

高能物理中的高性能计算技术

Wei Sun (孙玮)

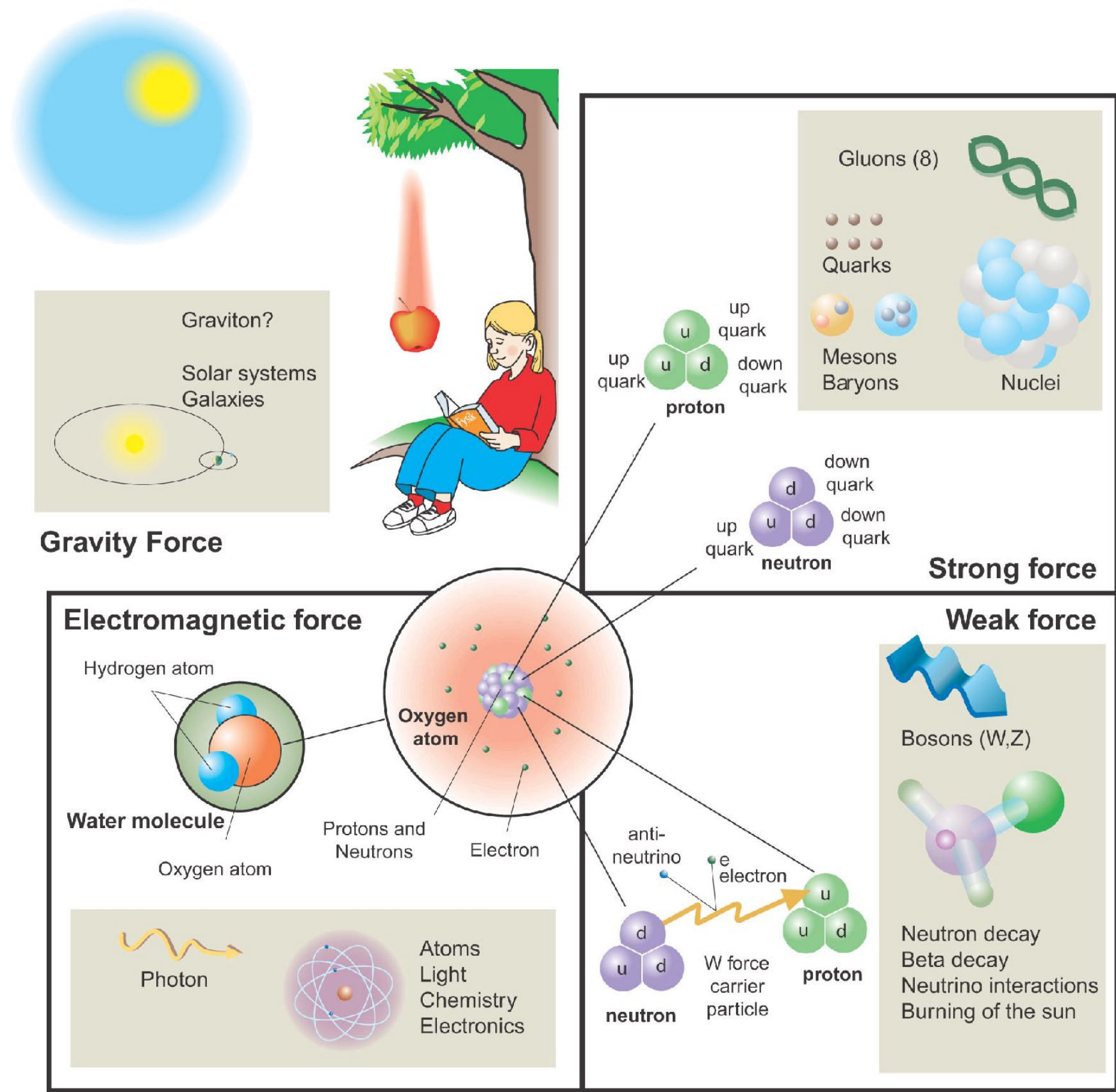
Computing Center, Institute of High Energy Physics

IHEP School of Computing 2022, 2022.8.17-19

Contents

- Introduction
- High performance computers and supercomputers
- Parallel programming models
- Case study with HPC in high energy physics
- Summary and tips

Introduction



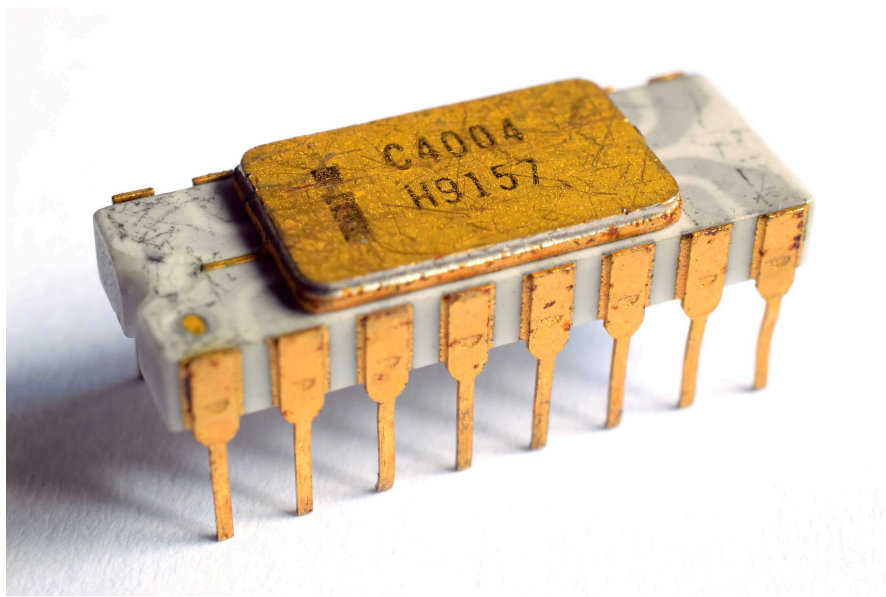
Standard Model

Weinberg–Salam theory

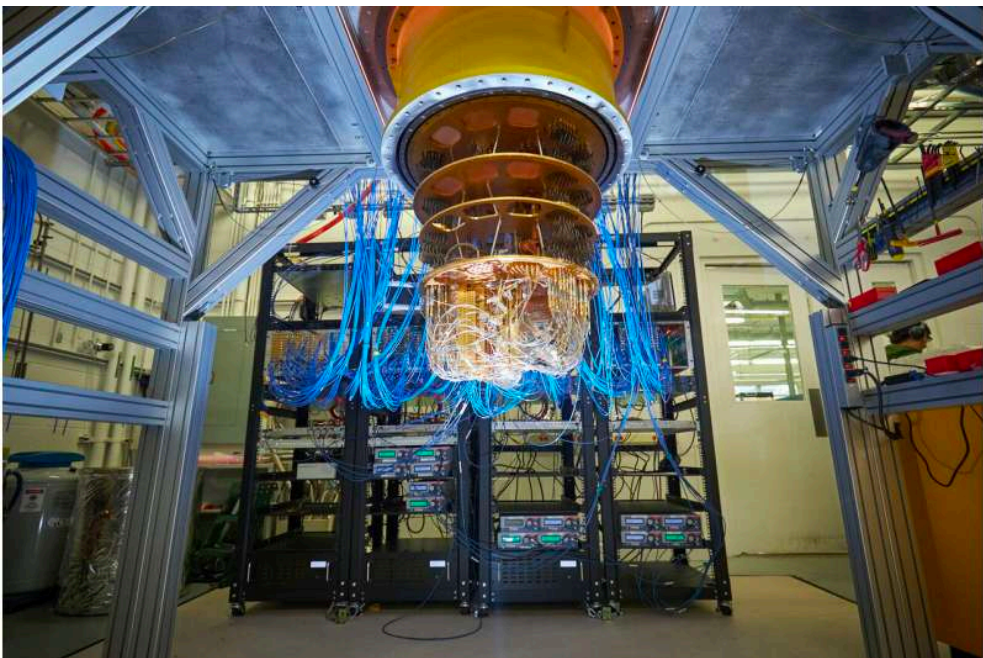
Electromagnetic interaction
Weak interaction

Quantum Chromodynamics

Strong interaction

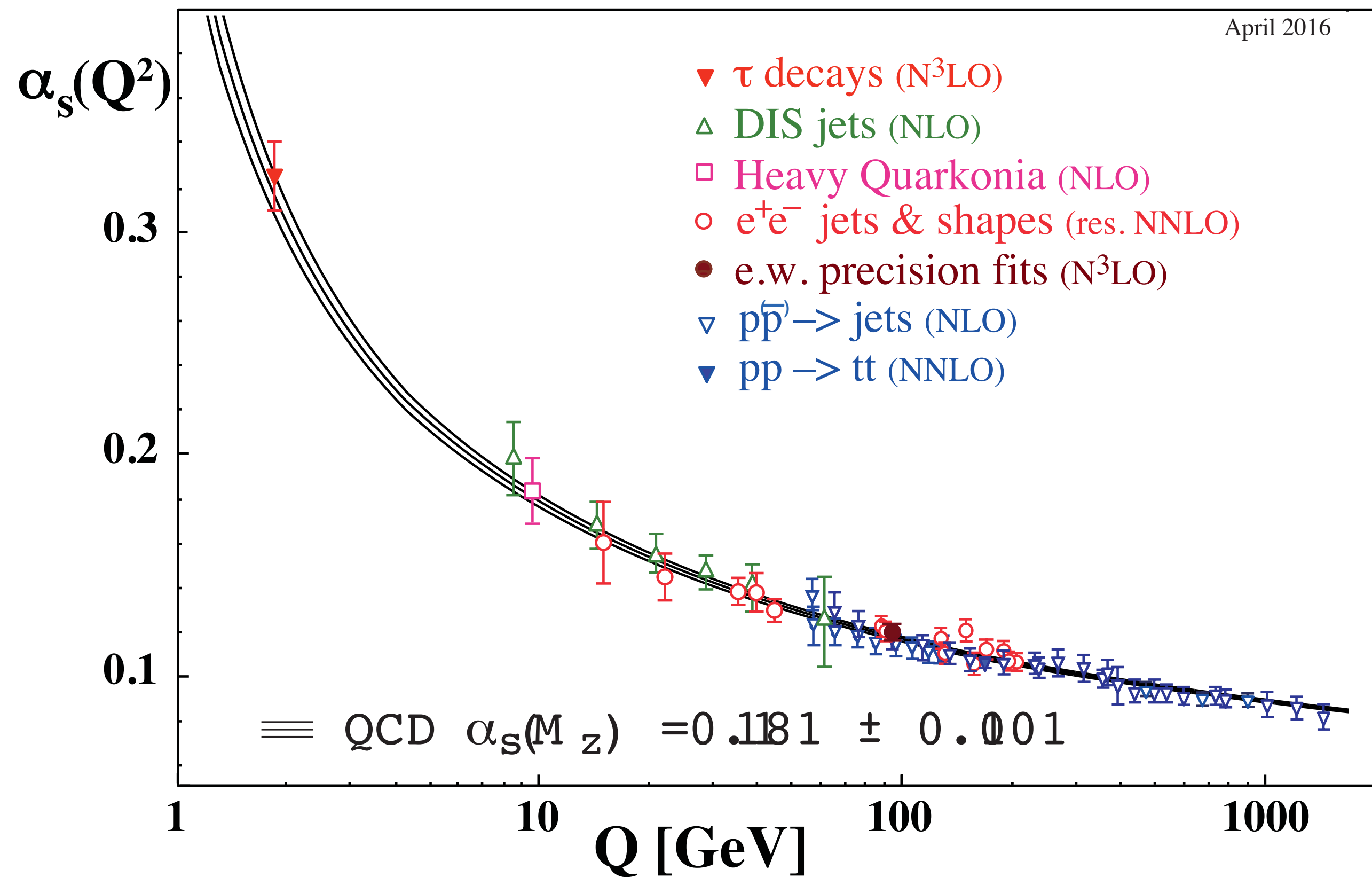


classical computer

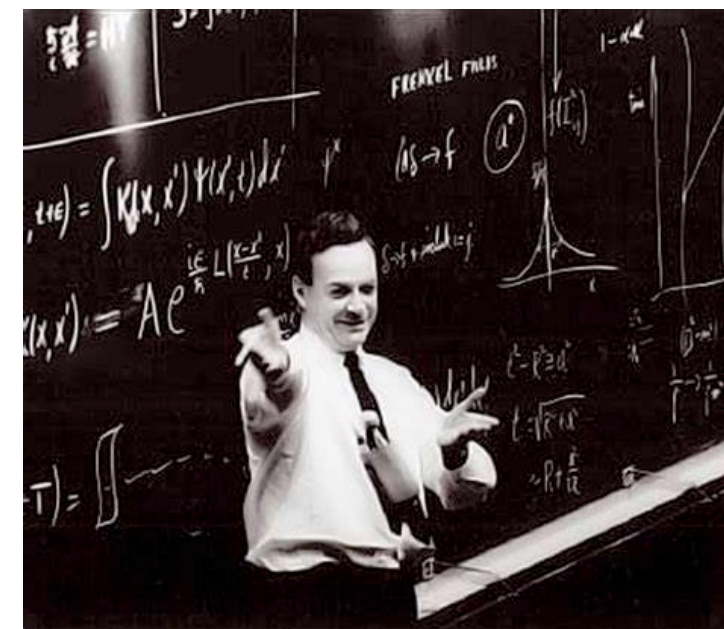
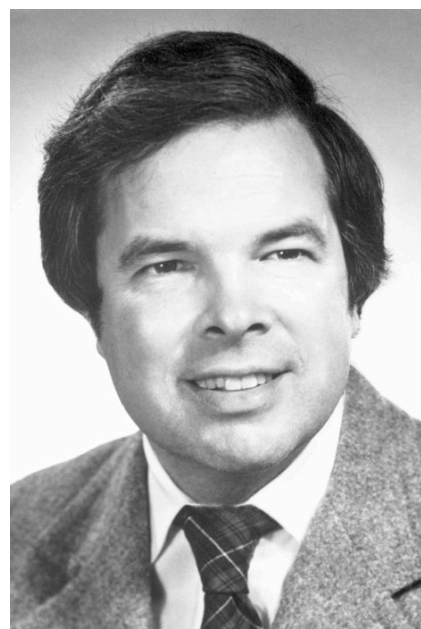
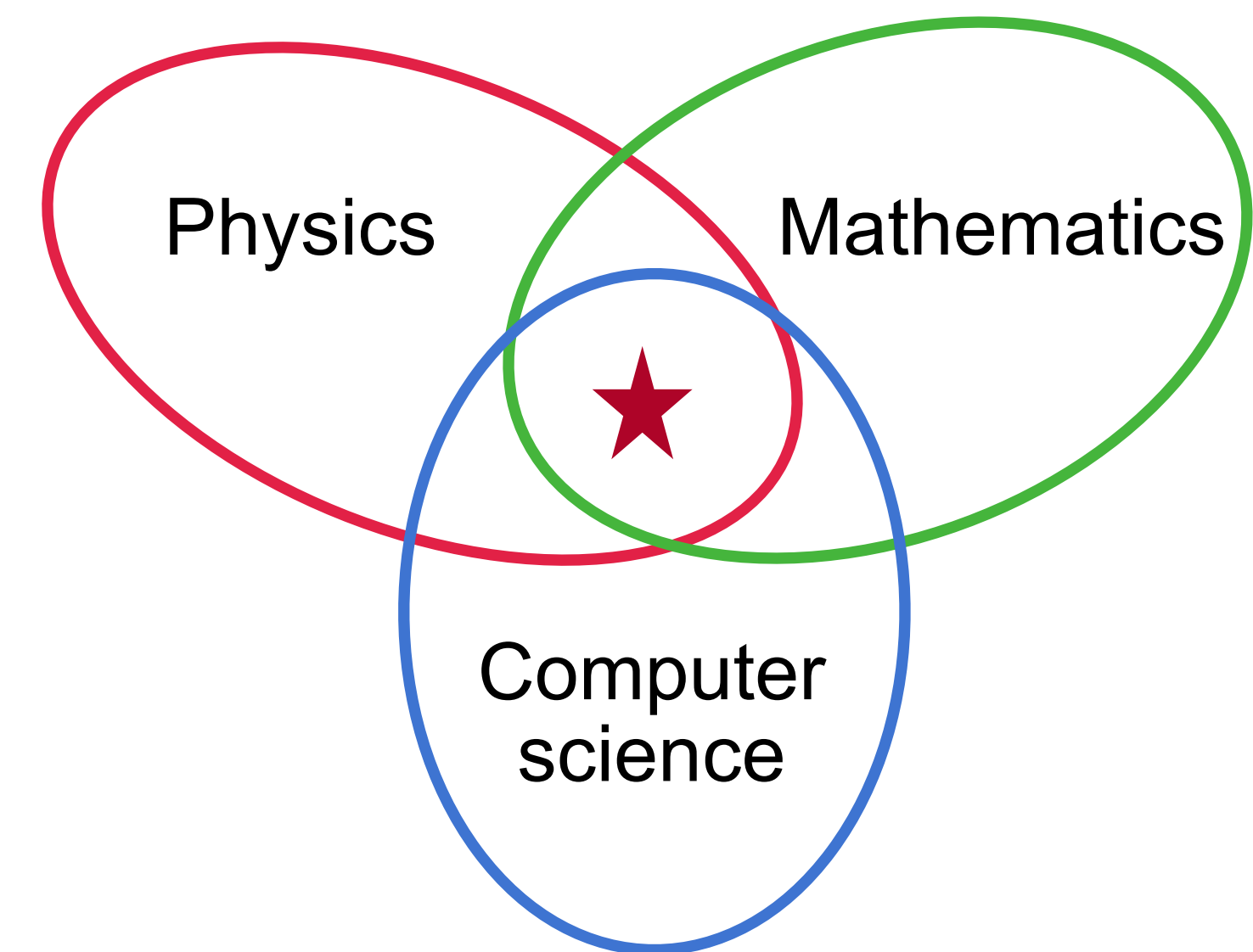


quantum computer

Introduction



- analytical method at high energy
- numerical **Monte Carlo** method at low energy



Introduction

Computational Task	Current Usage	2025 Usage	Current Storage (Disk)	2025 Storage (Disk)	2025 Network Requirements (WAN)
Accelerator Modeling	$\sim 10\text{M} - 100\text{M}$ core-hrs/yr	$\sim 10\text{G} - 100\text{G}$ core-hrs/yr			
Computational Cosmology	$\sim 100\text{M} - 1\text{G}$ core-hrs/yr	$\sim 100\text{G} - 1000\text{G}$ core-hrs/yr	$\sim 10\text{PB}$	$>100\text{PB}$	300Gb/s (burst)
Lattice QCD	$\sim 1\text{G}$ core-hrs/yr	$\sim 100\text{G} - 1000\text{G}$ core-hrs/yr	$\sim 1\text{PB}$	$>10\text{PB}$	
Theory	$\sim 1\text{M} - 10\text{M}$ core-hrs/yr	$\sim 100\text{M} - 1\text{G}$ core-hrs/yr			
Cosmic Frontier Experiments	$\sim 10\text{M} - 100\text{M}$ core-hrs/yr	$\sim 1\text{G} - 10\text{G}$ core-hrs/yr	$\sim 1\text{PB}$	10 – 100PB	
Energy Frontier Experiments	$\sim 100\text{M}$ core-hrs/yr	$\sim 10\text{G} - 100\text{G}$ core-hrs/yr	$\sim 1\text{PB}$	$>100\text{PB}$	300Gb/s
Intensity Frontier Experiments	$\sim 10\text{M}$ core-hrs/yr	$\sim 100\text{M} - 1\text{G}$ core-hrs/yr	$\sim 1\text{PB}$	10 – 100PB	300Gb/s

Introduction

my definition

High Performance Computing \approx Numerical Linear Algebra on Supercomputers

Introduction



Jack J. Dongarra, 2021 Turing Award Laureate

- primary implementor or principal investigator for many libraries such as LINPACK, BLAS, LAPACK, ScaLAPACK etc.
- **Autotuning**: automatically finding algorithmic parameters that produce linear algebra kernels of near-optimal efficiency
- **Mixed precision arithmetic**: Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy
- **Batch computations**: Performance, design, and autotuning of batched GEMM for GPUs
- **Message Passing Interface (MPI) standard**

ACM Turing Award Honors Jack Dongarra for Pioneering Concepts and Methods Which Resulted in World-Changing Computations

Dongarra's Algorithms and Software Fueled the Growth of High-Performance Computing and Had Significant Impacts in Many Areas of Computational Science from AI to Computer Graphics

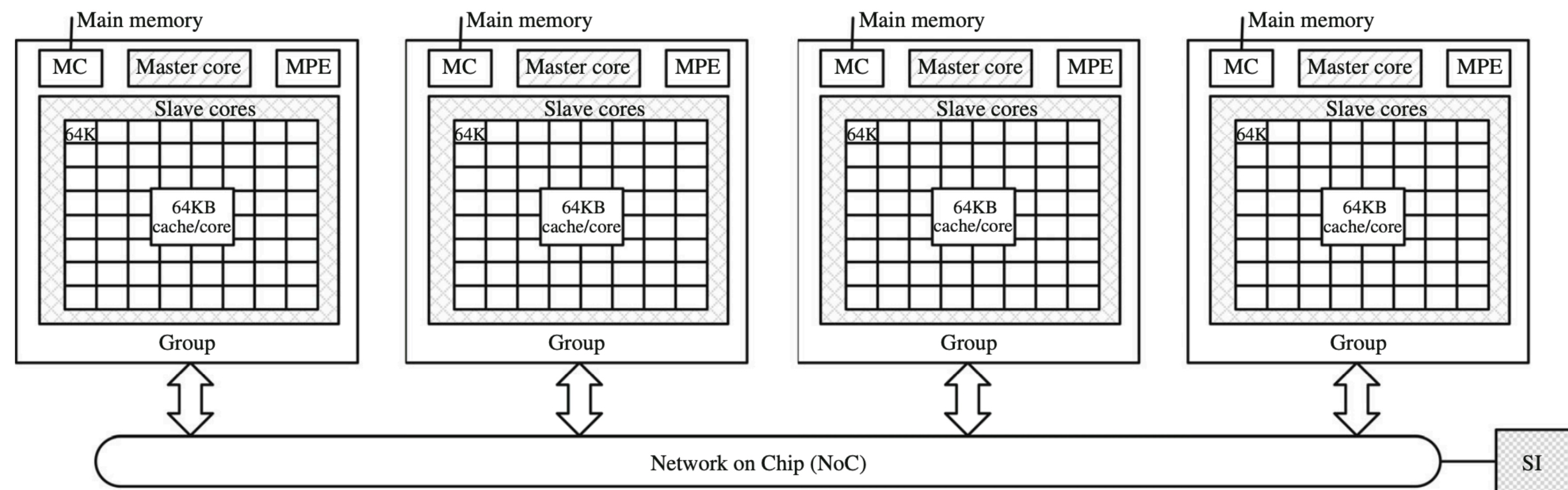
High performance computers and supercomputers

<https://www.top500.org/lists/top500/2022/06/>

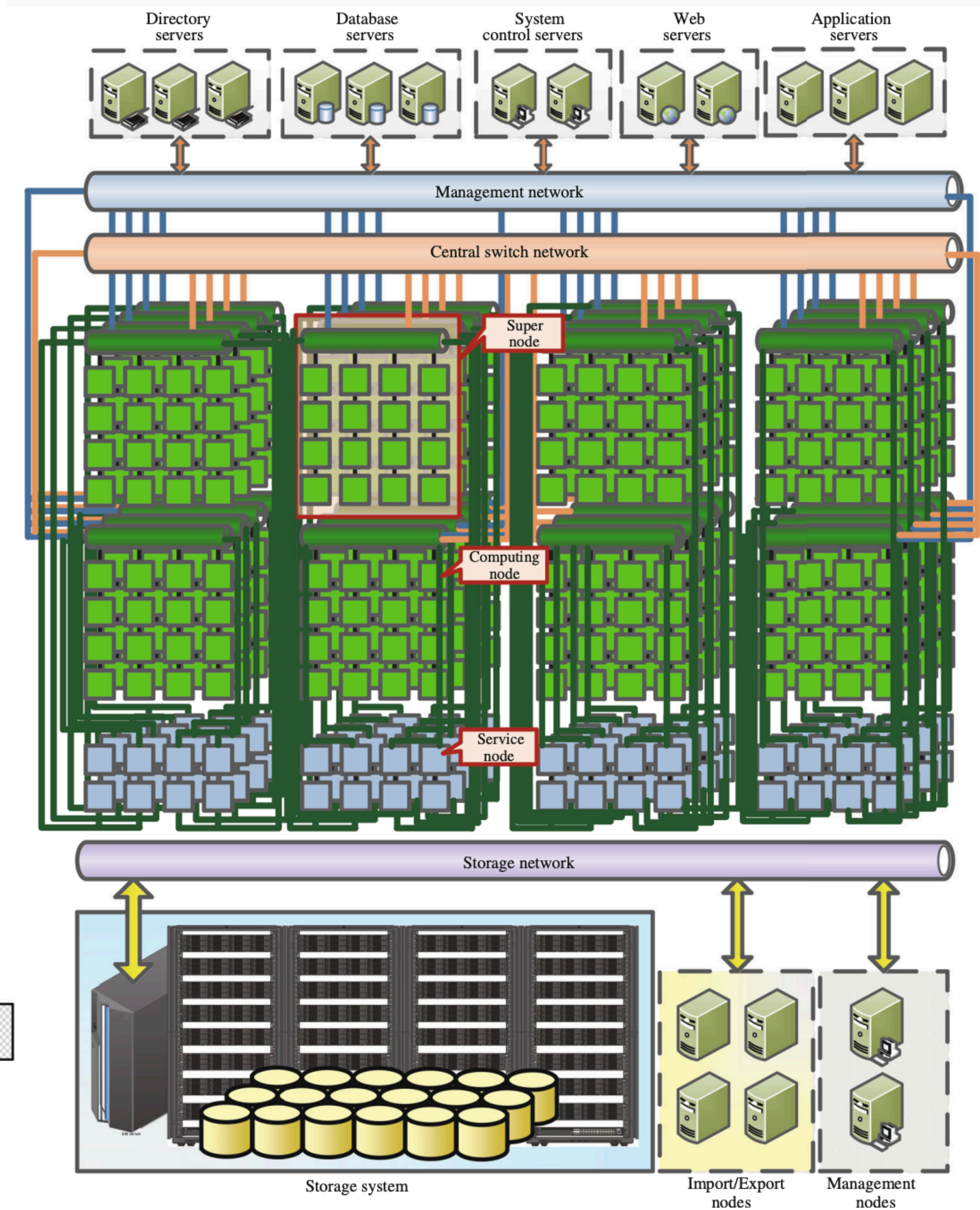
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438
6	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93.01	125.44	15,371
7	Perlmutter - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE DOE/SC/LBNL/NERSC United States	761,856	70.87	93.75	2,589
8	Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63.46	79.22	2,646
9	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61.44	100.68	18,482
10	Adastr a - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Grand Equipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Suprieur (GENCI-CINES) France	319,072	46.10	61.61	921

High performance computers and supercomputers

Supercomputer example: Sunway TaihuLight

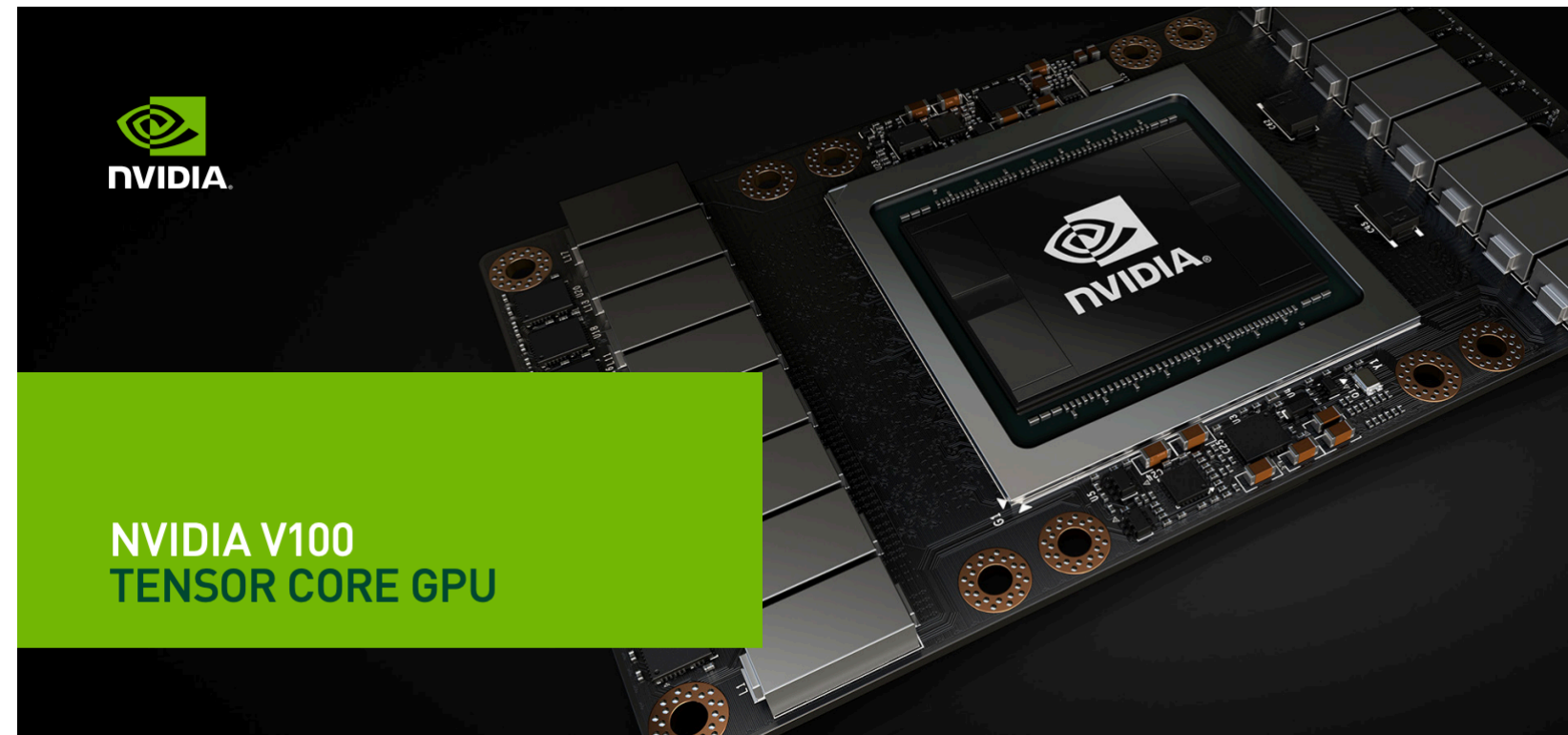


Fu, H., Liao, J., Yang, J. *et al.*, Sci. China Inf. Sci. **59**, 072001 (2016).



High performance computers and supercomputers

High performance clusters at IHEP



Nvidia V100 GPU: ≈ 300 cards



x86 CPU: $\approx O(10000)$ cores



ARM CPU: $\approx O(10000)$ cores

High performance computers and supercomputers

LQCD with HPC

- Decades ago - customized processors
- QCDOC (QCD On a Chip)
- Nowadays - supercomputers / clusters
- TOP 500

QCDOC Asic, 1 Gflop/s



- LQCD awarded 1995,1998,2006 Golden Bell Prize and 2018 finalist



Available online at www.sciencedirect.com



Computer Physics Communications 177 (2007) 631–639

Computer Physics
Communications

www.elsevier.com/locate/cpc

Lattice QCD as a video game **Before CUDA release!**

Győző I. Egri^a, Zoltán Fodor^{a,b,c,*}, Christian Hoelbling^b, Sándor D. Katz^{a,b}, Dániel Nógrádi^b,
Kálmán K. Szabó^b

^a *Institute for Theoretical Physics, Eötvös University, Budapest, Hungary*

^b *Department of Physics, University of Wuppertal, Germany*

^c *Department of Physics, University of California, San Diego, USA*

Received 2 February 2007; received in revised form 29 May 2007; accepted 7 June 2007

Available online 15 June 2007

Parallel programming models

Common programming language in HPC

- Fortran (Formula Translation)
 - Oldest high level programming language, first compiler released in 1957
 - Designed for numerical and scientific computing
 - Highly efficient, still widely used in high performance computing today
- C
 - Flexible, efficient, ...
- C++
 - Efficient, abstract, multi-paradigm (procedural, object oriented, functional)
- Assembly
 - Highly efficient but not portable across different processor architecture
- Python
 - Slow in python itself, but with great library such as Scipy, very suitable for data processing, analysis and visualization

Parallel programming models

MPI + X model (cluster level + node level + processor level + instruction level)

- MPI (Message Passing Interface)
 - MPI is a communication protocol for programming parallel computers
 - The dominant programming model in high performance computing today
 - Support point-to-point and collective communication
 - MPI version 1.0 standard released in 1994
 - Directly callable from C, C++, Fortran
 - Very suitable for distributed memory system, therefore supported by all kinds of supercomputers
- Major implementation
 - MPICH (<https://www.mpich.org/>)
 - Open MPI (<https://www.open-mpi.org/>)
 - Many others derived from MPICH and Open MPI, such as Intel MPI, Cray MPI, IBM Spectrum MPI

Parallel programming models

MPI Basics

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char** argv) {
5      // Initialize the MPI environment
6      MPI_Init(&argc, &argv);
7
8      // Get the number of processes
9      int size;
10     MPI_Comm_size(MPI_COMM_WORLD, &size);
11
12     // Get the rank of the process
13     int rank;
14     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16     // Get the name of the processor
17     char processor_name[MPI_MAX_PROCESSOR_NAME];
18     int name_len;
19     MPI_Get_processor_name(processor_name, &name_len);
20
21     // Print off a hello world message
22     printf("Hello world from processor %s, rank %d out of %d processors\n",
23           processor_name, rank, size);
24
25     // Finalize the MPI environment.
26     MPI_Finalize();
27 }
```

- Compile: `mpicc hello_world.c -o hello_world`
- Run: `mpirun -np 4 hello_world`
- NOTE: MPI is a library and mpicc is not a compiler, it is a wrapper over regular C compiler
- Use `mpicc -show` to see the compile and link flags
- **`gcc -I /path to MPI/include -L /path to MPI/lib -lmpi`**

output

```
Hello world from processor ui03.hep.ustc.edu.cn, rank 1 out of 4 processors
Hello world from processor ui03.hep.ustc.edu.cn, rank 2 out of 4 processors
Hello world from processor ui03.hep.ustc.edu.cn, rank 3 out of 4 processors
Hello world from processor ui03.hep.ustc.edu.cn, rank 0 out of 4 processors
```

Parallel programming models

MPI Basics (point-to-point communication)

- Total 400+ APIs

```
MPI_Send(  
    void* data,  
    int count,  
    MPI_Datatype datatype,  
    int destination,  
    int tag,  
    MPI_Comm communicator)
```

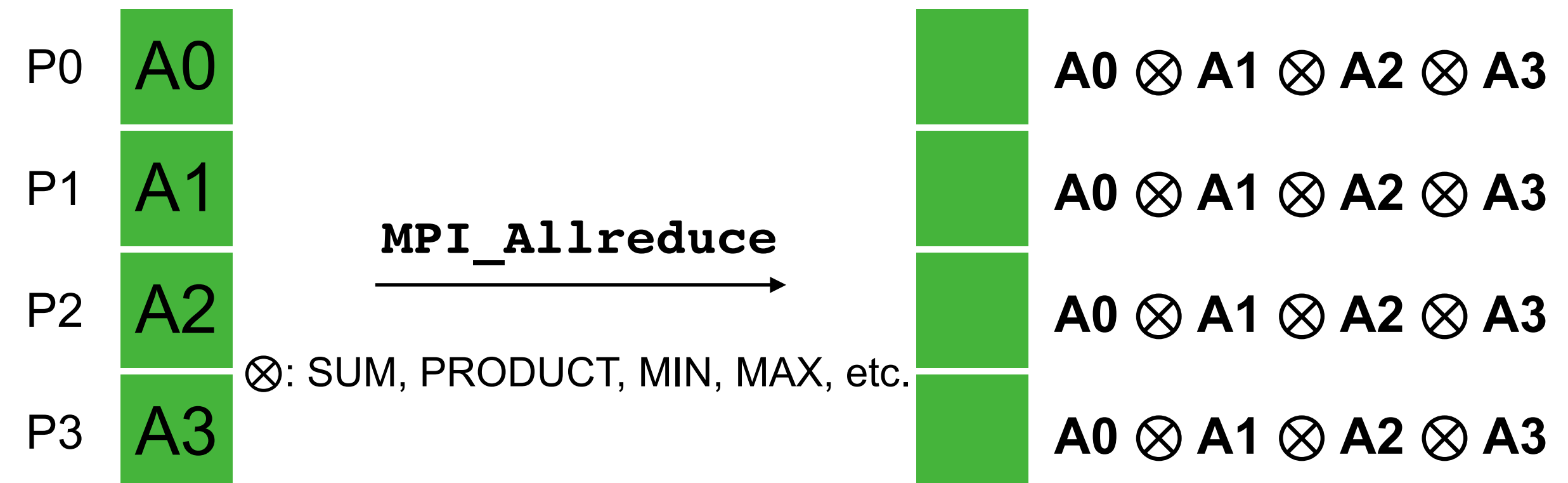
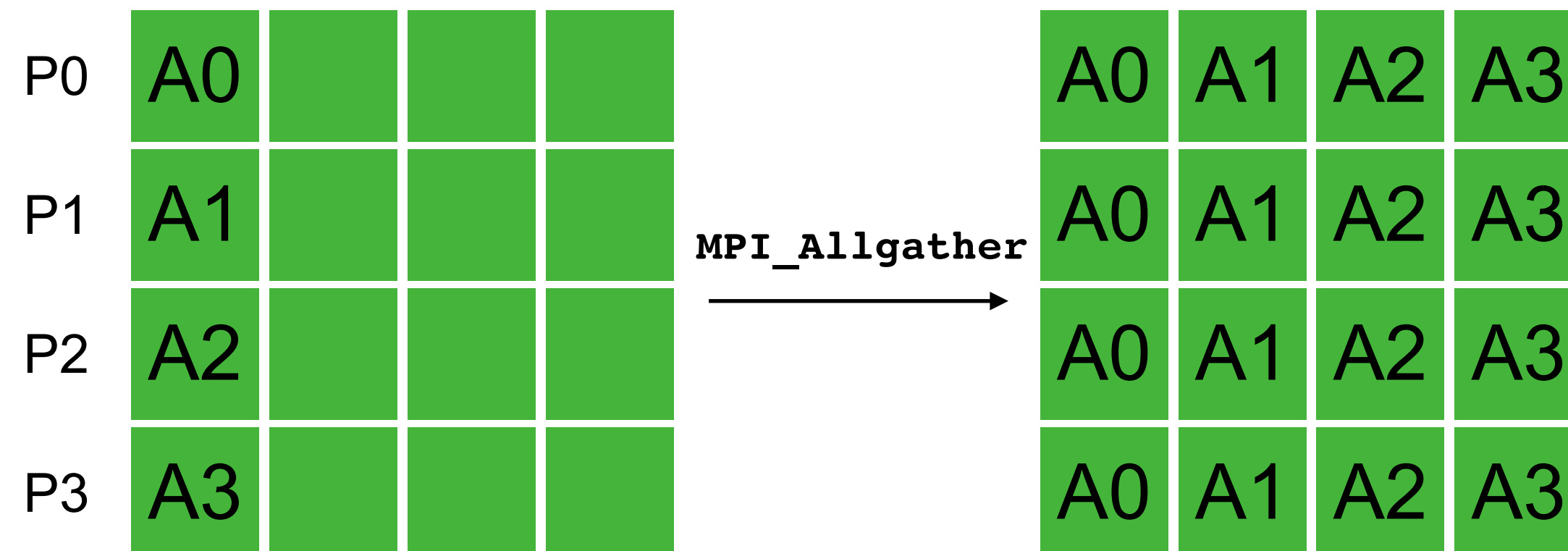
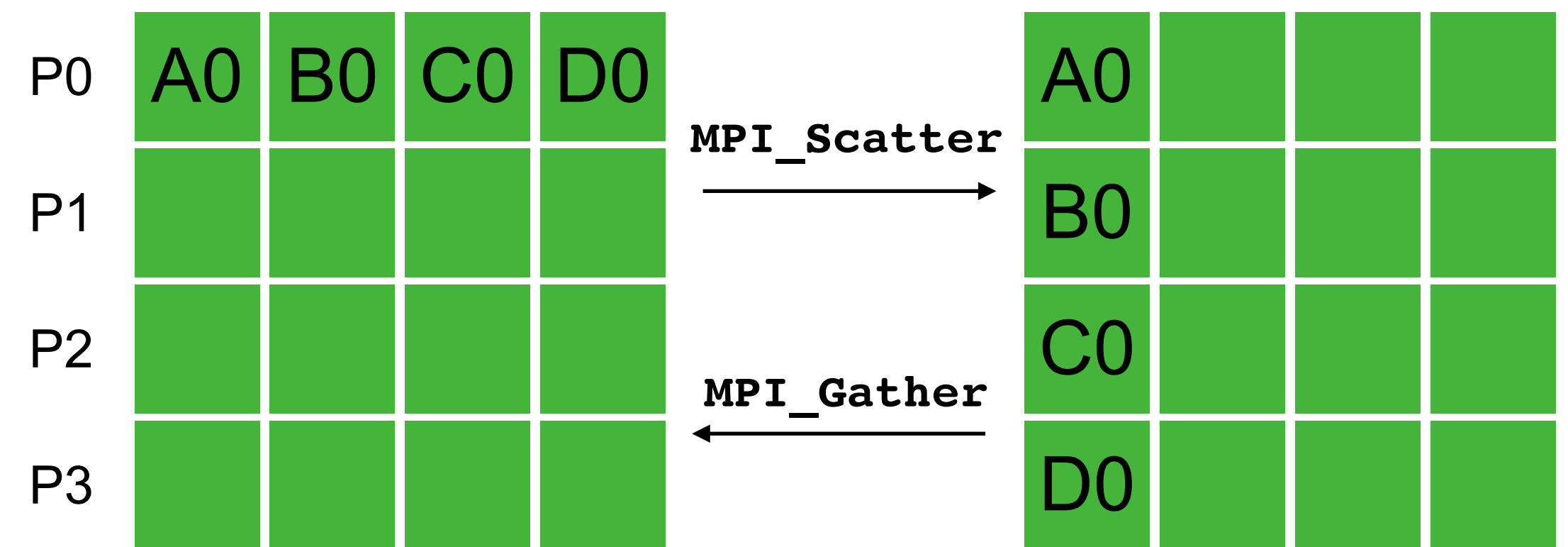
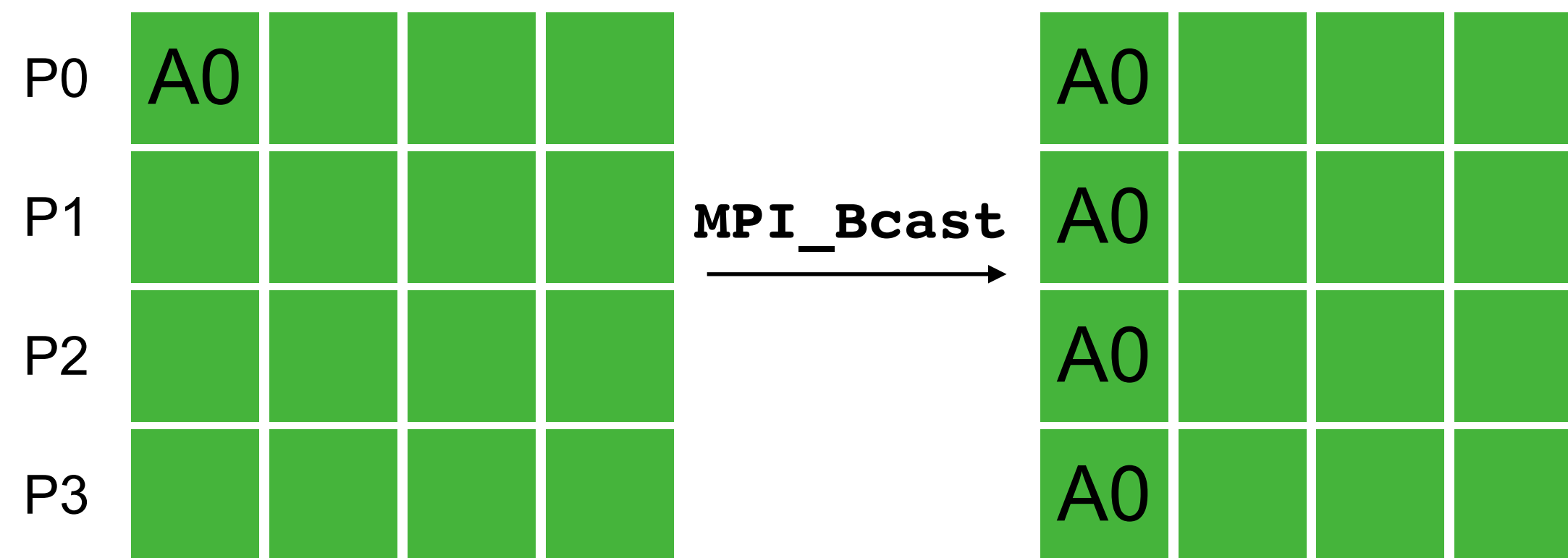
```
MPI_Recv(  
    void* data,  
    int count,  
    MPI_Datatype datatype,  
    int source,  
    int tag,  
    MPI_Comm communicator,  
    MPI_Status* status)
```

```
int world_rank;  
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);  
int world_size;  
MPI_Comm_size(MPI_COMM_WORLD, &world_size);  
  
int number;  
if (world_rank == 0) {  
    number = -1;  
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);  
} else if (world_rank == 1) {  
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,  
            MPI_STATUS_IGNORE);  
    printf("Process 1 received number %d from process 0\n",  
           number);  
}
```


Parallel programming models

MPI Basics (collective communication)

- Total 400+ APIs



Parallel programming models

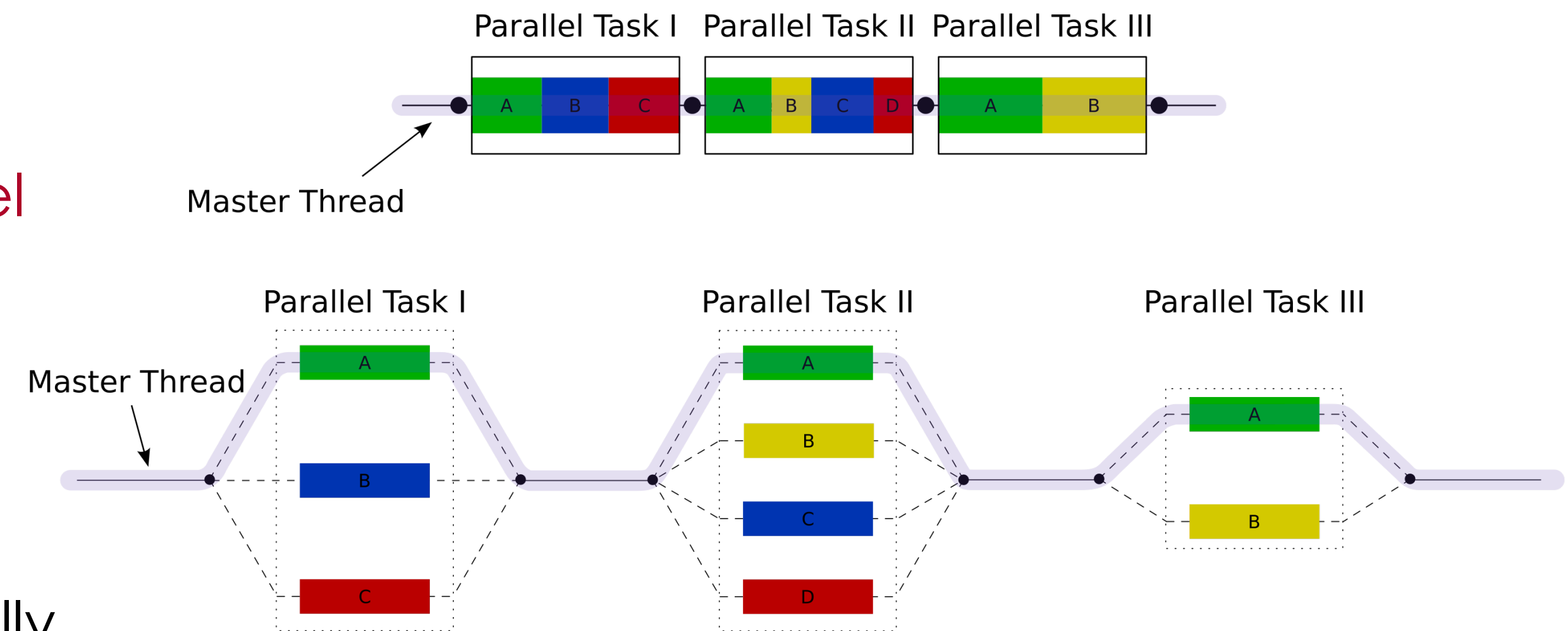
OpenMP (Open Multi-Processing)

- Pros

- API that supports various instruction set architectures, operating system, and C, C++, Fortran
- First standard released in 1997
- Compiler directive based
- Simple, flexible, portable, scalable
- Easy to modify existing serial code into parallel
- OpenMP 4.0 and later version support GPUs

- Cons

- Multi-threading programming is easy to implement but hard to debug in general
 - Need to deal with **race condition** very carefully
 - Only used for parallelism **within a node**
- Major implementation
 - GCC, Intel, Clang



source: wikipedia

Parallel programming models

OpenMP hello world example

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(int argc, char **argv) {
5  #pragma omp parallel
6  {
7      int threads_total = omp_get_num_threads();
8      int thread_id = omp_get_thread_num();
9      printf("Hello, world from thread %d,"
10           "out of %d threads.\n",
11           thread_id,
12           threads_total);
13  }
14  return 0;
15 }
```

```
1  #include <omp.h>
2  #include <math.h>
3
4  int main(int argc, char **argv) {
5      const int N = 1000000;
6      int a[N];
7
8      #pragma omp parallel for
9      for (int i = 0; i < N; i++) {
10         a[i] = sin(i);
11     }
12
13     return 0;
14 }
15
```

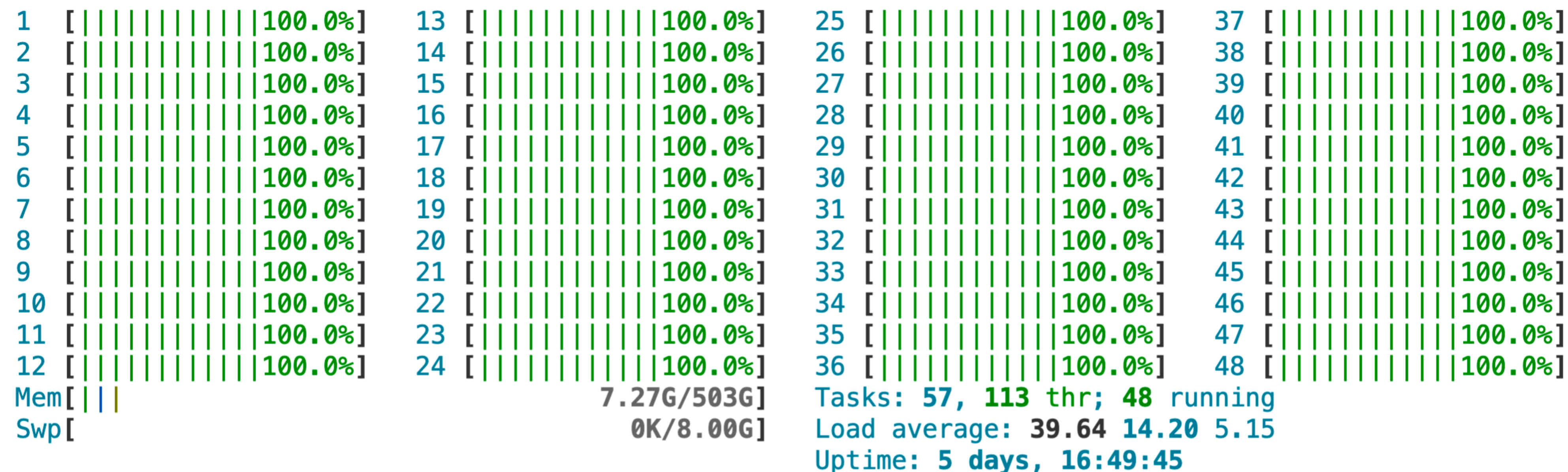
Compile: `gcc -fopenmp hello_world.c -o hello_world`

Run: `./hello_world` # use all cores / hardware threads available on single node

`OMP_NUM_THREADS=4 ./hello_world` # use 4 cores / hardware threads

Parallel programming models

OpenMP program monitored with htop

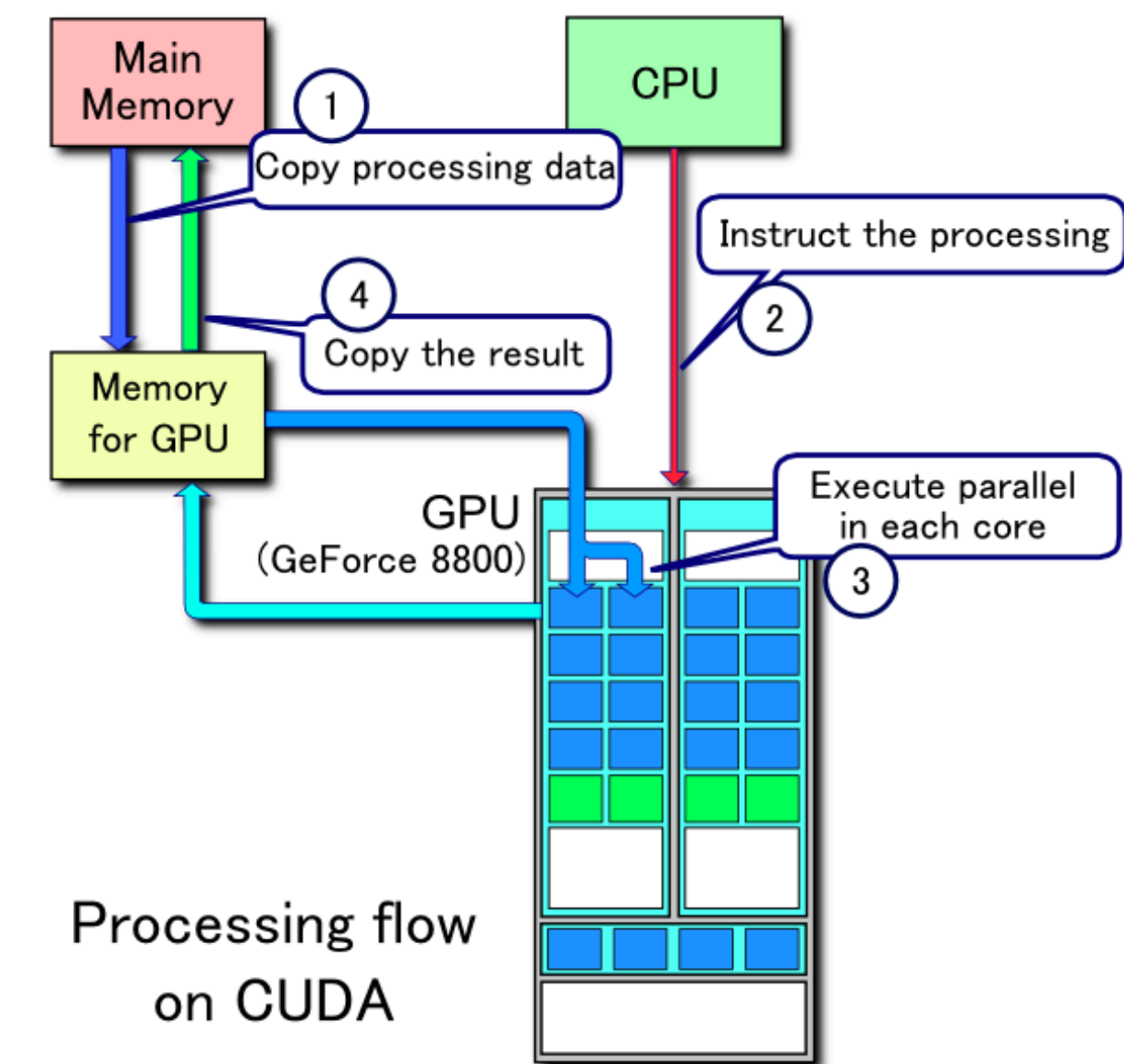


CPU	PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
26	47947	sunwei	20	0	391M	2996	616	R	4786	0.0	1h23:25	└─ ./parallel_for_openmp
24	47994	sunwei	20	0	391M	2996	616	R	100.	0.0	1:44.33	└─ ./parallel_for_openmp
33	47993	sunwei	20	0	391M	2996	616	R	100.	0.0	1:44.33	└─ ./parallel_for_openmp
23	47992	sunwei	20	0	391M	2996	616	R	100.	0.0	1:44.34	└─ ./parallel_for_openmp
25	47991	sunwei	20	0	391M	2996	616	R	100.	0.0	1:44.20	└─ ./parallel_for_openmp
22	47990	sunwei	20	0	391M	2996	616	R	99.4	0.0	1:44.33	└─ ./parallel_for_openmp
48	47989	sunwei	20	0	391M	2996	616	R	99.4	0.0	1:44.17	└─ ./parallel_for_openmp
21	47988	sunwei	20	0	391M	2996	616	R	100.	0.0	1:44.34	└─ ./parallel_for_openmp
47	47987	sunwei	20	0	391M	2996	616	R	100.	0.0	1:44.34	└─ ./parallel_for_openmp
46	47986	sunwei	20	0	391M	2996	616	R	98.7	0.0	1:44.30	└─ ./parallel_for_openmp
16	47985	sunwei	20	0	391M	2996	616	R	99.4	0.0	1:44.13	└─ ./parallel_for_openmp

Parallel programming models

CUDA for GPU computing

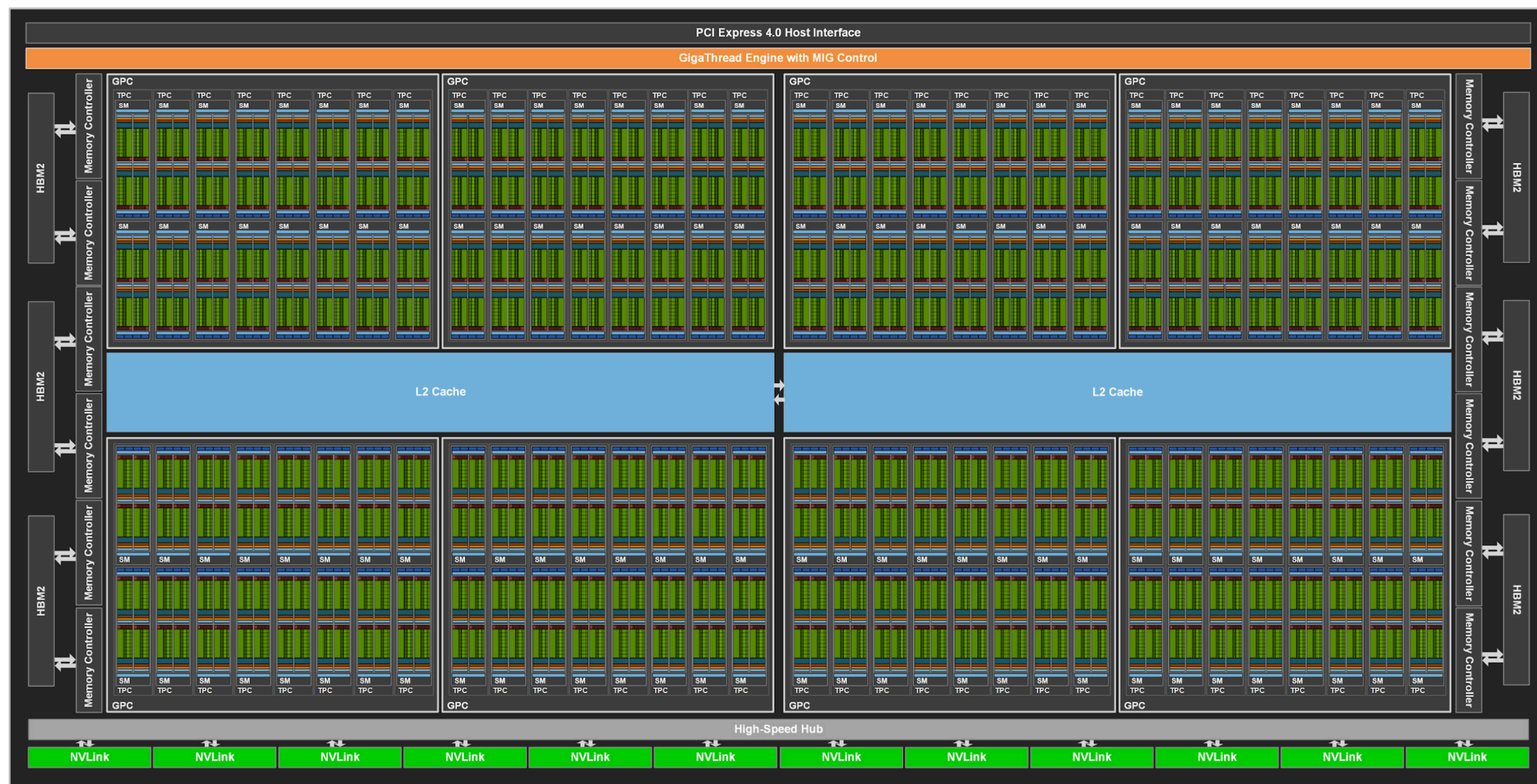
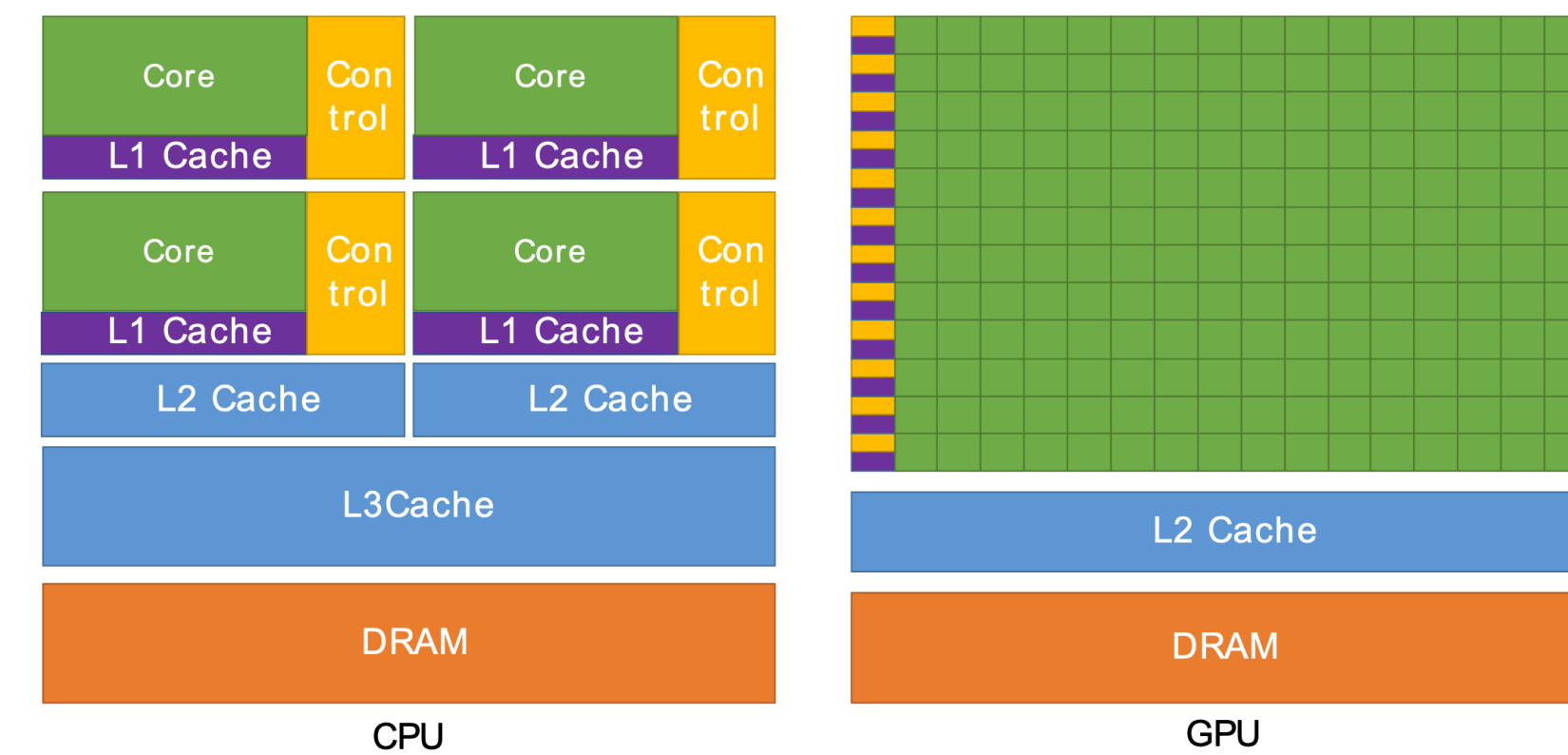
- CUDA (Compute Unified Device Architecture)
 - CUDA is a parallel programming framework and API for general purpose GPU (GP GPU) computing
 - Developed by Nvidia and support Nvidia' s GPUs
 - Supported Tesla -> Fermi -> Kepler -> Maxwell -> Pascal -> Volta -> Turing -> Ampere -> Hopper
 - Directly callable from C, C++, Fortran
 - Need CUDA Toolkit to compile
 - Free but not open source
 - Multi-node GPU programming with CUDA-aware MPI
 - The HIP (Heterogeneous Interface for Portability) developed by AMD can is portable both for AMD and Nvidia' s GPUs, and also free and open source



source: wikipedia

Parallel programming models

Details of GPU architecture



More on [CUDA C++ Programming Guide](#) and [Nvidia Ampere Architecture Whitepaper](#)

Parallel programming models

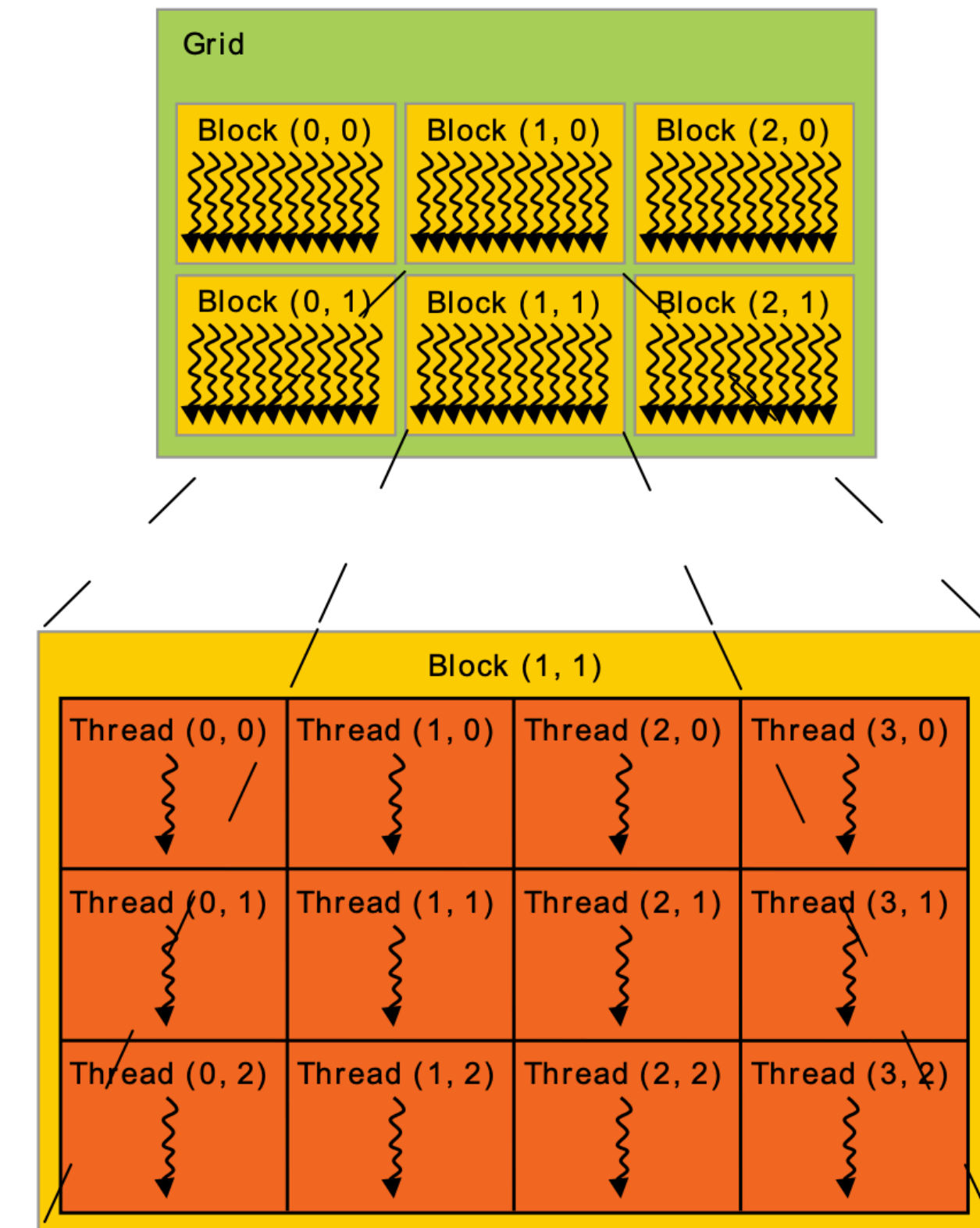
Simple CUDA programming example

```
int main(int argc, char **argv) {  
    int vectorSize = 50000;  
    size_t size = vectorSize * sizeof(float);  
  
    // Allocate the host memory for A, B, C  
    float *h_A = (float *)malloc(size);  
    float *h_B = (float *)malloc(size);  
    float *h_C = (float *)malloc(size);  
  
    // Initialize the host input vectors A and B  
    for (int i = 0; i < vectorSize; ++i) {  
        h_A[i] = rand() / (float)RAND_MAX;  
        h_B[i] = rand() / (float)RAND_MAX;  
    }  
  
    // Allocate the device memory for A, B, C  
    float *d_A = NULL;  
    cudaMalloc((void **)&d_A, size);  
    float *d_B = NULL;  
    cudaMalloc((void **)&d_B, size);  
    float *d_C = NULL;  
    cudaMalloc((void **)&d_C, size);  
  
    // Copy the host input vectors A and B into device memory  
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);  
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);  
}
```

```
// Launch the CUDA Kernel  
int threadsPerBlock = 256;  
int blocksPerGrid = (vectorSize + threadsPerBlock - 1) / threadsPerBlock;  
vectorAdd <<<blocksPerGrid, threadsPerBlock>>> (d_C, d_A, d_B, vectorSize);  
  
// Copy the device result vector back to the host result vector  
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);  
  
// Free the device and host memory  
cudaFree(d_A);  
cudaFree(d_B);  
cudaFree(d_C);  
free(h_A);  
free(h_B);  
free(h_C);  
  
return 0;  
}
```

```
/**  
 * CUDA Kernel Device code, C = A + B  
 */  
__global__ void vectorAdd(float *C, const float *A, const float *B,  
    int vectorSize) {  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if (i < vectorSize) {  
        C[i] = A[i] + B[i];  
    }  
}
```

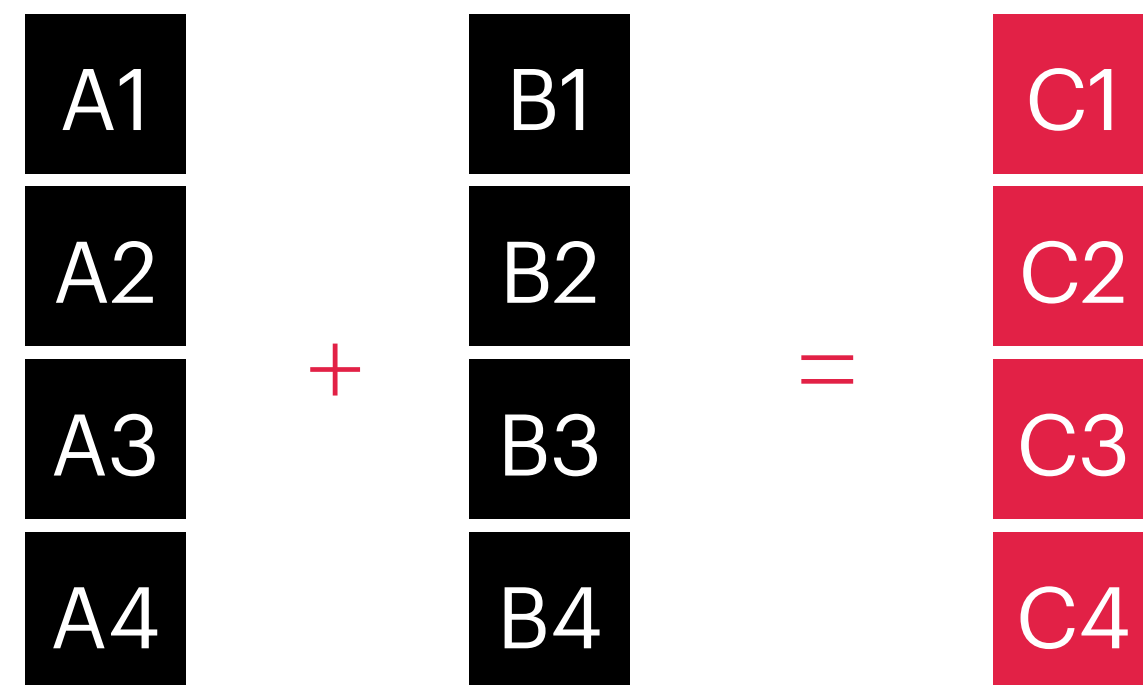
Thread Hierarchy



Parallel programming models

SIMD (Single Instruction Multiple Data)

- Vectorization: supported by x86 (SSE, AVX, AVX2, AVX512 etc.), Arm (NEON, SVE), PowerPC (AltiVec) etc.
- Implementation: optimized math libraries (such as Intel MKL), inline assembly, intrinsic function



Parallel programming models

SIMD with intrinsic functions

No explicit SIMD

```
void add(float* out, const float* input1, const float* input2, int N)
{
    for(int i=0; i<N; i++){
        out[i] = input1[i] + input2[i];
    }
}
```

x86 AVX SIMD

```
#include<immintrin.h>
//compile: g++ -O3 -mavx -o exe src.c

void add_avx(float* out, const float* input1,
             const float* input2, int N)
{
    for(int i=0; i<N; i+=8){
        __m256 v1 = _mm256_load_ps(input1+i);
        __m256 v2 = _mm256_load_ps(input2+i);

        __m256 v0 = _mm256_add_ps(v1, v2);
        _mm256_store_ps(out+i, v0);
    }
}
```

ARM NEON SIMD

```
#include<arm_neon.h>
//compile: g++ -O3 -march=armv8-a -o exe src.c

void add_neon(float* out, const float* input1,
              const float* input2, int N)
{
    for(int i=0; i<N; i+=4){
        float32x4_t v1 = vld1q_f32(input1+i);
        float32x4_t v2 = vld1q_f32(input2+i);

        float32x4_t v0 = vaddq_f32(v1, v2);
        vst1q_f32(out+i, v0);
    }
}
```


Parallel programming models

Software build tools

```
1 CC = gcc
2 CFLAGS = -O3 -fopenmp
3
4 objects = hello_world.o
5 all: hello_world
6
7 %.o : %.c
8     $(CC) -c $(CFLAGS) $< -o $@
9
10 hello_world: $(objects)
11     $(CC) $(CFLAGS) $^ -o $@
12
13 .PHONY: all
14 clean:
15     rm -f *.o hello_world
```

Makefile

Build: **make**

acinclude.m4	lib
AUTHORS	LICENSE
autogen.sh	mainprogs
ChangeLog	Makefile.am
chroma-config.in	metadata.yml
config	NEWS
configure.ac	other_libs
COPYING	README
docs	scripts
INSTALL	tests

GNU Autotools

Build: **autoreconf**
./configure
make && make install


cmake	lib
CMakeLists.txt	LICENSE
doc	NEWS
externals	README.md
include	tests
jenkins	

CMake

Build: **mkdir build && cd build**
cmake ..
make && make install

Parallel programming models

Version control and collaborative development



IHEP-LQCD Collaboration

Follow

Overview

Repositories12

Projects

Packages

Teams

People9

Settings

Popular repositories

utility_tools

Public

Useful tools for data analysis and visualization

Python

2

2

mg_proto

Public

Forked from JeffersonLab/mg_proto

C++

PyQuda

Public

Python wrapper for quda written in Cython.

Cython

.github

Public

Example Template for GitHub Actions

chroma_addons

Public

mirror of https://code.ihep.ac.cn/ihep-lqcd/chroma_addons

C++








View as: Public



You are viewing the README and pinned repositories as a public user.

You can [create a README file](#) or [pin repositories](#) visible to anyone.

[Get started with tasks](#) that most successful organizations complete.

People





Invite someone

Repositories

Find a repository...

Type

Language

Sort

New

Top languages

Python

C++

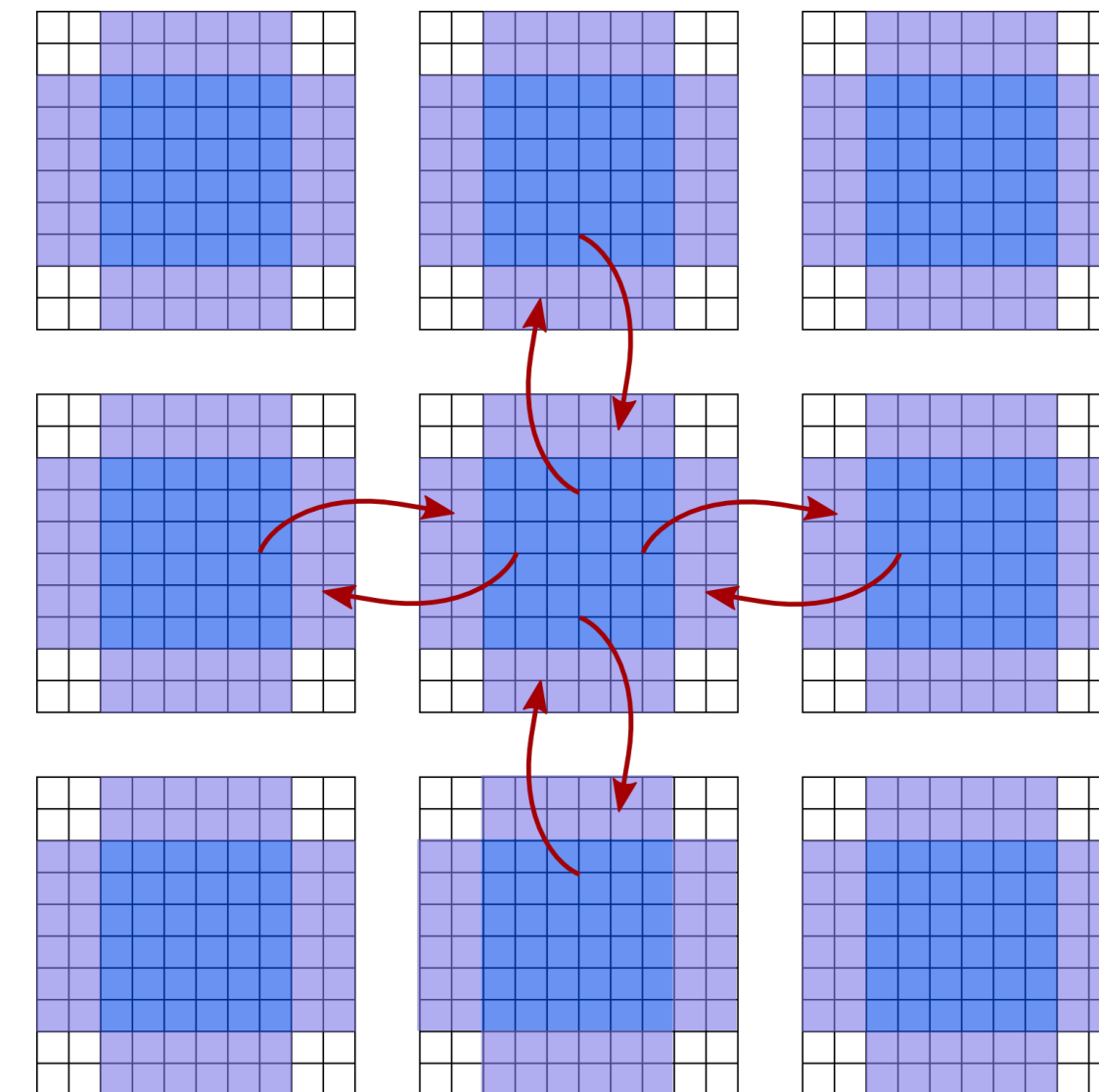
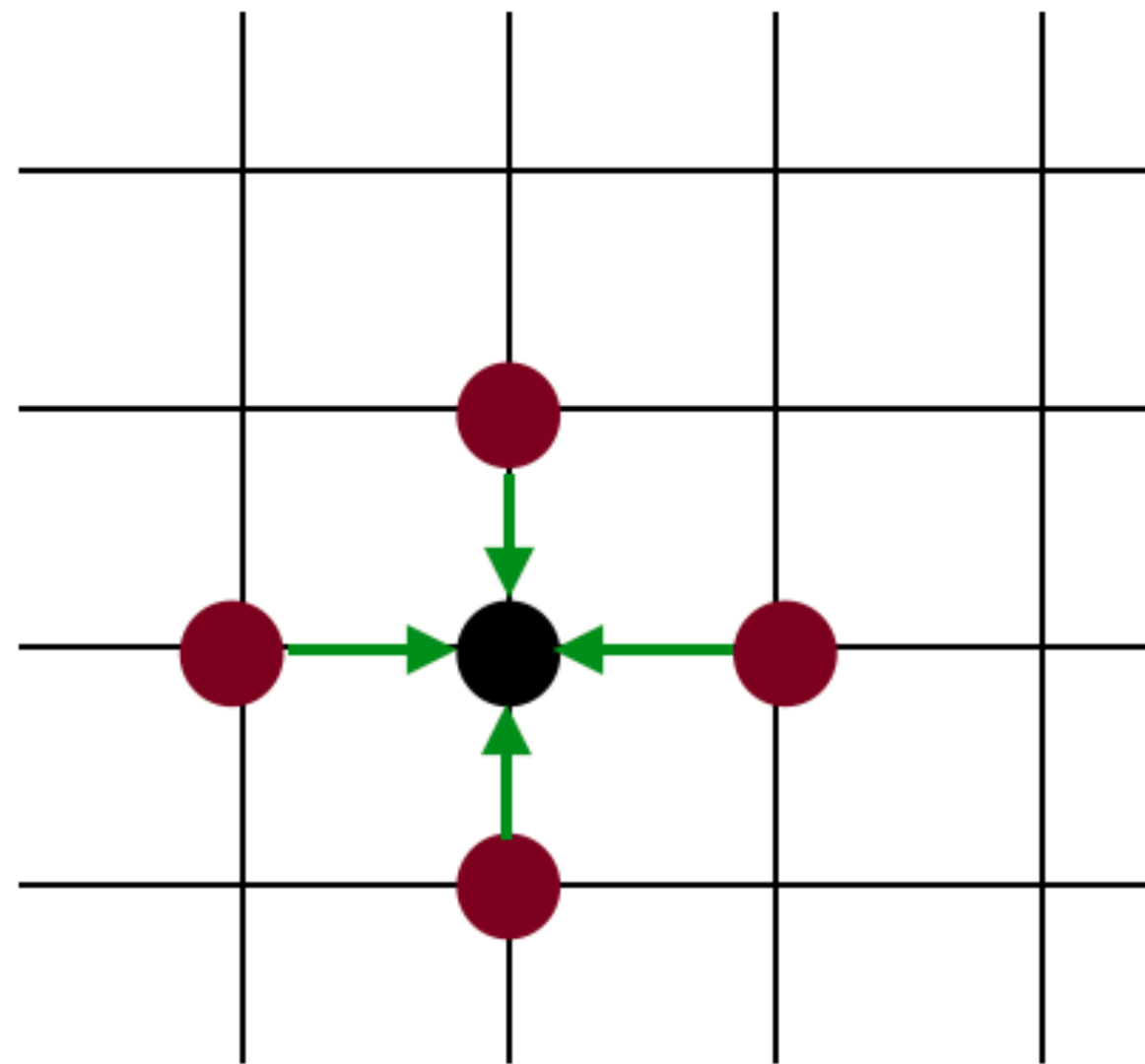
Scheme

Fortran

Cython

Case study with HPC in high energy physics

- Dirac equation on the lattice is to solve the large sparse linear system $Mx = b$
- $M = m + 4 - \frac{1}{2}D$, D is the Dslash operation
- $$D_{x,y} = \sum_{\mu=1}^4 U_{\mu}(x)(1 - \gamma_{\mu})\delta_{x+\hat{\mu},y} + U_{\mu}(x - \hat{\mu})^{\dagger}(1 + \gamma_{\mu})\delta_{x-\hat{\mu},y}$$
- 4 dimensional 8(9) point stencil operator
- Nearest neighbor communication, very suitable for parallel computing



Case study with HPC in high energy physics

Solve $Mx = b$ with iterative method

Algorithm 1 Solve $Mx = b$ with BiCGStab method

1: $r_0 = b - Mx_0$ $\longrightarrow x_0$ is initial guess

2: $p_0 = r_0$

3: **while** $\|r_j\| > \epsilon$ **do** $\longrightarrow \epsilon$ is the convergence tolerance error

4: $\alpha_j = \frac{(r_j, r_0^*)}{(Mp_j, r_0^*)}$

5: $s_j = r_j - \alpha_j Mp_j$

6: $\omega_j = \frac{(Ms_j, s_j)}{(Ms_j, Ms_j)}$

7: $x_{j+1} = x_j + \alpha_j p_j + \omega_j s_j$

8: $r_{j+1} = s_j - \omega_j Ms_j$

9: $\beta_j = \frac{\alpha_j}{\omega_j} \times \frac{(r_{j+1}, r_0^*)}{(r_j, r_0^*)}$

10: $p_{j+1} = r_{j+1} + \beta_j(p_j - \omega_j Mp_j)$

11: **end while**

- Matrix vector multiplications Mp_j etc. are nearest neighbor commutation (MPI_Send/Recv)
- (r_j, r_0^*) etc. are complex inner product (MPI_Allreduce)

Summary and tips

- Covered basics of high performance computing programming model and tools widely used in high energy physics
- Tips:
 - Select the right programming model and tools before writing the code
 - Correctness is the top priority, NOT performance at the beginning of the software development
 - Use well established and tested libraries, do NOT reinvent the wheels unless you know what you are doing
 - Use version control system such as git for code development, use github or gitlab for collaborative development