High Performance Linear Algebra

Lecture 7: Distributed Memory Machines and MPI

Ph.D. program in High Performance Scientific Computing

Fabio Durastante **Pasqua D'Ambra** **Salvatore Filippone**

January 12, 2026 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

Before the time-skip

- We have seen in the first half of the course:
  - The basic concepts of parallel computing
  - The main architectures for parallel computing
  - The main programming models for parallel computing
- We have introduced the BLAS libraries for linear algebra computations
  - We have seen the Level 1, Level 2 and Level 3 BLAS operations
  - We have discussed the performance of BLAS operations on shared memory architectures
  - Explored the OpenMP programming model
  - Employed the roofline model to analyze the performance of BLAS operations

Before the time-skip

- We have seen in the first half of the course:
  - The basic concepts of parallel computing
  - The main architectures for parallel computing
  - The main programming models for parallel computing
- We have introduced the BLAS libraries for linear algebra computations
  - We have seen the Level 1, Level 2 and Level 3 BLAS operations
  - We have discussed the performance of BLAS operations on shared memory architectures
  - Explored the OpenMP programming model
  - Employed the roofline model to analyze the performance of BLAS operations

And now *to boldly go out of shared memory architectures*...

Table of Contents

2 Distributed memory machines

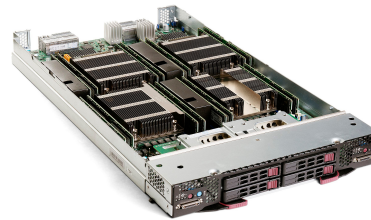
Distributed memory machines

- In distributed memory machines, each processor has its own private memory
- Processors communicate by passing messages through a network
- Examples of distributed memory machines:
  - Clusters of workstations connected by a high-speed network,
  - Massively parallel supercomputers (e.g., the machines of the TOP500 list)
- Programming models for distributed memory machines:
  - **M**essage **P**assing **I**nterface (MPI)
  - **P**artitioned **G**lobal **A**ddress **S**pace (PGAS) languages (e.g., Coarray Fortran)

Nodes

2 Distributed memory machines

- A distributed memory machine is composed of multiple nodes
- Each node contains one or more processors (CPUs) and its own private memory
- Nodes are connected by a high-speed network that allows them to communicate with each other
- Each node can run one or more processes that execute the parallel program
- Each node can have one or more accelerators (e.g., GPUs) to offload computations



General information on networks

2 Distributed memory machines

- In distributed memory machines, communication between processors occurs through a network
- Network performance is characterized by two key parameters:
  - Latency (α): time to send a message of zero length
  - Bandwidth (β): inverse of time to send one byte of data
- The time to send a message of size n bytes is modeled as:

$$T_{\text{comm}}(n) = \alpha + \frac{n}{\beta}$$

- **Latency** is typically measured in microseconds (µs)
- **Bandwidth** is typically measured in gigabits per second (Gbit s^{-1})

What determines these parameters?

- α Depends almost entirely on the operating system stack. To minimize latency: avoid TCP/IP protocols
- β Depends on both the operating system stack *and* the physical communication device hardware

Key insight: Low latency requires careful software optimization, while high bandwidth depends on specialized hardware (e.g., InfiniBand).

Some examples from the market

InfiniBand **H**igh **D**ynamic **R**ange (HDR) 2018:

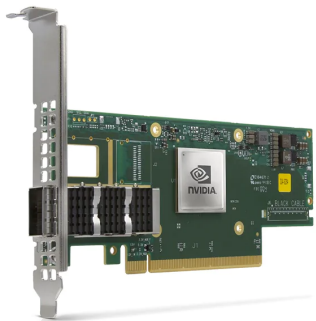
- Latency: $<0.6\,\mu s$
- Bandwidth: $200\,\text{Gbit s}^{-1}$

The *InfiniBand* technology is widely used in high-performance computing clusters, and it is the network technology used in many TOP500 supercomputers.

Some examples from the market

InfiniBand **H**igh **D**ynamic **R**ange (HDR) 2018:

- Latency: $< 0.6\,\mu s$
- Bandwidth: $200\,\text{Gbit}\,\text{s}^{-1}$



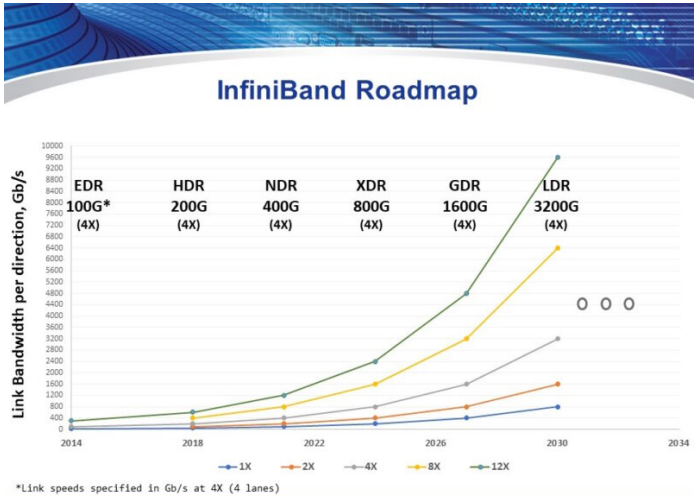
There are different vendors for Infiniband network adapters, e.g., Mellanox (now part of NVIDIA):

- NVIDIA/Mellanox Compatible AOC 20m InfiniBand HDR Active Optical Cable: 910 USD,
- NVIDIA/Mellanox MCX653105A-HDAT-SP ConnectX® -6 InfiniBand Adapter Card, HDR/200G: 1069 USD,
- NVIDIA MQM8700-HS2F Quantum HDR InfiniBand Switch, 40 x HDR QSFP56 Ports: 17538 USD.

Some examples from the market

2 Distributed memory machines

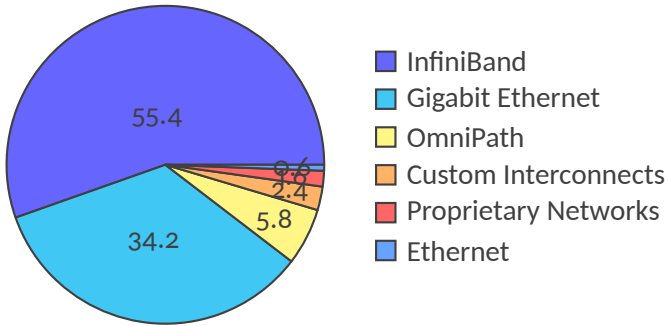
Newer InfiniBand standard exists, but are not yet widely used in HPC clusters



The TOP500 supercomputer situation

2 Distributed memory machines

From the November 2025 TOP500 list[1], the distribution of interconnects used in the top 500 supercomputers is as follows:



[1] https://www.top500.org/lists/top500/2025/11/

Should we care about network topology?

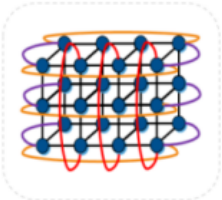
In large distributed memory machines, the network topology can have a significant impact on the performance of parallel applications.

- Different topologies have different characteristics in terms of latency, bandwidth, and scalability.
- Common network topologies include:
  - Fat-tree,
  - Torus,
  - Hypercube,
  - Dragonfly.
- The choice of network topology can affect the performance of collective communication operations.

Should we care about network topology?

In large distributed memory machines, the network topology can have a significant impact on the performance of parallel applications.

- Different topologies have different characteristics in terms of latency, bandwidth, and scalability.
- Common network topologies include:
  - Fat-tree,
  - Torus,
  - Hypercube,
  - Dragonfly.
- The choice of network topology can affect the performance of collective communication operations.

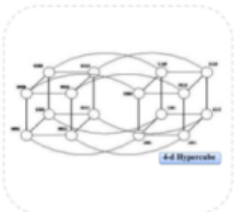


Should we care about network topology?

2 Distributed memory machines

In large distributed memory machines, the network topology can have a significant impact on the performance of parallel applications.

- Different topologies have different characteristics in terms of latency, bandwidth, and scalability.
- Common network topologies include:
  - Fat-tree,
  - Torus,
  - Hypercube,
  - Dragonfly.
- The choice of network topology can affect the performance of collective communication operations.



Should we care about network topology?

In large distributed memory machines, the network topology can have a significant impact on the performance of parallel applications.

- Different topologies have different characteristics in terms of latency, bandwidth, and scalability.
- Common network topologies include:
  - Fat-tree,
  - Torus,
  - Hypercube,
  - Dragonfly.
- The choice of network topology can affect the performance of collective communication operations.

Should we care about network topology?

2 Distributed memory machines

In large distributed memory machines, the network topology can have a significant impact on the performance of parallel applications.

- Different topologies have different characteristics in terms of latency, bandwidth, and scalability.
- Common network topologies
- The choice of network topology can affect the performance of collective communication operations.


Fat Tree


Dragonfly


Torus

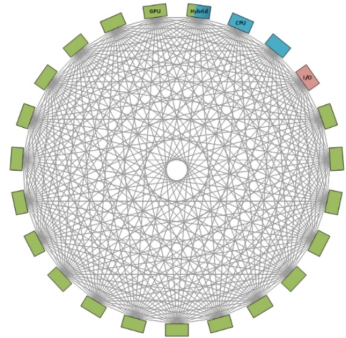

Hypercube

On the **implementation side**, we usually do not have to care, and think of the network as a black box with given latency and bandwidth.

An example: the Leonardo supercomputer *Dragonfly+*

- At top level, there are 23 cells fully connected in a dragonfly topology,
- Locally, intra-cell routers are organized in a **bipartite graph**
  - in which a *first tier* is directly connected to servers (leaf routers)
  - and a *second tier* (spine routers) is equally provisioned with down-links.

See the full description in: Turisini, Cestarti, Amati. “LEONARDO A Pan-European Pre-Exascale Supercomputer for HPC and AI applications”, Vol. 9 No. 1 (2024): Journal of large-scale research facilities.



Dragonfly topology of the internal network. Green is used for Booster cells, blue for Data-Centric cells, pink for the I/O.

Leonardo supercomputer: Network specifications

- Network Technology: 200 Gbps InfiniBand HDR (Mellanox/NVIDIA)
  - Switch latency: 90 nanoseconds port-to-port
  - Message rate: 390 million messages/second per port
  - Total switches: **823 QM8700 units**
- Node-level adapter: ConnectX-6 (CX6) card
  - 200 million messages per second capacity
  - 600 ns latency per Network Interface Card (NIC)
  - PCIe Gen4 on 32 lanes
- Maximum inter-node latency: 3 µs
  - Dominated by NIC delays: 1.2 µs
  - Fiber segments: 1 m (NIC to leaf), 5 m (leaf to spine), 20 m (spine to spine)
- External connectivity: 4 gateway routers with Ethernet-InfiniBand translators
  - 1.6 Tbit s^{-1} per unit, 6.4 Tbit s^{-1} aggregated

Table of Contents

3 How do we program such a machine?

Message Passing Interface (MPI)

3 How do we program such a machine?

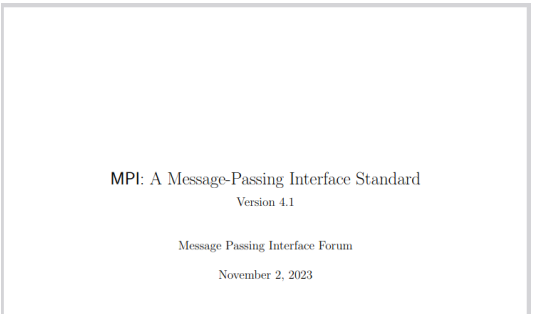
- The **M**essage **P**assing **I**nterface (MPI) is the de facto standard for programming distributed memory machines
- MPI provides a set of functions for:
  - Point-to-point communication (send/receive messages between two processes)
  - Collective communication (broadcast, scatter, gather, reduce, etc.)
  - Process management (creating and terminating processes)
- MPI is implemented as a library that can be used with different programming languages (C, C++, Fortran)
- There are several implementations of MPI
  - MPICH `https://www.mpich.org/`
  - OpenMPI `https://www.open-mpi.org/`
  - MVAPICH `https://mvapich.cse.ohio-state.edu/`
- The current stable version is 4.1, and work is underway to define version 5.0.

What is exactly MPI?

3 How do we program such a machine?

> *"MPI (Message-Passing Interface) is a message-passing library interface specification."*

All parts of this definition are significant.
See: https://www.mpi-forum.org/docs/


MPI: A Message-Passing Interface Standard
Version 4.1

Message Passing Interface Forum

November 2, 2023

MPI: Key aspects

3 How do we program such a machine?

- **Message-passing model**: Data moves from the address space of one process to another through cooperative operations
- **Specification, not implementation**: Multiple implementations exist (MPICH, OpenMPI, MVAPICH)
- **Library interface**: Operations expressed as functions, subroutines, or methods in C and Fortran
- **Extensions**: Collective operations, remote-memory access, dynamic process creation, parallel I/O

MPI basic concepts

3 How do we program such a machine?

An MPI program is composed of multiple processes that run concurrently on different processors

- Each process has a unique identifier called rank
- The total number of processes is called the size of the communicator
- The main communicator is called `MPI_COMM_WORLD`, which includes all processes
- Processes can communicate by sending and receiving messages using MPI functions

Why message passing?

- Each process has its **own private address space**
- Other processes **cannot access data directly**
- Processes must **cooperate** to exchange data through messages

An MPI hello world program in Fortran

Let us write a simple MPI program that prints “Hello, World!” from each process, we write a file called `mpi_hello.f90` with the following content:

```
program mpi_hello
    use mpi
    use iso_fortran_env, only: output_unit
    implicit none
    integer :: ierr, rank, size

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
    write(output_unit, *) 'Hello from process', rank, 'of', size
    call MPI_Finalize(ierr)
end program mpi_hello
```

Let us look at it line by line


```
use mpi
```

- This line includes the MPI module, which contains the definitions of MPI functions and constants

```
call MPI_Init(ierr)
```

- This line initializes the MPI environment, must be called before any other MPI function

```
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
```

- This line gets the rank of the current process in the communicator

```
call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
```

- This line gets the total number of processes in the communicator

Let us look at it line by line

3 How do we program such a machine?

```
write(output_unit, *) 'Hello from process', rank, 'of', size
```

- This line prints a message from each process, including its rank and the total size

```
call MPI_Finalize(ierr)
```

- This line finalizes the MPI environment, must be called at the end of the program

- The program declares three integer variables:
  - `rank`: to store the rank of the current process
  - `size`: to store the total number of processes
  - `ierr`: to store the error code returned by MPI functions

  The variable `ierr` is used to capture error codes from MPI functions, and should be used for error handling in a production code.

Compiling and running the MPI program

To compile the MPI program, we need to have installed an MPI implementation (e.g., MPICH, OpenMPI, MVAPICH).

On an Ubuntu system, we can install OpenMPI with the following command:

```
sudo apt-get install libopenmpi-dev openmpi-bin openmpi-common
```

If you are using Spack to manage your software, you can install OpenMPI with:

```
spack install openmpi
spack load openmpi
```

In a system **with multiple MPI implementations**, make sure to load the correct one using `module load` or `spack load`.

Compiling and running the MPI program

All these implementations provide a wrapper compiler that simplifies the compilation process by automatically including the necessary **MPI headers** and **linking** against the **MPI libraries**.

For example, using OpenMPI, we can compile the program with the following command:

```
mpifort -o mpi_hello mpi_hello.f90
```

If you are using MPICH or MVAPICH, the command is the same.

Compiling and running the MPI program

If you want to investigate what the wrapper compiler is doing behind the scenes:

```
mpifort --show:me
```

This will display the actual compilation command, e.g., on my machine it shows:

```
/usr/bin/gfortran -I/opt/spack/opt/spack/linux-skylake/openmpi-4.1.8/include
↪  -I/opt/spack/opt/spack/linux-skylake/openmpi-4.1.8/lib
↪  -L/opt/spack/opt/spack/linux-skylake/openmpi-4.1.8/lib
↪  -L/opt/spack/opt/spack/linux-skylake/hwloc-2.12.2/lib
↪  -L/opt/spack/opt/spack/linux-skylake/libevent-2.1.12/lib -Wl,-rpath
↪  -Wl,/opt/spack/opt/spack/linux-skylake/openmpi-4.1.8/lib -Wl,-rpath
↪  -Wl,/opt/spack/opt/spack/linux-skylake/hwloc-2.12.2/lib -Wl,-rpath
↪  -Wl,/opt/spack/opt/spack/linux-skylake/libevent-2.1.12/lib -lmpi_usempif08
↪  -lmpi_usempi_ignore_tkr -lmpi_mpifh -lmpi
```

showing that it is using `gfortran` as the underlying compiler, and including the necessary MPI headers and libraries from Spack.

Compiling and running the MPI program

To run the MPI program, we use the `mpirun` or `mpiexec` command, specifying the number of processes with the `-n` option:

```
mpirun -n 4 ./mpi_hello
```

This command runs the `mpi_hello` program with 4 processes, and we should see output similar to:

```
Hello from process           1 of           4
Hello from process           2 of           4
Hello from process           0 of           4
Hello from process           3 of           4
```

Note that **the order of the output may vary**, as the processes run concurrently.

Finding MPI via CMake

To find and use MPI in a CMake project, we can use the `FindMPI` module provided by CMake, i.e., we can add the following lines to our `CMakeLists.txt` file:

```
find_package(MPI REQUIRED COMPONENTS Fortran)
```

This command searches for an installed MPI implementation and sets the necessary variables to use MPI in our project.

To compile an MPI program, we need to link against the MPI libraries and include the MPI headers. We can do this by adding the following lines to our `CMakeLists.txt` file:

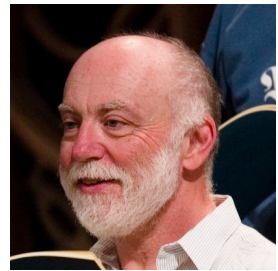
```
add_executable(mpi_hello mpi_hello.f90)
target_link_libraries(mpi_hello MPI::MPI_Fortran)
```

The fallacies of distributed computing

3 How do we program such a machine?

There are some common misconceptions about distributed computing that can lead to poor performance and scalability

- These misconceptions are known as the **fallacies of distributed computing**:
  2. The network is reliable
  8. Latency is zero
  1. Bandwidth is infinite
  4. The network is secure
  3. Topology doesn't change
  5. There is one administrator
  6. Transport cost is zero
  7. The network is homogeneous
- It is important to be aware of these fallacies when designing and implementing distributed applications



L. Peter Deutsch

Working on a cluster/shared machine with MPI

3 How do we program such a machine?

To work on a cluster or shared machine with MPI, we typically need to follow these steps:

- Connect to the cluster using SSH
- Load the MPI module using `module load` or `spack load`
- ⚠ Compile the MPI program using the MPI wrapper compiler (e.g., `mpifort`, `mpicc`)
- Submit the MPI job to the job scheduler (e.g., SLURM, PBS, LSF) using a job script
- Monitor the job status and retrieve the output files

The compile step ⚠ may have to be done on compute nodes, depending on the cluster configuration.

The SLURM job scheduler

3 How do we program such a machine?

SLURM (Simple Linux Utility for Resource Management) is a popular **job scheduler** used in many HPC clusters.

- SLURM manages the allocation of resources (e.g., nodes, CPUs, memory) for jobs submitted by users
- Users submit jobs to SLURM using a job script that specifies the resources required and the commands to execute
- SLURM schedules jobs based on resource availability and job priorities
- Users can monitor the status of their jobs using SLURM commands (e.g., `squeue`, `sacct`)
- Once a job is completed, users can retrieve the output files generated by their jobs

SLURM glossary

node: A single physical or virtual machine in the cluster

task: A single instance of a program running on a node

job: A collection of tasks that are submitted to SLURM for execution

partition: A group of nodes with similar characteristics (e.g., hardware, software)

allocation: A reservation of resources (nodes, CPUs, memory) for a job

script: A file that contains the commands to execute a job, along with SLURM directives

interactive session: A temporary allocation of resources for interactive use (e.g., debugging, testing, compilation)

We usually have a number of **tasks per node**, depending on the number of available CPUs/cores, and a number of **CPUs per task**, depending on the number of threads we want to use per task.

Running an interactive session with SLURM

To run an interactive session with SLURM, we can use the `salloc` command, specifying the resources we need:

```
salloc -N 1 -n 4 --cpus-per-task=2 --time=01:00:00 --partition=cl1
```

This command requests an interactive session with:

- 1 nodes (`-N 1`),
- 4 tasks (`-n 4`),
- 2 CPUs per task (`--cpus-per-task=2`),
- a time limit of 1 hour (`--time=01:00:00`),
- on the `cl1` partition (`--partition=cl1`).

Running an interactive session with SLURM

To run an interactive session with SLURM, we can use the `salloc` command, specifying the resources we need:

```
salloc -N 1 -n 4 --cpus-per-task=2 --time=01:00:00 --partition=cl1
```

Which will print on screen something like:

```
salloc: Pending job allocation 20864
salloc: job 20864 queued and waiting for resources
salloc: job 20864 has been allocated resources
salloc: Granted job allocation 20864
```

After a while, when the resources are allocated, we will get a shell prompt on the compute node.

Running an interactive session with SLURM

Another option to run an interactive session is to use the `srun` command with the `--pty` option:

```
srun --pty -N 1 -n 4 --cpus-per-task=2 --time=01:00:00 --partition=cl1 bash
```

This command has the same effect as the previous one, but it directly starts a bash shell on the compute node.

If the cluster supports it, you can also use `ssh` to connect directly to a compute node for interactive work (after allocating resources with `salloc`), but this is less common.

Preparing a SLURM job script for MPI

An example of a SLURM job script `launch.sh` for running an MPI program:

```
#!/bin/bash
#SBATCH --job-name=mpi_hello
#SBATCH --output=mpi_hello.out
#SBATCH --error=mpi_hello.err
#SBATCH --nodes=2
#SBATCH --ntasks=4
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=1
#SBATCH --time=01:00:00
#SBATCH --partition=cl2
```

- The script starts with a shebang line (*#!/bin/bash*) to specify the shell to use
- The *#SBATCH* directives specify the job:
  - job name (`--job-name`),
  - output file (`--output`),
  - error file (`--error`),
  - number of nodes (`--nodes`),
  - number of tasks (`--ntasks`),
  - tasks per node (`--ntasks-per-node`),
  - CPUs per task (`--cpus-per-task`),
  - time limit (`--time`),
  - partition (`--partition`).

Output files options in SLURM job scripts

The `--output` and `--error` options in SLURM job scripts specify the files where the **standard output** and **standard error** streams of the job will be redirected.

By default, if these options are not specified, SLURM will create output files named `slurm-<jobid>.out` in the directory where the job was submitted.

You can customize the names of these files using the `--output` and `--error` options, together with some *special placeholders*:

- `%j`: Job ID
- `%N`: Node name
- `%n`: Task ID

Preparing a SLURM job script for MPI + OpenMP

If we want to use OpenMP in addition to MPI, we need to set the number of threads per process using the `OMP_NUM_THREADS` environment variable in the job script:

```
export OMP_NUM_THREADS=4
```

This line should be added before the command that runs the MPI program.

⚠ It is crucial to ensure that `OMP_NUM_THREADS` does not exceed the total number of available CPU cores on the allocated nodes to avoid oversubscription, i.e., the number of threads should be less than or equal to the number passsed to `--cpus-per-task`.

The execution command in SLURM job scripts

To run the MPI program in the SLURM job script, we use the `srun` command:

```
srun ./mpi_hello
```

This command launches the MPI program `mpi_hello` using the resources allocated by SLURM.

⚠ It is important to use `srun` instead of `mpirun` or `mpiexec` in SLURM job scripts, as `srun` is integrated with SLURM and ensures proper resource allocation and management; in many clusters `mpirun` and `mpiexec` are disabled or not recommended.

Submitting the SLURM job script and checking job status

To submit the SLURM job script, we use the `sbatch` command:

```
sbatch launch.sh
```

This command submits the job script `launch.sh` to SLURM for execution.

To check the status of the submitted job, we can use the `squeue` command:

```
squeue -u your_username
```

This command lists all the jobs submitted by the user `your_username`, showing their job IDs, statuses, and other information.
If the job is running or completed, we can check the output and error files specified in the job script.

Table of Contents

4 The Toeplitz cluster at DMPISA

The Toeplitz cluster at DMPISA

- The Toeplitz cluster is a distributed memory machine available at DMPISA for high-performance computing tasks
- It consists of multiple nodes, each equipped with powerful processors and a significant amount of memory
- The nodes are connected by a $10\,\text{Gbit}\,\text{s}^{-1}/25\,\text{Gbit}\,\text{s}^{-1}$ network providing communication between nodes
- The cluster is managed using the SLURM job scheduler, which allows users to submit and manage their jobs effectively
- Users can access the cluster remotely via SSH and utilize MPI for parallel programming

Specifications of the Toeplitz cluster at DMPISA

- The Toeplitz cluster consists of 9 nodes:
  - 4 AMD EPYC 7763 nodes: 2 threads per core, 64 cores per socket, 2 sockets, 2 TB of memory (1.96 TB usable).
  - 4 Intel Xeon E5-2650 v4 at 2.20 GHz nodes: 2 threads per core, 12 cores per socket, 2 sockets, 256 GB of memory (250 GB usable).
  - 1 Intel Xeon E5-2643 v4 at 3.40 GHz node: 2 threads per core, 6 cores per socket, 2 sockets, 128 GB of memory (125 GB usable).
- Network connectivity:
  - The first 4 AMD nodes are connected via fiber at 25 Gbit s^{-1}.
  - The remaining nodes use Intel Ethernet Controller X540-AT2 10-Gigabit NICs over copper, with a 10 Gbit s^{-1} switch.

Accessing the Toeplitz cluster

Once you receive your account credentials, you can log in to the cluster with the command:

```
ssh username@toeplitz.cs.dm.unipi.it
```

At the first connection, you will be asked to accept the machine's fingerprint.

! If you intend to use services with a graphical interface on the remote machine, you need to request SSH to forward the X11 server by adding the `-X` option to the previous command.

Setting up SSH key authentication

In general, it is useful to connect via SSH key. A key can be generated on your system by following the instructions given by:

```
ssh-keygen
```

Important: Set a passphrase for the generated key.

You can use the ssh-key we generated for GitHub if you want to.

Once the procedure is complete, you must copy the key to the remote machine with:

```
ssh-copy-id username@toeplitz.cs.dm.unipi.it
```

Every subsequent login from your machine will not require a password; the first login from your machine in each session will require the passphrase.

Partitions on Toeplitz

Toeplitz contains three different partitions:

Partition	Description	Time Limit	Nodes	Node List
`gpu`	2 threads/core, 128 threads/socket 2 sockets, 4 NVIDIA A40 (48GB RAM)	infinite	4	`gpu0[1-4]`
`cl1`	2 threads/core, 6 cores/socket 2 sockets	infinite	1	`lnx1`
`cl2`	2 threads/core, 12 cores/socket 2 sockets	infinite	4	`lnx[2-5]`
`all`	All nodes in the cluster	infinite	9	`lnx[1-5]`, `gpu0[1-4]`

Software management on Toeplitz

The software management on the cluster is performed using **Spack**, a package manager for supercomputers.

- Simplifies installation and management of scientific software
- Not tied to a specific programming language
- Allows creating software stacks in Python or R, linking to libraries in C, C++, or Fortran
- Easily switch compilers or program for specific microarchitectures

Environment Modules on Toeplitz

Environment Modules is a tool that simplifies shell initialization and allows users to modify their environment during the session.
To view available modules:

```
module avail
```

Modules are named with the following pattern:
`programname/version-compiler-version`
Example output:

```
anaconda3/2021.05-gcc-12.2.0
cmake/3.23.3-gcc-12.2.0
gcc/12.2.0
intel-oneapi-compilers/2022.1.0
openmpi/4.1.4-gcc-12.2.0
openblas/0.3.20-gcc-12.2.0
 valgrind/3.19.0-oneapi-2022.1.0
```

⚠ All loaded modules must refer to the **same compiler** for a consistent environment.

Loading and managing modules

Load modules:

```
module load programname1/version-compiler-version
↪  programname2/version-compiler-version
```

Remove modules:

```
module unload programname1/version-compiler-version
```

Revert to original state:

```
module purge
```

View active modules:

```
module list
```

Anaconda on Toeplitz

For installing and managing Python environments we use **Anaconda**.

- Anaconda is a Python environment designed to work with multiple isolated **environments**
- Each environment can contain different versions of software and modules
- This approach minimizes conflicts and undesired interactions between different projects

Best practices when starting a new project:

- Create a new environment (do not use the base environment)
- Install required software in the new environment
- Use the `$SCRATCH` directory for environments (more storage than home)

Anaconda on Toeplitz

Quick start guide:

```
module load anaconda3
conda create -p $SCRATCH/my-env-project
conda activate $SCRATCH/my-env-project
```

For subsequent uses, simply reload the module and activate the environment. Install packages with:

```
conda install pytorch torchvision torchaudio pytorch-cuda=12.1 -c pytorch
↪  -c nvidia
```

Conclusions and next steps

5 Conclusions

We have:

- ✔ Introduced distributed memory machines and MPI programming
- ✔ Explained how to compile and run MPI programs on a cluster
- ✔ Presented the Toeplitz cluster at DMPISA and its software management

Next steps:

- Explore communication routines in MPI