DTU Compute
Department of Applied Mathematics and Computer Science

DTU

High-Performance Computing

02616

Large-scale Modelling

Why Large-Scale Modelling?

- Computations on models of high complexity or with a large number of degrees of freedom
- Physical, chemical or biological processes
  - Fluid dynamics, reactor processes, proteins
  - Many-body problems
- Complex mathematical models of networks
  - Traffic simulations
  - Rule-based interactions
  - ...

Why Large-Scale Modelling?

- Common theme: large models require
  - Many computations (e.g. flop/s)
  - Large amounts of memory
- Solution:
  - Divide problem into smaller chunks, e.g. domain decomposition
  - Use computational resources in parallel, e.g. many CPU cores, many computers (clusters)
- Requires parallel programming models
  - We get there later

Why Large-Scale Modelling?

- Hardware – looking 15 years back:
  - Typical servers in the DCC cluster had
    - 16-20 cores
    - 24 GB RAM
  - Solving a problem with a 512 GB memory usage required approx. 22 cluster nodes
- Need for a programming model that could be used on a cluster with ‘distributed’ resources

Why Large-Scale Modelling?

- Hardware – today:
  - Typical servers in the DCC cluster have
    - 32-64 cores, up to 128 cores
    - 384 GB – 1.5 TB RAM
  - Solving a problem with a 512 GB memory usage requires maybe a single node, only!
- Do we still need a ‘distributed’ programming model?
- Yes – there are even larger problems
  - Want to simulate more particles
  - Increased model complexity

Why Large-scale Modelling?

- Supercomputers: the really big iron is usually a cluster of smaller nodes!
- TOP500 no. 1 (June 2025): El Capitan
  - 11.136 compute nodes
  - 4 APUs (combined CPU and GPU) per node
  - 512 GB memory per node
  - fast interconnect

Why Python?

- Python is “slow”!
  - Lots of comparisons/benchmarks show this
- … but there a ways to get Python up to speed
  - Calling optimized libraries
  - Using ‘JIT’ tools, like Numba
  - Remember what you have in learned in 02613 – Python and HPC
- … and Python is an agile language
  - Very popular, large community
  - Easy to use
  - _The_ programming language at DTU!

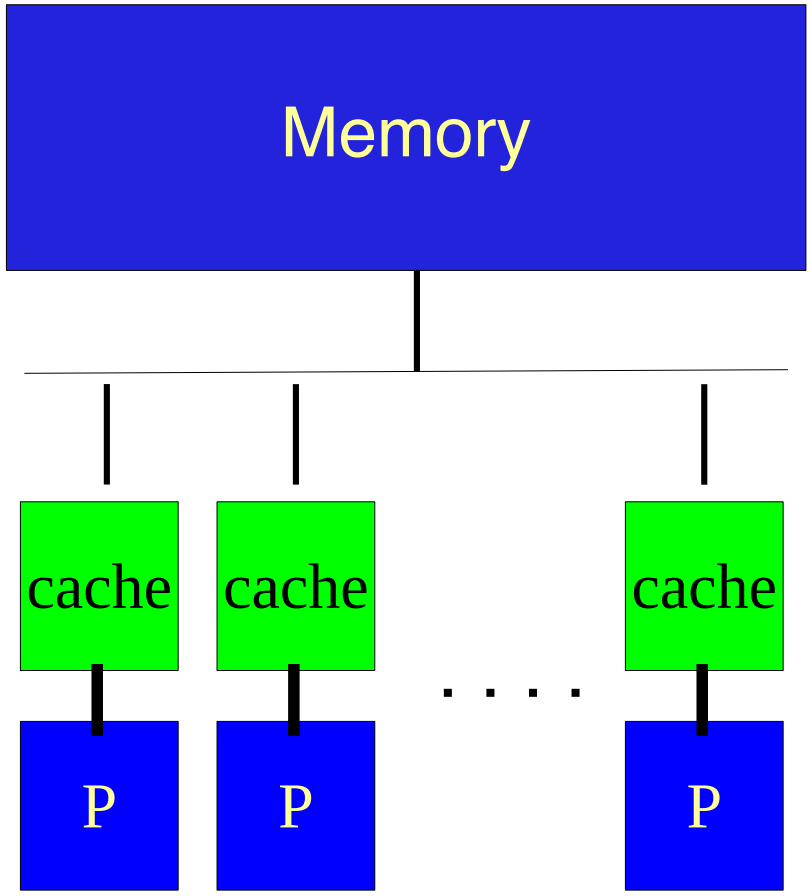
Why Python?

- Python in this course:
  - We need to teach you to go beyond a ‘single computer’ (common theme in both 02613 and 02614)
  - We want you to understand and focus on the aspects of ‘multi-computer’ (or: ‘multi-node’) computations
  - … using a programming language you are familiar with

Hardware overview

Classification of parallel systems

Uniform Memory Access (UMA)



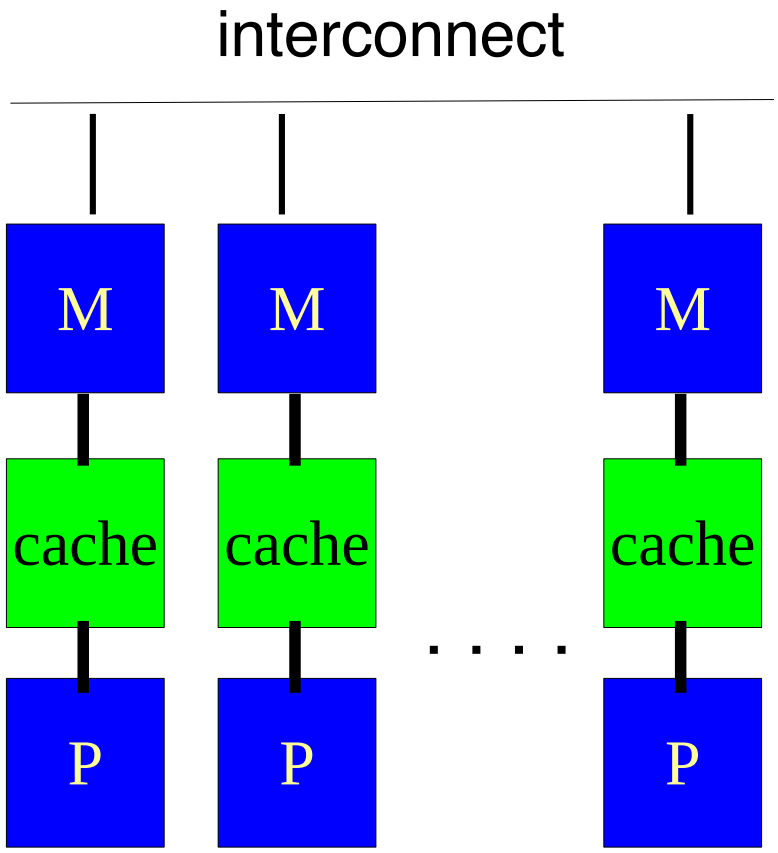
Memory access:

- uniform for all P
- P: single or multiple cores

Example systems:

- single-socket multi-core CPU (e.g. your laptop)

Non-Uniform Memory Access (NUMA)



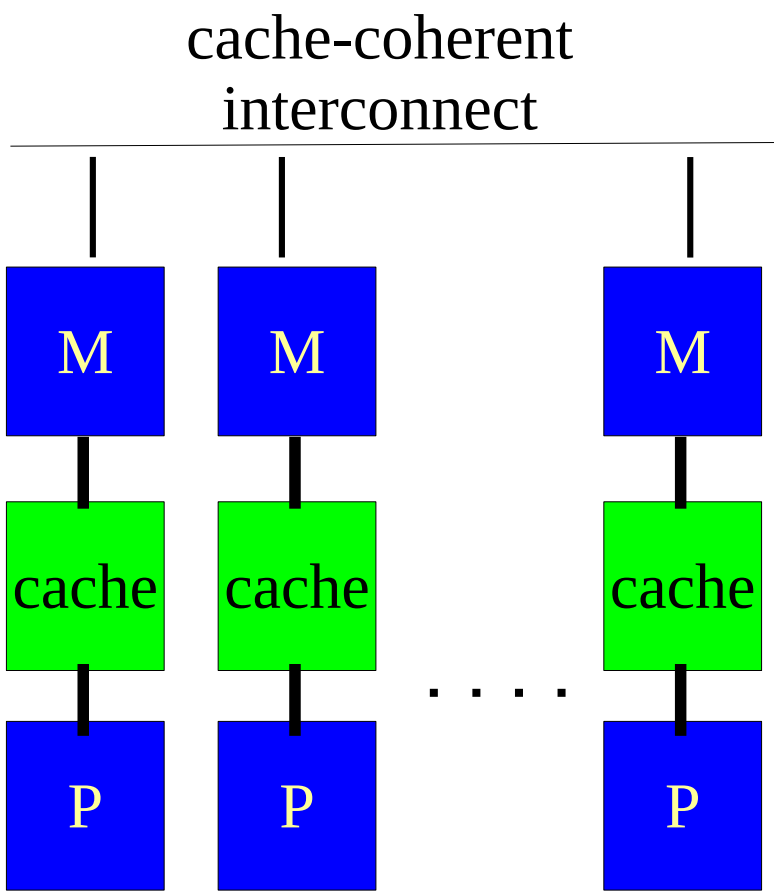
Memory access:

- non-uniform across P

Example systems:

- cluster of nodes, connected by a (fast) network

cc-NUMA



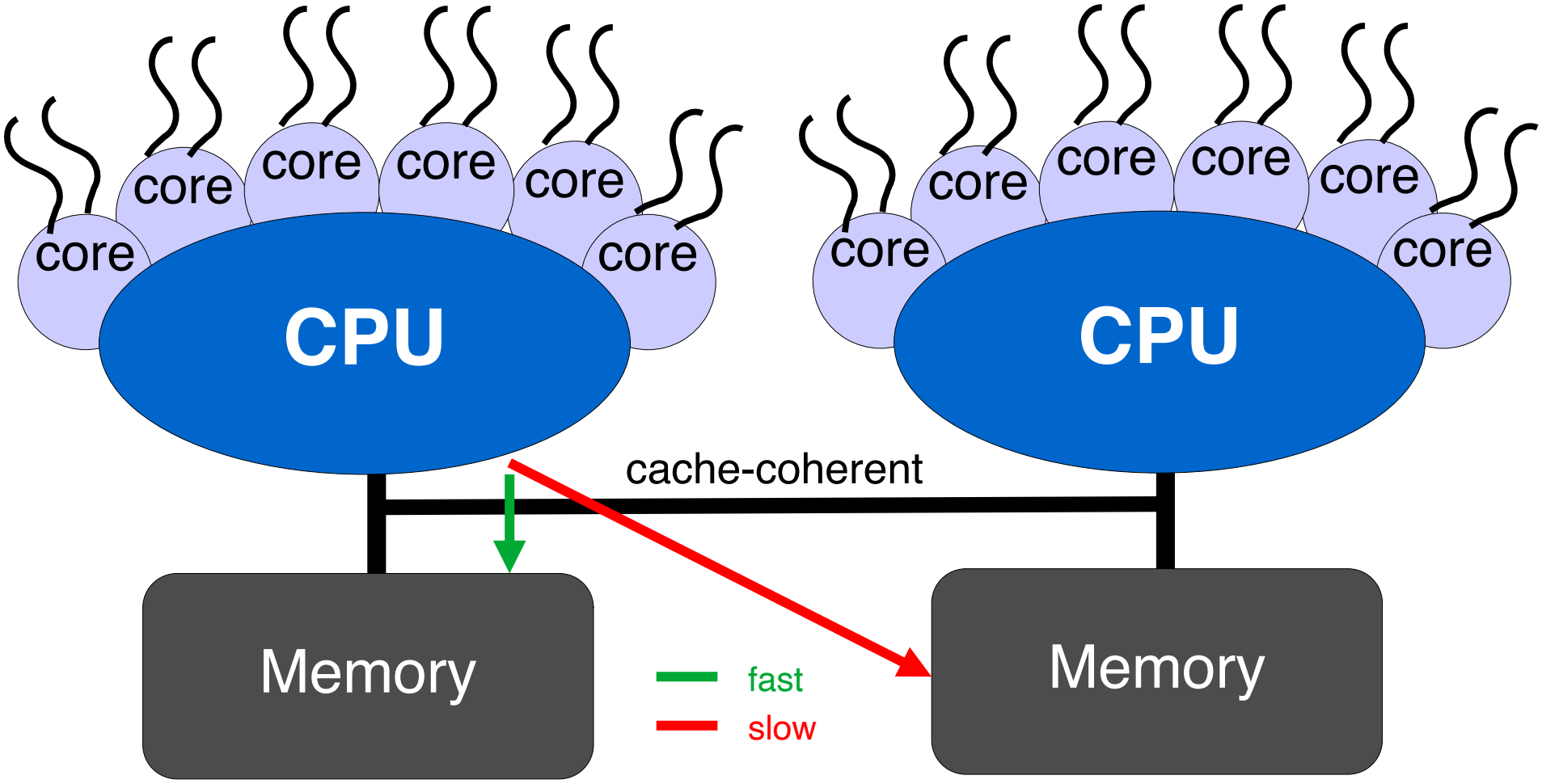
Memory access:

- non-uniform across P
- ... but cache-coherency on the interconnect
- one global memory space

Example systems:

- (almost) all modern multi-socket servers (x64 and arm)

A typical multi-core server



a 2-socket, 12-core, 24-(hyper-)threads server ⇒ 24 logical CPUs

Note: we do not use hyperthreading in our setup!

The “classical” view

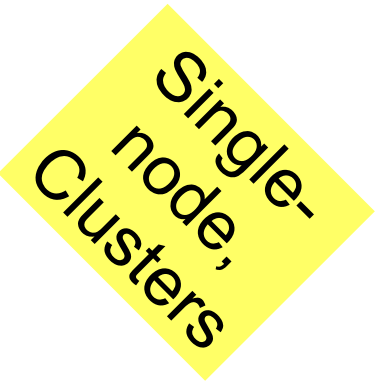
02616 – Week 01

Parallel Programming Models

Parallel Programming Models

Two “classical” parallel programming models:

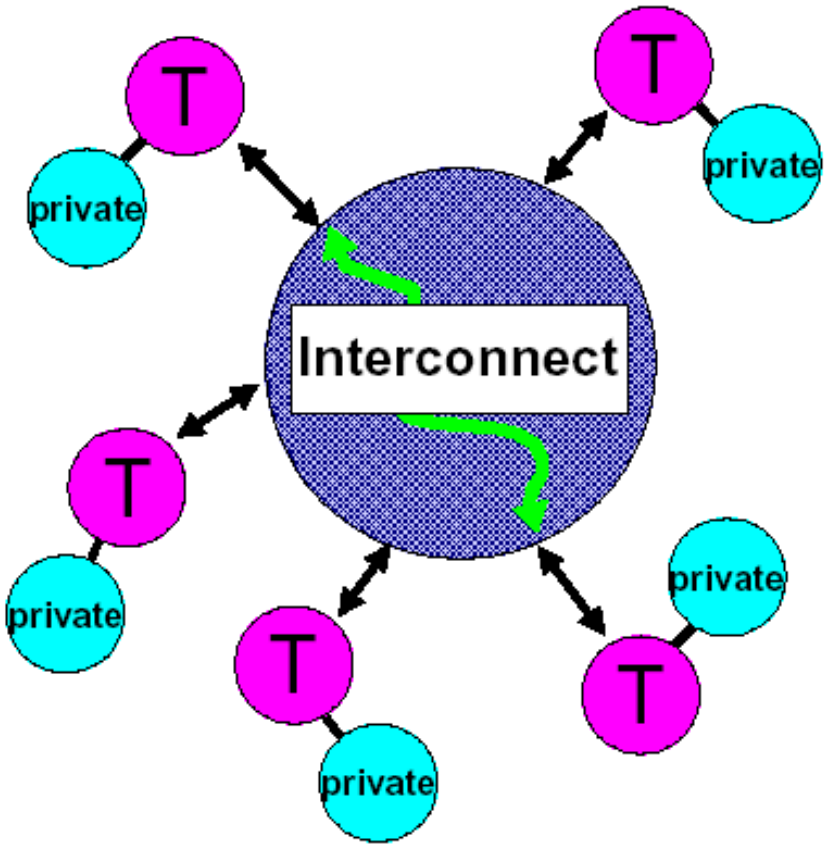
- Distributed memory
  - “process based”
  - PVM (standardized)
  - MPI (de-facto standard, widely used)
- Shared memory
  - “threads based”
  - Pthreads (standardized)
  - OpenMP (de-facto standard)
  - C++11 threads



Parallel Programming Models

Distributed memory programming model:

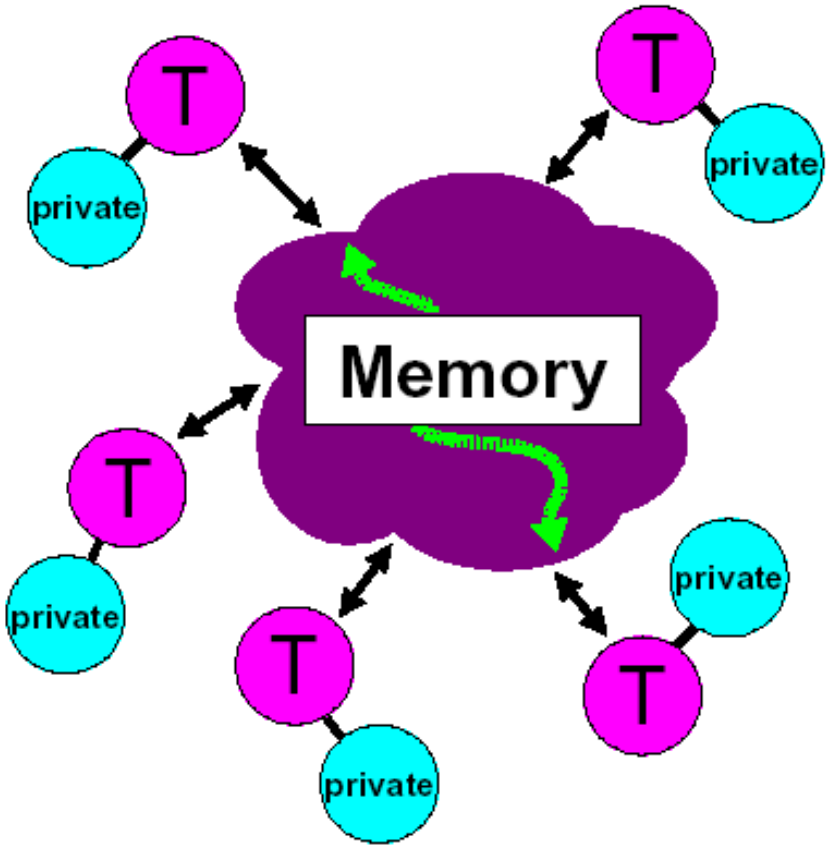
- all data is private to the ranks (T)
- data is shared by exchanging buffers over an interconnect (shared memory, network, ...)
- explicit communication:
  - data transfer
  - synchronization



Parallel Programming Models

Shared memory model:

- all threads (T) have access to the same global memory
- data “transfer” is transparent to the programmer
- synchronization is (mostly) implicit
- there is private data as well



Parallel Programming Models

Other programming models

- PGAS (Partitioned Global Address Space):
  - UPC (Unified Parallel C)
  - Co-Array Fortran
- GPUs: massively parallel & shared memory
  - CUDA
  - OpenCL
  - Shader languages
  - OpenMP, OpenACC (offloading to GPU)
- Hybrid models: MPI+X (X=OpenMP, CUDA, ...)

Parallel Programming in Python

- The Python interpreter itself is multi-threaded
- … but there is GIL
  - Global Interpreter Lock
  - only one thread can execute interpreter code at a time
  - assures correctness
  - … for the price of speed
- writing fast computations using ‘native’ threads in Python is not possible
  - the GIL will go away (be relaxed) in Python versions above 3.13 (optional feature)

Parallel Programming in Python

- How to make use of multi-threading in Python?
  - implicit, through libraries/modules that implement threading by other mechanism, e.g. POSIX threads
  - example: NumPy, SciPy
  - limitation: single computer, only!
- Another way to parallelize Python code:
  - multi-processing
  - replicates Python into “independent” processes (each of them having a GIL!)
  - mostly task-queue based implementations (pools)
  - limited to a single computer (most implementations)

What to do on clusters?

- Search for ‘Parallel Python distributed’
- … and you might end up here:
  - https://wiki.python.org/moin/ParallelProcessing
- Under ‘Cluster computing’ you will find 35+ suggestions
  - … but most of them are very specialized solutions for specific tasks or problem classes
- There are MPI (and PVM(!)) based solutions, too
  - e.g. mpi4py
- … and that’s what we will use in this course!

MPI Basics

02616 – Week 01

A very brief introduction to MPI
(using C)

MPI Basics

MPI version of “Hello world” in C:

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

int
main(int argc, char **argv) {
    int myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    printf("Hello world from %d!\n", myrank);

    MPI_Finalize();
    return 0;
}
```

MPI Basics

MPI version: compile and run

```
$ module load mpi/<version>
$ mpicc -o hello_mpi hello_mpi.c

$ ./hello_mpi
Hello world from 0!

$ mpirun -np 4 ./hello_mpi
Hello world from 1!
Hello world from 3!
Hello world from 0!
Hello world from 2!
```

MPI Basics

- What is MPI?
  - MPI is an API standard to implement parallel programs using “Message Passing”
  - allows you to “write” portable programs
- MPI components:
  - a library of functions (API) – standardized (MPI-4)
  - a runtime environment
    - launchers (mpirun, mpiexec), runtime libraries, ...
    - not standardized (a standard ABI will come in MPI-5)
  - set of helper commands, like mpicc (compiler, linker)

MPI Basics

A closer look:

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

int
main(int argc, char **argv) {
    int myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("Hello world from %d!\n", myrank);

    MPI_Finalize();
    return 0;
}
```

include MPI library header file

get in touch with MPI runtime

ask for my rank (ID) in the communication setup

tell the MPI runtime we are done

MPI Basics

A closer look at the execution:

```
$ ./hello_mpi
Hello world from 0!
```

← no launcher, no runtime -> serial program

```
$ mpirun -np 4 ./hello_mpi
Hello world from 1!
Hello world from 3!
Hello world from 0!
Hello world from 2!
```

← launch program 4 times, in the MPI runtime

```
$ mpirun -np 2 /bin/echo "Hello World!"
Hello World!
Hello World!
```

← mpirun is an "advanced" replicator – if the code is not using MPI, it executes simply 'np' copies

MPI Basis

- The tasks of mpirun (or mpiexec):
  - set up the runtime environment for the program
  - set up the communication channels
    - single node: shared memory or (emulated) network
    - multiple nodes: across the “selected” interconnect (ethernet, Infiniband, etc)
    - selection can be fixed, or controlled in “collaboration” with the scheduler (Slurm, LSF, etc)
  - start and control ‘np’ copies of the program
  - control the communication among the copies
  - close down everything at the end

MPI Basics

Improved MPI version of “Hello world” in C:

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

int main(int argc, char **argv) {
    int myrank, p;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    printf("Hello world from %d of %d!\n",
            myrank, p);
    MPI_Finalize();
    return 0;
}
```

MPI Basics

Improved MPI version: compile and run

```
$ module load mpi/<version>
$ mpicc -o hello_mpi hello_mpi.c

$ ./hello_mpi
<That's your task in the labs!>

$ mpirun -np 4 ./hello_mpi
<That's your task in the labs!>
```

MPI Basics

- By now, we know 4 basic MPI functions:
  - MPI_Init()
  - MPI_Comm_size()
  - MPI_Comm_rank()
  - MPI_Finalize()
- What is missing to have a working MPI program?
- We need to communicate, i.e. sending and receiving messages (the M in MPI!)
  - MPI_Send()
  - MPI_Recv()

MPI Basics

The six function prototypes:

```
int MPI_Init(int *argc, char ***argv);

int MPI_Comm_rank(MPI_Comm comm, int *rank);
int MPI_Comm_size(MPI_Comm comm, int *size);

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_Finalize(void);
```

MPI Basics

- More details we need to know:
  - return values of the functions: either MPI_SUCCESS (usually defined as 0) or different from it (values are defined in mpi.h)
  - MPI_Comm is a data structure for communicators, and there are some pre-defined values, like MPI_COMM_WORLD (global communicator)
  - MPI_Datatype: matches the basic C datatypes, e.g. MPI_INT, MPI_DOUBLE, etc
  - MPI_Status is a data structure, that holds information about source, tag, size and error.

MPI Basics

- More details on MPI_Send() and MPI_Recv()
  - all message send need to have a destination
  - all messages received need to have a source
  - all messages need to have a message tag
  - the receiving buffer needs to be at least large enough to hold the sent amount of elements of the MPI_Datatype
  - MPI_Send() and MPI_Recv() are “blocking”, i.e. MPI_Send() will first return, when the receiving end has acknowledged that the data sent was completely copied.

MPI Basics

- With the help of those six basic functions, you will be able to solve a wide range of problems.
- More advanced and complex functionality, like collectives, non-blocking, etc, will be introduced later in this course, using the mpi4py embeddings

MPI Basics

Simplest send-receive example: hello_neighbour.c

```
int main(int argc, char **argv) {
    char message[20];
    int i, rank, size, tag = 99;
    MPI_Status status;

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

    if (rank == 0) {
        strcpy(message, "Hello!");
        MPI_Send(message, 6, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
        printf("Rank %d says: %s\n", rank, message);
    } else {
        MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
                 &status);
        printf("Rank %d received: %.13s\n", rank, message);
    }
    MPI_Finalize();
}
```

MPI Basics

Compile and run:

```
$ module load mpi/<version>
$ mpicc -o hello_neighbour hello_neighbour.c

$ mpirun -np 2 ./hello_neighbour
Rank 0 says: Hello!
Rank 1 received: Hello!

$ mpirun -np X ./hello_neighbour
<For different X: that's your task in the
labs!>
```

MPI in batch jobs

02616 – Week 01

How to run MPI programs
in LSF batch jobs

MPI in batch jobs

Hello world batch job script (single host, 8 ranks):

```
#BSUB -J MPIhello
#BSUB -o MPIhello_%J.out
#BSUB -e MPIhello_%J.err
#BSUB -n 8
#BSUB -R "span[hosts=1]"
#BSUB -q hpcintro
#BSUB -W 00:05
#BSUB -R "rusage[mem=1GB]"

module load mpi

mpirun ./hello_c
```

ask for 8 cores – one core per rank

no '-np 8' needed – mpirun gets this from the scheduler!