

# Introduction to High-Performance Computing

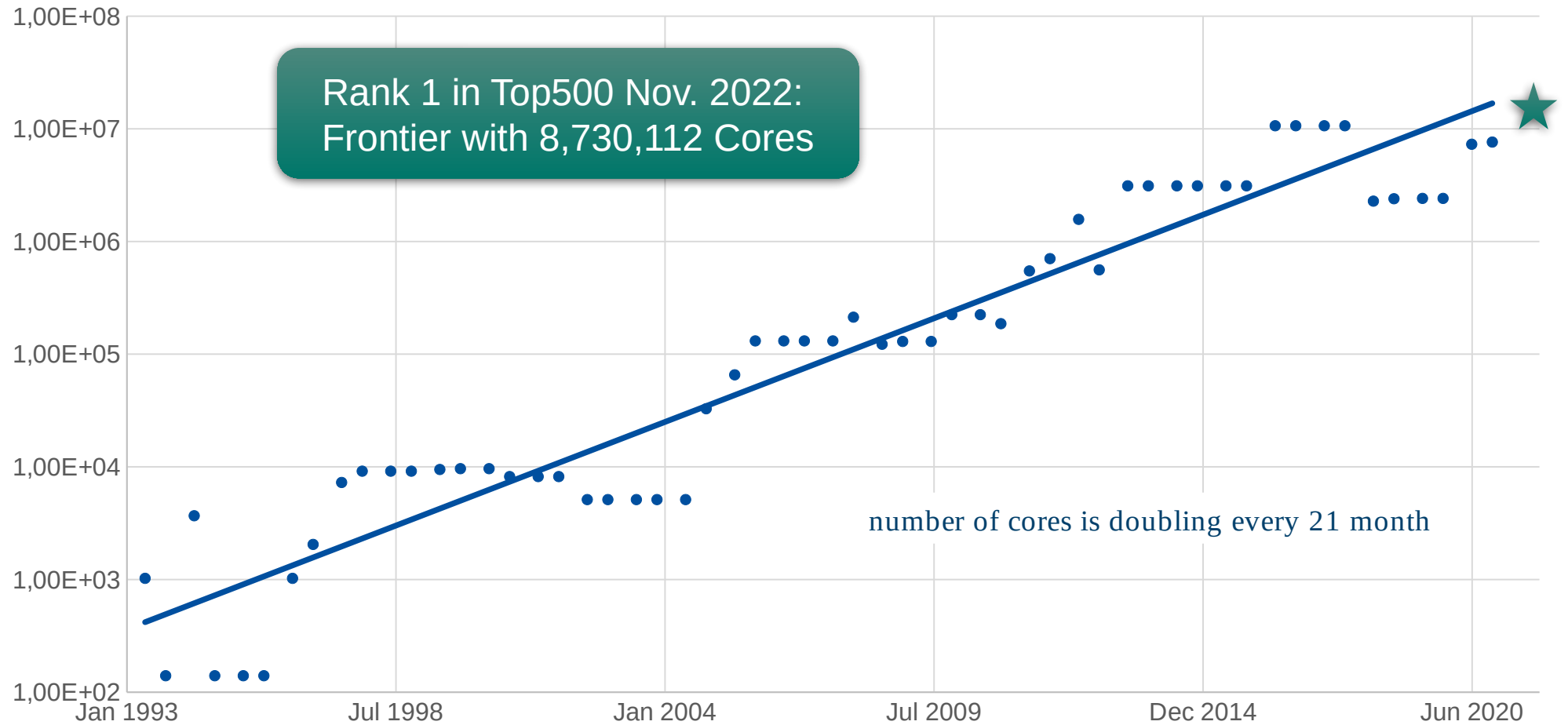
Session 04

Introduction to Parallel Computing

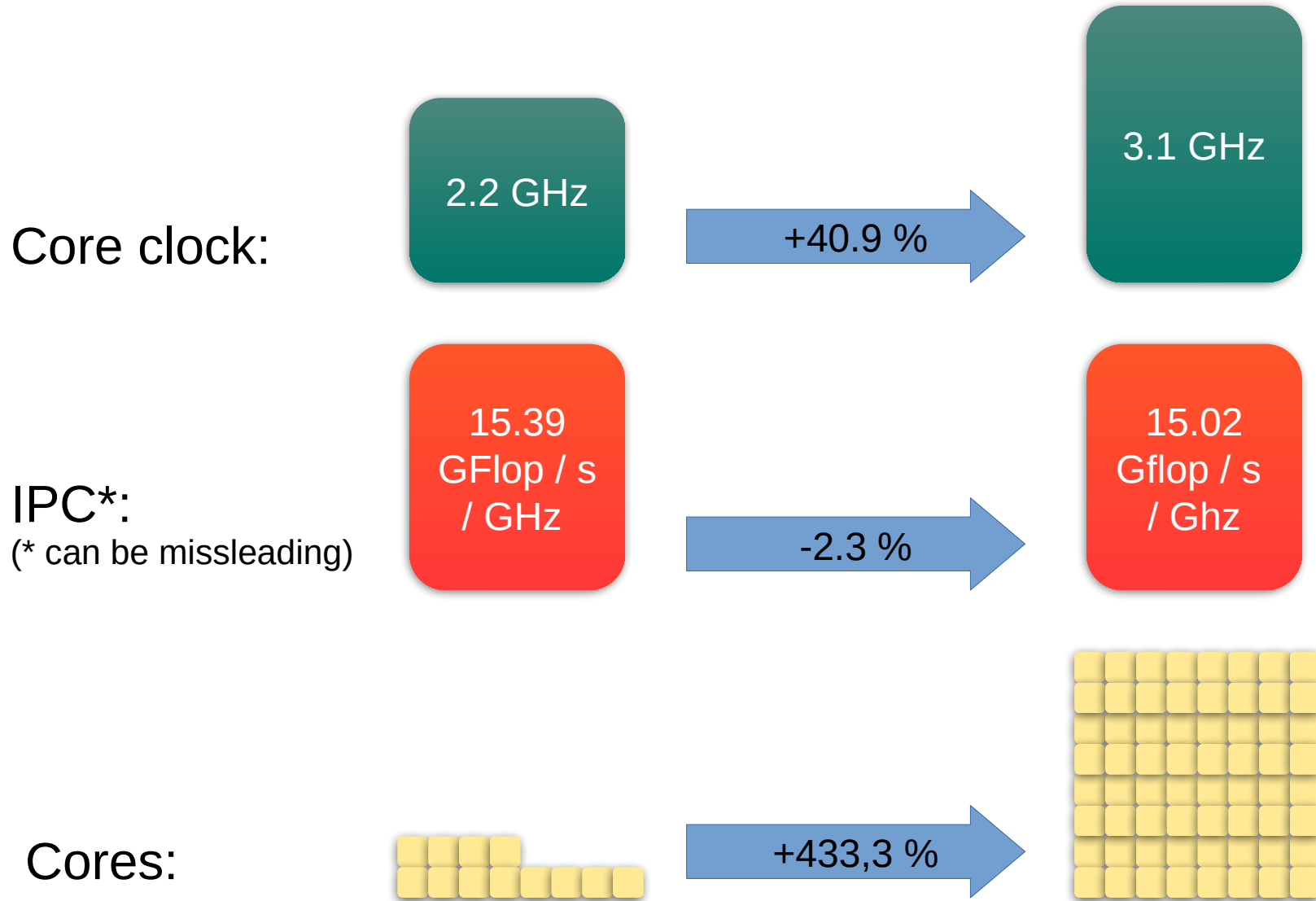


# Why Parallel Computing?

Number of Cores in the fastest Supercomputers

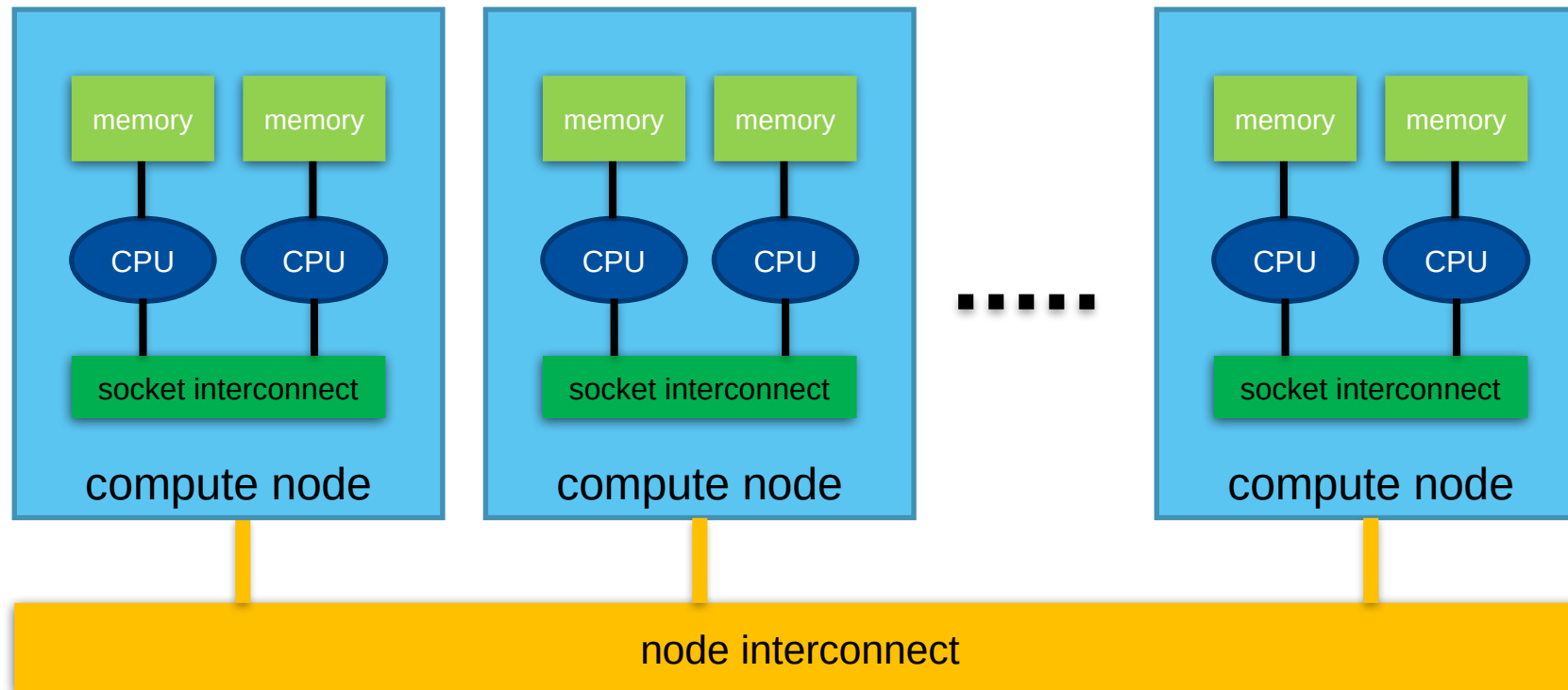


# HPC at University of Oldenburg 2016 to 2023



# Parallel Hardware Architectures

- most modern HPC systems (e.g. CARL and EDDY) are clusters of SMP/ccNUMA nodes

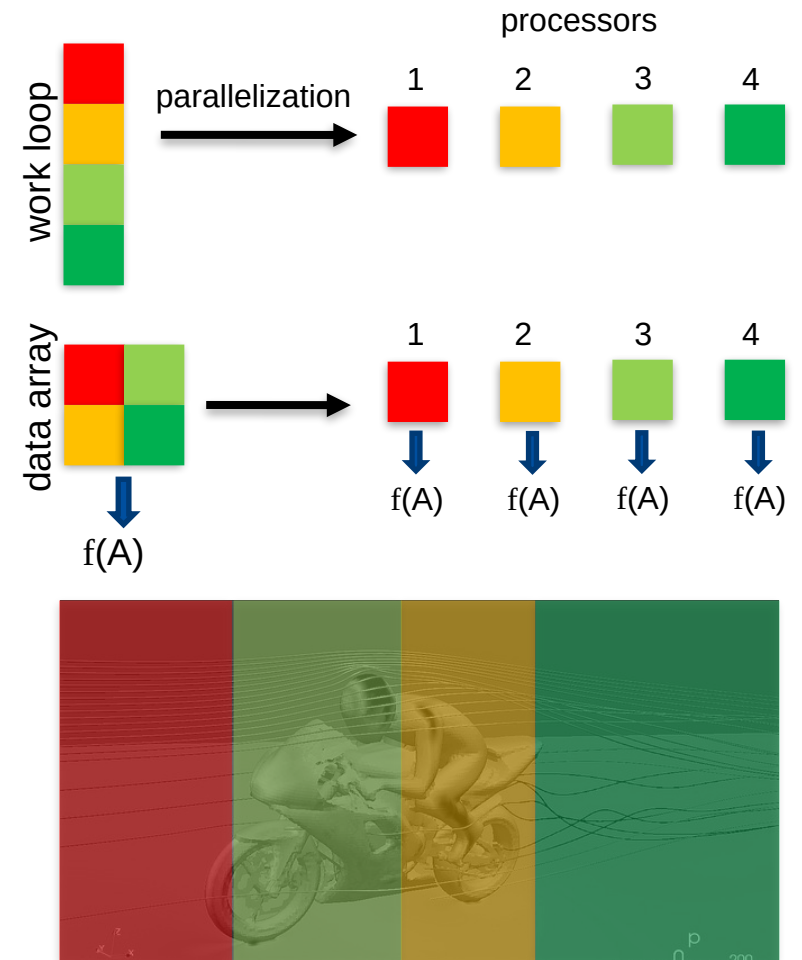


# Parallelization Considerations

- two major resources for computations
  - processor
  - memory
- parallelization means
  - distributing the work
  - distributing the data (on distributed memory machines)
  - synchronization of work
  - communication of data (on distributed memory machines)
- parallel programming models provide the methods to achieve the above goals

# Distributing Work and Data

- Work decomposition
  - based on loop decomposition
- Data decomposition
  - all the work for a local chunk of the data is done by the local processor
- Domain decomposition
  - work and data are distributed according to a higher model, e.g. reality



# Parallel Programming Models

- two dominating programming models:
  - OpenMP: uses directives to define work decomposition
  - MPI: standardized message-passing interface
- other programming models
  - HPF (high-performance Fortran)
  - PGAS (Partitioned Global Address Space), e.g. Co-Array Fortran UPC (Unified Parallel C)
- programming models for compute devices
  - CUDA
  - OpenCL
  - OpenACC
  - ...

# Parallel Programming Models

- two dominating programming models:
  - OpenMP: uses directives to define work decomposition
  - MPI: standardized message-passing interface
- other programming models
  - HPF (high-performance Fortran)
  - PGAS (Partitioned Global Address Space), e.g. Co-Array Fortran UPC (Unified Parallel C)
- programming models for compute devices
  - CUDA
  - OpenCL
  - OpenACC
  - ...



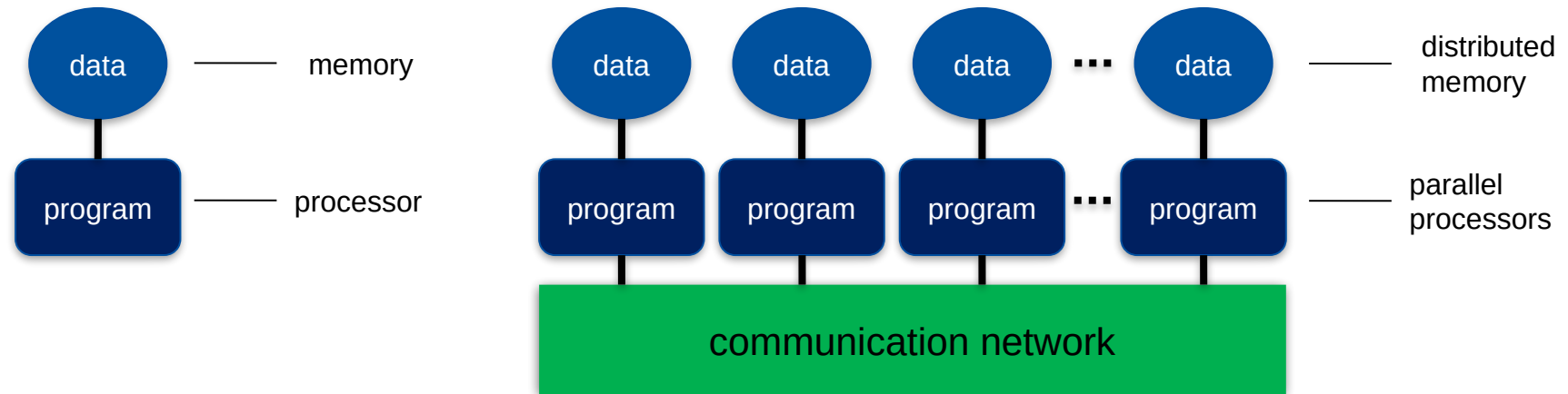
# History of MPI

- MPI is a standard with the prime goals
  - to provide a message-passing interface
  - to provide source-code portability
  - to allow efficient implementations
- MPI exists for more than 25 years
  - MPI-1.0 was released in June, 1994
  - MPI-2.0 was released in July, 1997 and provided additional functionality
  - MPI-3.0 was released in October, 2012 and was developed for better platform and application support (in particular clusters of SMP nodes)
  - MPI-4.0 was released June 2021.

<http://mpi-forum.org/docs/>

# A Message-Passing Interface

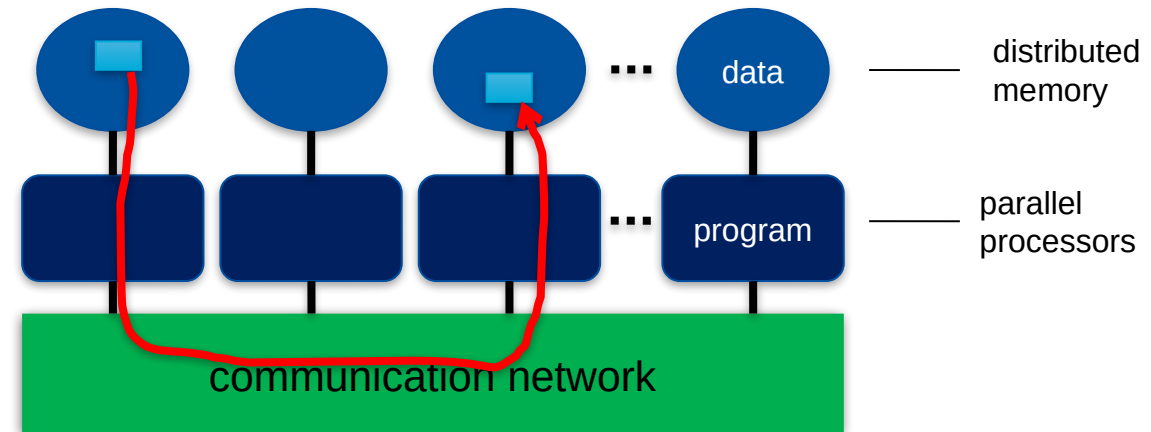
- sequential program vs. message-passing program



- message-passing programming paradigm:
  - each processor runs a (sub)program, typically the same (SPMD)
  - variables of subprograms have the same name but different (distributed) data
  - communication by special library routines □ message passing

# Message Passing

- messages are passed through the communication network
- messages require the following information:
  - sending and receiving process
  - data location
  - data type
  - data size



- in order to use the message-passing interface the program must be
  - connected to the MPI library (at compile time)
  - started with the MPI startup tool (**mpirun** or **mpiexec**)
  - at runtime MPI is initialized with special library calls (**MPI\_Init()**)

# Example MPI Program in C/C++

```
#include <mpi.h>

using namespace std;

int main(int argc, char *argv[]) {
    // initialization of MPI
    MPI_Init(&argc, &argv);

    // do some computation in parallel
    int partial_result = some_computation();
    int global_result = 0;

    // collect the result by an all-to-one communication
    MPI_Reduce(&partial_result, &global_result, 1,
              MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // finalization of MPI
    MPI_Finalize();
}
```

other examples for MPI programs  
are shown in the videos

# Parallel Programming Models

- two dominating programming models:
  - OpenMP: uses directives to define work decomposition
  - MPI: standardized message-passing interface
- other programming models
  - HPF (high-performance Fortran)
  - PGAS (Partitioned Global Address Space), e.g. Co-Array Fortran UPC (Unified Parallel C)
- programming models for **compute devices**
  - CUDA
  - OpenCL
  - OpenACC
  - ...

# GPUs in HPC

- GPUs appeared in the early 2000s in HPC
  - good cost/performance ratio due to mass production for gaming
- initially consumer-grade graphic cards were used
  - limited general-purpose computing
  - algorithms had to mimic graphics display
- today special GPUs are used in HPC
  - no display port
  - run real algorithms

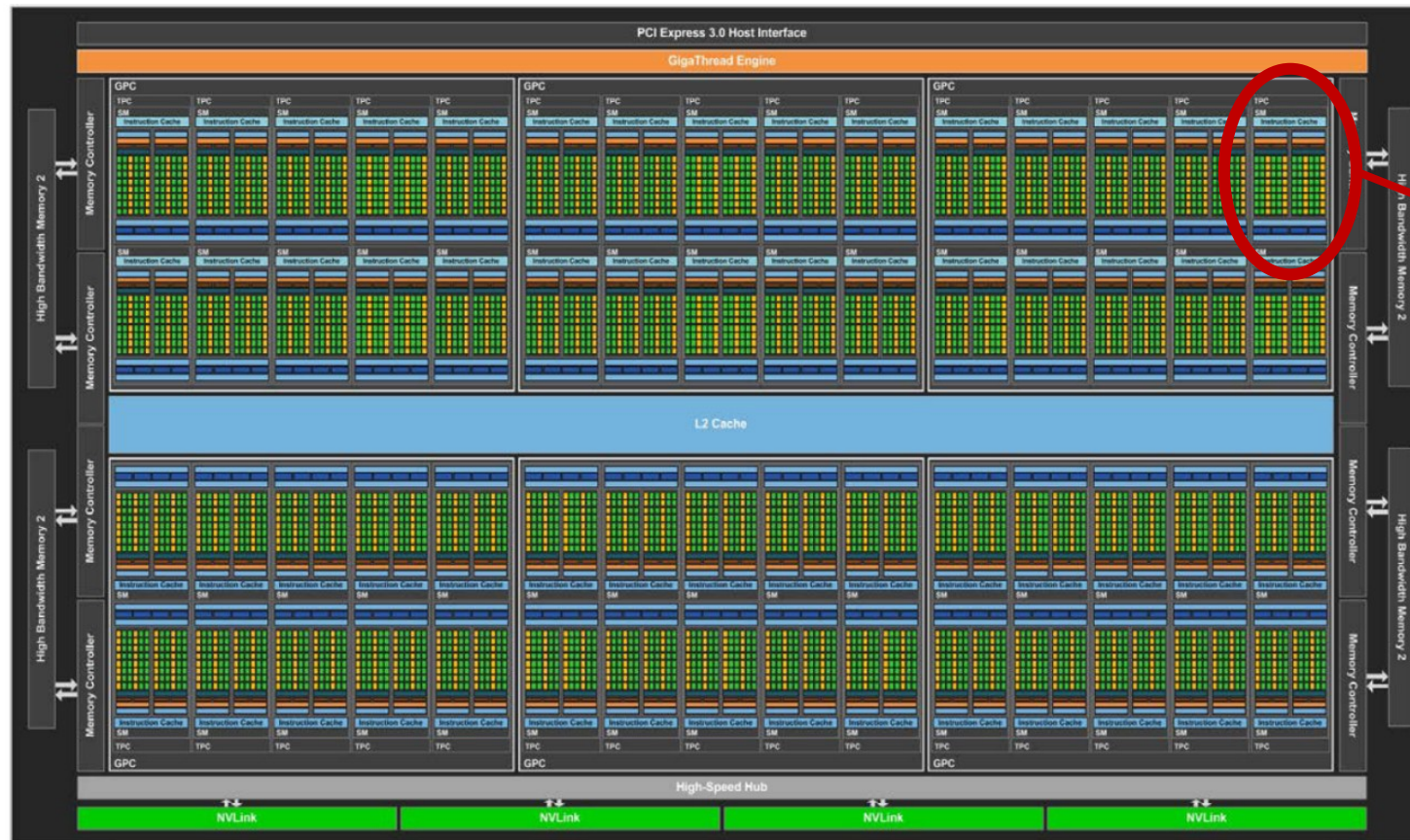
Geforce 8800 GTX (2006)



Tesla P100 (2016)

# Design of GPUs

- Example NVIDIA P100



- organized in Graphics (GPCs) and Texture (TPCs) Processing Clusters
- 60 streaming multiprocessor (SM)
  - basic compute resource
  - each SM has 64 CUDA cores
- 4 MB L2 Cache
  - accessed by 8 memory controllers

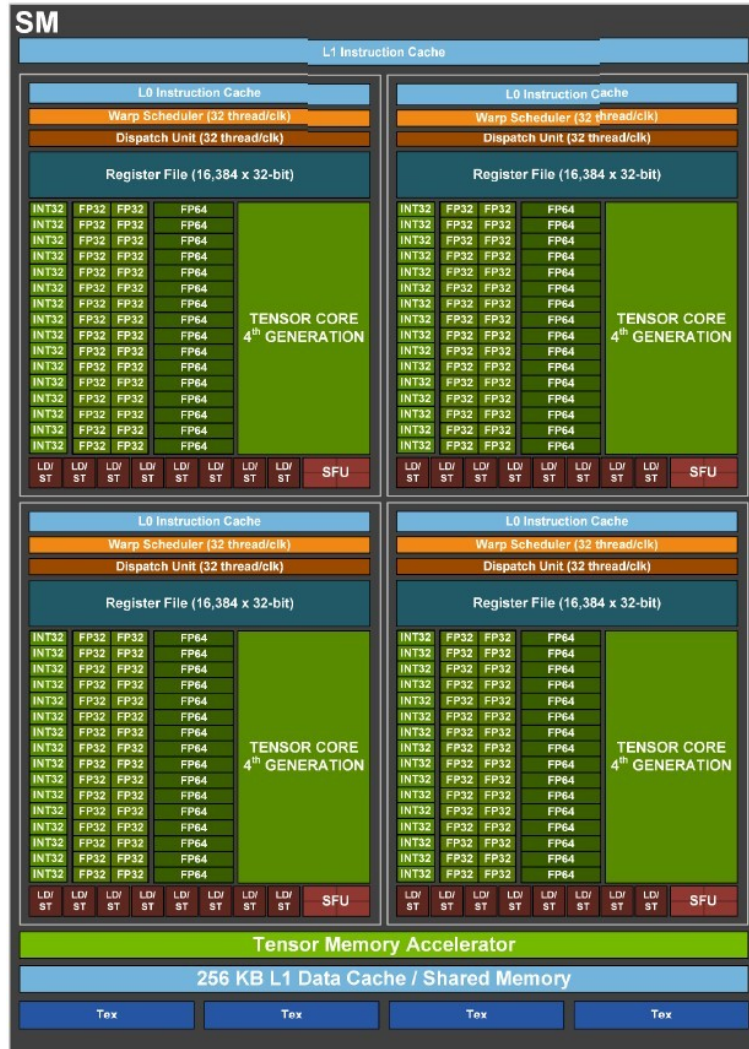
# Design of GPUs (GP100)



- the SM is divided into two blocks
  - each has 32 SP core and 16 DP cores
  - 8 Special Function Units (SFUs)
- 64kB of shared memory

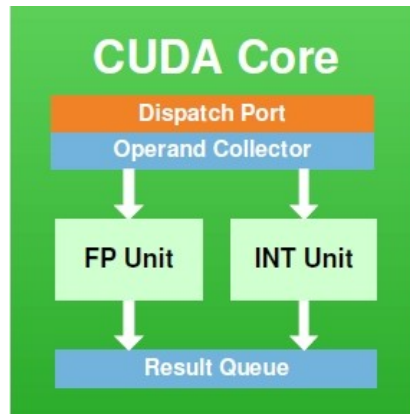
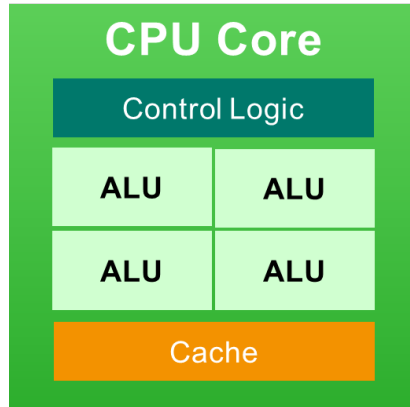


# Design of GPUs (GH100 – New HPC Cluster)

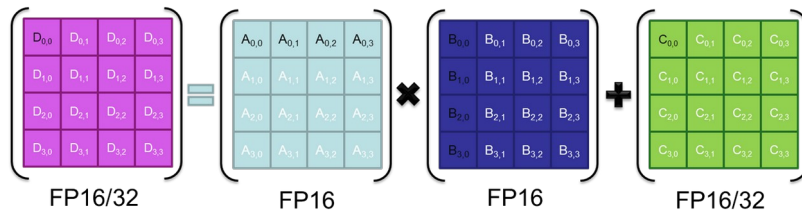


- the SM is divided into 4 blocks, each block has
  - 32 FP32 and 16 FP64 units
  - 16 INT32 units
  - 1 Tensor core
  - 4 SFUs
- 256kB of shared L1-cache

# CPU Core vs. CUDA Core vs. Tensor Core



- CPU cores perform tasks independently
  - control logic is included in the core
  - designed for serial computation
  - modern CPU cores include small SIMD units (e.g. AVX)
- CUDA cores have no control logic
  - control logic is in SM only
  - all cores must perform same instruction
  - SM is a SIMD unit



- Tensor cores operate on 4x4 matrices or larger

# Hybrid Parallel Programming Models

- parallel programming models can be combined in a hybrid approach for better performance or special needs
- common approach is MPI + OpenMP to reduce the number of MPI process (communication overhead)
  - example: use MPI to start a parallel program on multiple dual-socket nodes, one MPI process per socket and OpenMP to utilize the available cores per socket
- MPI + CUDA/OpenACC to use GPUs across multiple nodes or OpenMP + CUDA for multiple GPUs in a single node
  - NVLink (or similar) may allow you to address multiple GPUs within a node as a single device

# Slurm Options for Parallel Computing

# Slurm Options for Parallel Computing

a Slurm job can request to run multiple tasks

- the option **--ntasks** or a combination of **--nodes** and **--tasks-per-node** can be used to set the number of tasks
- tasks can be executed using with **srun** (but this is not a typical use case)
- a process in a parallel MPI programs corresponds to a task and **mpirun** is aware of the requested number of tasks

a Slurm job can also request multiple (logical) cores per task

- the option **--cpus-per-task** can be used for that
- a Slurm **cpu** can be a physical core or a logical (hyper)thread

# Variables in Job Scripts

if you have a parallel application and you have requested multiple tasks and/or CPUs you can use corresponding variables in your job script

- **SLURM\_JOB\_NODELIST:** List of nodes allocated to the job
- **SLURM\_JOB\_NUM\_NODES:** Total number of nodes in the job's resource allocation
- **SLURM\_NTASKS:** Number of tasks requested
- **SLURM\_NTASKS\_PER\_NODE:** Number of tasks requested per node
- **SLURM\_CPUS\_PER\_TASK:** Number of CPUs requested per task

# Slurm Options for GPU Computing

to use the GPU nodes your job script should include

- selection of an appropriate partition

```
#SBATCH --partition mpcg.p      # or mpcb.p or cfdg.p
```

- request for one or two GPUs (Generic Resource in Slurm)

```
#SBATCH --gres=gpu:1           # 1 or 2 gpus
```

- you also need to load the CUDA Toolkit

```
module load CUDA               # add version if needed
```

- note that the driver (and as a result **libcuda.so**) is only available on the GPU nodes