HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 11

Fabian Wermelinger
Harvard University
CS205

Tuesday, February 28th 2023

LAST TIME

- A few more OpenMP details regarding overhead and wait policies for threads
- What limits your performance on a single node?
- Computing nominal peak performance and peak bandwidth for CPUs (and GPUs)
- The operational intensity of compute kernels (algorithms in software)
- The roofline model

TODAY

Main topic: ***Introduction to distributed memory programming and MPI***

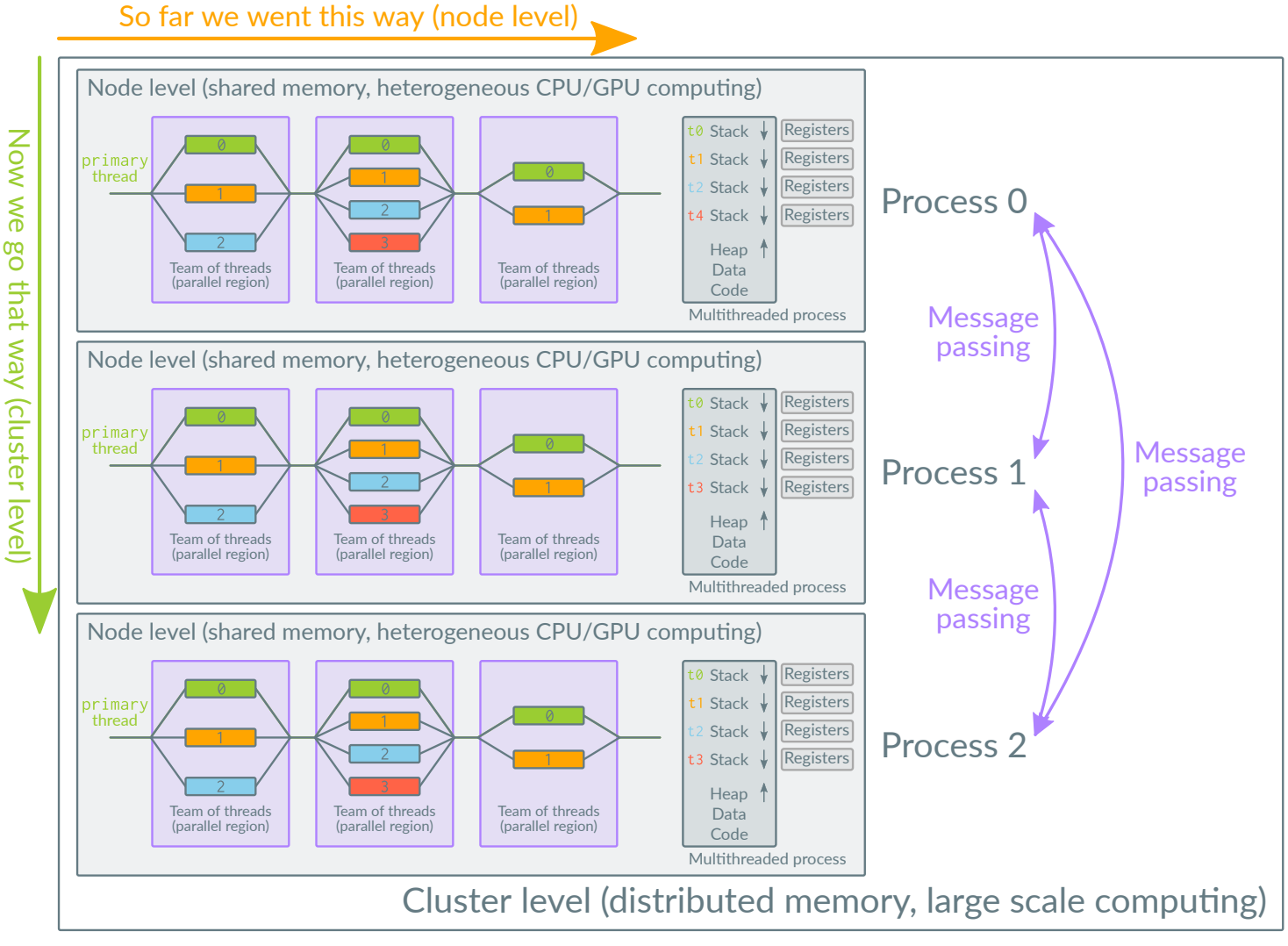
Details:

- Why distributed memory parallelism and what are differences to shared memory models.
- SPMD and MPMD high-level programming models.
- Introduction to MPI and its history
- MPI communicators and groups
- Point-to-point and collective communication
- Synchronous and asynchronous communication
- Anatomy of a MPI message
- Getting started with MPI and first examples

AGENDA CHECK:

- Reading assignment for Thursday is about blocking point-to-point communication in the MPI standard sections 3.1, 3.2, 3.4 and 3.5. The standard can be found in the Git repository: https://code.harvard.edu/CS205/main/blob/master/manuals/mpi40-report.pdf.
- Reading assignment for next Tuesday is about non-blocking point-to-point communication and the intro section for collective communication → MPI standard sections 3.7 and 6.1.
- Project M2 feedback will be posted today via GitHub issue in your project repository. Great proposals → we look forward to your parallel solutions.
- Quiz 2 review session this Thursday at 5pm in SEC 1.312 lounge.

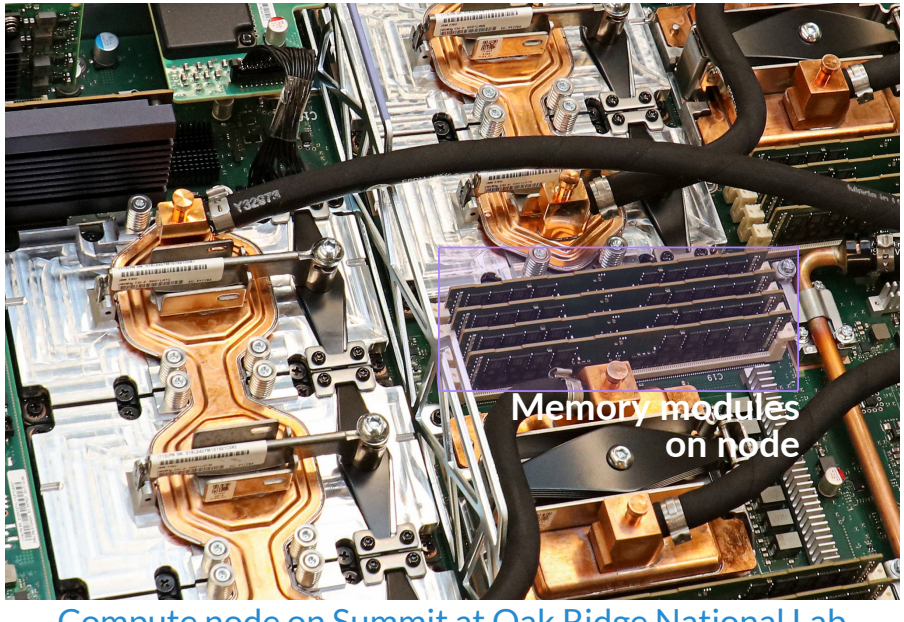
ORTHOGONALITY



- Shared memory and distributed memory are two orthogonal programming models
- Recall that we can not simply access the virtual memory of a *process* but since threads are created by individual processes, these entities can access the memory of its creator process easily.
- Memory is what we use for communication.

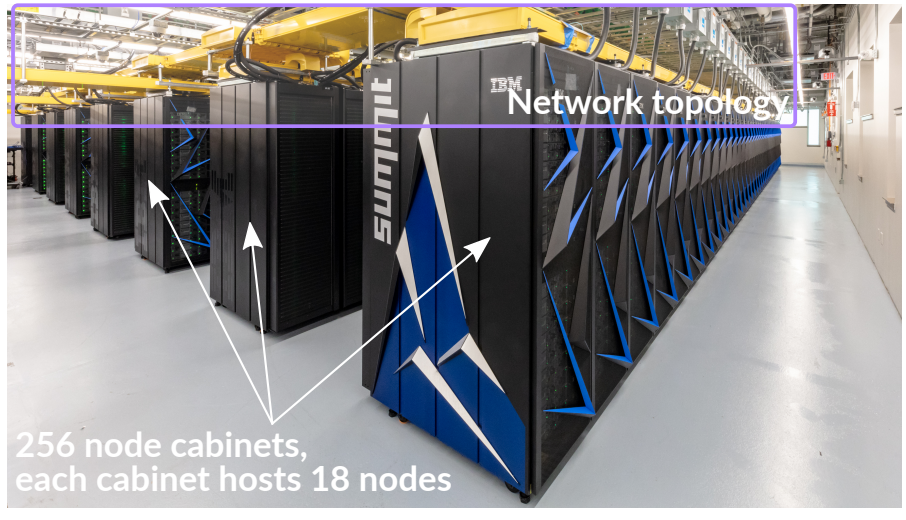
WHY DISTRIBUTED PARALLELISM?

- There is a limit for the amount of memory you can install on one node.
- For large computing tasks, the memory on a single node is simply not enough (e.g. fluid dynamics, molecular dynamics, machine learning or large scale data processing).



Compute node on Summit at Oak Ridge National Lab

Nodes are connected by a non-blocking Fat-Tree topology



- ***Solution:*** put together many nodes (example Summit at ORNL: 4608 nodes).
- Connect the nodes with a (HPC) network and choose a *topology* how to connect the nodes together.
- Memory is now *distributed* among the connected nodes. Communication is achieved by *sending* or *receiving* messages over the network.

WHY DISTRIBUTED PARALLELISM?

- In the Summit example on the previous slide, the network is a dual-rail EDR InfiniBand network providing a node injection bandwidth of 23 GB/s.
- The total wiring in Summit is estimated to be 219 kilometer (136 miles).
- All the nodes are within a distance of less than 300 meters.
- For HPC workloads this is necessary.
- But distributed memory systems exist in the cloud as well, the distributed memory programming model is not limited to HPC only.
- The *heterogeneity* of nodes and network in the cloud may not be most efficient for highest performance.
- Parallelism is enabled through concurrency in a general cloud application.
- Amazon Web Services (AWS) has special HPC nodes which are not in the cloud but close by in a computing center.

SHARED VERSUS DISTRIBUTED MEMORY

Shared memory:

- Accessing data is easy
- Race conditions, mutual exclusion
- False sharing, cache coherency
- NUMA architectures, first-touch policy

(Highlighted keywords are *hard to debug* because they occur at the low level of the hardware)

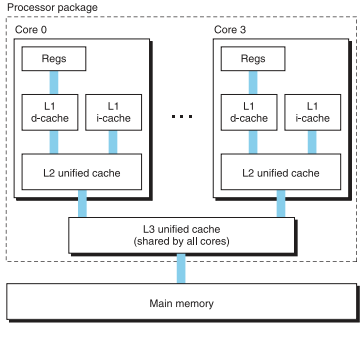
Distributed memory:

- No direct access to distributed data
- You must take care of communication *explicitly*, mutual exclusion
- No race conditions for distributed communication
- Can scale to much larger number of parallel processors

(Highlighted text marks main added complexity for distributed memory systems)

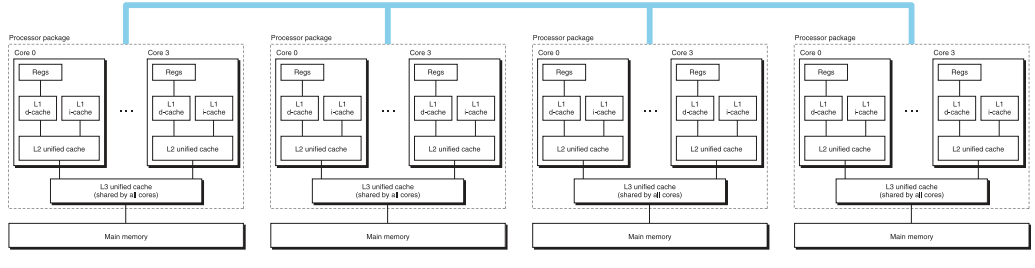
On-Chip Network (OCN):

- ***Fast*** communication
- Maximum distance ***few centimeters***
- *Examples:* memory bus, Intel Ultra Path



System Area Network (SAN):

- ***Slow*** communication
- Maximum distance ***~300 meters***
- *Examples:* InfiniBand, Cray Slingshot or Ethernet



RECALL PROCESSES AND THREADS

Processes and threads are not the same:

Processes:

- Is an execution sequence within the operating system (OS) → a *running* program with a process ID
- Owns a *virtual address space*
- Has an *execution state* (program counter, register values)
- Is managed by the OS

OS creates processes. Their creation is expensive and each has its *own* virtual address space.

Logical (software) threads:

- Can execute a *sequence of instructions* within a running process
- All threads associated to the same process *share* the application data in main memory
- Threads are *cheaper to create* than processes
- The programmer can *control* the creation of threads (they are *logical* entities)

Share the process resources. Cheap to create because the expensive process creation already happened.

RECALL PROCESSES AND THREADS

Processes and threads are not the same:

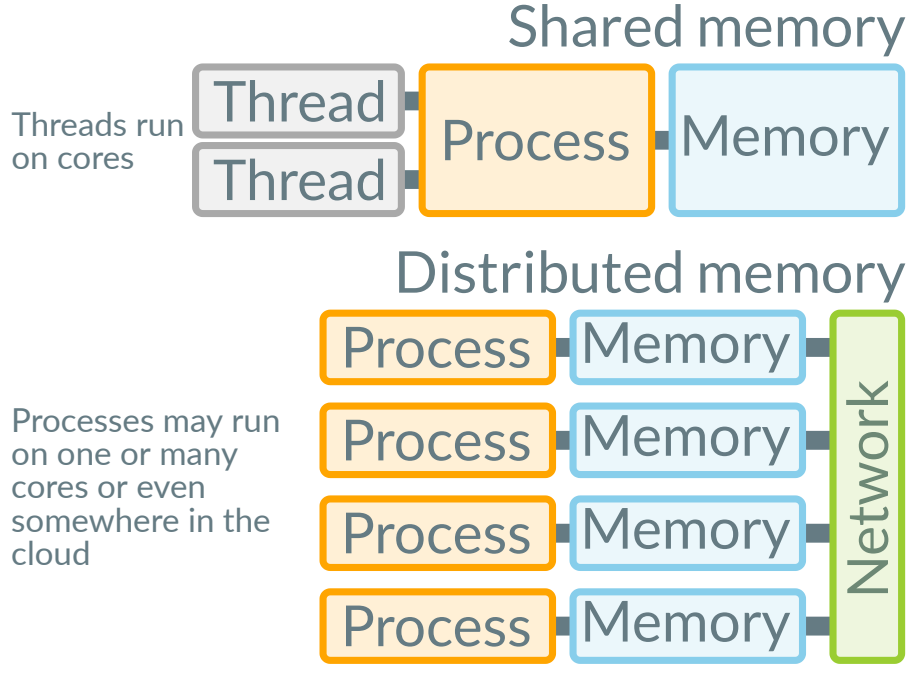
Processes:

OS creates processes. Their creation is expensive and each has its *own* virtual address space.

Logical (software) threads:

Share the process resources. Cheap to create because the expensive process creation already happened.

- This *does not* mean that each process must live in a separate physical memory space (or on different nodes)
- Many processes run simultaneously on your laptop for example. ***It does not matter whether these processes originate from individual programs or from a distributed memory code.***



SIMD VERSUS MIMD

Recall from Flynn's taxonomy: we are interested in SIMD and MIMD parallel computers

SIMD:

- The instruction stream is controlled by a *single* unit upon which the data transformation is *dependent*.
- Execution is usually *synchronous*. *For example:* vector intrinsics on a x86 architecture execute in *lockstep*. A *warp* on Nvidia GPUs no longer has that requirement (they used to be synchronous before CUDA 9).

MIMD:

- The instruction stream is controlled by *different* units (e.g. multiple processes) which *independently* transform the data.
- Execution is usually *asynchronous* and requires *explicit synchronization*. *For example: threads* in shared memory programming or *MPI processes* in distributed memory programming.

HIGH-LEVEL PROGRAMMING MODELS

Let us take a step back from shared memory or distributed memory programming and look at these programming models from a higher abstraction level

Single Program Multiple Data (SPMD)

- An SPMD program can be built with what we have seen so far, including distributed memory programs.
- *Single program* means that all tasks execute their copy of *the same* program simultaneously. The program is abstract and can be threads, MPI processes or a combination of MPI+X (where X is some other programming model) or SIMD lanes.
- *Multiple data* means that all tasks may use different data.
- SPMD programs usually have some logic programmed into them to allow different tasks to branch or conditionally execute code. *Tasks may not necessarily execute the whole program or the same execution path.*
- The SPMD programming model with MPI or a hybrid MPI+X model is one of the most commonly used parallel programming model for many node clusters.

Multiple Program Multiple Data (MPMD)

- A MPMD program can be built with what we have seen so far, including distributed memory programs.
- *Multiple program* means that tasks may execute different programs simultaneously. The programs can be threads, MPI, MPI+X or other possible models.
- *Multiple data* means that all tasks may use different data
- MPMD applications are not as common as SPMD applications, but may be better suited for certain types of problems that address functional decompositions rather than domain decompositions.

MESSAGE PASSING INTERFACE (MPI)

Recall lecture 4:

- No process can access one others virtual memory (and therefore physical memory). A basic safety requirement in multi-user environments such as Linux
- Communication between processes is important when they cooperate on a problem (e.g. MPI)
- Inter-process communication is implemented by the OS with several mechanisms: signals, files, pipes, sockets or shared memory

Simplified representation of the virtual address space of a Linux process (see lecture 2)

- Many different MPI *implementations* of the *specification* exist. Major ones are:
  - OpenMPI (https://www.open-mpi.org/)
  - MPICH (https://www.mpich.org/)
  - MVAPICH (https://mvapich.cse.ohio-state.edu/)
- Optimized vendor specific implementations exist for Cray/HPE, Intel, IBM, Microsoft, etc.

- Implementing *inter-process communication protocols* ourselves is an overkill and error-prone.
- Because it is such an ubiquitous requirement, there is a *standard* for it called *Message Passing Interface* or ***MPI*** for short.

MPI is a standard *(specification)* for an Application Programming Interface *(API)*.

- The MPI code you write does not depend on a specific *implementation*. Your code will run on a compute cluster just like it will run on your laptop.
- All you need is to include the API declarations in your code:

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

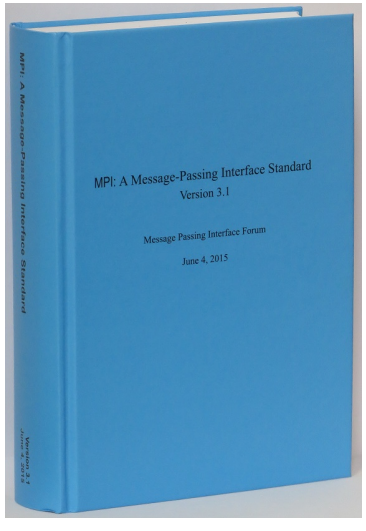
MPI HISTORY

- In the 1980's to the early 1990's, parallel computing with distributed memory develops, as do a number of incompatible software tools for writing such programs—usually with trade-offs between portability, performance, functionality and price. Recognition of the need for a standard arose.
- Working group meets in Minneapolis. MPI draft proposal (MPI-1) from ORNL presented. Group adopts procedures and organization to form the MPI Forum. It eventually comprised of about 175 individuals from 40 organizations including parallel computer vendors, software writers, academia and application scientists.

Version	Year	Description
1.0	1994	The MPI Forum with participation from over 40 organizations releases the first standard for a message passing interface. The MPI Forum is not sanctioned or supported by any official standards organization.
2.0	1997	More focus on process creation, added one-sided communications, extended collective communications and added parallel I/O.
3.0	2012	Significant extensions to MPI functionality including non-blocking collectives, new one-sided communication operations and Fortran 2008 bindings. This standard is a major update to earlier versions. *C++ bindings were removed*.
4.0	2021	Major update with significant new functionality including persistent collectives, partitioned communications and improvements to error handling.

THE CHURCH OF MPI

Where to get help → the bible



> *Most people find the concept of programming obvious, but the doing impossible.*
> *A. Perlis, Epigrams in programming, 115*

- Full documentation with useful examples
- This is the main MPI reference

- *MPI-4.0:* https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (also in the class `git` repo)
- *MPI-3.1:* https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

Practice is most important:

- https://hpc-tutorials.llnl.gov/mpi
- https://www.mcs.anl.gov/research/projects/mpi/tutorial
- https://mpitutorial.com

The best way to lookup the docs are the `man`-pages!

- A complete MPI installation includes the docs and `man`-pages.
- Invaluable when you work in the command line.
- When you have loaded a MPI module on the cluster, e.g.:

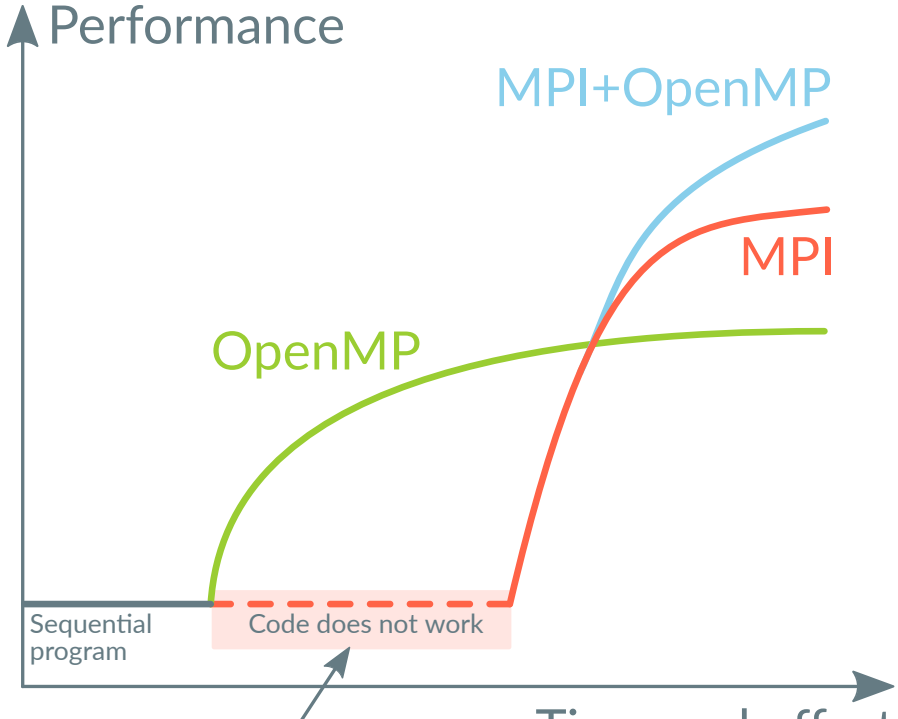
```
1  module load gcc/12.1.0-fasrc01 openmpi/4.1.3-fasrc01
```

you can query the `man`-pages to get instant help for any MPI routine. *For example:*

```
1  man MPI_Init
2  MPI_Init(3)              Open MPI              MPI_Init(3)
3
4  NAME
5         MPI_Init - Initializes the MPI execution environment
6
7  SYNTAX
8  C Syntax
9         #include <mpi.h>
10        int MPI_Init(int *argc, char ***argv)
11 ...
```

OVERHEAD BETWEEN OPENMP AND MPI

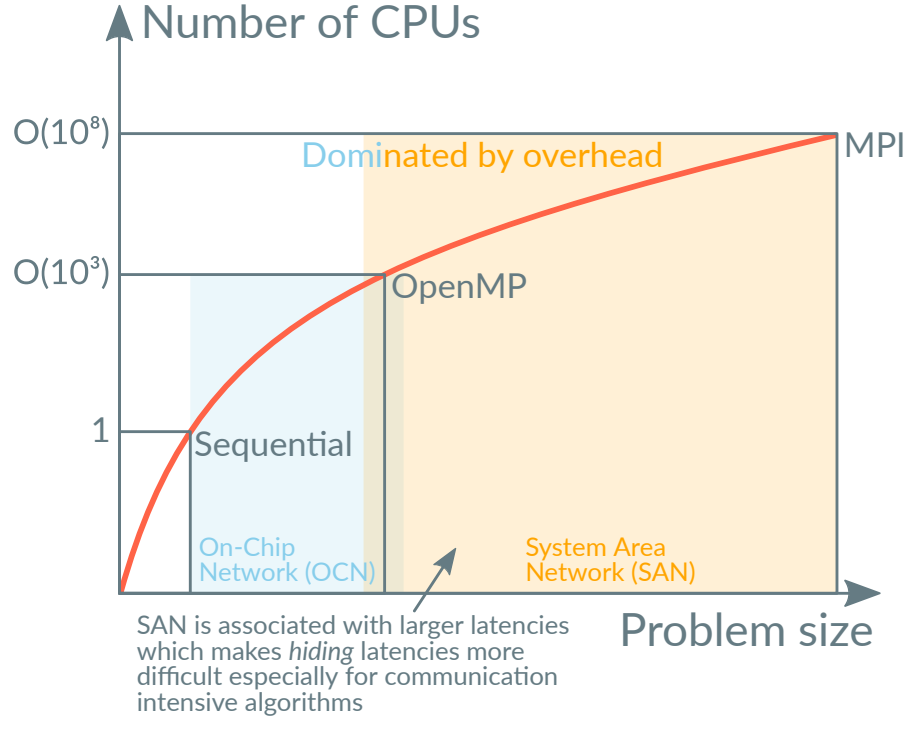
Time to (parallel) software:



MPI code involves API only (OpenMP is mainly compiler directives and some runtime routines). This overhead is mostly due to *explicit communication*.

Code does not work in initial parallelization phase because communication must be handled *explicitly* by API calls.

Communication overhead:



SAN is associated with larger latencies which makes *hiding* latencies more difficult especially for communication intensive algorithms

A SAN can cover larger area at the cost of slower speed (same story with caches, cell phones, etc.) and therefore higher latencies. Hiding these latencies is more difficult the more complex the system becomes.

WHY USE MPI?

Standardization

MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries. *Everyone uses MPI!*

Portability

There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

Performance opportunities

Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.

Functionality

There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.

Note: Most MPI programs can be written using a dozen or less routines.

Availability

A variety of implementations are available, both vendor and public domain.

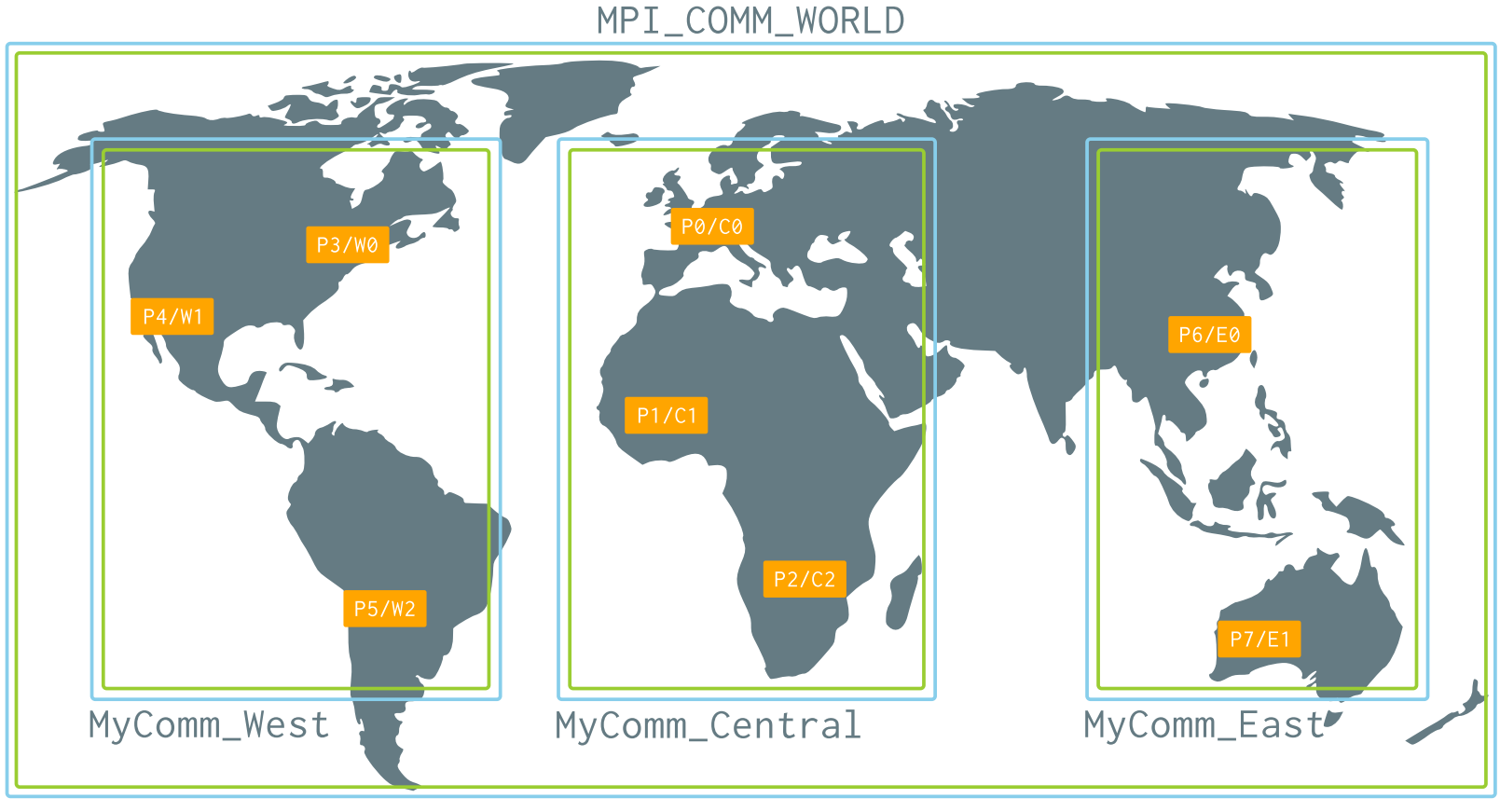
MPI COMMUNICATORS AND GROUPS

- MPI uses objects called *communicators* and *groups* to define which collection of *processes* may communicate with each other.
- Most MPI routines require you to specify a *communicator as an argument*.
- `MPI_COMM_WORLD` is the predefined default communicator that includes all of your MPI processes.

An MPI communicator encapsulates:

- ***Contexts*** (the communication "universe")
- ***Groups*** (the *processes* that make up the universe, or a subset of it)
- ***Virtual topologies*** (e.g. arrange the processes on a Cartesian grid. You will use a Cartesian topology in HW3)
- Caching (allows to add new attributes to communicators. For advanced usage, not covered here)

MPI COMMUNICATORS AND GROUPS

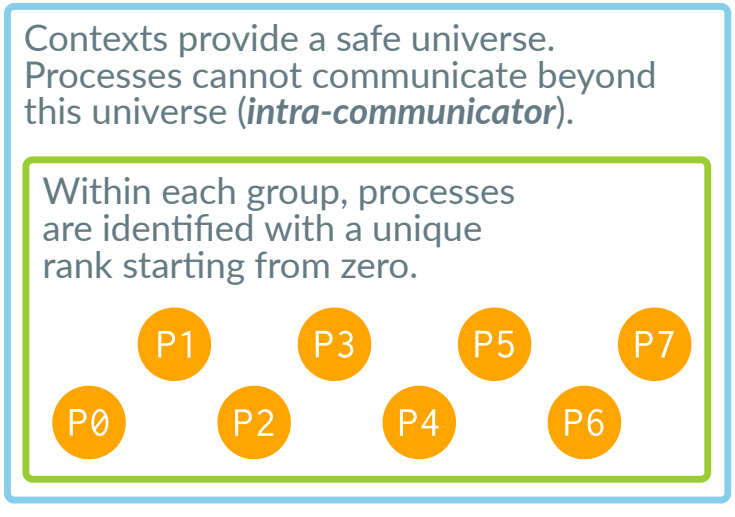


- A MPI *group* is an ordered set of processes.
- Each process is associated with an integer ***rank***. Ranks are *contiguous* and start from zero.
- A group is used within a communicator to describe the participants in a communication "universe".
- Communicators can be split into *sub-communicators*. E.g., `MPI_COMM_WORLD` is split into `MyComm_West`, `MyComm_Central` and `MyComm_East` above.

MPI COMMUNICATORS AND GROUPS

More abstract:

`MPI_COMM_WORLD` (Default communicator after `MPI_Init`)



See Section 7.1 and 7.2 in the MPI 4.0 standard.

Intra-communicator:

- Most commonly used for message passing in MPI.
- Contain one group of valid processes and contexts for *point-to-point and collective* communication.
- An intra-communicator allows you to define a virtual topology for the processes.
- The image on the left is an *intra-communicator*.

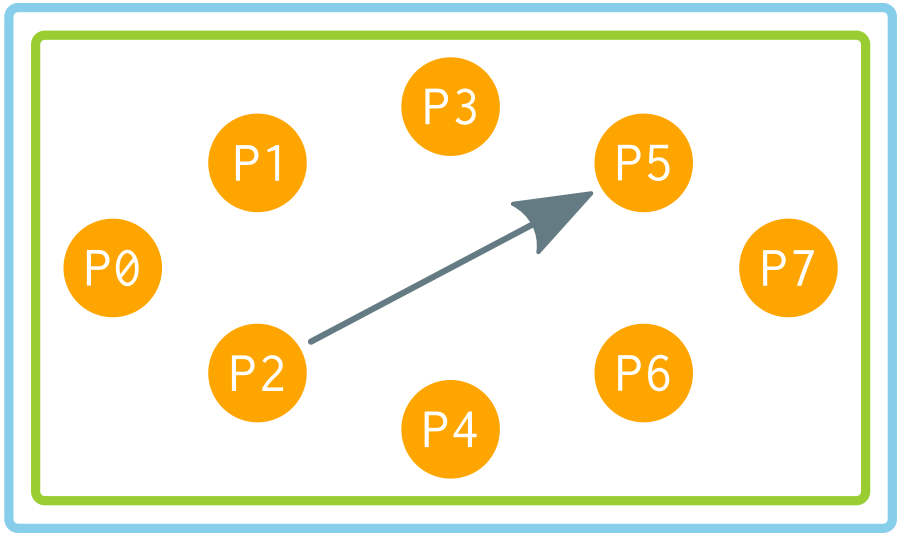
Inter-communicator:

- Allows for communication between two non-overlapping groups of processes (communicate across "universes").
- Useful to communicate between parallel modules in an application and client-server computing paradigm.
- Bind two groups together with communication contexts *shared* by both groups.
- You cannot define a virtual topology on an inter-communicator.

POINT-TO-POINT AND COLLECTIVE COMMUNICATION

The fundamental operation in MPI is point-to-point communication:

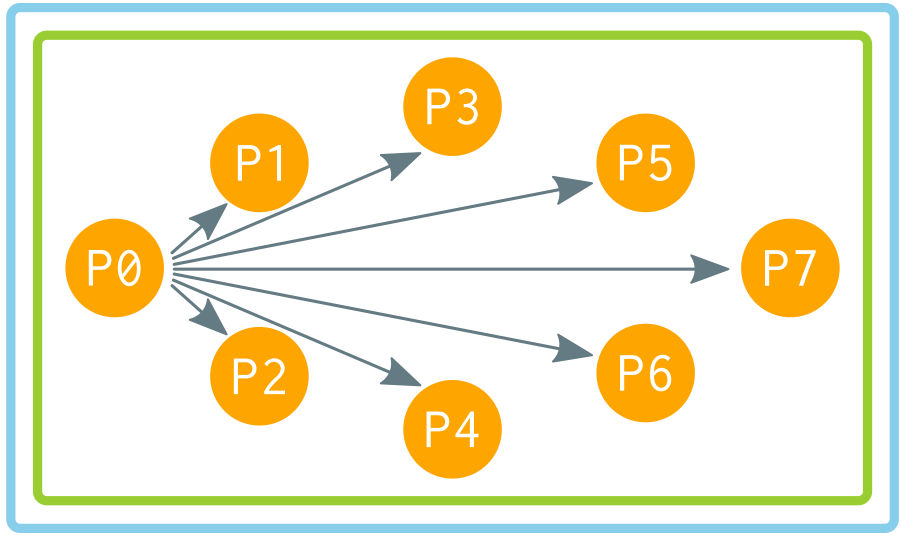
MPI_COMM_WORLD



- It is simply a *matching* send/receive pair.
- Think of "point" as a process within a group.
- The two points must be in the same group (intra-communicators).

Collective communication involves ALL ranks in a group:

MPI_COMM_WORLD



- Any collective communication can be composed with point-to-point communication only.
- They exist for convenience and *performance* reasons.
- Collective primitives are generally *expensive*.

Examples:

- Barrier
- Broadcast
- Reduction
- Scan

SYNCHRONOUS AND ASYNCHRONOUS COMMUNICATION

MPI specifies two communication behaviors:

Blocking

When the API function call for a blocking primitive returns, MPI *guarantees* that it is safe to use the message buffer again (for sender and/or receiver).

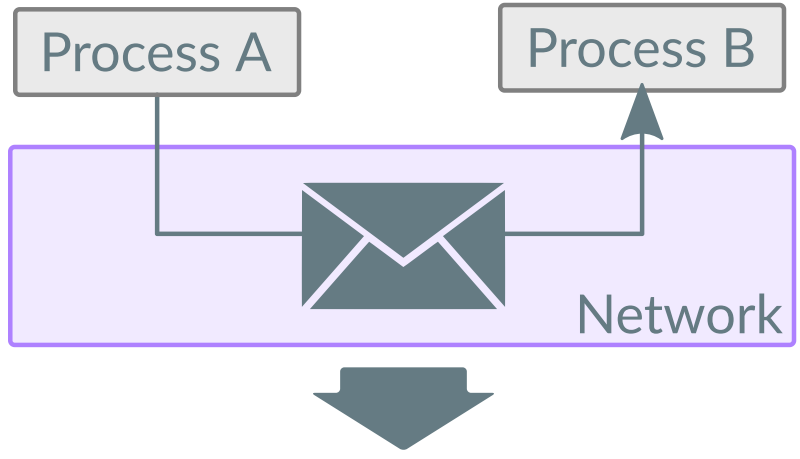
Non-Blocking

The API function call for a non-blocking primitive *returns immediately*. There is *no guarantee* that the communication has concluded. Communication buffers *are not* safe for read/write operations.

Blocking communication means synchronization which results in sequential code!

A *blocked process does not make progress* (it does not perform useful work while blocked). Non-blocking communication allows to do work while messages are in flight. *The single most useful property of non-blocking communication is compute-transfer overlap.*

ANATOMY OF AN MPI MESSAGE



Envelope (meta-data, *fixed size*)			
source	destination	communicator	tag
source/destination: sender/receiver ranks **communicator:** describes communication context **tag:** identifier to select incoming messages on receiver side			

Body (message data, *variable size*)		
count	datatype	buffer
count: number of *elements* in message (*not bytes*) **datatype:** describes the type of one element **buffer:** address of the first data element in the message		

Example:

```
1 if (Process_A) {
2     // destination = Process B
3     MPI_Send(send_buffer, count, datatype, destination, tag, MPI_COMM_WORLD);
4 } else if (Process_B) {
5     // source = Process A
6     MPI_Recv(recv_buffer, count, datatype, source, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
7 }
```

GETTING STARTED WITH MPI

General MPI program structure:

```
1  // headers and declarations
2  #include <mpi.h> // includes the MPI API
3  // other headers and declarations
4  int main(int argc, char *argv[]) // Program start
5  {
6      // 1. Serial code
7
8      // 2. Initialize MPI environment
9      MPI_Init(&argc, &argv); // parallel code begins
10
11     // 3. Do work and MPI calls here
12
13     // 4. Terminate MPI environment
14     MPI_Finalize(); // parallel code ends
15
16     // 5. Serial code
17
18     return 0;
19 } // Program end
```

- It is *necessary* to initialize and terminate the MPI environment.
- MPI calls can only be carried out after initialization and before termination of the MPI environment.
- Before initialization and after termination of the MPI environment the code is serial (i.e., you cannot use MPI).
- You must include the `mpi.h` header to make use of MPI library calls.
- The names of the MPI routines all start with `MPI_` (or `PMPI_` for the profiling interface) and *are case-sensitive* in C/C++ code (they are not in Fortran).
- Your code ***must not*** declare variables or functions that begin with `MPI_` or `PMPI_`.

HELLO MPI

Hello MPI:

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

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    std::cout << "Hello MPI\n";

    MPI_Finalize();
    return 0;
}
```

MPI compiler wrapper for C++:

```
$ mpic++ -show # output shortened
g++ -pthread -L/usr/lib/openmpi -lmpi
```

Compilation is similar as with g++:

```
$ mpic++ -o hello_mpi hello_mpi.cpp
```

MPI runtime wrapper:

```
$ mpirun -n 2 ./hello_mpi
Hello MPI
Hello MPI
```

- The `-n 2` argument creates 2 copies of the program.
- We can check that this creates 2 processes:

```
$ ps aux | grep hello_mpi
fabs  584178  20:00  0:00 mpirun -n 2 ./hello_mpi
fabs  584180  20:00  0:00 ./hello_mpi
fabs  584181  20:00  0:00 ./hello_mpi
```

Running MPI on SLURM:

```
$ srun -n2 -c1 ./hello_mpi
Hello MPI
Hello MPI
```

- SLURM uses `srun` to run parallel jobs.
- You have already used it in your job scripts. The option `-n2` requests 2 tasks (`--ntasks`) where each task has `-c1` cores to run on (`--cpus-per-task`).

MPI ENVIRONMENT MANAGEMENT ROUTINES

`MPI_Init` and `MPI_Init_thread`

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For thread support you must initialize the environment with `MPI_Init_thread`.

`MPI_Comm_size`

Returns the total number of MPI processes in the specified communicator, such as `MPI_COMM_WORLD`.

`MPI_Comm_rank`

Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator `MPI_COMM_WORLD`.

`MPI_Abort`

Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates *all* processes regardless of the communicator specified.

`MPI_Wtime`

Returns an elapsed wall clock time in seconds (`double` precision) on the calling processor.

`MPI_Get_version`

Returns the version and subversion of the MPI standard that is implemented by the library.

`MPI_Finalize`

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program—no other MPI routines may be called after it.

PRINT HOSTNAME WITH MPI

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

int main(int argc, char *argv[])
{
    int size, rank, len, err;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    // initialize MPI
    err = MPI_Init(&argc, &argv);
    assert(err == MPI_SUCCESS);

    // get number of ranks in the MPI_COMM_WORLD communicator
    err = MPI_Comm_size(MPI_COMM_WORLD, &size);
    assert(err == MPI_SUCCESS);

    // get my rank
    err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    assert(err == MPI_SUCCESS);

    // query processor name (the return value is
    // implementation dependent)
    err = MPI_Get_processor_name(hostname, &len);
    assert(err == MPI_SUCCESS);
    std::cout << "Rank " << rank << "/" << size <<
              << " running on: " << hostname << '\n';

    // call other MPI routines

    // terminate the MPI environment
    err = MPI_Finalize();
    assert(err == MPI_SUCCESS);

    return 0;
}
```

- The code uses `MPI_Comm_size` and `MPI_Comm_rank` to determine the number of ranks and the process rank (or task ID) in the `MPI_COMM_WORLD` communicator.
- Almost all MPI routines return an error code. It is good practice to check it (but often skipped when you read other code).
- You can find this code in the lecture repository:

```
$ make
mpic++ -O1 -Wall -o rank_size_proc rank_size_proc.cpp
$ mpirun -n 4 ./rank_size_proc
Rank 0/4 running on: pilatus
Rank 1/4 running on: pilatus
Rank 3/4 running on: pilatus
Rank 2/4 running on: pilatus
```

- Similar as in shared memory, the order of the output from each process is arbitrary.

MPI WITH THREAD SUPPORT

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

int main(int argc, char *argv[])
{
    int rank, size, version, subversion, provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);

    MPI_Get_version(&version, &subversion);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    const bool isroot = (0 == rank);
    if (isroot) { // only the root rank executes this code
        std::cout << "MPI version:           " << version << '.' << subversion
                  << '\n';
        std::cout << "Total ranks:           " << size << '\n';
        std::cout << "Thread support:        " << provided
                  << " (requested MPI_THREAD_FUNNELED)\n";
        std::cout << "MPI_THREAD_SINGLE:     " << MPI_THREAD_SINGLE << '\n';
        std::cout << "MPI_THREAD_FUNNELED:   " << MPI_THREAD_FUNNELED << '\n';
        std::cout << "MPI_THREAD_SERIALIZED: " << MPI_THREAD_SERIALIZED << '\n';
        std::cout << "MPI_THREAD_MULTIPLE:   " << MPI_THREAD_MULTIPLE << '\n';
    }

    MPI_Finalize();
    return 0;
}
```

Output:

```
MPI version:           3.1
Total ranks:           4
Thread support:        1 (requested MPI_THREAD_FUNNELED)
MPI_THREAD_SINGLE:     0
MPI_THREAD_FUNNELED:   1
MPI_THREAD_SERIALIZED: 2
MPI_THREAD_MULTIPLE:   3
```

- If your MPI application is multi-threaded you need to initialize MPI with thread support using `MPI_Init_thread`.
- There are four possible levels:
  1. `MPI_THREAD_SINGLE`: Only one thread will execute.
  2. `MPI_THREAD_FUNNELED`: If the process is multithreaded, only the thread that called `MPI_Init_thread` will make MPI calls.
  3. `MPI_THREAD_SERIALIZED`: If the process is multi-threaded, only one thread will make MPI library calls at one time.
  4. `MPI_THREAD_MULTIPLE`: If the process is multithreaded, multiple threads may call MPI at once with no restrictions.

RECAP

- SPMD (single-program-multiple-data) is the most commonly used programming model for MPI.
- MPI has a long history and is the *industry standard* for distributed memory programming.
- MPI processes are arranged in groups with unique ranks (task IDs). An intra-communicator contains one group of ranks and adds contexts for point-to-point and collective communication for that group.
- The default intra-communicator in MPI is called `MPI_COMM_WORLD`.
- In order to use MPI you must initialize the MPI environment in your code. All MPI routines have the following structure `MPI_Xxxx` and are case sensitive.

Further reading:

- Chapter 1, Sections 2.1, 2.2, 2.7, 7.1 and 7.2 in *MPI 4.0 Standard*, MPI Forum, 2021 (pdf in `git` repo as well)
- Section 3.1 in Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011
- Section 9.1 and 9.2 in Hager and Wellein, *"Introduction to High Performance Computing for Scientists and Engineers"*, CRC Press, 2011