I** for *immediate*
- `request` in addition to parameters from blocking version
- receive does not have a status
- `request` is an object attached to the communication
- the communications starts but is not completed
- **S**, **B**, and **S** variant are also defined

Non-blocking send/receive

Completion

Syntax

```
1 int MPI_Wait(MPI_Request *request, MPI_Status *status);
2
3 int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

- completion of communication should be checked
- `MPI_Test` or `MPI_Wait`
- send completed means the buffer can be reused
- receive completed means the buffer can be read
- `status` is set a completion
- `flag` is **true** if completed, **false** otherwise

mpi/sendrecv.cc

```
22 MPI_Request request;
23 if (rank == 0) {
24   fill_buffer(buf);
25   MPI_Isend(buf.data(), buf.size(), MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
26 } else if (rank == 1) {
27   MPI_Recv(buf.data(), buf.size(), MPI_INT, 0, 0, MPI_COMM_WORLD,
28            MPI_STATUS_IGNORE);
29 }
30 // here I can do computation as long as buf is not modified
31 MPI_Wait(&request, MPI_STATUS_IGNORE);
```

Ring reduction of π

Using non-blocking send

- Modify the previous exercise to use `MPI_Isend` and `MPI_Recv`
- Do not forget to **wait**

Non-blocking send/receive

Multiple completions

- `MPI_Waitall`, `MPI_Testall` wait or test completion of all the pending requests
- `MPI_Waitany`, `MPI_Testany` wait or test completion of one out on many
- `MPI_Waitsome`, `MPI_Testsome` wait or test completion of all the enabled requests
- for arrays of statuses can use `MPI_STATUSES_IGNORE`
- `MPI_Request_get_status` equivalent to `MPI_Test` but does not free completed requests

Probing

Syntax

```
1 int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
2 MPI_Status *status);
3
4 int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
```

- check incoming messages without receiving
- Immediate variant returns **true** if matching message exists

Syntax

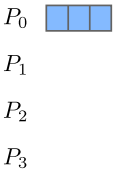
```
1 int MPI_Barrier(MPI_Comm comm);
```

- collective communications **must** be called by all processes in the communicator
- barrier is hard synchronization
- avoid as much a possible

Syntax

```
1 int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
2 MPI_Comm comm);
```

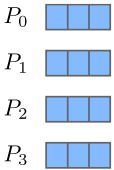
- the **root** process sends data to every other process



Syntax

```
1 int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
2 MPI_Comm comm);
```

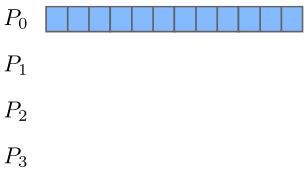
- the **root** process sends data to every other process



Syntax

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm);
```

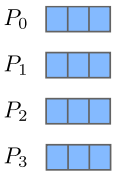
- the **root** process sends a piece of the data to all processes
- the `sendbuf`, `sendcount` and `sendtype` are only relevant on the **root**



Syntax

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm);
```

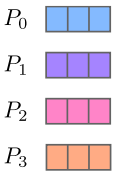
- the **root** process sends a piece of the data to all processes
- the `sendbuf`, `sendcount` and `sendtype` are only relevant on the **root**



Syntax

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm);
```

- all process their data to the **root** process
- the `recvbuf`, `recvcount` and `recvtype` are only relevant on the **root**
- `recvcount` is the size per process not the total size



Syntax

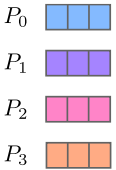
```
1 int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
2 void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
3 MPI_Comm comm);
```

- all process their data to the **root** process
- the `recvbuf`, `recvcount` and `recvtype` are only relevant on the **root**
- `recvcount` is the size per process not the total size

Syntax

```
1 int MPI_Allgather(const void *sendbuf, int sendcount,
2 MPI_Datatype sendtype, void *recvbuf, int recvcount,
3 MPI_Datatype recvtype, MPI_Comm comm);
```

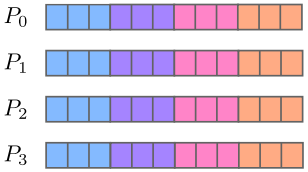
- all process send their data to all other process



Syntax

```
1 int MPI_Allgather(const void *sendbuf, int sendcount,
2 MPI_Datatype sendtype, void *recvbuf, int recvcount,
3 MPI_Datatype recvtype, MPI_Comm comm);
```

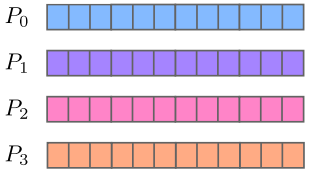
- all process send their data to all other process



Syntax

```
1 int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
2 void *recvbuf, int recvcount, MPI_Datatype recvtype,
3 MPI_Comm comm);
```

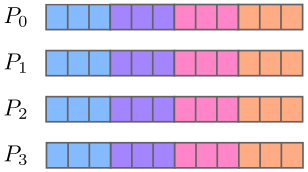
- all process send their a piece of their data to all other process



Syntax

```
1 int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
2 void *recvbuf, int recvcount, MPI_Datatype recvtype,
3 MPI_Comm comm);
```

- all process send their a piece of their data to all other process



Ring reduction of π

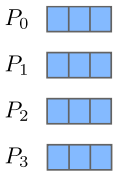
Using collective communications

- `MPI_Gather` the partial sums to the root process.
- `MPI_Bcast` the total sum all the process

Syntax

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

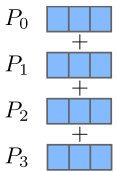
- data from all process are reduced on the **root** process
- common operations being `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`
- a `MPI_Allreduce` variant exists where all the process have the results
- `MPI_IN_PLACE` can be passed in the `sendbuf` of **root** for a *reduce* a of all process for a *allreduce*



Syntax

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

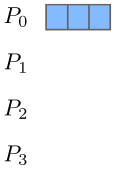
- data from all process are reduced on the **root** process
- common operations being `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`
- a `MPI_Allreduce` variant exists where all the process have the results
- `MPI_IN_PLACE` can be passed in the `sendbuf` of **root** for a *reduce* a of all process for a *allreduce*



Syntax

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

- data from all process are reduced on the **root** process
- common operations being `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`
- a `MPI_Allreduce` variant exists where all the process have the results
- `MPI_IN_PLACE` can be passed in the `sendbuf` of **root** for a *reduce* a of all process for a *allreduce*



Ring reduction of π

Using collective communications

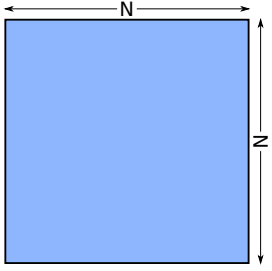
- Modify the previous exercise to use `MPI_Reduce` and `MPI_Bcast`
- Modify it again to use `MPI_Allreduce`

Syntax

```
1 int MPI_Scan(const void *sendbuf, void *recvbuf, int count,
2 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
3
4 int MPI_Exscan(const void *sendbuf, void *recvbuf, int count,
5 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

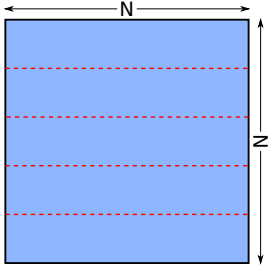
- performs the prefix reduction on data
- `MPI_Scan` on process i contains the reduction of values from processes $[0, i]$
- `MPI_Exscan` on process i contains the reduction of values from processes $[0, i[$
- `MPI_IN_PLACE` can be passed a `sendbuf`

Parallelization of the poisson code



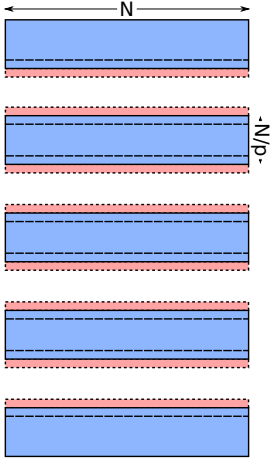
- Parallelize the Poisson 2D problem using the Messages Passing Interface (MPI)

Parallelization of the poisson code



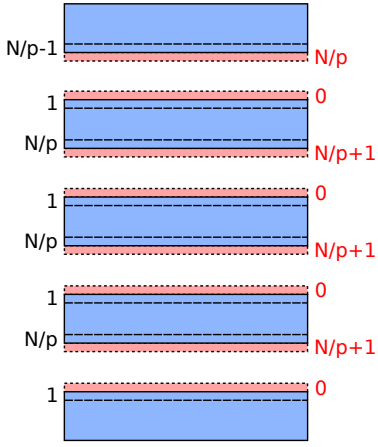
- The memory allocation is done in the C default manner, "Row-Major Order": make your domain decomposition by lines

Parallelization of the poisson code



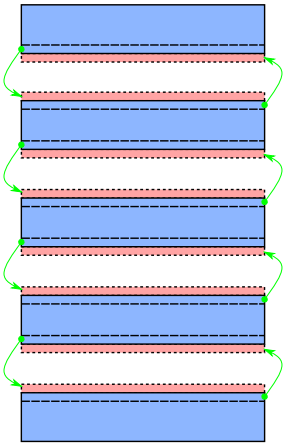
- p domains of size N/p each (1 per process)

Parallelization of the poisson code



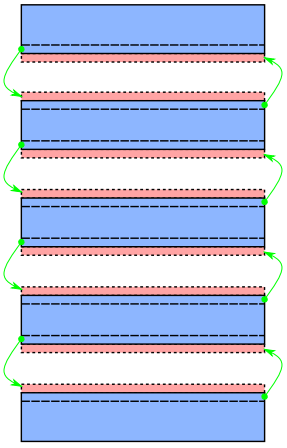
- Adding *ghost* lines before and after

Parallelization of the poisson code



- Use the *ghost* lines to receive the missing local data

Parallelization of the poisson code



- Start using `MPI_Sendrecv` to implement the communications
- You can use the number of iteration as a check
- Remove the `dump()` function to start
- Once it is working try to use *non-blocking* communications