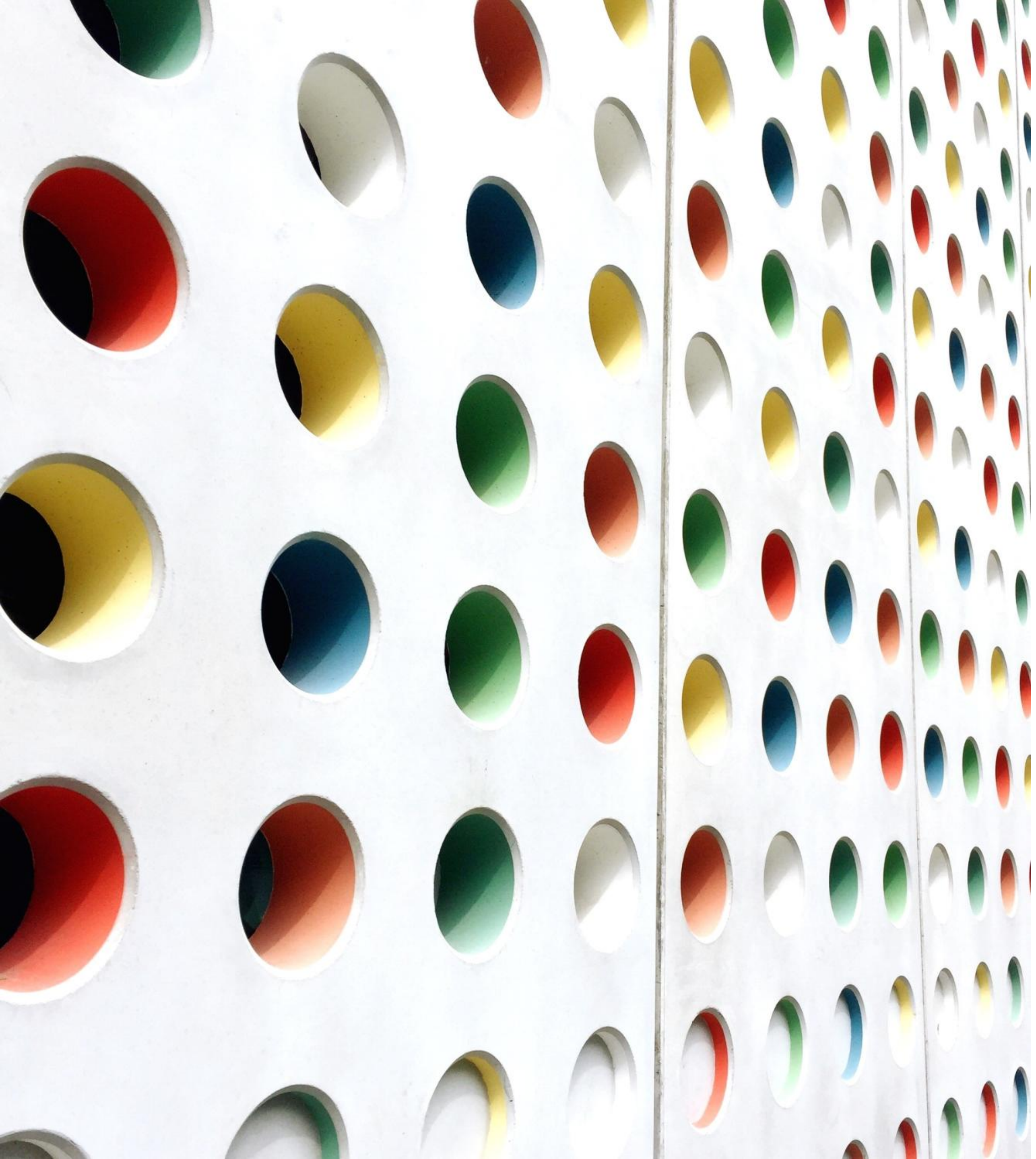


Parallelism beyond the node: Introduction to MPI Programming

DANIELE CESINI – INFN-CNAF

FELICE PANTALEO – CERN

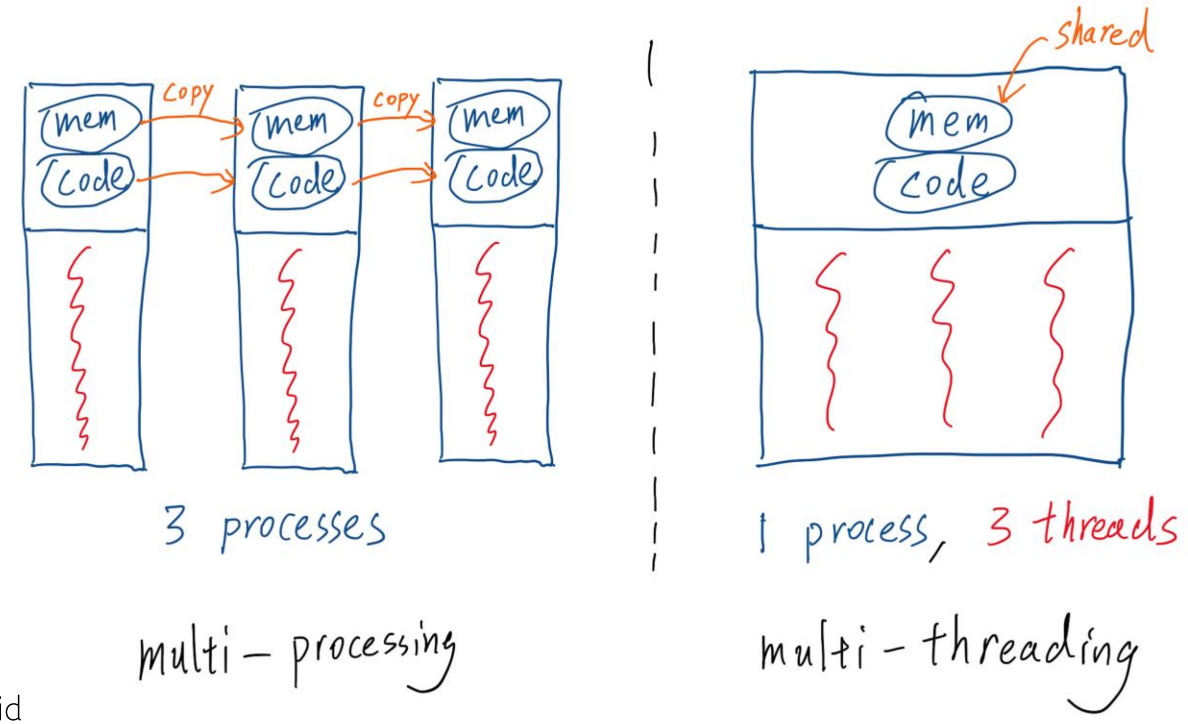
A lot of material from Tim Mattson's
"Hands-on" Introduction to MPI" at ESC15

Multithread vs Multiprocess

- Multithreading and multiprocessing are two ways to achieve multitasking
- A process has its own memory
- A thread shares the memory with the parent process and other threads within the process.
- pid is process identifier; tid is thread identifier
  - (*)But as it happens, the kernel doesn't make a real distinction between them: threads are just like processes but they share some things (memory, fds...) with other instances of the same group

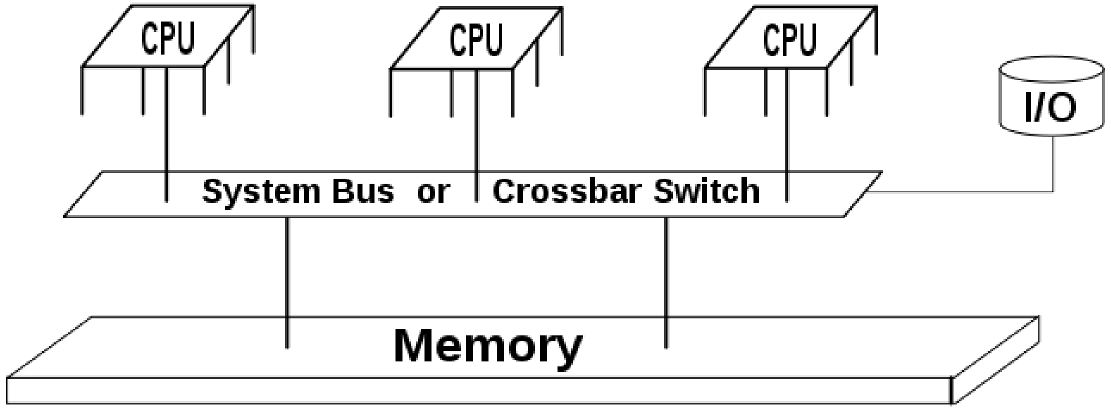
(*)https://stackoverflow.com/questions/4517301/difference-between-pid-and-tid

- Inter-process communication is slower due to isolated memory



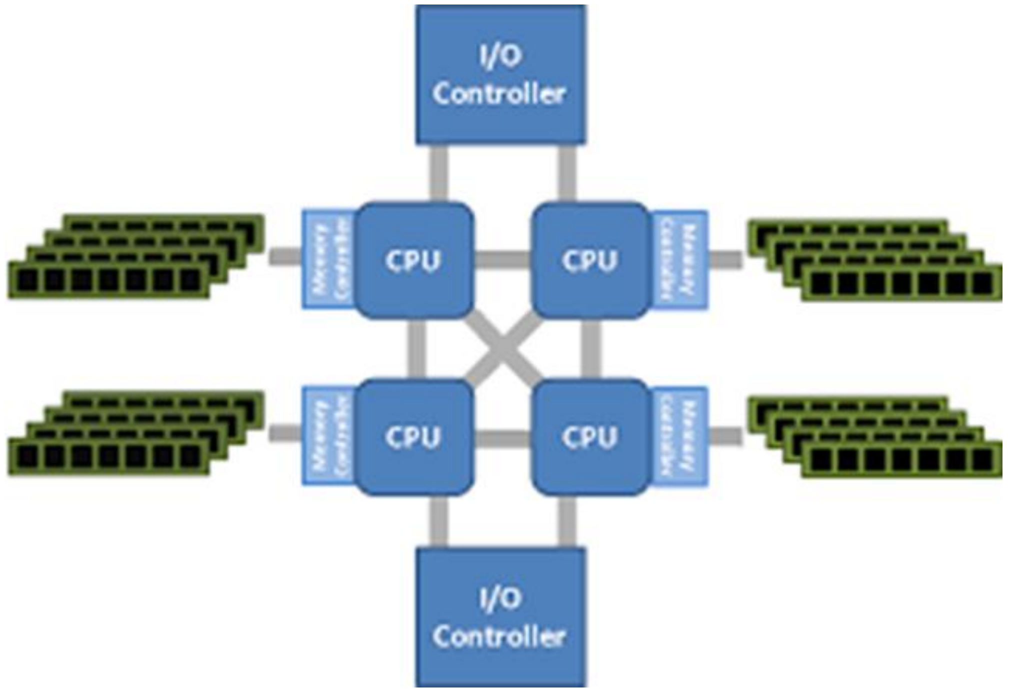
Shared Memory Systems

- Shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies
- Shared memory is an efficient means of passing data between programs
- Shared memory systems may use uniform memory access (UMA): all the processors share the physical memory uniformly
- Non-uniform memory access (NUMA): memory access time depends on the memory location relative to a processor



NUMA Architecture Programming

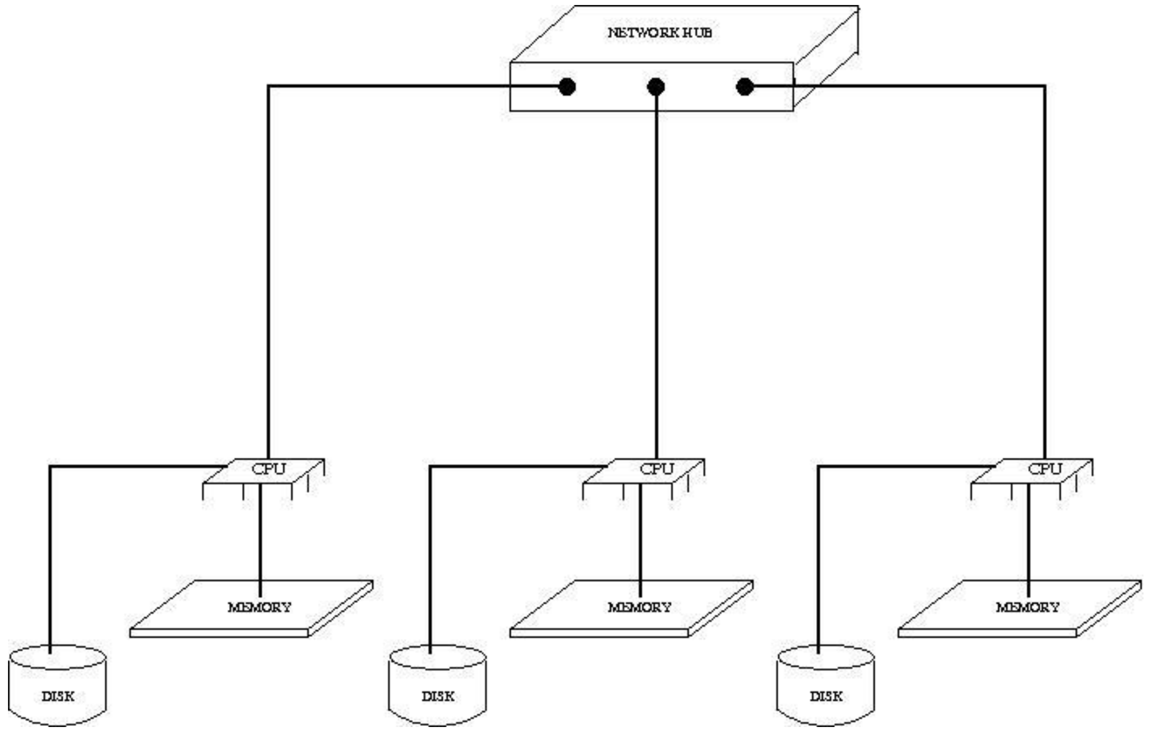
- A programmer can set an allocation policy for its program using a component of NUMA API called libnuma.
  - a user space shared library that can be linked to applications
  - provides explicit control of allocation policies to user programs.
- The NUMA execution environment for a process can also be set up by using the numactl tool
- Numactl can be used to control process mapping to cpuset and restrict memory allocation to specific nodes without altering the program's source code



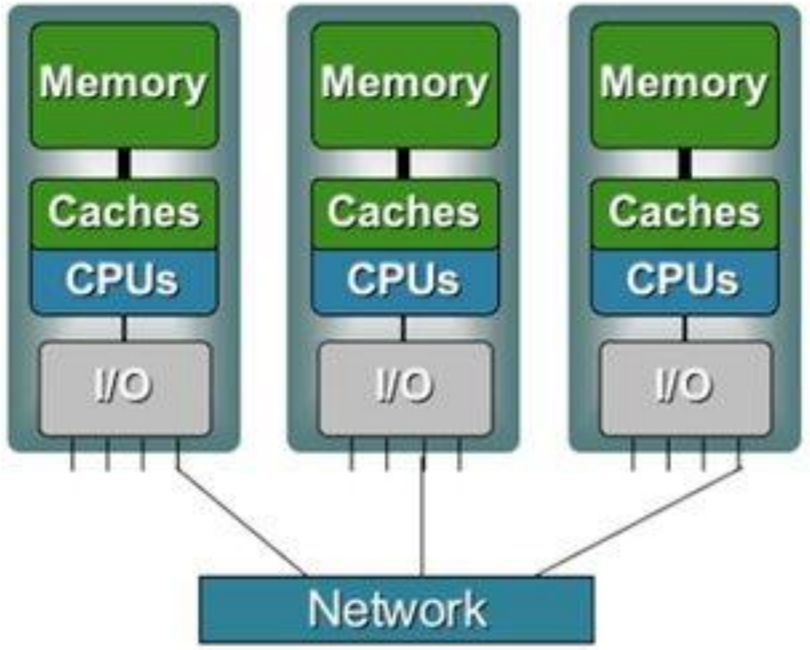
http://halobates.de/numaapi3.pdf

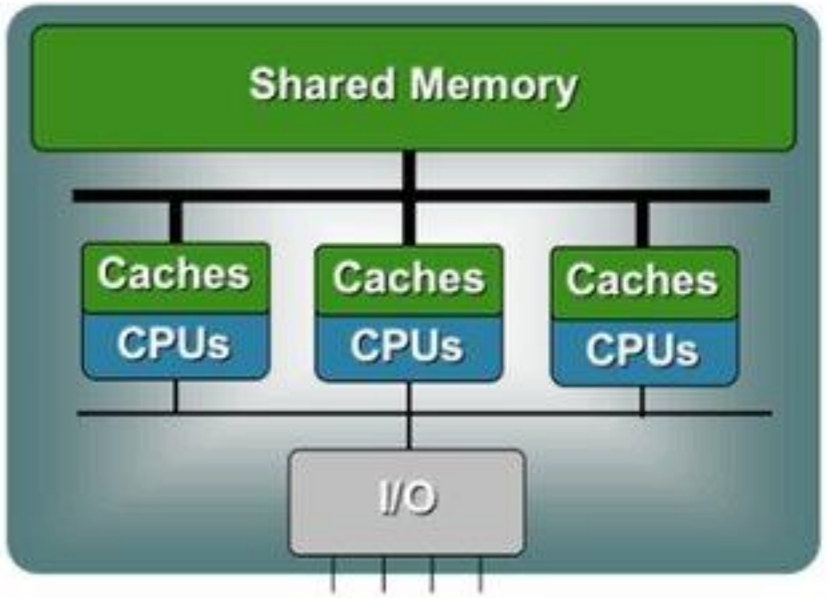
Distributed Memory Systems

- Distributed memory refers to a multiprocessor computer system in which each processor has its own private memory
- Computational tasks can only operate on local data
- if remote data is required, the computational task must communicate with one or more remote processors
- In contrast, a shared memory multiprocessor offers a single memory space used by all processors



Shared vs Distributed Memory Systems





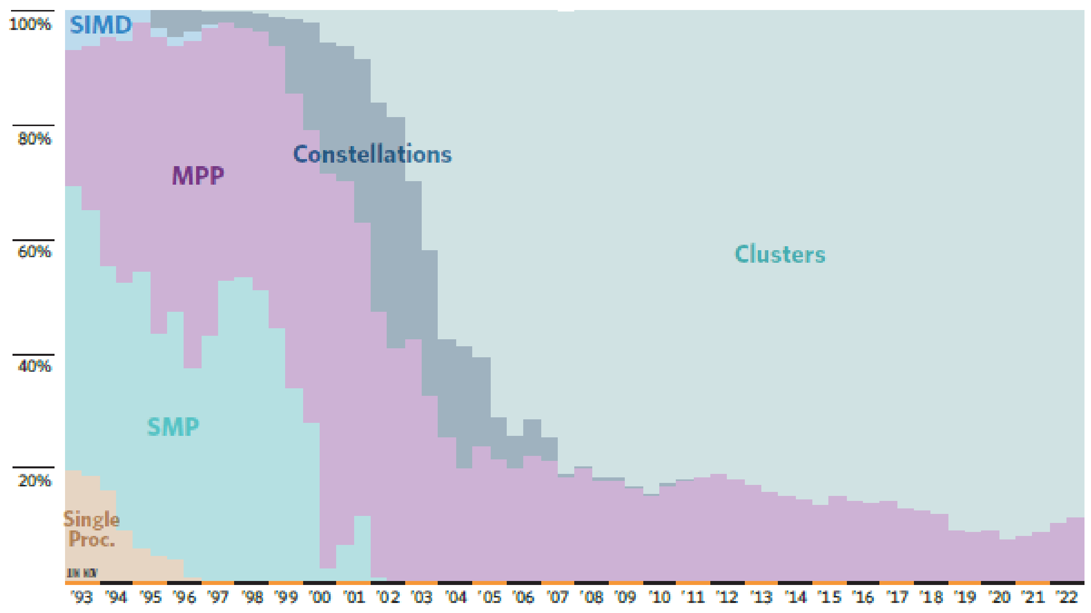
Clusters

[a cluster is a] parallel computer system comprising an integrated collection of independent nodes, each of which is a system in its own right, capable of independent operation and derived from products developed and marketed for other stand-alone purposes

© Dongarra et al. : "High-performance computing: clusters, constellations, MPPs, and future directions", Computing in Science & Engineering (Volume:7 , Issue: 2)

(*) Picture from: http://en.wikipedia.org/wiki/Computer_cluster

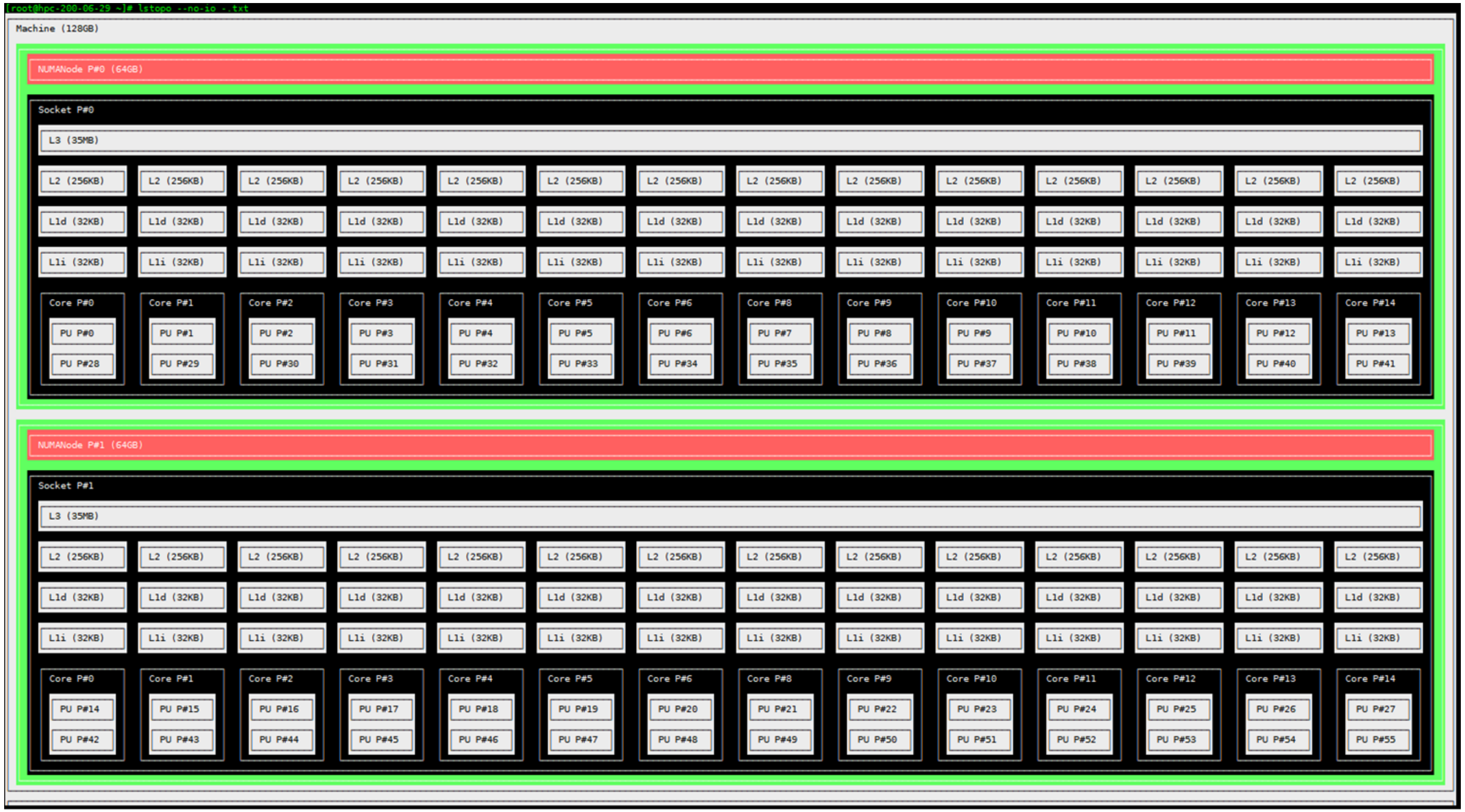
ARCHITECTURES Top500.org 2023 stats



System Topology

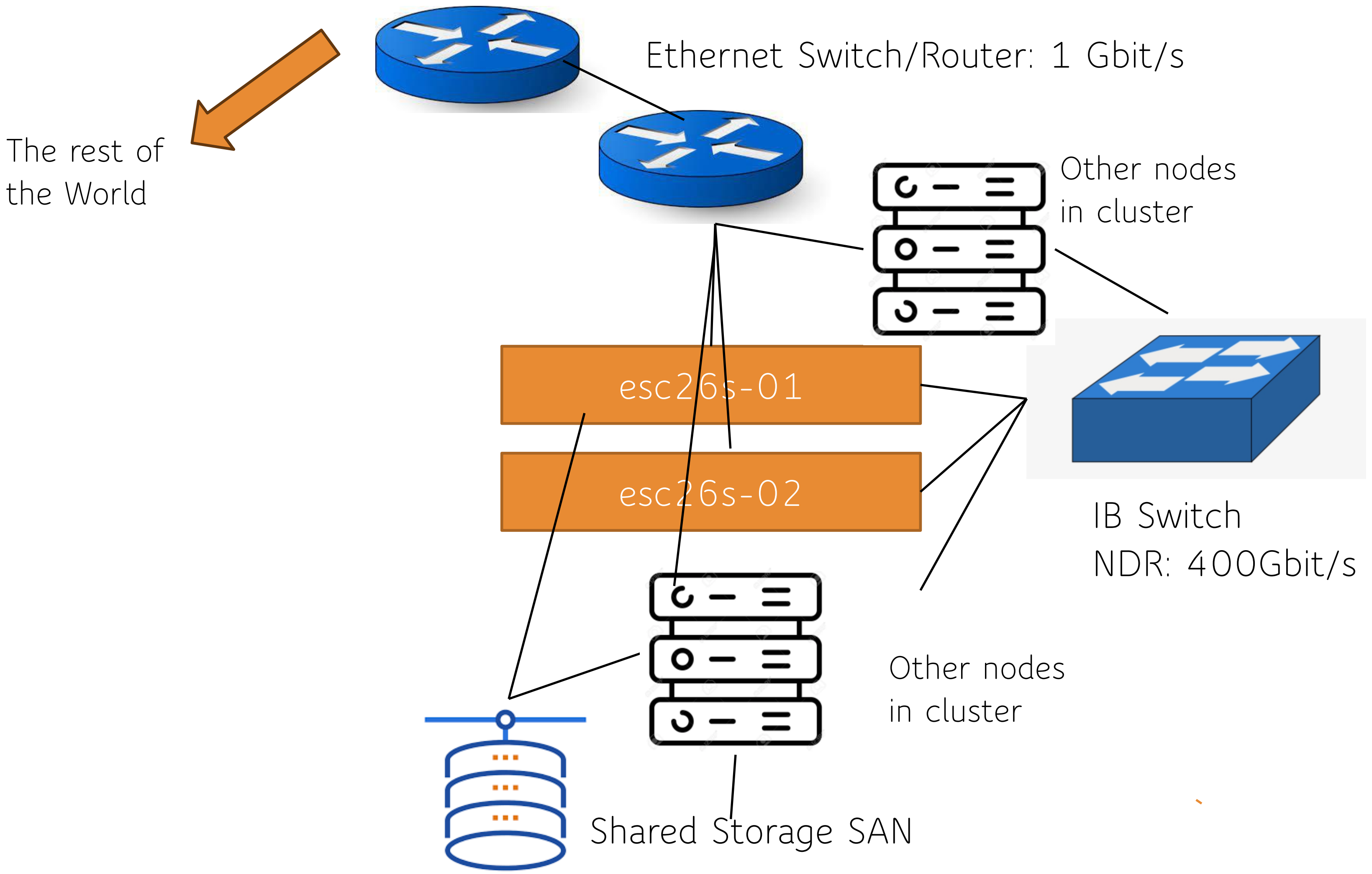
- Knowing where you are is important!!
  - Always try to understand the details of the system you are running on

`# lstopo --no-io -.txt`



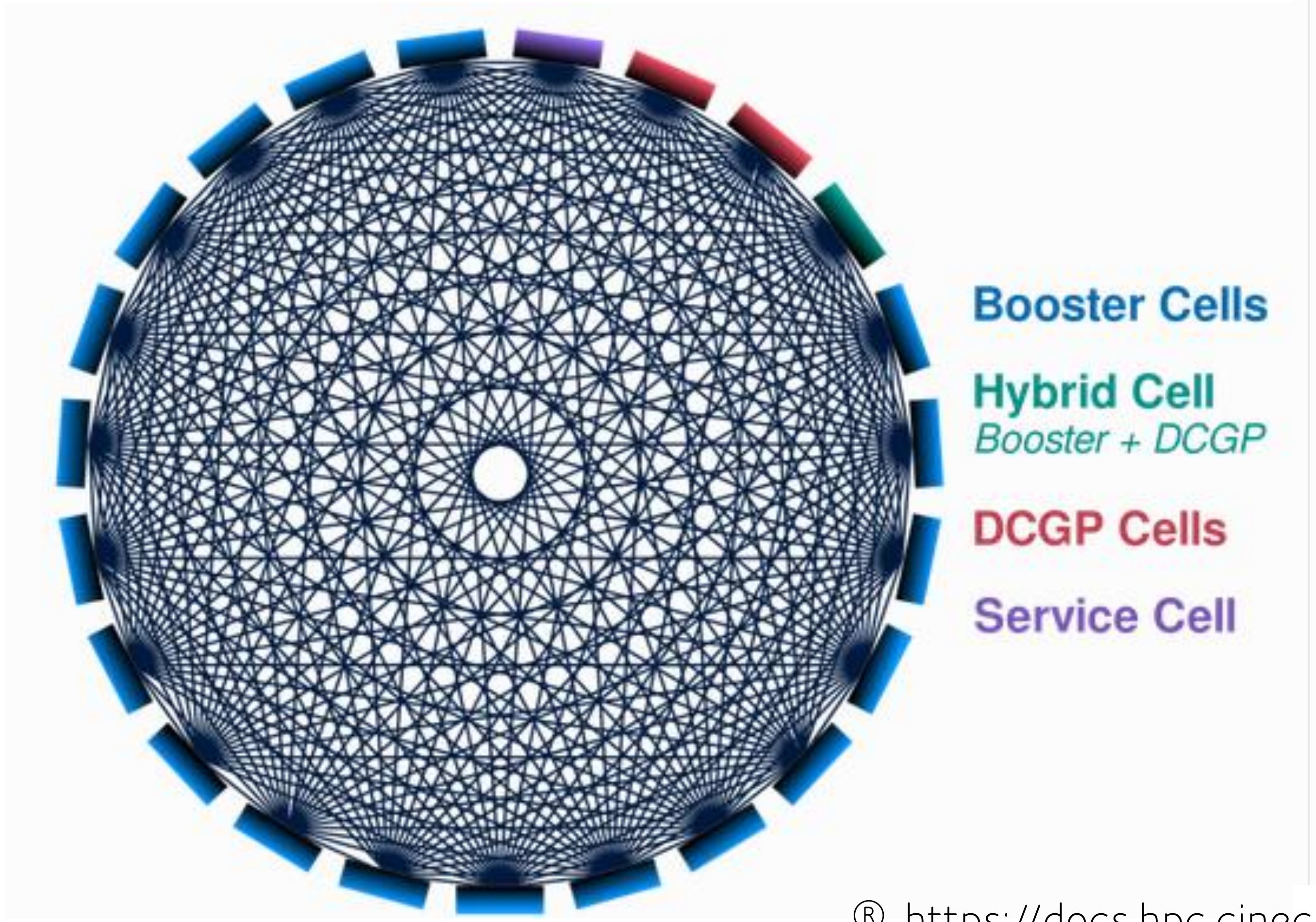
```
[root@hpc-200-06-29 ~]# lstopo --no-io -.txt
Machine (128GB)
  NUMANode P#0 (64GB)
  Socket P#0
    L3 (35MB)
    L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)
    L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)
    L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)
    Core P#0    Core P#1    Core P#2    Core P#3    Core P#4    Core P#5    Core P#6    Core P#8    Core P#9    Core P#10   Core P#11   Core P#12   Core P#13   Core P#14
      PU P#0      PU P#1      PU P#2      PU P#3      PU P#4      PU P#5      PU P#6      PU P#7      PU P#8      PU P#9      PU P#10     PU P#11     PU P#12     PU P#13
      PU P#28     PU P#29     PU P#30     PU P#31     PU P#32     PU P#33     PU P#34     PU P#35     PU P#36     PU P#37     PU P#38     PU P#39     PU P#40     PU P#41
  NUMANode P#1 (64GB)
  Socket P#1
    L3 (35MB)
    L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)  L2 (256KB)
    L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)
    L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)
    Core P#0    Core P#1    Core P#2    Core P#3    Core P#4    Core P#5    Core P#6    Core P#8    Core P#9    Core P#10   Core P#11   Core P#12   Core P#13   Core P#14
      PU P#14     PU P#15     PU P#16     PU P#17     PU P#18     PU P#19     PU P#20     PU P#21     PU P#22     PU P#23     PU P#24     PU P#25     PU P#26     PU P#27
      PU P#42     PU P#43     PU P#44     PU P#45     PU P#46     PU P#47     PU P#48     PU P#49     PU P#50     PU P#51     PU P#52     PU P#53     PU P#54     PU P#55
```

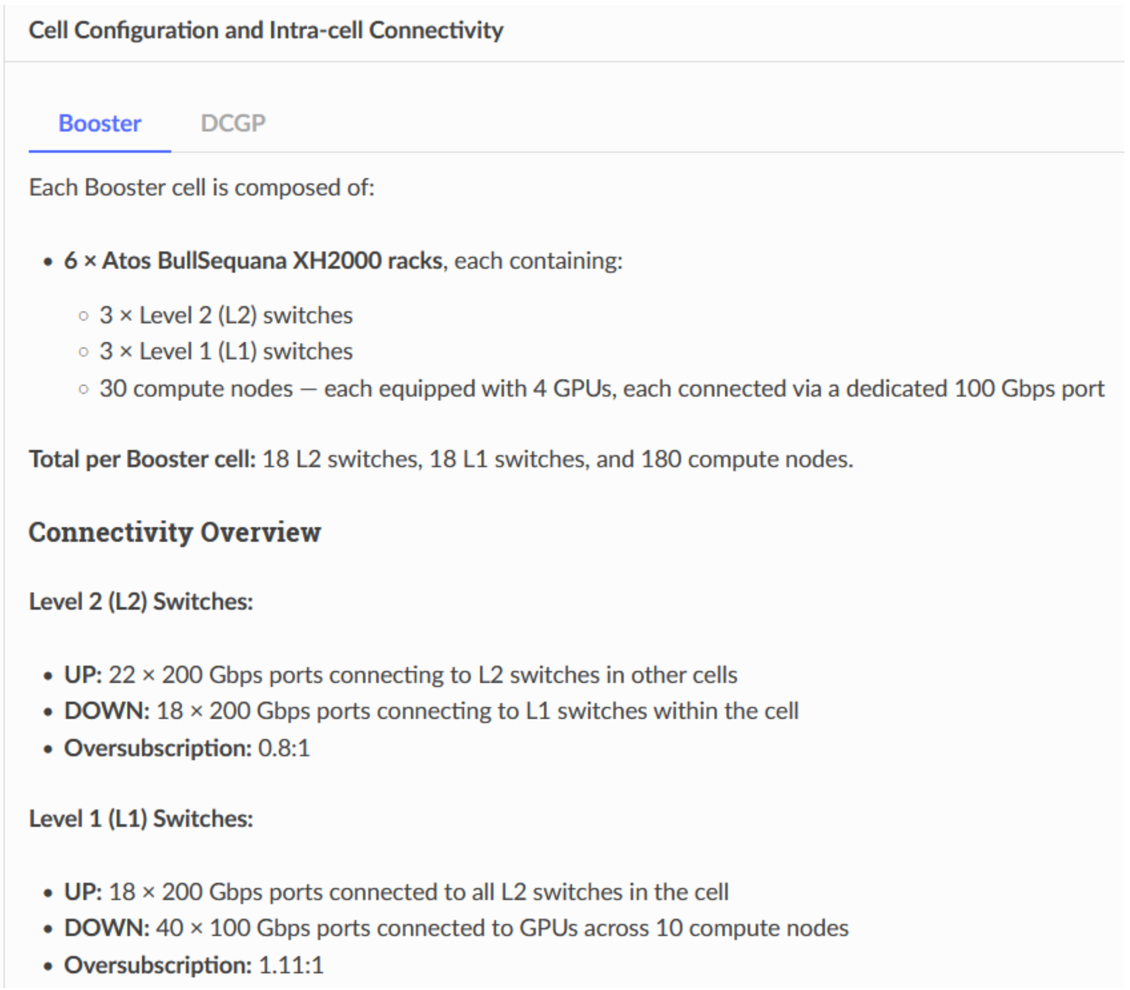
System Networking



Topologies can be complex…

LEONARDO@CINECA Dragonfly+ IB Network topolog





Cell Configuration and Intra-cell Connectivity

Booster DCGP

Each Booster cell is composed of:

- **6 × Atos BullSequana XH2000 racks**, each containing:
  - 3 × Level 2 (L2) switches
  - 3 × Level 1 (L1) switches
  - 30 compute nodes — each equipped with 4 GPUs, each connected via a dedicated 100 Gbps port

Total per Booster cell: 18 L2 switches, 18 L1 switches, and 180 compute nodes.

Connectivity Overview

Level 2 (L2) Switches:

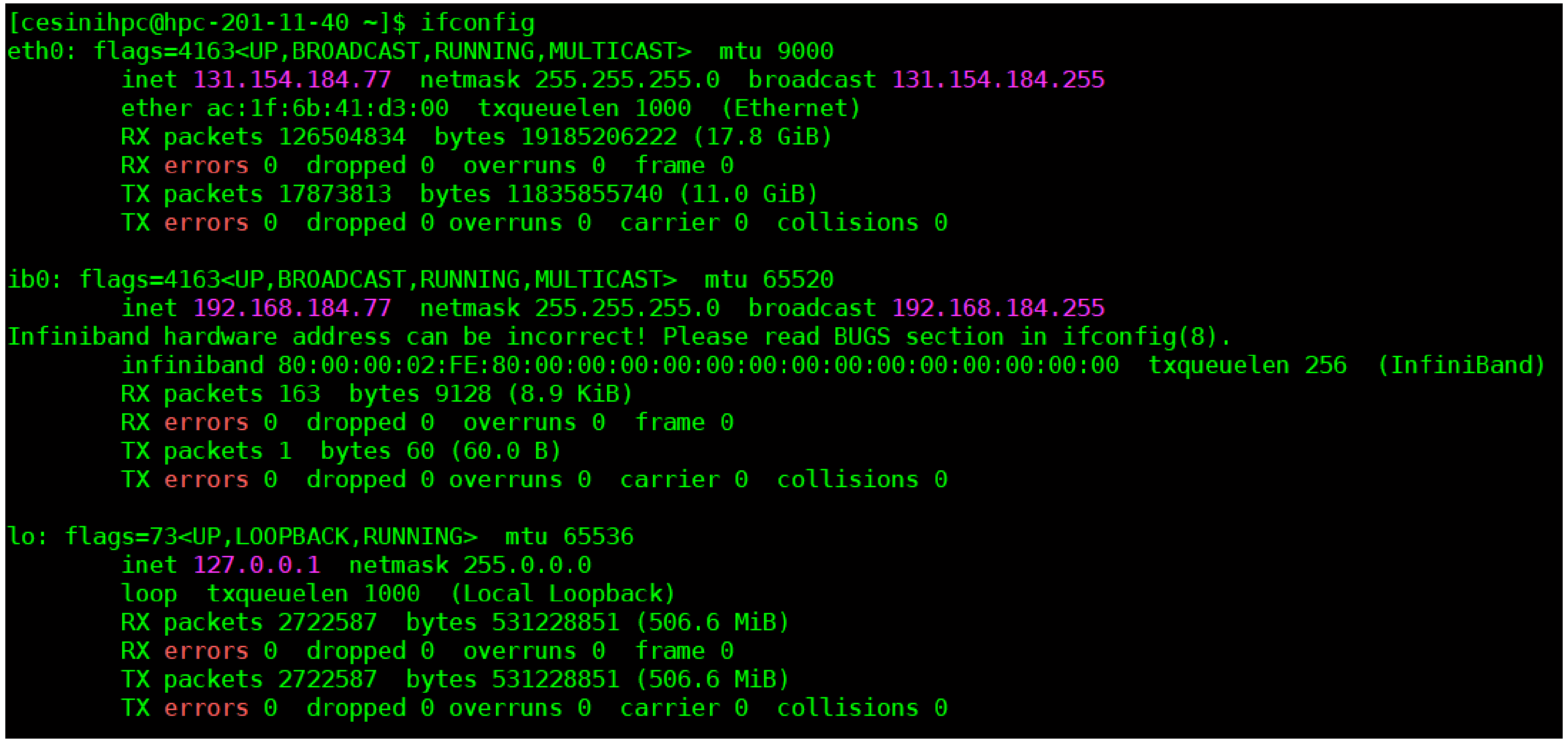
- **UP:** 22 × 200 Gbps ports connecting to L2 switches in other cells
- **DOWN:** 18 × 200 Gbps ports connecting to L1 switches within the cell
- **Oversubscription:** 0.8:1

Level 1 (L1) Switches:

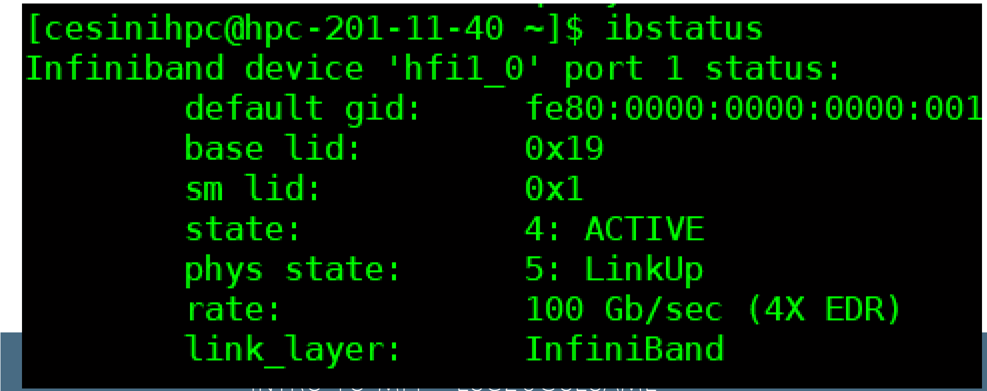
- **UP:** 18 × 200 Gbps ports connected to all L2 switches in the cell
- **DOWN:** 40 × 100 Gbps ports connected to GPUs across 10 compute nodes
- **Oversubscription:** 1.11:1

® https://docs.hpc.cineca.it/hpc/leonardo.html

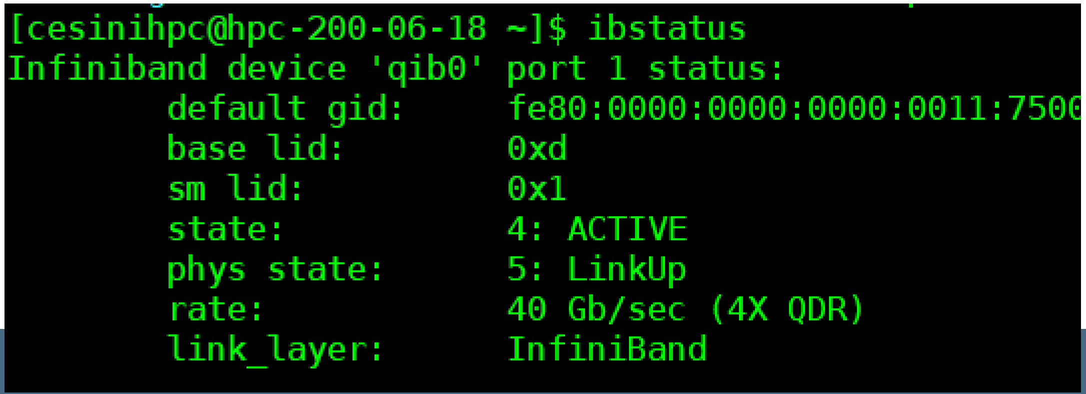
System Networking



```
[cesinihpc@hpc-201-11-40 ~]$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9000
        inet 131.154.184.77  netmask 255.255.255.0  broadcast 131.154.184.255
        ether ac:1f:6b:41:d3:00  txqueuelen 1000  (Ethernet)
        RX packets 126504834  bytes 19185206222 (17.8 GiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 17873813  bytes 11835855740 (11.0 GiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ib0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 65520
        inet 192.168.184.77  netmask 255.255.255.0  broadcast 192.168.184.255
Infiniband hardware address can be incorrect! Please read BUGS section in ifconfig(8).
        infiniband 80:00:00:02:FE:80:00:00:00:00:00:00:00:00:00:00:00:00:00:00  txqueuelen 256  (InfiniBand)
        RX packets 163  bytes 9128 (8.9 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1  bytes 60 (60.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 2722587  bytes 531228851 (506.6 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2722587  bytes 531228851 (506.6 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```



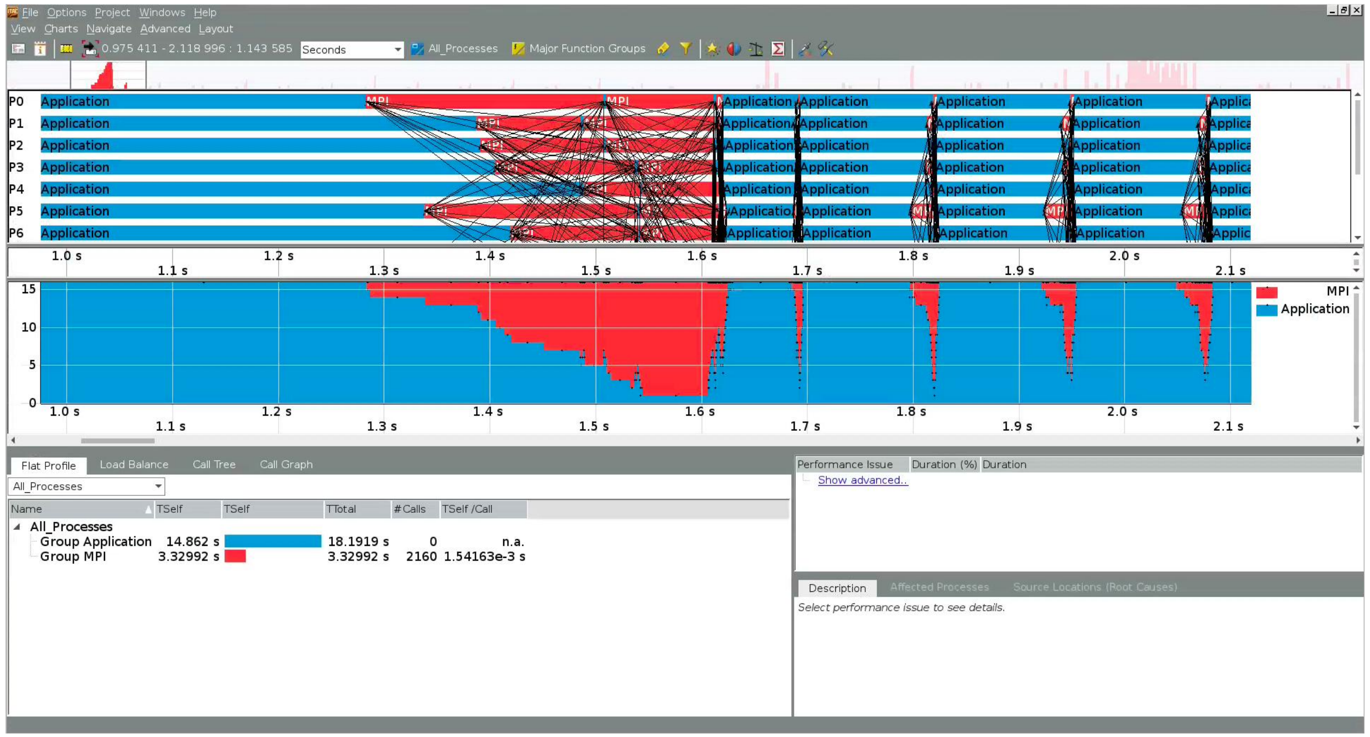
```
[cesinihpc@hpc-201-11-40 ~]$ ibstatus
Infiniband device 'hfi1_0' port 1 status:
        default gid:     fe80:0000:0000:0000:001
        base lid:        0x19
        sm lid:          0x1
        state:           4: ACTIVE
        phys state:      5: LinkUp
        rate:            100 Gb/sec (4X EDR)
        link_layer:      InfiniBand
```



```
[cesinihpc@hpc-200-06-18 ~]$ ibstatus
Infiniband device 'qib0' port 1 status:
        default gid:     fe80:0000:0000:0000:0011:7500
        base lid:        0xd
        sm lid:          0x1
        state:           4: ACTIVE
        phys state:      5: LinkUp
        rate:            40 Gb/sec (4X QDR)
        link_layer:      InfiniBand
```

The Message Passing Programming Model

- Program consists of a collection of named processes
  - Number of processes almost always fixed at program startup time
  - Local address space per node
    - NO physically shared memory.
  - Logically shared data is partitioned over local processes



All_Processes

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
All_Processes					
Group Application	14.862 s	🟦	18.1919 s	0	n.a.
Group MPI	3.32992 s	🟥	3.32992 s	2160	1.54163e-3 s

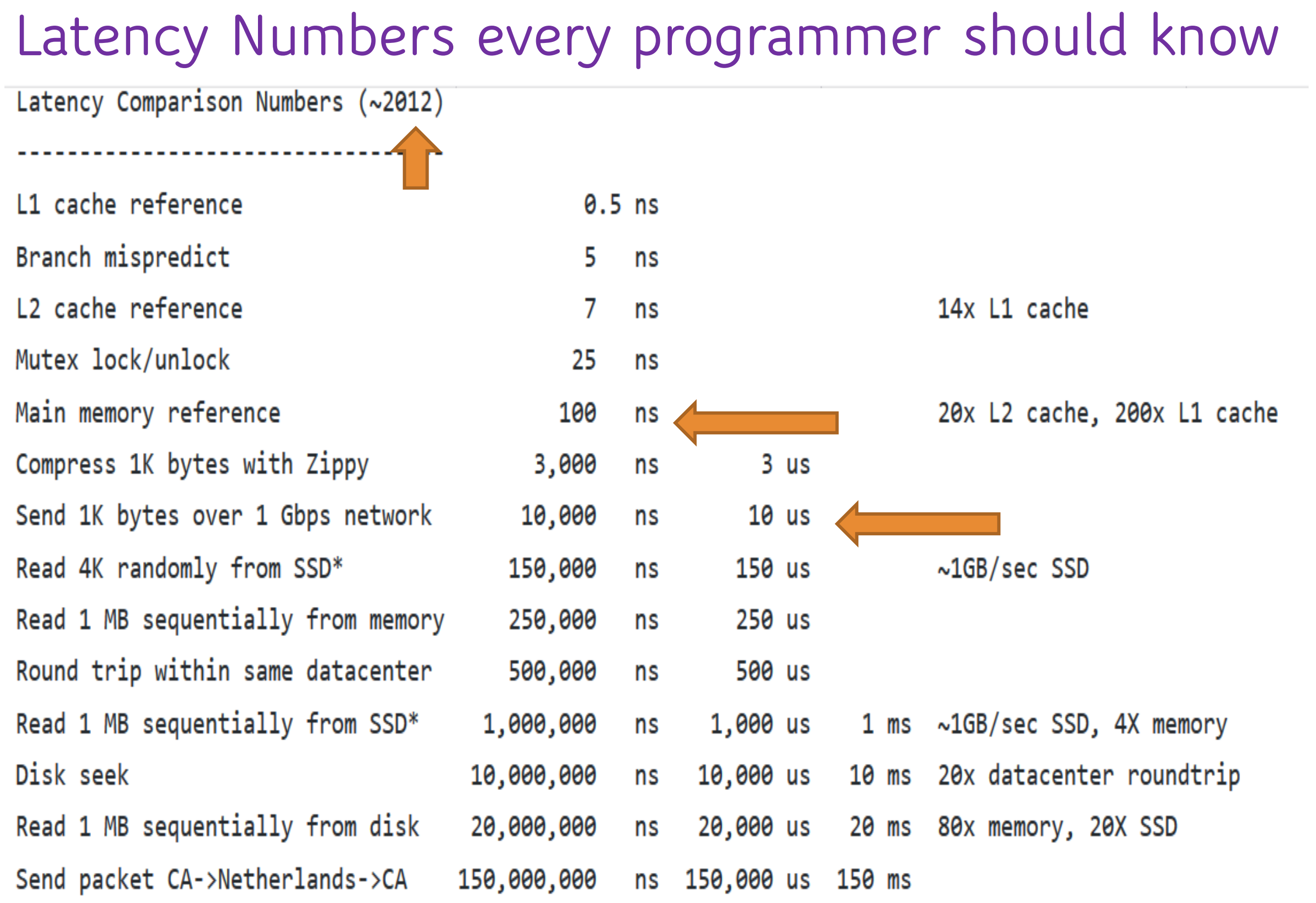
Select performance issue to see details.

- Communication happens by explicit send/receive statements

- Message can be passed over a network infrastructure <u>*or via the main memory, "shared" memory*</u>

Performance and Efficiency Loss?

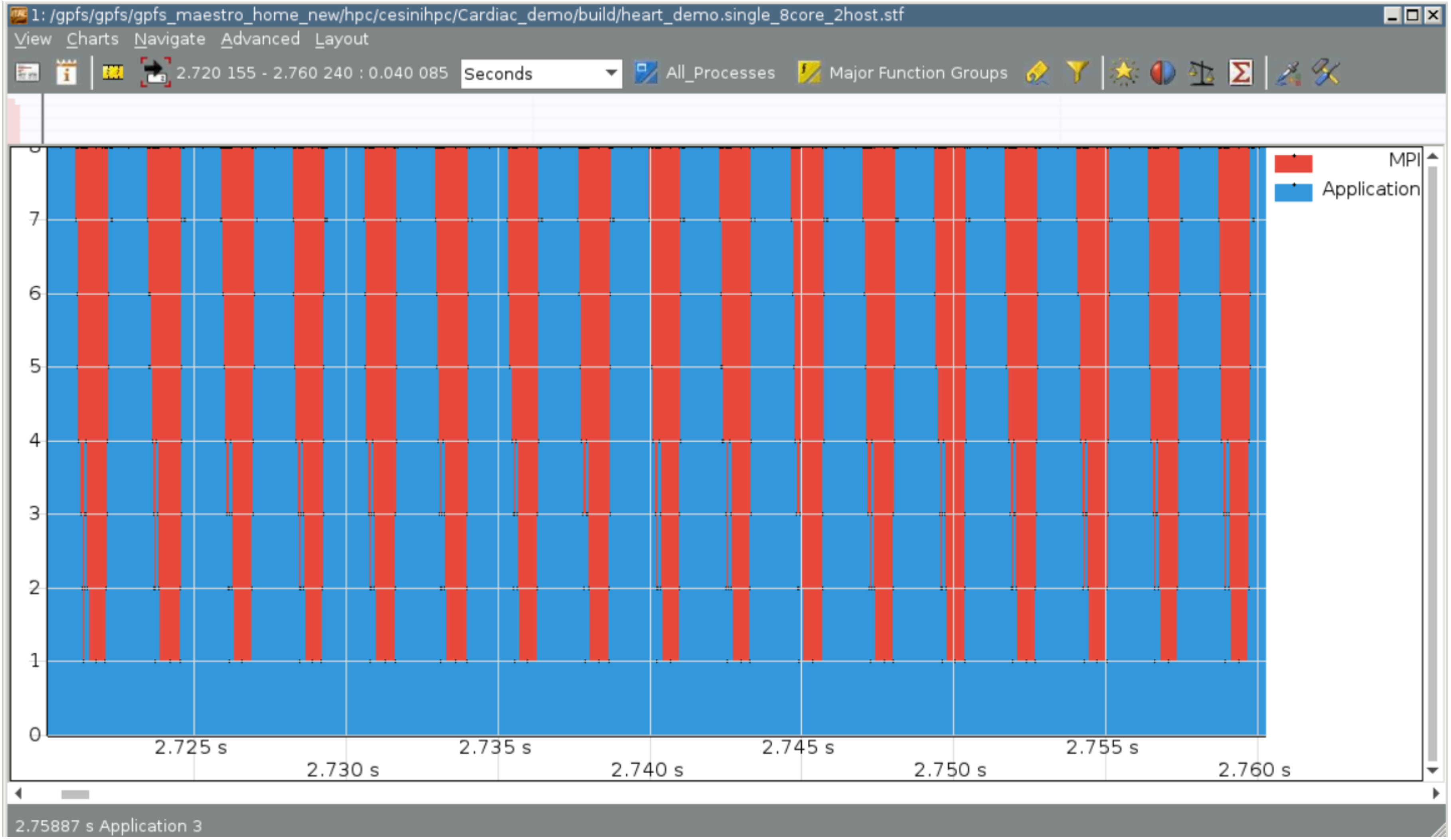
- The latency of the DRAM can be measured in tens of nanoseconds
- Sending a byte to a **networked computer** can take **2-3 orders of magnitude longer than DRAM**, depending on the interconnect technology
- In using Message Passing, try hard to minimize communication
- In any case, the interconnection technology greatly affects the program performances
  - Ethernet 1Gbs latency O(10.000ns)
  - Infiniband NDR latency O(200ns)
  - DDR4-3600 latency O(60ns)
  - DDR5-5600 latency O(10ns)



Latency Numbers every programmer should know

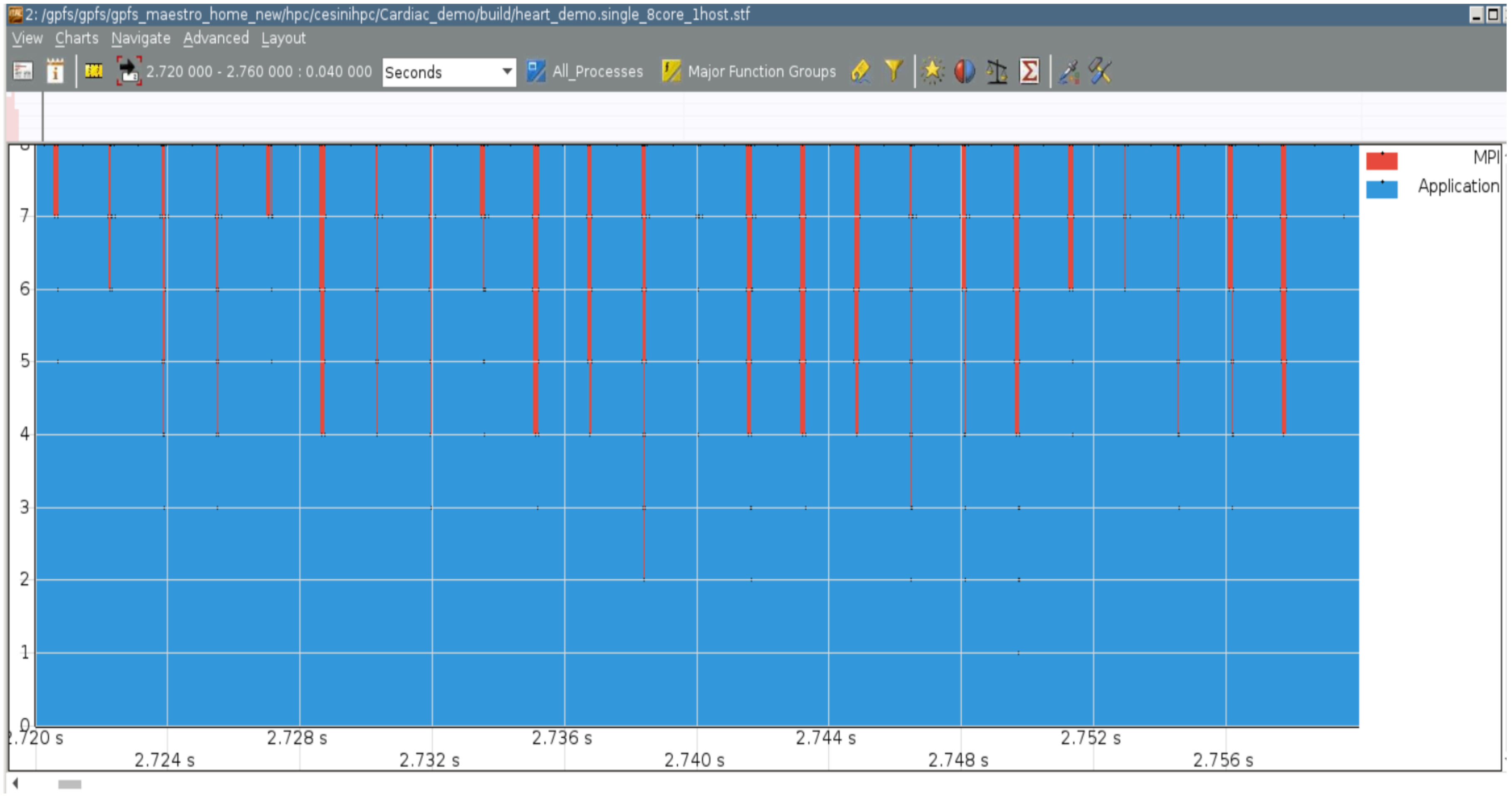
```
Latency Comparison Numbers (~2012)
----------------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns                      20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us          ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD, 4X memory
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datacenter roundtrip
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms
```

Communication performances in MPI Applications



8 processes
2 hosts
MPI send/receive
over ethernet

Communication performances in MPI Applications

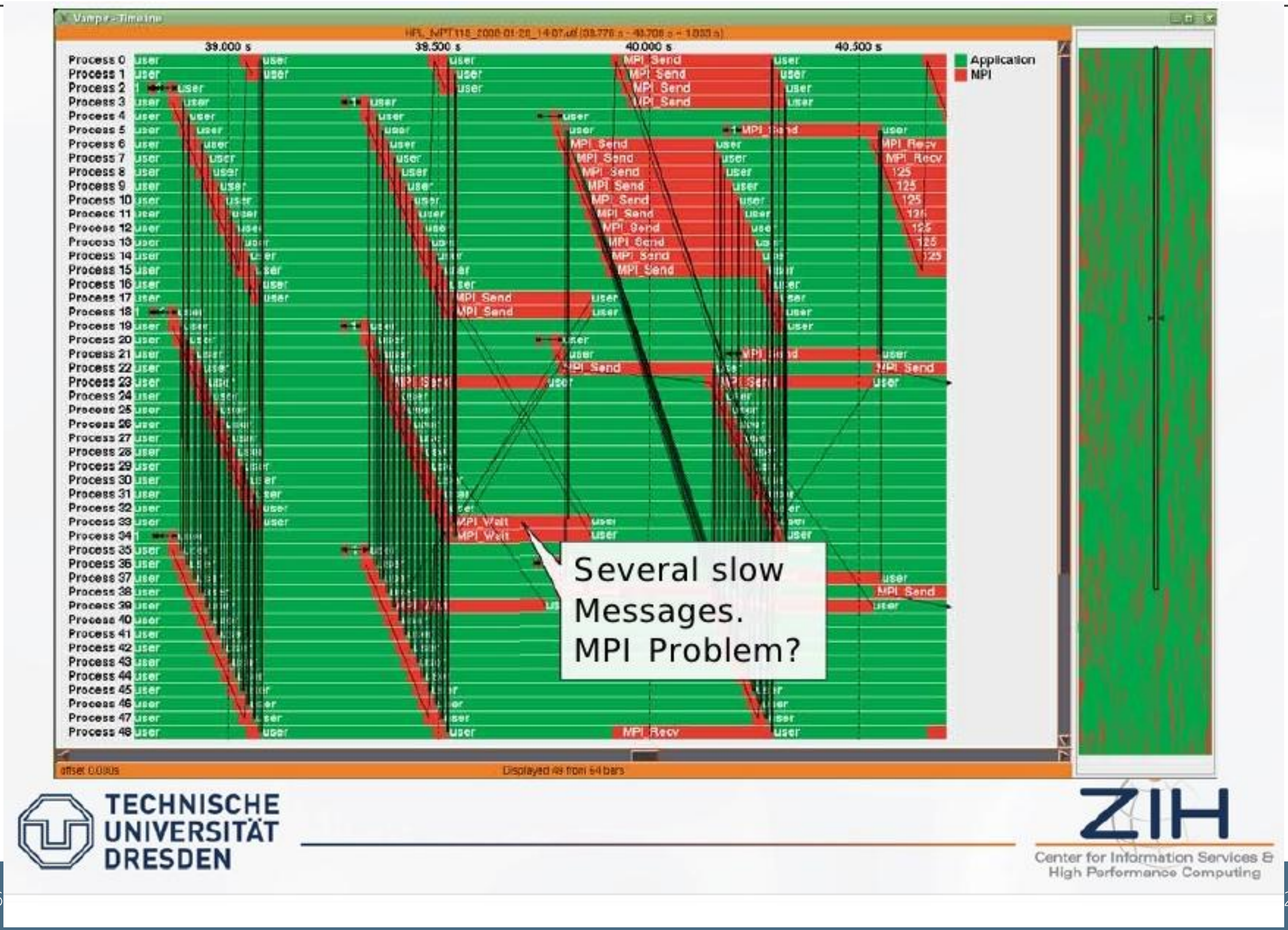


8 processes

1 host

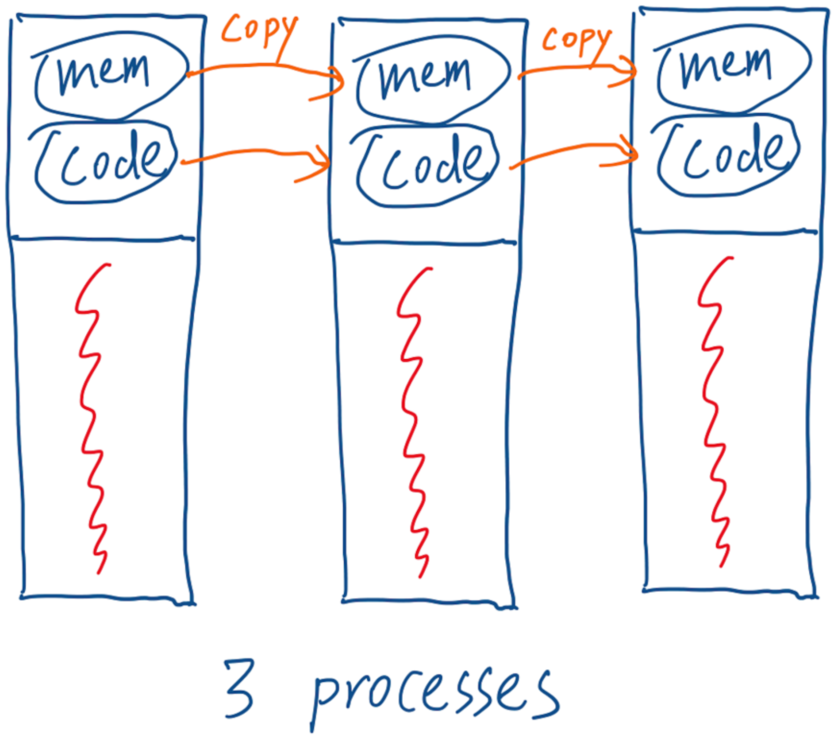
MPI send/receive via shared memory

Communication performances in MPI Applications


Vampir - Timeline
39.000 s
39.500 s
40.000 s
40.500 s
Application
MPI
Process 0
Process 48
user
MPI_Send
MPI_Recv
MPI_Wait
Several slow Messages. MPI Problem?
Displayed 49 from 64 bars
TECHNISCHE UNIVERSITÄT DRESDEN
ZIH
Center for Information Services & High Performance Computing

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
- When you run an MPI program, multiple processes all running the same program are launched working on their own block of data



SPMD – Single Program Multiple Data

- Every process runs the same program…
  - ……on P processing elements where P can be arbitrarily large
- Each process has a unique identifier and runs the version of the program with that particular identifier
  - the rank - an ID ranging from 0 to (P-1)
- Each process access its own private data
- You usually run one process per socket/core depending on the parallelization strategy
  - And on the system topology

SPMD – Single Program 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++
```

MPI Implementations

- MPICH
  - The initial implementation of the MPI 1.x standard, from Argonne National Laboratory (ANL) and Mississippi State University.
  - ANL has continued developing MPICH for over a decade, and now offers MPICH-3.2, implementing the MPI-3.1 standard
- IBM also was an early implementor, and most early 90s supercomputer companies either commercialized MPICH, or built their own implementation.
- LAM/MPI from Ohio Supercomputer Center
  - another early open implementation..
- Open MPI (not to be confused with OpenMP) was formed by the merging FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI, and is found in many TOP-500 supercomputers.
  - We will use OpenMPI for our exercises!!
- Many other efforts are derivatives of MPICH, LAM, and other works, including, but not limited to, commercial implementations from HP, Intel, Microsoft, and NEC.

MPI HelloWorld

https://github.com/infn-esc/sesame26/blob/main/hands-on/mpi/MPI_Hello.cpp

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

int main(intargc,char**argv){
   int rank, world_size;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&world_size);
   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();
   return 0;
}
```

MPI_Init and MPI_Finalize

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

int main(int argc,char**argv){
   int rank, world_size;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&world_size);
   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();
   return 0;
}
```

Called before any other MPI functions
- Initializes the library
- Argc and argv are the command line args passed to main
  - - Open MPI accepts the C/C++ *argc* and *argv* arguments to main, but neither modifies, interprets, nor distributes them

Called to close any MPI program
- Frees memory allocated by MPI
- Must be invoked by all Ranks

How many processes?

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

int main(int argc, char** argv){
   int rank, world_size;

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

  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();
   return 0;
}
```

Communicators consist of two parts, a context and a process group. The communicator lets us control how groups of messages interact.

`int MPI_Comm_size (MPI_Comm comm, int* size)`

- **`MPI_Comm`**, an *opaque data type called a communicator.* Default context: **MPI_COMM_WORLD (all processes)**
- **`MPI_Comm_size`** returns the number of processes in the process group associated with the communicator

Who am I? (which is my rank?)

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

int main(int argc, char** argv){
   int rank, world_size;

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

  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();
   return 0;
}
```

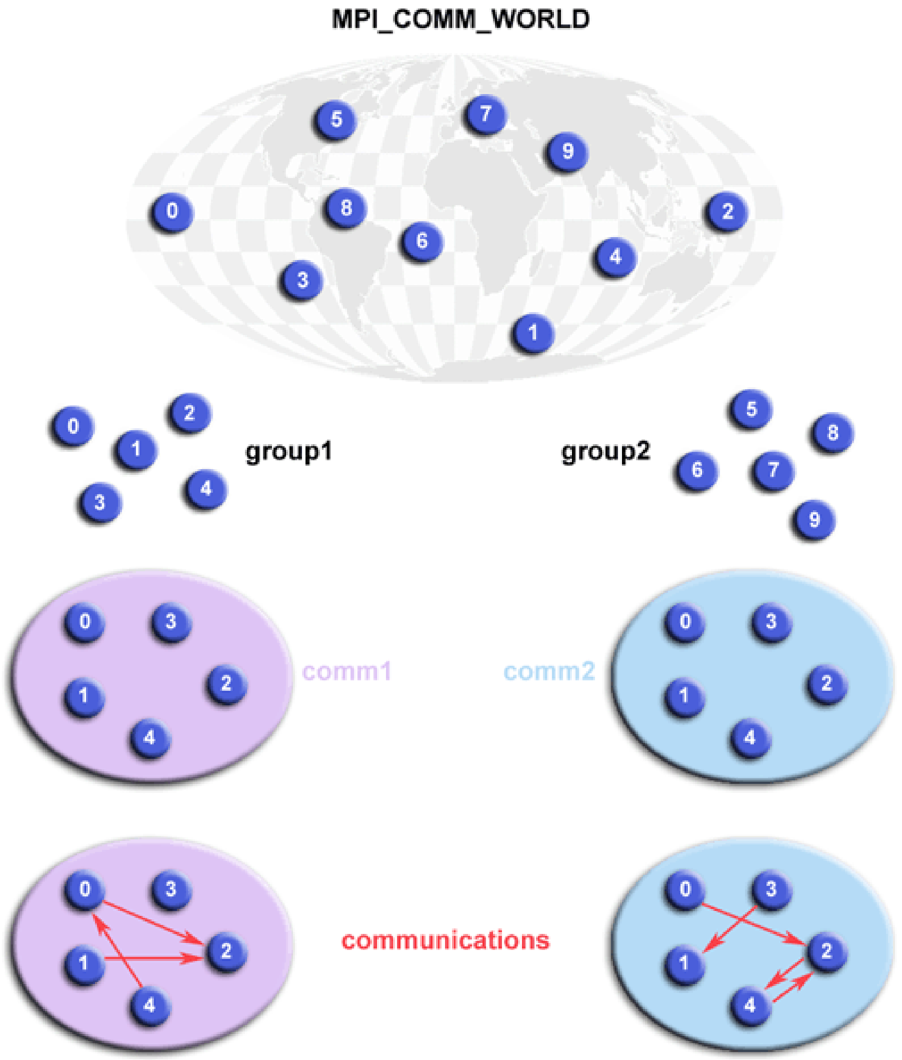
Note that other than init() and finalize(), every MPI function has a communicator which defines the context and group of processes that the MPI functions impact

`int MPI_Comm_rank (MPI_Comm comm, int* rank)`

- **`MPI_Comm`**, an *opaque data type called a communicator.* Default context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_rank`** returns an integer ranging from 0 to “(num of procs)-1”

Communicators and Groups - 1

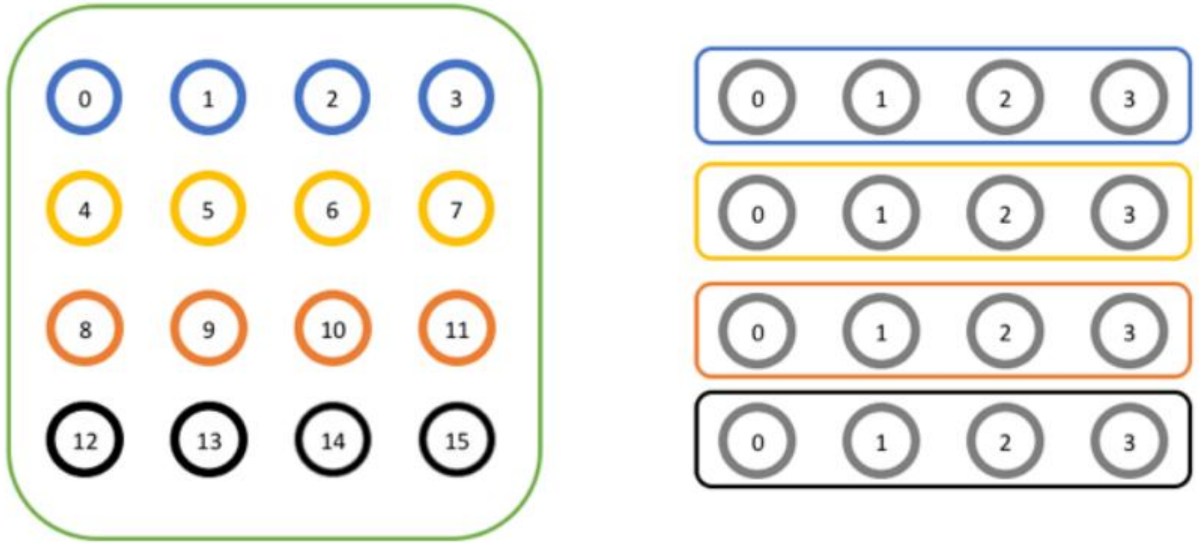
- Internally, MPI has to keep up with (among other things) two major parts of a communicator
  - the context (or ID) that differentiates one communicator from another
    - prevents an operation on one communicator from matching with a similar operation on another communicator
  - the group of processes contained by the communicator
- Communicators provides a separate communication space
- It's not unusual to do everything using MPI_COMM_WORLD, but for more complex use cases, it might be helpful to have more communicators.
  - MPI_Comm_split is the simplest way to create a new communicator
- A Group is a little simpler, since it is just the set of all processes in the communicator.
  - MPI offers function to manage Groups: Union or Intersection
  - Groups can be used to create Communicators



Communicators and Groups - 2

```
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank,
&row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE:
%d/%d\n",
                world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

Split a Large Communicator into a Smaller ones



© https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/

How do I run it?

- Compile it:
  - mpic++ -o MPI_Hello.out MPI_Hello.cpp
- Run it:
  - mpirun –hostfile machinefile.txt –np <np> MPI_Hello.out
    - the command is implementation dependent

```
[dcesini@ esc26s-01]$ cat machinefile.txt
esc26s-01 slots=2
esc26s-02 slots=2
```

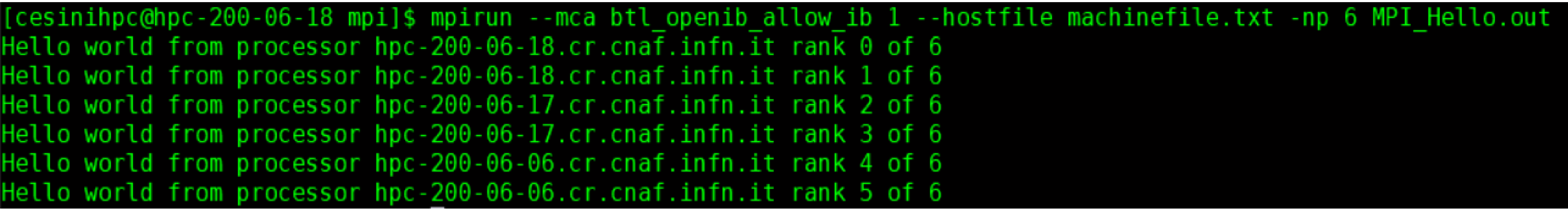
The same of running this

```
mpirun -H esc26s-01 :2, esc26s-01 :2 -np 4 MPI_Hello.out
```

An MPI version is installed on our systems

Normally you should "export" the right PATH and LD_LIBRARY_PATH in -bashrc

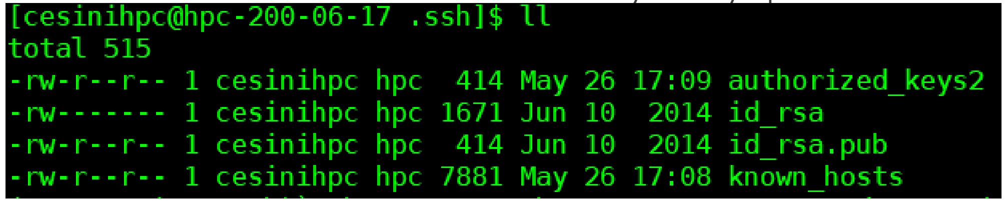
done for you automatically via .bash_profile



```
[cesinihpc@hpc-200-06-18 mpi]$ mpirun --mca btl_openib_allow_ib 1 --hostfile machinefile.txt -np 6 MPI_Hello.out
Hello world from processor hpc-200-06-18.cr.cnaf.infn.it rank 0 of 6
Hello world from processor hpc-200-06-18.cr.cnaf.infn.it rank 1 of 6
Hello world from processor hpc-200-06-17.cr.cnaf.infn.it rank 2 of 6
Hello world from processor hpc-200-06-17.cr.cnaf.infn.it rank 3 of 6
Hello world from processor hpc-200-06-06.cr.cnaf.infn.it rank 4 of 6
Hello world from processor hpc-200-06-06.cr.cnaf.infn.it rank 5 of 6
```

A couple of notes

- <u>The executable must be present in all the hosts used, in the same path</u>
  - You are lucky in the school nodes – shared home directories!!
- OpenMPI in our cluster uses ssh to connect to the remote hosts
  - ssh should work passwordless
    - i.e. HostBasedAuthentication yes in sshd_config
    - i.e. Exchange and authorize keys – see the esc26@sesame Environment setup page
  - Create an identity key pair and add the public part to the authorized_keys file in .ssh



```
[cesinihpc@hpc-200-06-17 .ssh]$ ll
total 515
-rw-r--r-- 1 cesinihpc hpc  414 May 26 17:09 authorized_keys2
-rw------- 1 cesinihpc hpc 1671 Jun 10  2014 id_rsa
-rw-r--r-- 1 cesinihpc hpc  414 Jun 10  2014 id_rsa.pub
-rw-r--r-- 1 cesinihpc hpc 7881 May 26 17:08 known_hosts
```

```
[dcesini@esc26s-01 .ssh]$ cat id_rsa.pub >> authorized_keys
```

  - During login the OpenMPI environment should be loaded
    - Typically via the .basrc file

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
- Send/Receive Functions
  - Can be Blocking and Non blocking
- Send and Receive calls/messages must match, otherwise deadlocks can occur

Messages – Send and Receive

```
int MPI_Send (const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

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

- Int MPI_Send performs a blocking send of the specified data ("count" copies of type "datatype," stored in "buf") to the specified destination (rank "dest" within communicator "comm"), with message ID "tag"
- int MPI_Recv (void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)
- "blocking" means the functions return as soon as the buffer, "buf", can be safely used.

MPI_Send, MPI_Isend, MPI_SSend

- MPI_Send (Standard Send)

  Type: **Blocking send**
  Behavior: The function **returns only after the message data has been safely stored away** – either in the receiver's buffer or in an internal system buffer.
  Use case: Good for general-purpose communication when you want to ensure the message has been sent before proceeding.
  Blocking? Yes – the sender waits until it is safe to reuse the send buffer.

- MPI_ISend (Immediate Send)

  Type: **Non-blocking send**
  Behavior: Initiates the send operation and **returns immediately, allowing the program to continue executing.** You must later call MPI_Wait or MPI_Test to ensure the send has completed.
  Use case: Useful for **overlapping communication with computation**, improving performance in parallel applications.
  Blocking? No – but you must manage completion manually.

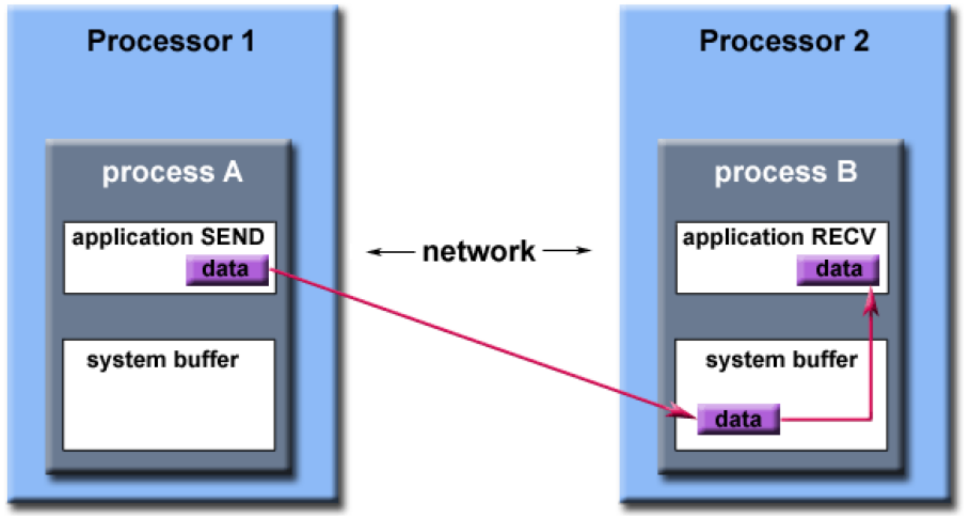
- MPI_SSend (Synchronous Send)

  Type: Blocking synchronous send
  Behavior: The send operation only **completes when the receiver has started receiving the message**. This ensures synchronization between sender and receiver.
  Use case: Useful when you want to ensure that the receiver is ready before sending data, which can help avoid buffer overflows or race conditions.
  Blocking? Yes – waits until the receiver has initiated a matching receive.

Function	Blocking	Synchronization	Buffering	Use Case
MPI_Send	Yes	No	May use internal buffer	General-purpose send
MPI_ISend	No	No	May use internal buffer	Overlap communication and computation
MPI_SSend	Yes	Yes	No buffering; waits for receiver	Ensures receiver is ready

MPI Message Buffer

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case.
- The MPI implementation must be able to deal with storing data when the two tasks are out of sync.
- Consider the following two cases:
  - A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
  - Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?
- The MPI implementation (not the MPI standard) decides what happens to data in these types of cases.
- Typically, a system buffer area is reserved to hold data in transit



- Opaque to the programmer and managed entirely by the MPI library
- A finite resource that can be easy to exhaust
- Often mysterious and not well documented
- Able to exist on the sending side, the receiving side, or both
- Something that may improve program performance because it allows send - receive operations to be asynchronous

Blocking vs Non-Blocking

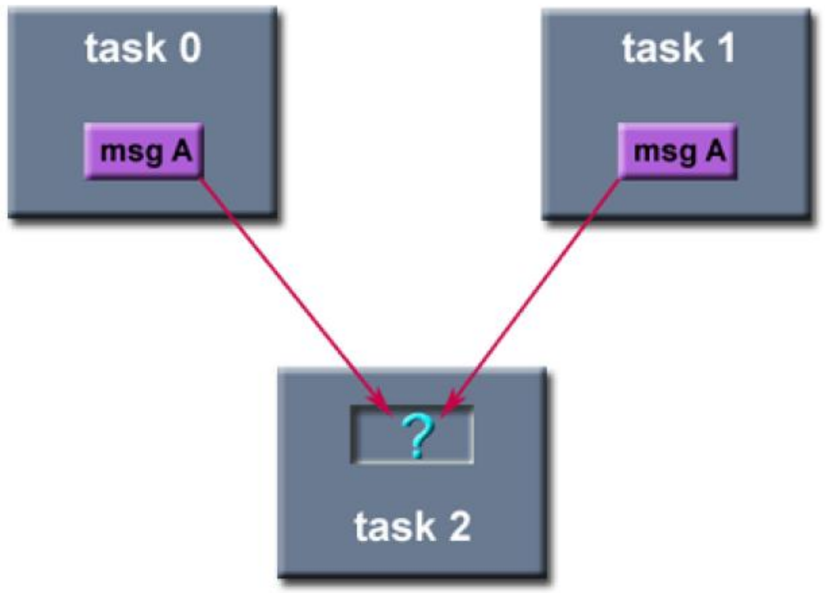
- Blocking:
  - A blocking send routine will only "return" after it is safe to modify the application buffer (your sent data) for reuse.
  - Safe means that modifications will not affect the data intended for the receive task.
  - Safe does not imply that the data was actually received - it may very well be sitting in a system buffer
- Non-Blocking
  - Non-blocking send and receive routines behave similarly - they will return almost immediately.
  - They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message
  - Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user cannot predict when that will happen.
  - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains

Order

- MPI guarantees that messages will not overtake each other.
- If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
- If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.

Fairness

- MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
- Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete



The scenario requires that the receive used the wildcard **MPI_ANY_SOURCE** as its source argument.

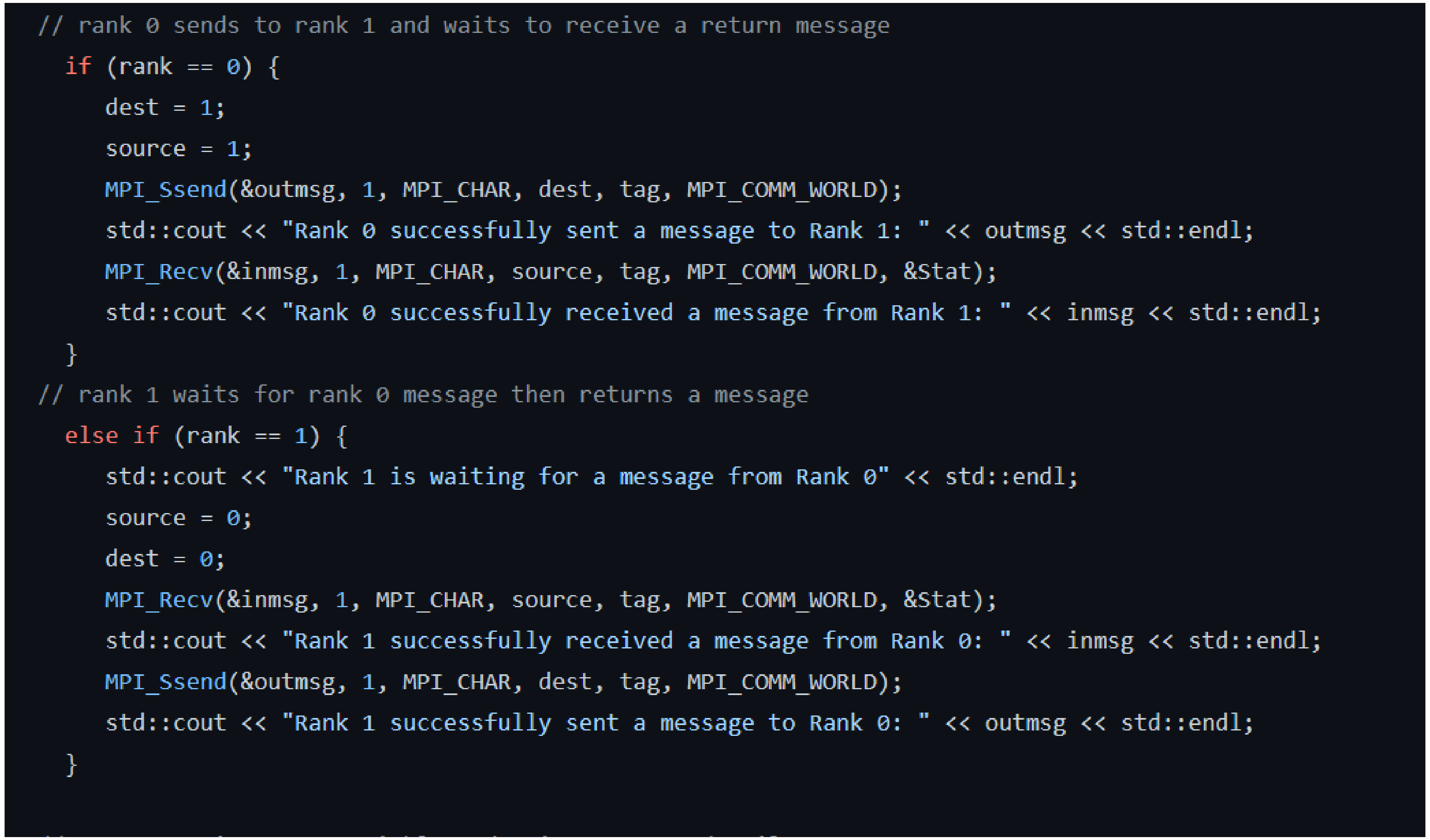
Non-blocking Send and Receive

```
int MPI_ISend (const 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)
```

- int MPI_Isend begins a non-blocking send of the variable buf to destination dest.
- Int MPI_Irecv begins a non-blocking receive
- Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number".
  - The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine completion of the non-blocking operation
    - MPI_Wait(request, status)
    - MPI_Test(request, flag, status)
- Anywhere you use MPI_Send or MPI_Recv, you can use the pair of MPI_Isend/MPI_Wait or MPI_Irecv/MPI_Wait

Send and Receive exercise – the PingPong

https://github.com/infn-esc/sesame26/blob/main/hands-on/mpi/PingPong.cpp



```
// rank 0 sends to rank 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);
   std::cout << "Rank 0 successfully sent a message to Rank 1: " << outmsg << std::endl;
   MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
   std::cout << "Rank 0 successfully received a message from Rank 1: " << inmsg << std::endl;
 }
// rank 1 waits for rank 0 message then returns a message
 else if (rank == 1) {
   std::cout << "Rank 1 is waiting for a message from Rank 0" << std::endl;
   source = 0;
   dest = 0;
   MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
   std::cout << "Rank 1 successfully received a message from Rank 0: " << inmsg << std::endl;
   MPI_Ssend(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   std::cout << "Rank 1 successfully sent a message to Rank 0: " << outmsg << std::endl;
 }
```

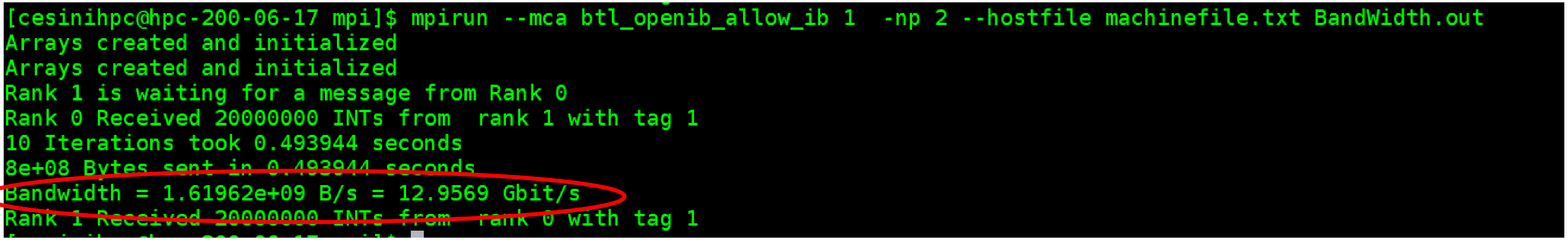
Change the Network interface

- Following the so-called "Law of Least Astonishment", Open MPI assumes that if you have both an IP network and at least one high-speed network (such InfiniBand), you will likely only want to use the high-speed network(s) for MPI message passing
  - AT least up to version3
  - This changed in version4
  - OpenMPI may still use TCP for setup and teardown information – so you'll see traffic across your IP network during startup and shutdown of your MPI job. This is normal and does not affect the MPI message passing channels.
- mpirun --mca btl_openib_allow_ib 1 -np 2 --hostfile machinefile.txt BandWidth.out

```
[cesinihpc@hpc-200-06-17 esc25]$ cat machinefile.txt
hpc-200-06-17 slots=1
hpc-200-06-18 slots=1
```

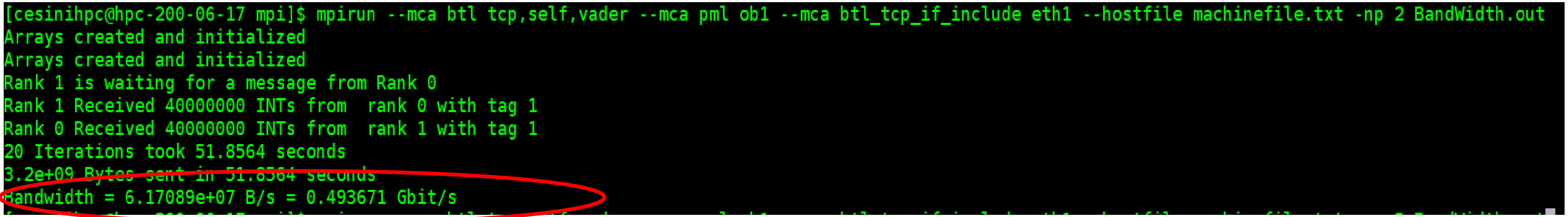
Effect of changing the network interface

- mpirun --mca btl_openib_allow_ib 1 -np 2 --hostfile machinefile.txt BandWidth.out
  (it's the default, you can avoid using this mca option)



```
[cesinihpc@hpc-200-06-17 mpi]$ mpirun --mca btl_openib_allow_ib 1  -np 2 --hostfile machinefile.txt BandWidth.out
Arrays created and initialized
Arrays created and initialized
Rank 1 is waiting for a message from Rank 0
Rank 0 Received 20000000 INTs from  rank 1 with tag 1
10 Iterations took 0.493944 seconds
8e+08 Bytes sent in 0.493944 seconds
Bandwidth = 1.61962e+09 B/s = 12.9569 Gbit/s
Rank 1 Received 20000000 INTs from  rank 0 with tag 1
```

- mpirun --mca btl tcp,self,vader --mca pml ob1 --mca btl_tcp_if_include eth0 --hostfile machinefile.txt -np 2 BandWidth.out

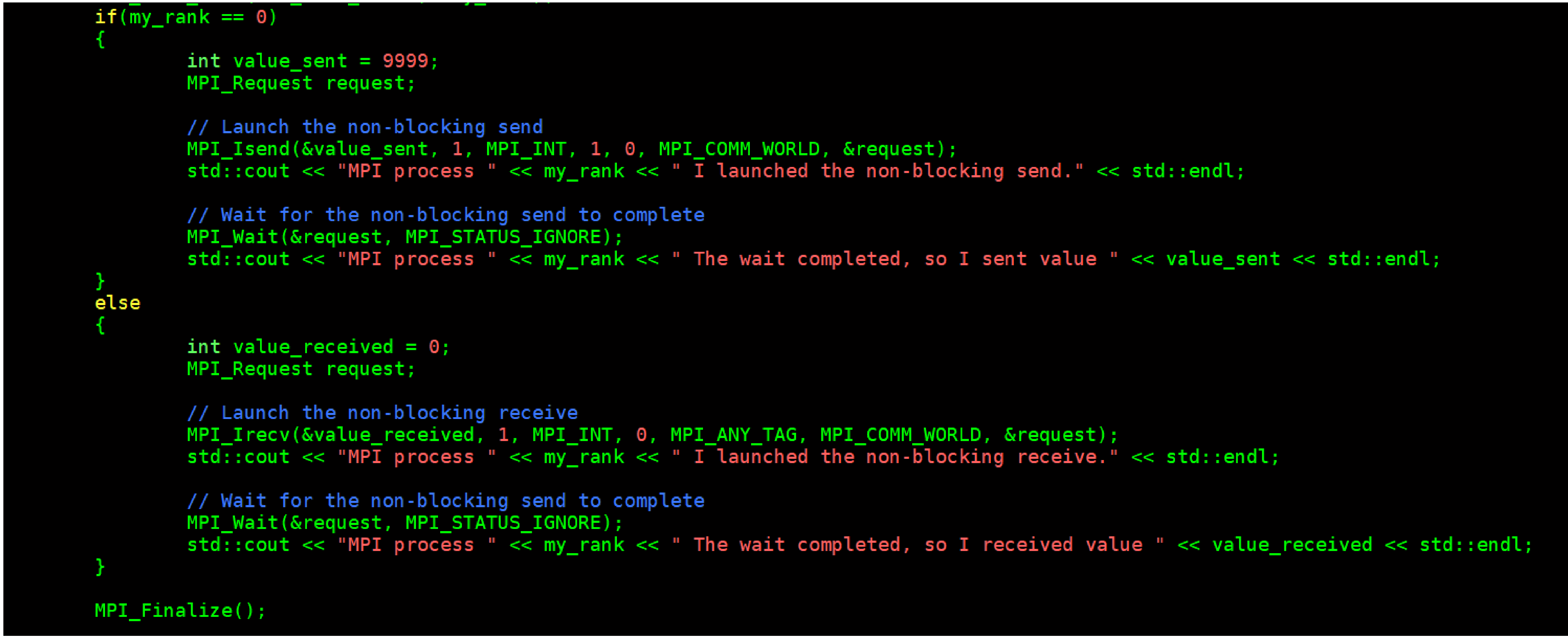


```
[cesinihpc@hpc-200-06-17 mpi]$ mpirun --mca btl tcp,self,vader --mca pml ob1 --mca btl_tcp_if_include eth1 --hostfile machinefile.txt -np 2 BandWidth.out
Arrays created and initialized
Arrays created and initialized
Rank 1 is waiting for a message from Rank 0
Rank 1 Received 40000000 INTs from  rank 0 with tag 1
Rank 0 Received 40000000 INTs from  rank 1 with tag 1
20 Iterations took 51.8564 seconds
3.2e+09 Bytes sent in 51.8564 seconds
Bandwidth = 6.17089e+07 B/s = 0.493671 Gbit/s
```

Now try using the SH (shared memory) MCA…any improvement?

Non Blocking PingPong

https://github.com/infn-esc/sesame26/blob/main/hands-on/mpi/NoBloc_PingPong.cpp



```
    if(my_rank == 0)
    {
            int value_sent = 9999;
            MPI_Request request;

            // Launch the non-blocking send
            MPI_Isend(&value_sent, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
            std::cout << "MPI process " << my_rank << " I launched the non-blocking send." << std::endl;

            // Wait for the non-blocking send to complete
            MPI_Wait(&request, MPI_STATUS_IGNORE);
            std::cout << "MPI process " << my_rank << " The wait completed, so I sent value " << value_sent << std::endl;
    }
    else
    {
            int value_received = 0;
            MPI_Request request;

            // Launch the non-blocking receive
            MPI_Irecv(&value_received, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
            std::cout << "MPI process " << my_rank << " I launched the non-blocking receive." << std::endl;

            // Wait for the non-blocking send to complete
            MPI_Wait(&request, MPI_STATUS_IGNORE);
            std::cout << "MPI process " << my_rank << " The wait completed, so I received value " << value_received << std::endl;
    }

    MPI_Finalize();
```

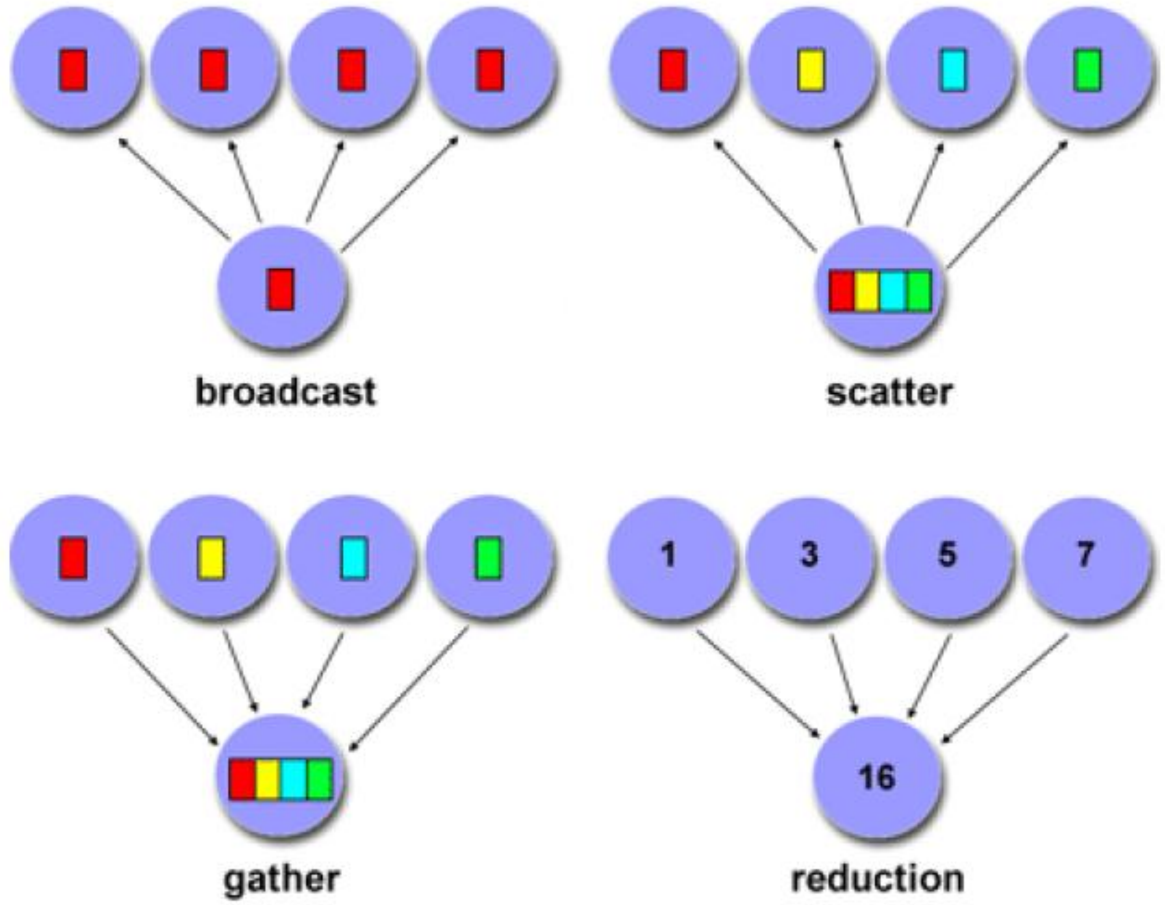
Collective Communication

Scope

- A message can be sent to/received from a group of processes
- Collective communication routines must involve all processes within the scope of a communicator
- It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.
- Use collective communication when possible
  - They are implemented more efficiently than the sum of their point-to-point equivalent calls

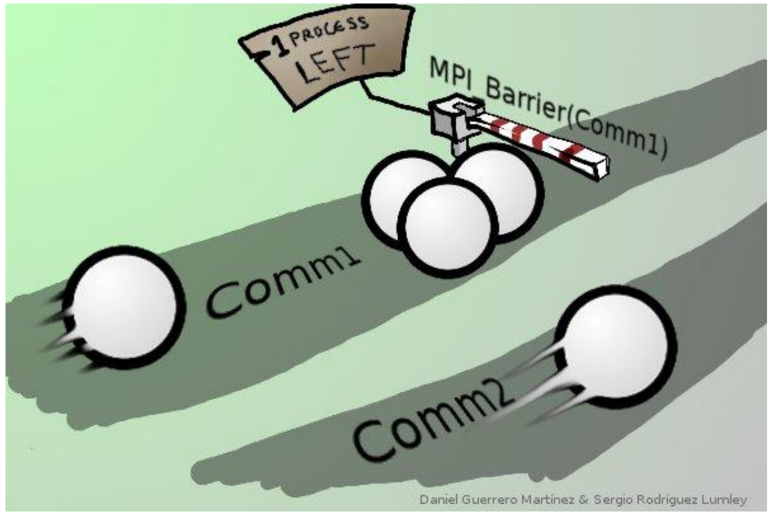
Types of Collective Operations

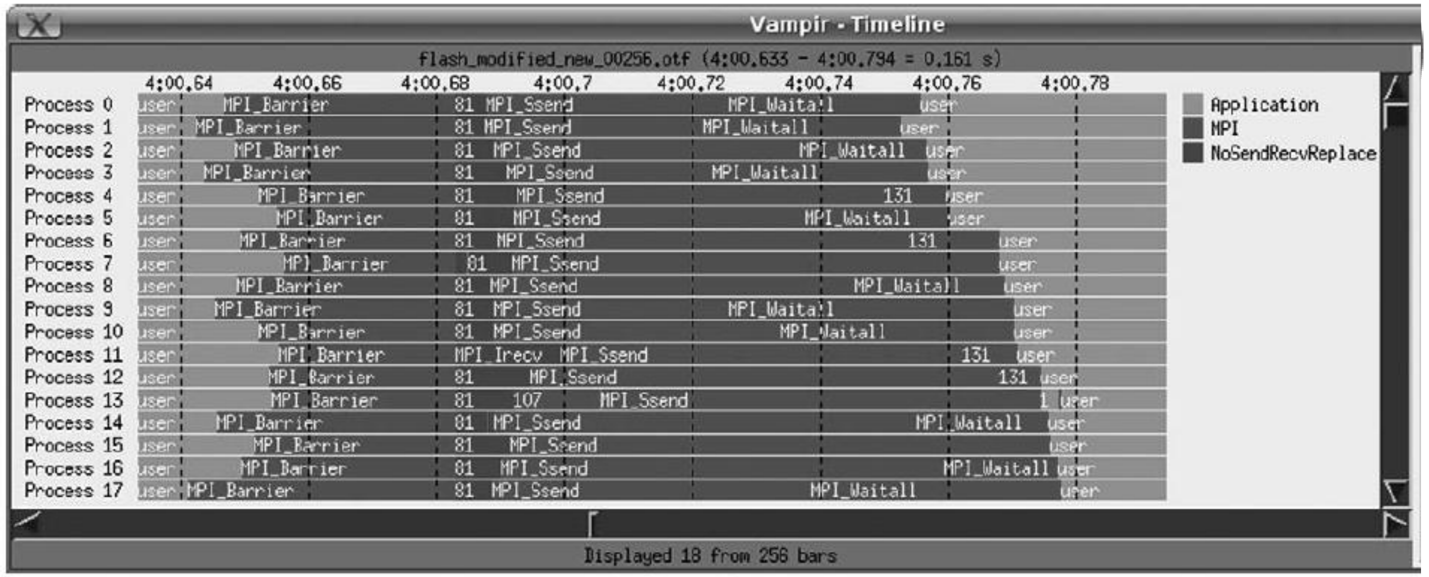
- Synchronization - processes wait until all members of the group have reached the synchronization point.
  - Data Movement
  - Broadcast
  - Scatter/gather
  - All-to-All.
- Collective Computation (reductions)
  - one member of the group collects data from the other members and performs an operation on that data
    - Min
    - Max
    - Add
    - multiply

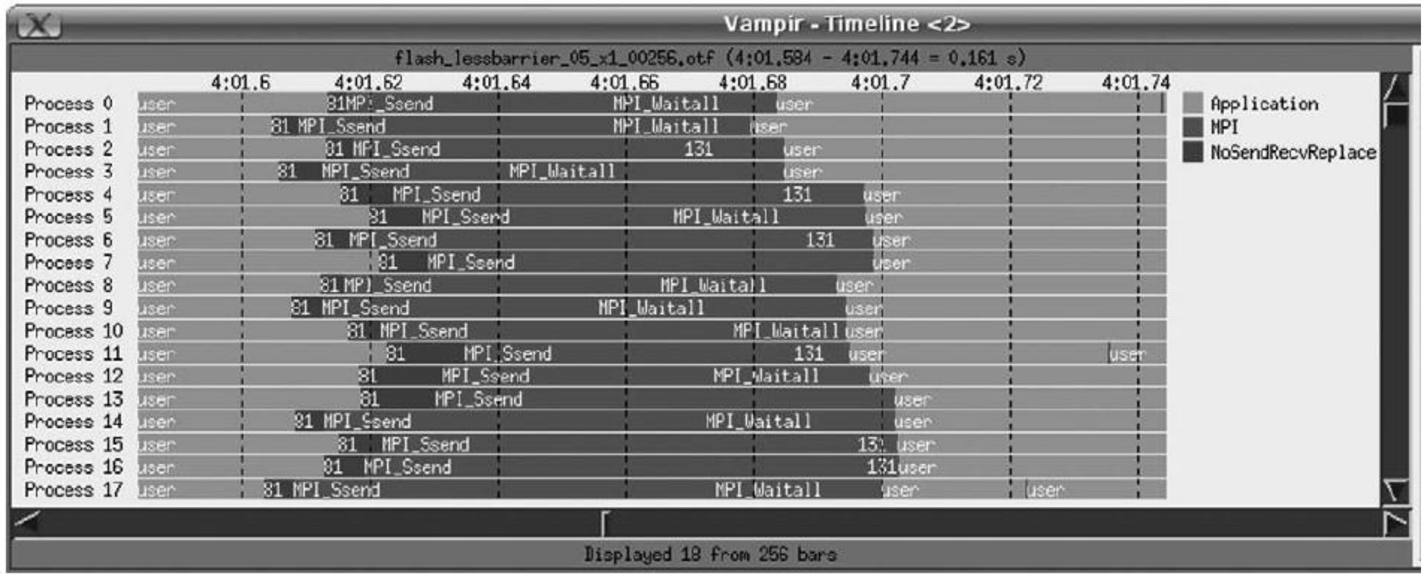


MPI_Barrier

- MPI_Barrier(MPI_Comm Comm)
- Blocks until all processes have reached this routine
  - Blocks the caller until all group members have called it



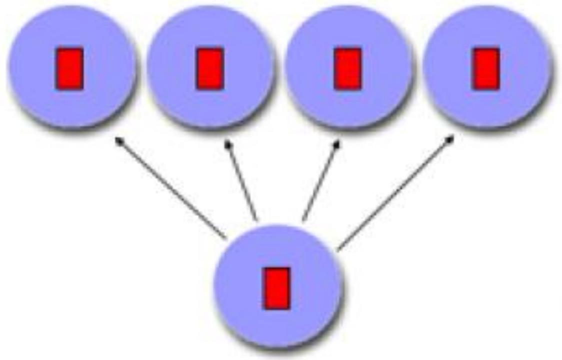

Vampir - Timeline
flash_modified_new_00256.otf (4:00.633 - 4:00.794 = 0.161 s)
Application
MPI
NoSendRecvReplace
MPI_Barrier
MPI_Ssend
MPI_Waitall
Displayed 18 from 256 bars


Vampir - Timeline <2>
flash_lessbarrier_05_x1_00256.otf (4:01.584 - 4:01.744 = 0.161 s)
Application
MPI
NoSendRecvReplace
MPI_Ssend
MPI_Waitall
Displayed 18 from 256 bars

An MPI Barrier call before a communication phase ensures a synchronized start of the communication calls (top). When removing the barrier there is an un-synchronized start (bottom)

Broadcast

- int MPI_Bcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype, int root, MPI_Comm comm)
  - broadcasts a message from the process with rank root to all processes of the group, itself included.
  - It is called by all members of the group using the same arguments for comm and root.
  - On return, the content of root's buffer is copied to all other processes.

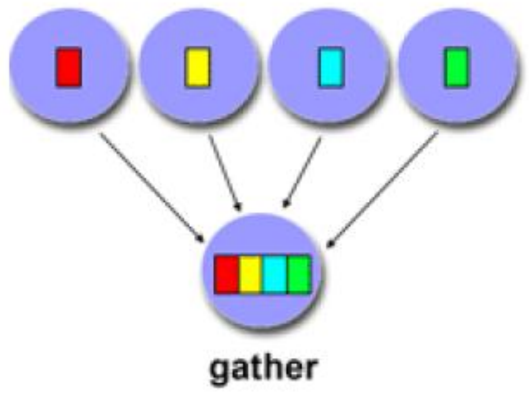

broadcast

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (non-negative integer)
IN	datatype	datatype of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

Gather

- int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
  - each process (root process included) sends the contents of its send buffer to the root process.
  - The root process receives the messages and stores them in rank order
  - The receive buffer is ignored for all non-root processes
  - Note that the recvcount argument at the root indicates the number of items it receives from each process, not the total number of items it receives

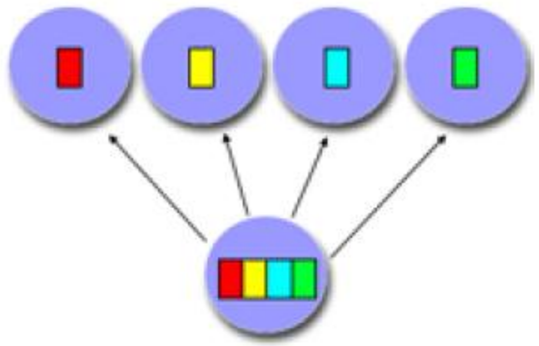


MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	datatype of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)
IN	recvtype	datatype of recv buffer elements (handle, significant only at root)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

Scatter

- 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 sends a message with MPI_Send(sendbuf,sendcount, sendtype,…). This message is split into n equal segments, the i-th segment is sent to the i-th process in the group, and each process receives this message as above.
  - The send buffer is ignored for all non-root processes



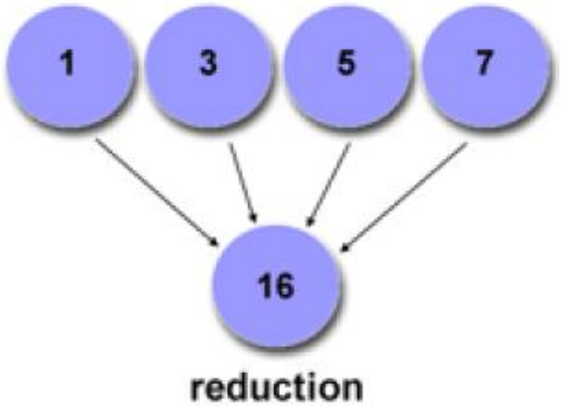
scatter

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)
IN	sendtype	datatype of send buffer elements (handle, significant only at root)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	datatype of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

Reduce

- int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root.
  - The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype;



MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	sendbuf	address of send buffer (choice)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of elements of send buffer (handle)
IN	op	reduce operation (handle)
IN	root	rank of root process (integer)
IN	comm	communicator (handle)

Reduce Operations

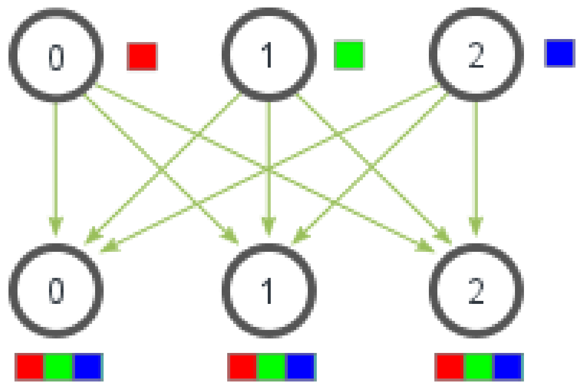
- MPI_MAX - Returns the maximum element
- MPI_MIN - Returns the minimum element
- MPI_SUM - Sums the elements.
- MPI_PROD - Multiplies all elements.
- MPI_LAND - Performs a logical and across the elements
- MPI_LOR - Performs a logical or across the elements
- MPI_BAND - Performs a bitwise and across 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

Other Collective Opertaions

- MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root

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

MPI_Allgather



`sendbuf` [in]: The starting address of the send buffer.

`sendcount` [in]: The number of elements in the send buffer.

`sendtype` [in]: The data type of each element in the send buffer (e.g., `MPI_INT`, `MPI_FLOAT`).

`recvbuf` [out]: The starting address of the receive buffer. This buffer must be large enough to hold data from all processes. The data from the *j*-th process is placed in the *j*-th block of the receive buffer.

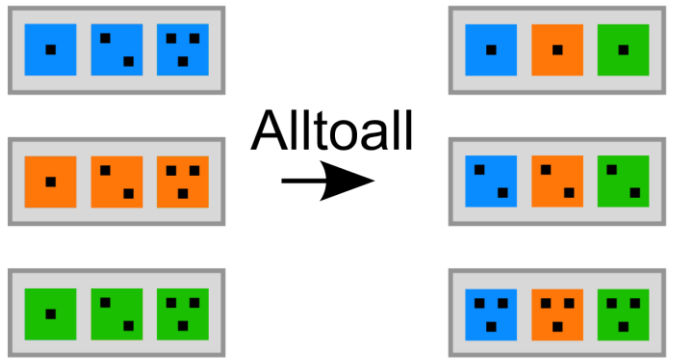
`recvcount` [in]: The number of elements to be received *from each process*, not the total count from all processes.

`recvtype` [in]: The data type of each element in the receive buffer.

`comm` [in]: The communicator (e.g., `MPI_COMM_WORLD`).

Other Collective Opertaions

- MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf.



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

`sendbuf` **[in]:** The starting address of the send buffer. Each process has a buffer containing elements that will be scattered across all processes.

`sendcount` **[in]:** The number of elements to send to each process. Note that this is the count for each *individual* destination, not the total size of the send buffer.

`sendtype` **[in]:** The data type of the send buffer elements (e.g., `MPI_INT`, `MPI_FLOAT`).

`recvbuf` **[out]:** The starting address of the receive buffer.

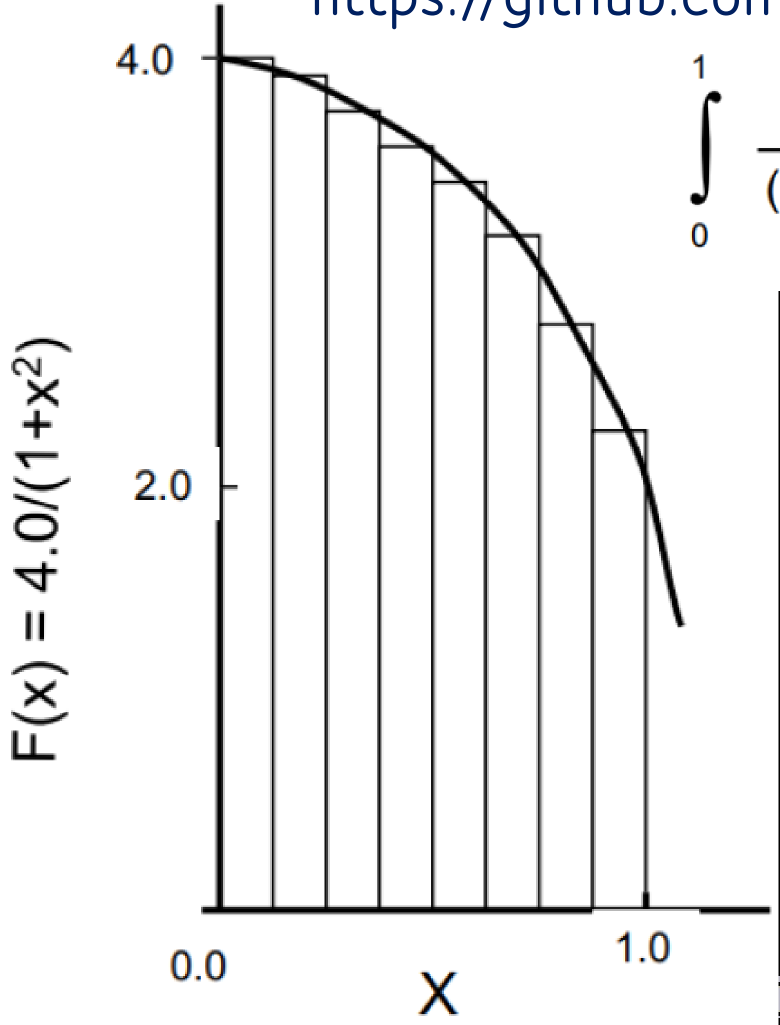
`recvcount` **[in]:** The number of elements to receive from any process. The total size of the receive buffer should be `recvcount` * (number of processes in `comm`).

`recvtype` **[in]:** The data type of the receive buffer elements.

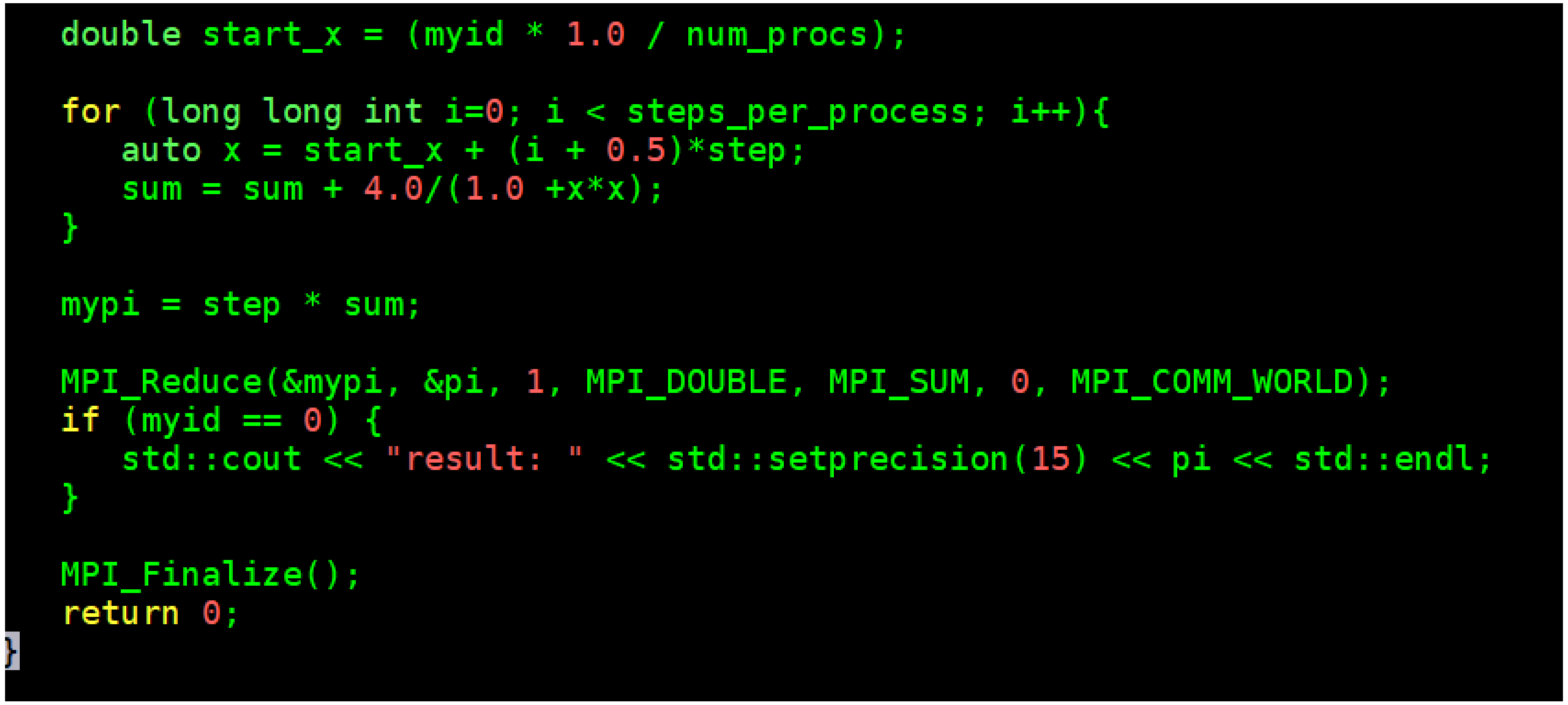
`comm` **[in]:** The communicator (handle) within which the operation takes place.

The MPI_Pi

https://github.com/infn-esc/sesame26/blob/main/hands-on/mpi/MPI_Pi.cpp



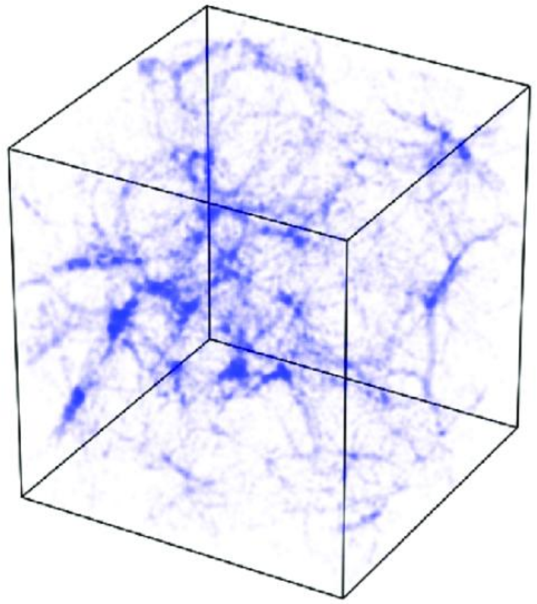
$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi \quad \Rightarrow \quad \sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$



```
    double start_x = (myid * 1.0 / num_procs);

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

    mypi = step * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) {
       std::cout << "result: " << std::setprecision(15) << pi << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

The trivial MPI N_Body

- Each process handles a subset of bodies
- At each time step, forces are computed and positions/velocities are updated.
- MPI is used to share body data across processes.
- Disclaimers
  - The code uses **explicit Euler integration**, which is simple but **not stable** for systems with strong forces or long time steps.
    - It can lead to **energy drift** and unrealistic trajectories over time.
  - Force Singularities:
    - The gravitational force becomes infinite as distance → 0.
    - The code adds a small softening term (+1e-9) to avoid division by zero, but this is a crude fix.
  - Time Step Sensitivity:
    - A fixed time step (dt = 0.01) may be too large or too small depending on the system.
    - Adaptive time stepping is often better



$$F = \frac{Gm_1m_2}{r^2}$$

$$F = \frac{Gm_1m_2}{(r^2 + \varepsilon^2)}$$

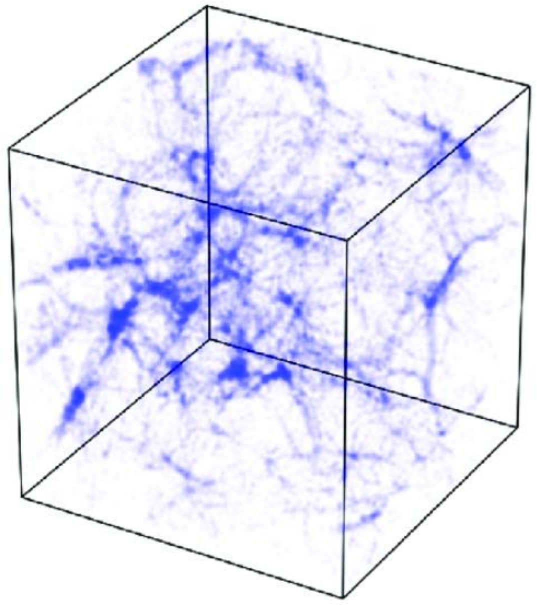
Where:

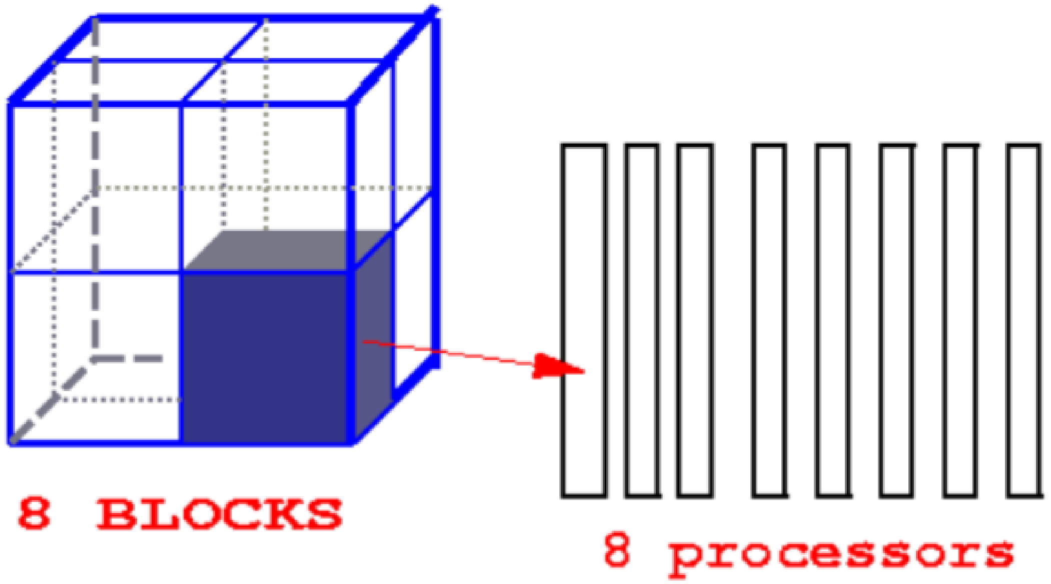
$\varepsilon$is the **softening length**.

It prevents the force from becoming infinite as $r \to 0$

The trivial MPI N_Body

- Each process handles a subset of bodies
- At each time step, forces are computed and positions/velocities are updated.
- MPI is used to share body data across processes.



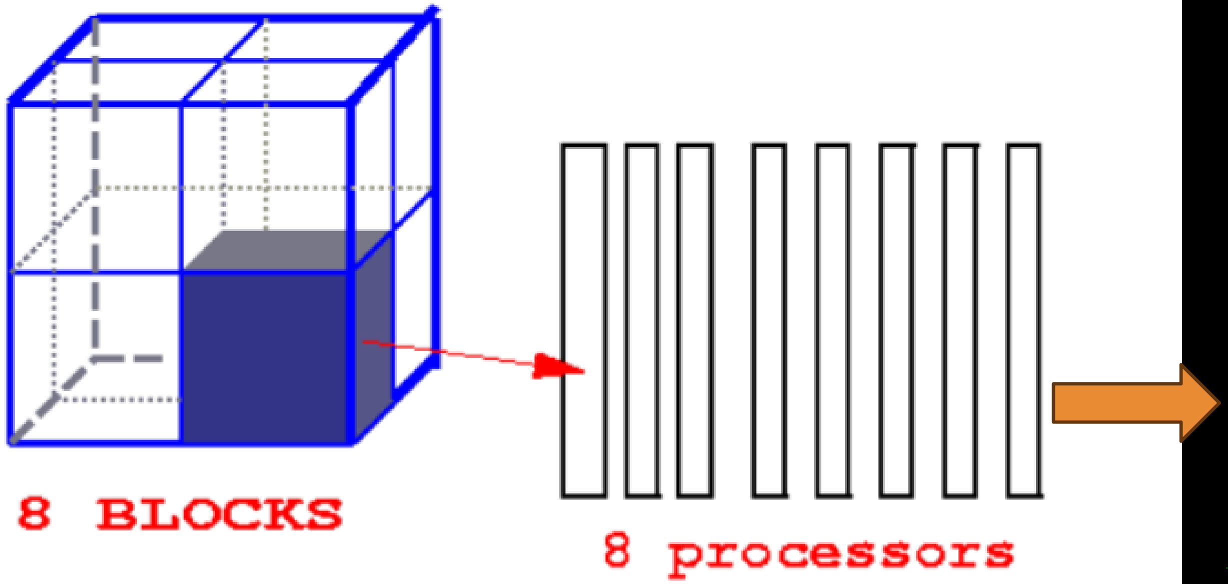


Each process maintains a local copy of the entire integration domain which is updated at each integration step by all the other processes via MPI_Allgather

The trivial MPI N_Body simulation

- Each process handles a subset of bodies
- At each time step, forces are computed and positions/velocities are updated.
- MPI is used to share body data across processes.

```
int local_N = N / size;
int start = rank * local_N;
int end = (rank == size - 1) ? N : start + local_N;
```



```
for (int step = 0; step < STEPS; ++step) {
    std::vector<Body> new_bodies = bodies;

    for (int i = start; i < end; ++i) {
        double fx = 0, fy = 0, fz = 0;
        for (int j = 0; j < N; ++j) {
            if (i != j) {
                compute_force(bodies[i], bodies[j], fx, fy, fz);
            }
        }

        new_bodies[i].vx += fx / bodies[i].mass * dt;
        new_bodies[i].vy += fy / bodies[i].mass * dt;
        new_bodies[i].vz += fz / bodies[i].mass * dt;

        new_bodies[i].x += new_bodies[i].vx * dt;
        new_bodies[i].y += new_bodies[i].vy * dt;
        new_bodies[i].z += new_bodies[i].vz * dt;
    }

    MPI_Allgather(&new_bodies[start], local_N * sizeof(Body), MPI_BYTE,
                  bodies.data(), local_N * sizeof(Body), MPI_BYTE,
                  MPI_COMM_WORLD);

    if (rank == 0 && step % 10 == 0) {
        std::cout << "Completed step " << step << " of " << STEPS << "\n";
    }
}
```

Process affinity

Run-time Tuning: Process Affinity

- Open MPI supports processor affinity on a variety of systems through process binding
  - Each MPI process is "bound" to a specific subset of processing resources (cores, sockets, L* cache, hwthread etc.).
  - The operating system will constrain that process to run on only that subset
- Affinity can improve performance by inhibiting excessive process movement
  - for example, away from "hot" caches or NUMA memory.
- Judicious bindings can improve performance
  - by reducing resource contention (by spreading processes apart from one another)
  - improving inter-process communications (by placing processes close to one another).
- Binding can also improve performance reproducibility by eliminating variable process placement.
- Unfortunately, binding can also degrade performance by inhibiting the OS capability to balance loads.
- Depending on how processing units on your node are numbered, the binding pattern may be good, bad, or even disastrous
  - If you want to control affinity you have to know what you are doing

Mapping, Ranking, and Binding: Oh My!

- Open MPI employs a three-phase procedure for assigning process locations and ranks:
  - Mapping
    - Assigns a default location to each process
  - Ranking
    - Assigns an MPI_COMM_WORLD rank value to each process
  - Binding
    - Constrains each process to run on specific processors
- To control process mapping in the command line:
  - --map-by <foo>
  - Map to the specified object, defaults to socket.
  - <foo> can be: slot, hwthread, core, L1cache, L2cache, L3cache, socket, numa, board, node, sequential, distance, and ppr.

Often a good choice is to let MPI decide for you. But if you want to master the MPI mapping, the mpirun manual is a good starting point, i.e.: https://www.open-mpi.org/doc/v4.1/man1/mpirun.1.php

Run-time Tuning - Binding

- In Open-MPI - mpirun automatically binds processes as of the start of the v1.8 series
  - Two binding patterns are used in the absence of any further directives:
    - Bind to core: when the number of processes is <= 2
    - Bind to socket: when the number of processes is > 2
- To control process binding in the command line:
  - --bind-to <foo>:
    - Bind processes to the specified object, defaults to core.
    - Supported options include slot, hwthread, core, l1cache, l2cache, l3cache, socket, numa, board, and none.
  - -report-bindings, --report-bindings: Report any bindings for launched processes.

Fine binding: The rankfile

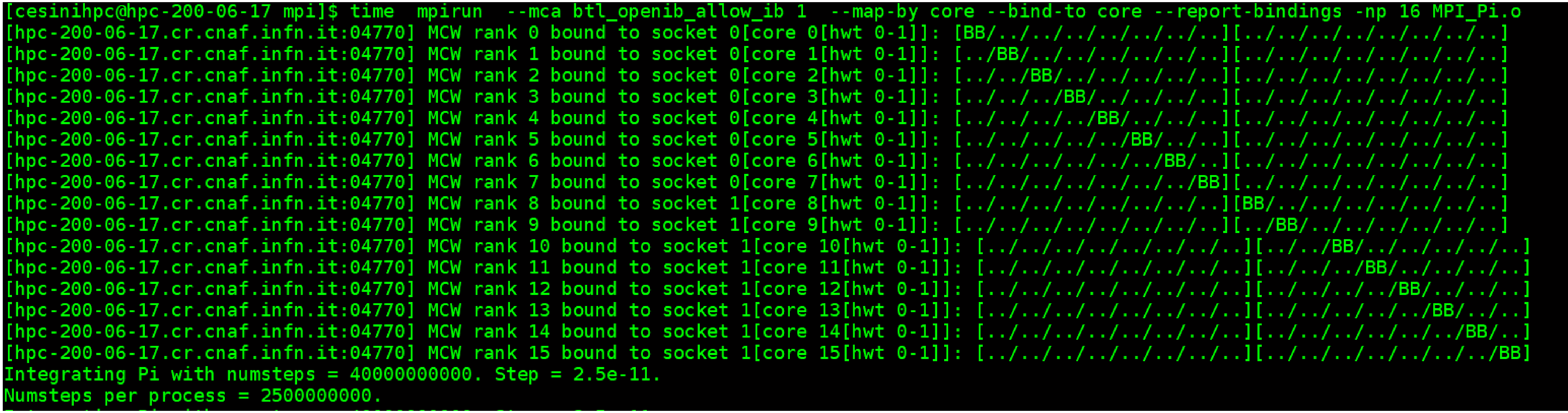
- -rf, --rankfile <rankfile>
  - Provide a rankfile file for fine control of the process allocation
  - rank <N>=<hostname> slot=<slot list>

For example:

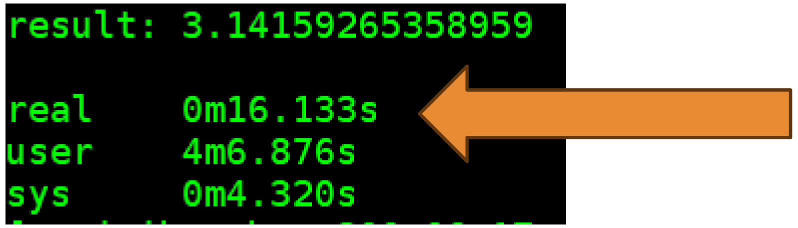
```
$ cat myrankfile
rank 0=aa slot=1:0-2
rank 1=bb slot=0:0,1
rank 2=cc slot=1-2
```

Rank 0 runs on node aa, bound to logical socket 1, cores 0-2.
Rank 1 runs on node bb, bound to logical socket 0, cores 0 and 1.
Rank 2 runs on node cc, bound to logical cores 1 and 2.

Bind-to core example

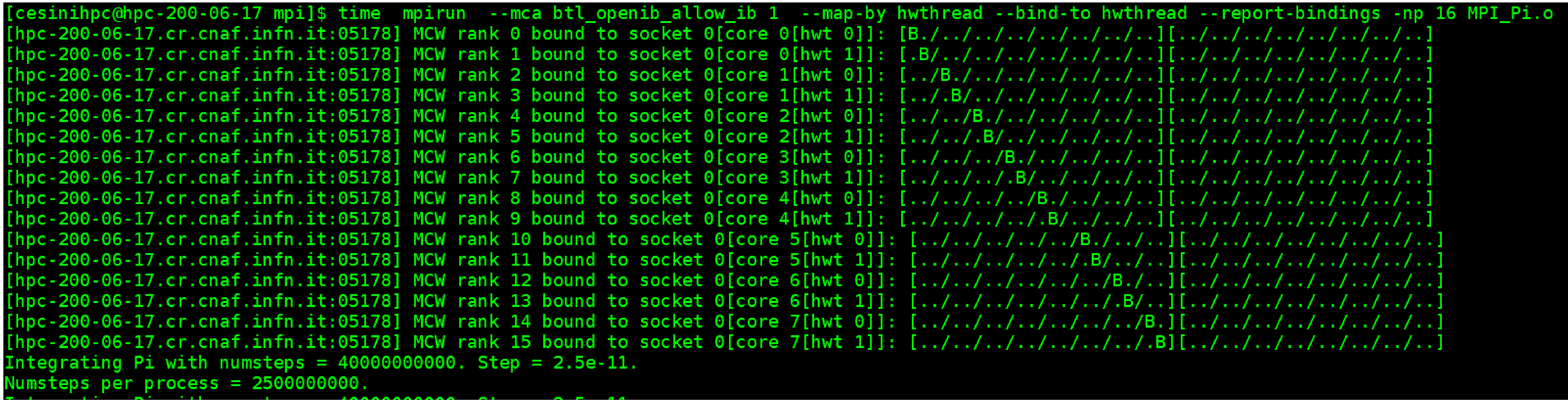


```
[cesinihpc@hpc-200-06-17 mpi]$ time  mpirun  --mca btl_openib_allow_ib 1  --map-by core --bind-to core --report-bindings -np 16 MPI_Pi.o
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 0 bound to socket 0[core 0[hwt 0-1]]: [BB/../../../../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 1 bound to socket 0[core 1[hwt 0-1]]: [../BB/../../../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 2 bound to socket 0[core 2[hwt 0-1]]: [../../BB/../../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 3 bound to socket 0[core 3[hwt 0-1]]: [../../../BB/../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 4 bound to socket 0[core 4[hwt 0-1]]: [../../../../BB/../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 5 bound to socket 0[core 5[hwt 0-1]]: [../../../../../BB/../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 6 bound to socket 0[core 6[hwt 0-1]]: [../../../../../../BB/..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 7 bound to socket 0[core 7[hwt 0-1]]: [../../../../../../../BB][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 8 bound to socket 1[core 8[hwt 0-1]]: [../../../../../../../..][BB/../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 9 bound to socket 1[core 9[hwt 0-1]]: [../../../../../../../..][../BB/../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 10 bound to socket 1[core 10[hwt 0-1]]: [../../../../../../../..][../../BB/../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 11 bound to socket 1[core 11[hwt 0-1]]: [../../../../../../../..][../../../BB/../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 12 bound to socket 1[core 12[hwt 0-1]]: [../../../../../../../..][../../../../BB/../../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 13 bound to socket 1[core 13[hwt 0-1]]: [../../../../../../../..][../../../../../BB/../..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 14 bound to socket 1[core 14[hwt 0-1]]: [../../../../../../../..][../../../../../../BB/..]
[hpc-200-06-17.cr.cnaf.infn.it:04770] MCW rank 15 bound to socket 1[core 15[hwt 0-1]]: [../../../../../../../..][../../../../../../../BB]
Integrating Pi with numsteps = 40000000000. Step = 2.5e-11.
Numsteps per process = 2500000000.
```

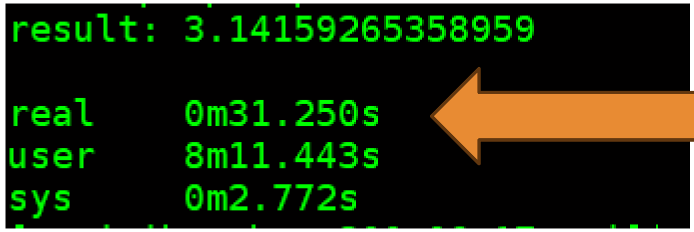


```
result: 3.14159265358959

real    0m16.133s
user    4m6.876s
sys     0m4.320s
```

A disastrous binding example



```
[cesinihpc@hpc-200-06-17 mpi]$ time  mpirun  --mca btl_openib_allow_ib 1  --map-by hwthread --bind-to hwthread --report-bindings -np 16 MPI_Pi.o
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 0 bound to socket 0[core 0[hwt 0]]: [B./../../../../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 1 bound to socket 0[core 0[hwt 1]]: [.B/../../../../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 2 bound to socket 0[core 1[hwt 0]]: [../B./../../../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 3 bound to socket 0[core 1[hwt 1]]: [../.B/../../../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 4 bound to socket 0[core 2[hwt 0]]: [../../B./../../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 5 bound to socket 0[core 2[hwt 1]]: [../../.B/../../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 6 bound to socket 0[core 3[hwt 0]]: [../../../B./../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 7 bound to socket 0[core 3[hwt 1]]: [../../../.B/../../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 8 bound to socket 0[core 4[hwt 0]]: [../../../../B./../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 9 bound to socket 0[core 4[hwt 1]]: [../../../../.B/../../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 10 bound to socket 0[core 5[hwt 0]]: [../../../../../B./../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 11 bound to socket 0[core 5[hwt 1]]: [../../../../../.B/../..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 12 bound to socket 0[core 6[hwt 0]]: [../../../../../../B./..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 13 bound to socket 0[core 6[hwt 1]]: [../../../../../../.B/..][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 14 bound to socket 0[core 7[hwt 0]]: [../../../../../../../B.][../../../../../../../..]
[hpc-200-06-17.cr.cnaf.infn.it:05178] MCW rank 15 bound to socket 0[core 7[hwt 1]]: [../../../../../../../.B][../../../../../../../..]
Integrating Pi with numsteps = 40000000000. Step = 2.5e-11.
Numsteps per process = 2500000000.
```



```
result: 3.14159265358959

real    0m31.250s
user    8m11.443s
sys     0m2.772s
```

Run-time tuning: Memory Affinity

- Open MPI supports general and specific memory affinity,
- it generally tries to allocate all memory local to the processor that asked for it.
- When shared memory is used for communication, Open MPI uses memory affinity to make certain pages local to specific processes in order to minimize memory network/bus traffic.

Reference Material & More Exercises

- MPI Standard: https://www.mpi-forum.org/docs/
- Open-mpi.org: https://www.open-mpi.org/software/ompi/v5.0/
  - https://www.open-mpi.org/faq/
- MPICH.org: https://www.mpich.org/
- MPI Tutorial: https://mpitutorial.com/
- Message Passing Interface (MPI). Author: Blaise Barney, Lawrence Livermore National
  - https://hpc-tutorials.llnl.gov/mpi/
- Tutorial and exercises @ Argonne National Laboratory: https://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/contents.html
- www.google.com