

WSIZ, 2023  
Computer systems architecture

# **Lecture 14. Architecture and programming technology of supercomputers**

Dmitry Zaitsev

<http://daze.ho.ua>

# High Performance Computing

- High Performance Computing most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business.

# Supercomputer

- A particularly powerful (mainframe) computer
- A supercomputer is a computer that performs at or near the highest operational rate for computers
- Requires special – massively parallel – programming technology to utilize performance

# FLOPS as measure of computing power

- **Floating Point Operations Per Second**
- **Teraflops  $10^{12}$**
- **Petaflops  $10^{15}$**
- **Exaflops  $10^{18}$  – Frontier, Oak Ridge, USA**
- **Zettaflops  $10^{21}$**

# Typical applications of supercomputers

- weather forecasting to predict the impact of extreme storms and floods;
- oil and gas exploration to collect huge quantities of geophysical seismic data to aid in finding and developing oil reserves;
- molecular modeling for calculating and analyzing the structures and properties of chemical compounds and crystals;
- physical simulations like modeling supernovas and the birth of the universe;
- aerodynamics such as designing a car with the lowest air drag coefficient;
- nuclear fusion research to build a nuclear fusion reactor that derives energy from plasma reactions;
- medical research to develop new cancer drugs, understand the genetic factors that contribute to opioid addiction and find treatments for COVID-19;
- next-gen materials identification to find new materials for manufacturing; and
- cryptanalysis to analyze cyphertext, ciphers and cryptosystems to understand how they work and identify ways of defeating them.

# top500.org

- The TOP500 project was launched in 1993
- Uses a benchmark to rank systems and to decide on whether or not they qualify for the TOP500 list
- LINPACK Benchmark – systems are ranked by their ability to solve a set of linear equations,  $Ax = b$ , using a dense random matrix  $A$ .

# LINPACK

- <https://www.netlib.org/linpack/>
- LINPACK is a collection of Fortran subroutines that analyze and solve linear equations and linear least-squares problems.
- The package solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square.
- In addition, the package computes the QR and singular value decompositions of rectangular matrices and applies them to least-squares problems.

# HPL

- <https://netlib.org/benchmark/hpl/>
- A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers
- Solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers



# Top 3

| Rank | System                                                                                                                                                                                      | Cores     | Rmax<br>(PFlop/s) | Rpeak<br>(PFlop/s) |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|-------------------|--------------------|
| 1    | <a href="#"><u>Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory</u></a><br>United States | 8,699,904 | 1,194.00          | 1,679.82           |
| 2    | <a href="#"><u>Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science</u></a><br>Japan                          | 7,630,848 | 442.01            | 537.21             |
| 3    | <a href="#"><u>LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC</u></a><br>Finland                                    | 2,220,288 | 309.10            | 428.70             |

# Efficiency of modern supercomputers



2021 ACM Turing Award Recipient Jack Dongarra

Turing Lecture: "A Not So Simple Matter..."



Watch later Share

## HPCG Top 10, November 2022

Press Esc to exit full screen

| Rank | Site                                   | Computer                                                                                                                 | Country      | Cores     | Rmax<br>[Pflop/s] | Top 500<br>Rank | HPCG<br>[Pflop/s] | % of<br>Peak |
|------|----------------------------------------|--------------------------------------------------------------------------------------------------------------------------|--------------|-----------|-------------------|-----------------|-------------------|--------------|
| 1    | RIKEN Center for Computational Science | Fugaku, Fujitsu A64FX 48C 2.2 GHz, Tofu D, Fujitsu                                                                       | Japan        | 7,630,848 | 422               | 2               | 16.0              | 3.0%         |
| 2    | DOE / SC / ORNL                        | Frontier, HPE Cray Ex235a, AMD 3rd EPYC 64C 2 GHz, AMD Instinct MI250X, Slingshot 10                                     | USA          | 8,730,112 | 1,102             | 1               | 14.1              | 0.8%         |
| 3    | EuroHPC / CSC                          | LUMI, HPE Cray EX235a, AMD Zen 3 (Milan) 64C 2GHz, AMD MI250X, Slingshot-11                                              | Finland      | 2,174,976 | 304               | 3               | 3.41              | 0.8%         |
| 4    | DOE / SC / ORNL                        | Summit, AC922, IBM POWER9 22C 3.7 GHz, Dual-rail Mellanox FDR, NVIDIA Volta V100, IBM                                    | USA          | 2,414,592 | 149               | 5               | 2.93              | 1.5%         |
| 5    | EuroHPC / CINECA                       | Leonardo, BullSequana XH2000, Xeon Platinum 8358 32C 2.6 GHz, NVIDIA A100 SXM4 40 GB, Quad-rail NVIDIA HDR100 InfiniBand | Italy        | 1,463,616 | 175               | 4               | 2.57              | 1.0%         |
| 6    | DOE / SC / LBNL                        | Perlmutter, HPE Cray EX235n, AMD EPYC 7763 64C 2.45 GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10                            | USA          | 761,856   | 70.9              | 8               | 1.91              | 2.0%         |
| 7    | DOE / NNSA / LLNL                      | Sierra, S922LC, IBM POWER9 20C 3.1 GHz, Mellanox EDR, NVIDIA Volta V100, IBM                                             | USA          | 1,572,480 | 94.6              | 6               | 1.80              | 1.4%         |
| 8    | NVIDIA                                 | Selene, DGX SuperPOD, AMD EPYC 7742 64C 2.25 GHz, Mellanox HDR, NVIDIA Ampere A100                                       | USA          | 555,520   | 63.5              | 9               | 1.62              | 2.0%         |
| 9    | Forschungszentrum Juelich (FZJ)        | JUWELS Booster Module, Bull Sequana XH2000, AMD EPYC 7402 24C 2.8 GHz, Mellanox HDR InfiniBand, NVIDIA Ampere A100, Atos | Germany      | 449,280   | 44.1              | 12              | 1.28              | 1.8%         |
| 10   | Saudi Aramco                           | Dammam-7, Cray CS-Storm, Xeon Gold 6252 20C 2.5 GHz, InfiniBand HDR 100, NVIDIA Volta V100, HPE                          | Saudi Arabia | 672,520   | 22.4              | 20              | 0.88              | 1.6%         |

MORE VIDEOS



52:17 / 1:09:27



YouTube



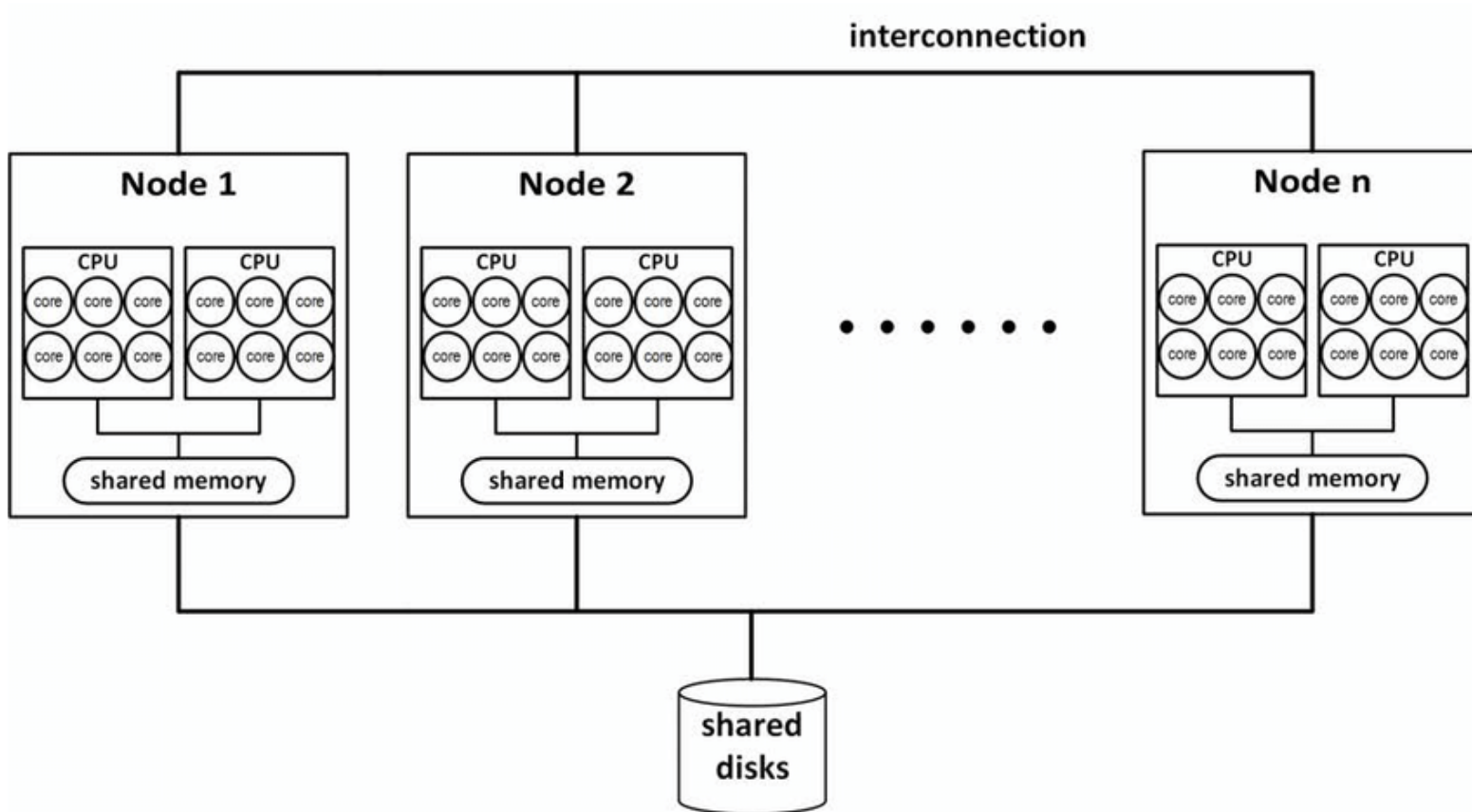
# HPCG Benchmark

- <https://hpcg-benchmark.org/>
- High Performance Conjugate Gradients
- HPCG: Sparse matrix-vector multiplication; Vector updates; Global dot products; Local symmetric Gauss-Seidel smoother; Sparse triangular solve (as part of the Gauss-Seidel smoother); Driven by multigrid preconditioned conjugate gradient algorithm that exercises the key kernels on a nested set of coarse grids
- Reference implementation is written in C++ with MPI and OpenMP support.

# Generalized HPC architecture

- High performance low latency interconnect
- Computing node:
  - multicore processor(s)
  - shared memory
  - GPU(s)

# HPC architecture scheme



# Fugaku (Fujitsu, RIKEN)





# Processor A64FX

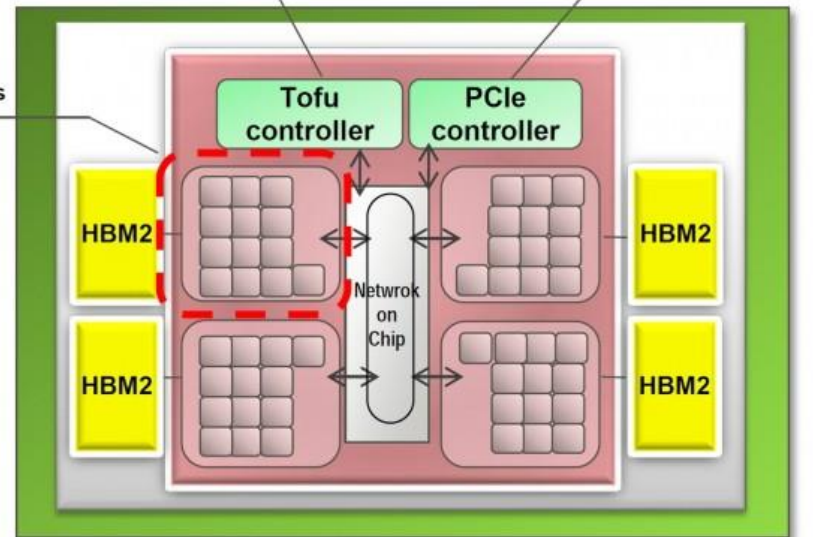


CMG specification  
13 cores  
L2\$ 8MiB  
Mem 8GiB, 256GB/s

<A64FX>

Tofu  
28Gbps 2 lanes 10 ports

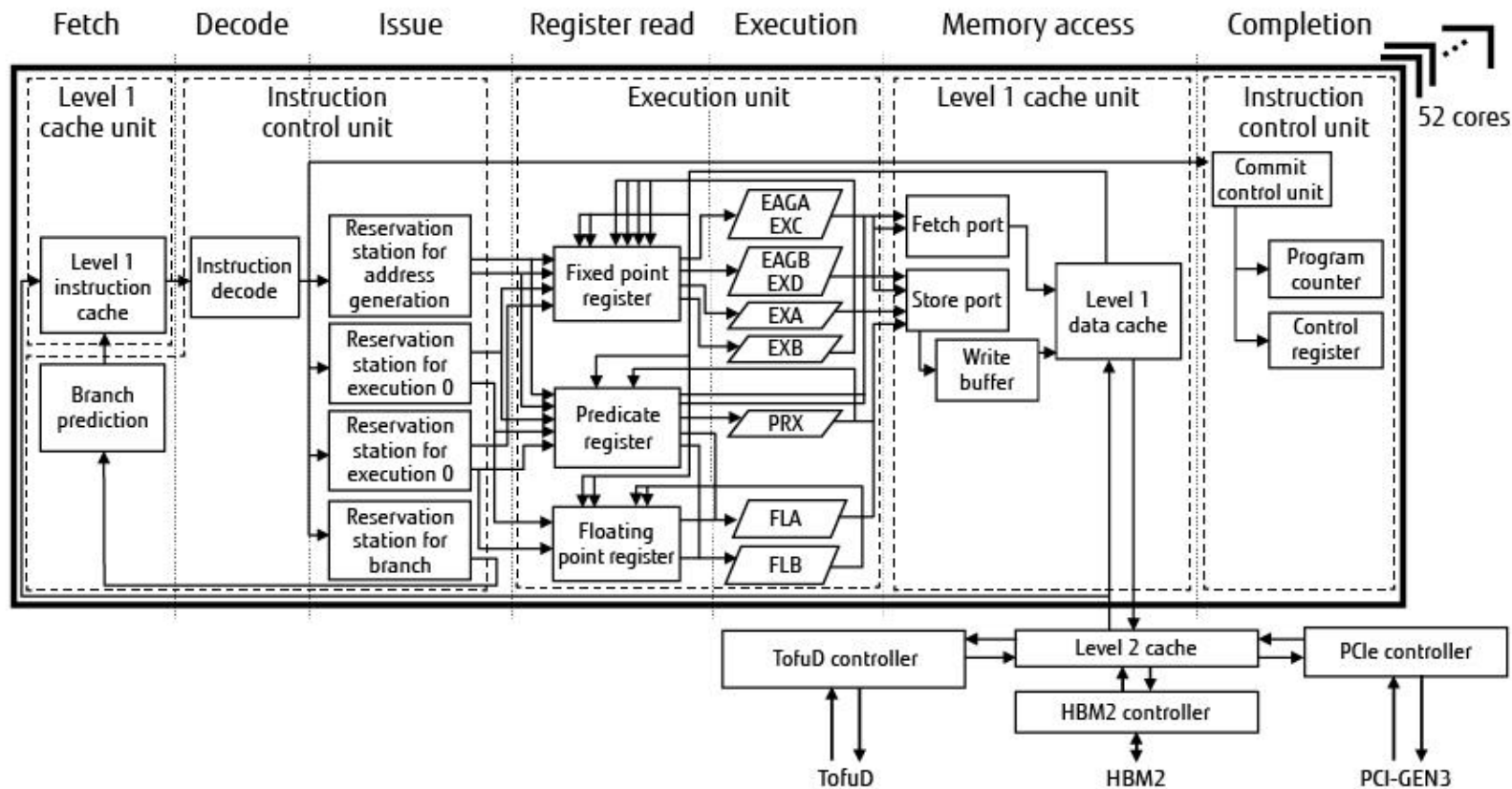
I/O  
PCIe Gen3 16 lanes



A64FX® Microarchitecture Manual

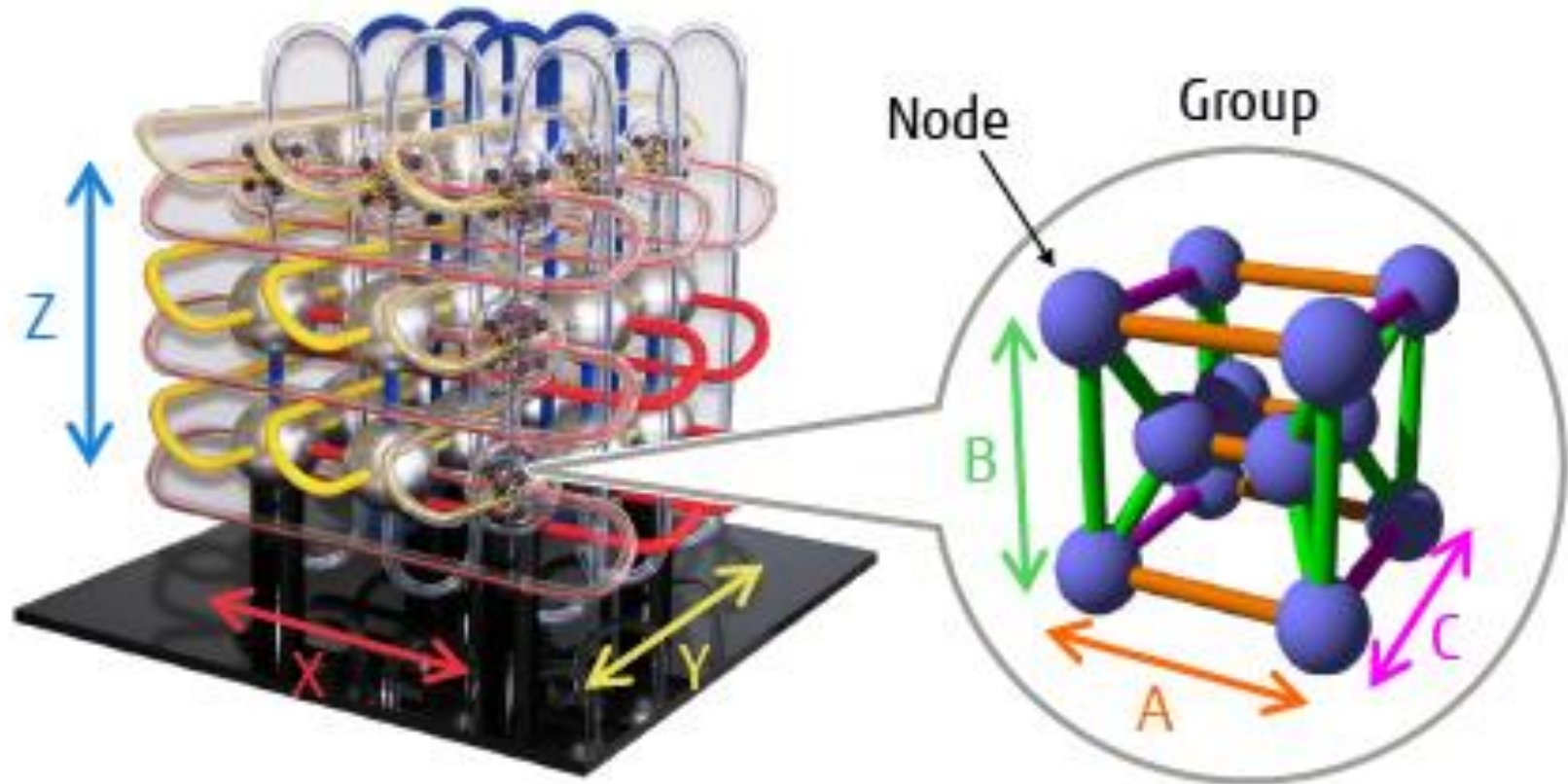
<http://github.com/fujitsu/A64FX>

# A64FX detailed architecture

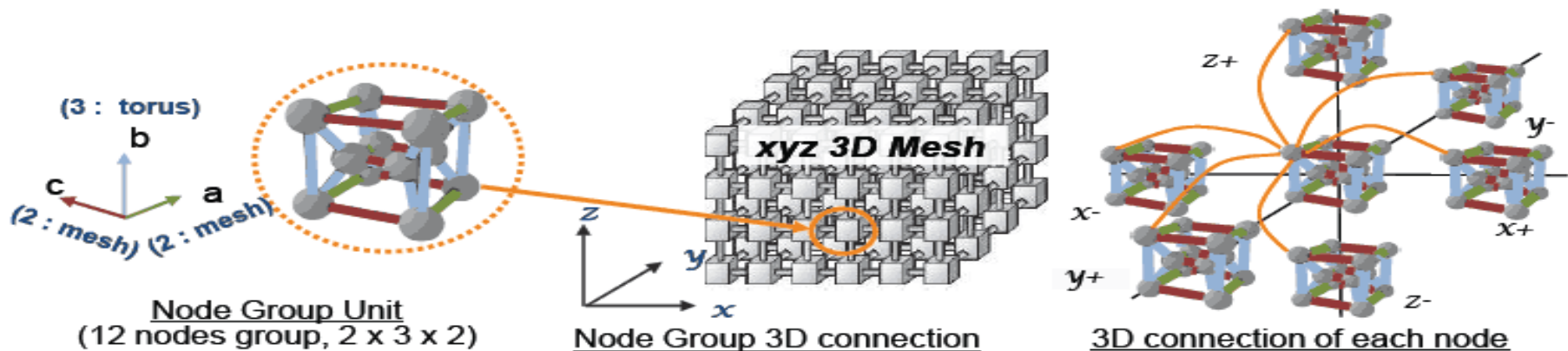




# Tofu Interconnect D

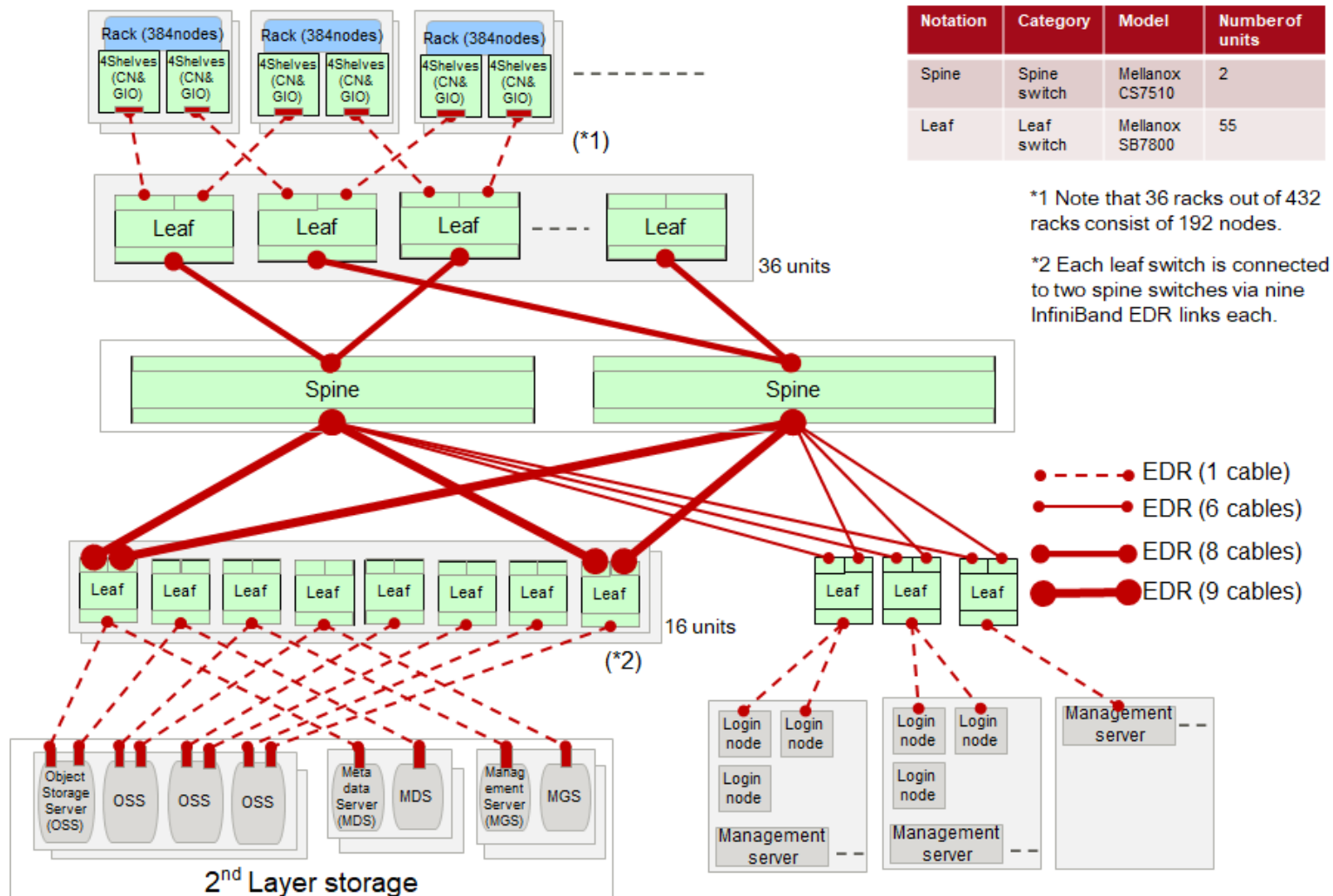


# Tofu: Fujitsu's original 6D mesh/torus interconnect



Tofu has a six-dimensional [mesh/torus topology](#), a scalability of over 100,000 nodes, and [full-duplex](#) links that have a peak bandwidth of 10 GB/s (5 GB/s per direction)

# Fugaku composition



# HPC programming technology

- OpenMP – uses multiply cores of multicore processors
  - MPI – computing on distributed nodes
  - CUDA/OpenCL – programming on GPUs
- 

SLURM – open source, fault-tolerant, and highly scalable cluster management and job scheduling system

# Open Multi-Processing (OpenMP) architecture



# Parallel loop – the best case – independent passages

```
#pragma omp parallel for  
for(i=0;i<n;i++)  
{  
    c[i] = a[i] + b[i];  
}
```

# Sum of vectors, example $z=x+y$

| Vector\Index | 0  | 1 | 2  | 3 | 4  | 5  |
|--------------|----|---|----|---|----|----|
| x            | -2 | 1 | -7 | 0 | -1 | 4  |
| y            | 5  | 1 | -2 | 1 | 1  | -6 |
| z            | 3  | 2 | -9 | 1 | 0  | -2 |

We can sum up all elements in parallel having the number of threads equal to the length of array!

# An example for $n=6$ , $th=3$

- Thread 1:

$c[0]=a[0]+b[0]$

$c[3]=a[3]+b[3]$

- Thread 2:

$c[1]=a[1]+b[0]$

$c[4]=a[4]+b[4]$

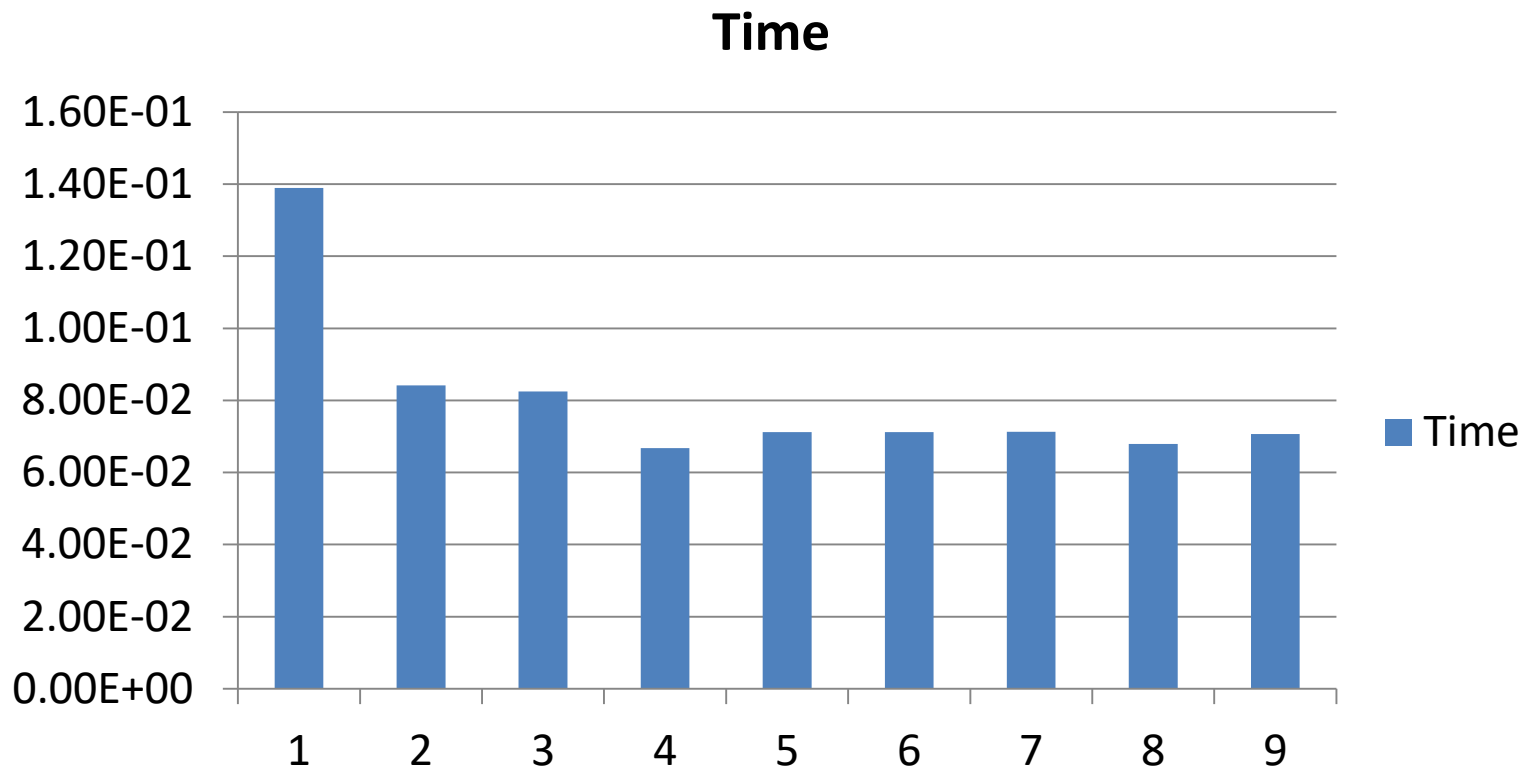
- Thread 3:

$c[2]=a[2]+b[2]$

$c[5]=a[5]+b[5]$



# Benchmarks time diagram



# Graphical Processing Unit (GPU)

- Mass parallel processing device – thousands of cores (threads)
- Reduced set of commands
- Own memory
- Initially adjusted for processing graphical information in real time
- Lately applied as general purpose mass parallel computer similar to FPGA

# NVIDIA GeForce RTX 3090



**GRAPHICS PROCESSOR GA102**

**CORES 10496**

**TMUS 328**

**ROPS 112**

**MEMORY SIZE 24 GB**

**MEMORY TYPE GDDR6X**

**BUS WIDTH**  
**384 bit**

# CUDA architecture

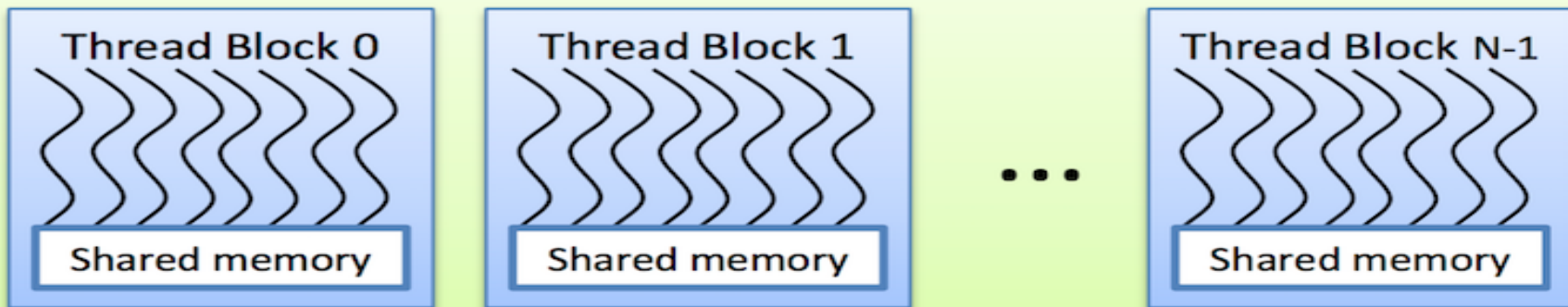
- **Host** – controls computations and code/data exchange (CPU – for instance i7/i9)
- **Device** – executes fast definite reduced set of instructions over big volume of data (GPU – for instance GeForce RTX 3090)
- **Kernel** – a program that is executed on device

# GPU logical structure

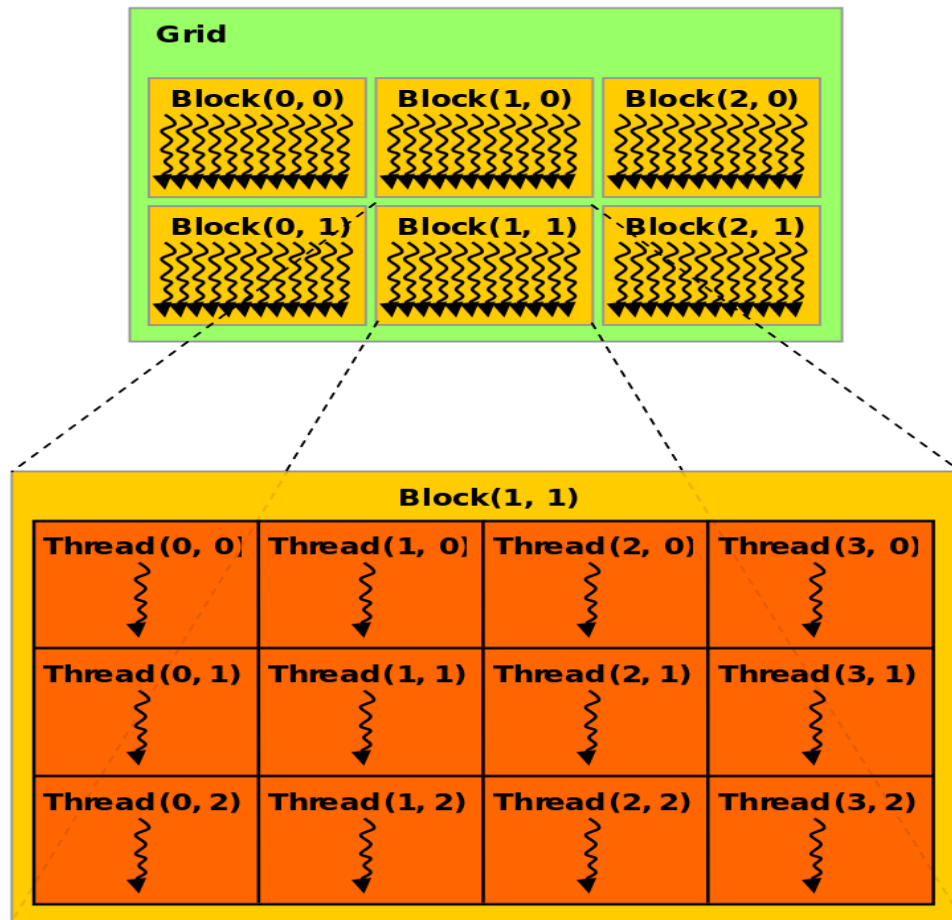
- **Grid** – 1, 2, or 3 dimensional array of blocks
  - **Block** – 1, 2, or 3 dimensional array of threads
  - **Thread** – a single process of computations
- 
- Warp – a bunch of threads executed on the same Multi Processor

# Abstract structure of computations: 1D grid, 1D block

## Grid



Abstract  
structure of  
computations:  
2D grid,  
2D block



**Kernel to compute sum of vectors  
(grid consists of one 1D block)**

```
__global__ void sumArraysOnGPU  
(float *A, float *B, float *C, const int N)  
{  
    int i = threadIdx.x;  
  
    if (i < N) C[i] = A[i] + B[i];  
}
```



# How it works

- The number of array elements is less or equal to the number of threads
- No loop at all
- All threads started at once
- A thread computes one element
- Some threads may be

# Copy data from host to device

```
// initialize device
```

```
cudaSetDevice(dev);
```

```
// allocate device global memory
```

```
float *d_A, *d_B, *d_C;
```

```
cudaMalloc((float**)&d_A, nBytes);
```

```
cudaMalloc((float**)&d_B, nBytes);
```

```
cudaMalloc((float**)&d_C, nBytes);
```

```
// transfer data from host to device
```

```
cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_C, gpuRef, nBytes, cudaMemcpyHostToDevice);
```

# Launch computations on device and copy obtained results to host

```
// specify block and grid  
dim3 block (nElem);  
dim3 grid (1);
```

```
// start computations on device  
sumArraysOnGPU<<<grid, block>>>(d_A, d_B, d_C, nElem);
```

```
// copy results to host  
cudaMemcpy(gpuRes, d_C, nBytes, cudaMemcpyDeviceToHost));
```

# Examples of indexing

- 1D grid of 1D blocks:

$i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$

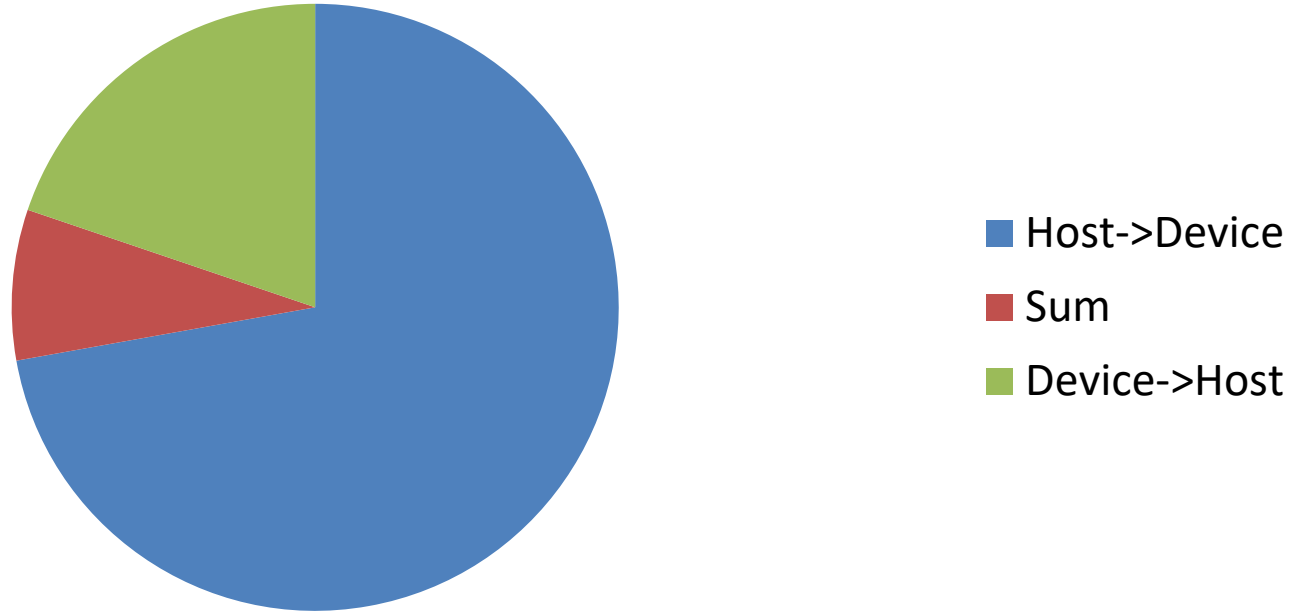
- 2D grid of 2D blocks:

$\text{blockId} = \text{blockIdx.x} + \text{blockIdx.y} * \text{gridDim.x};$

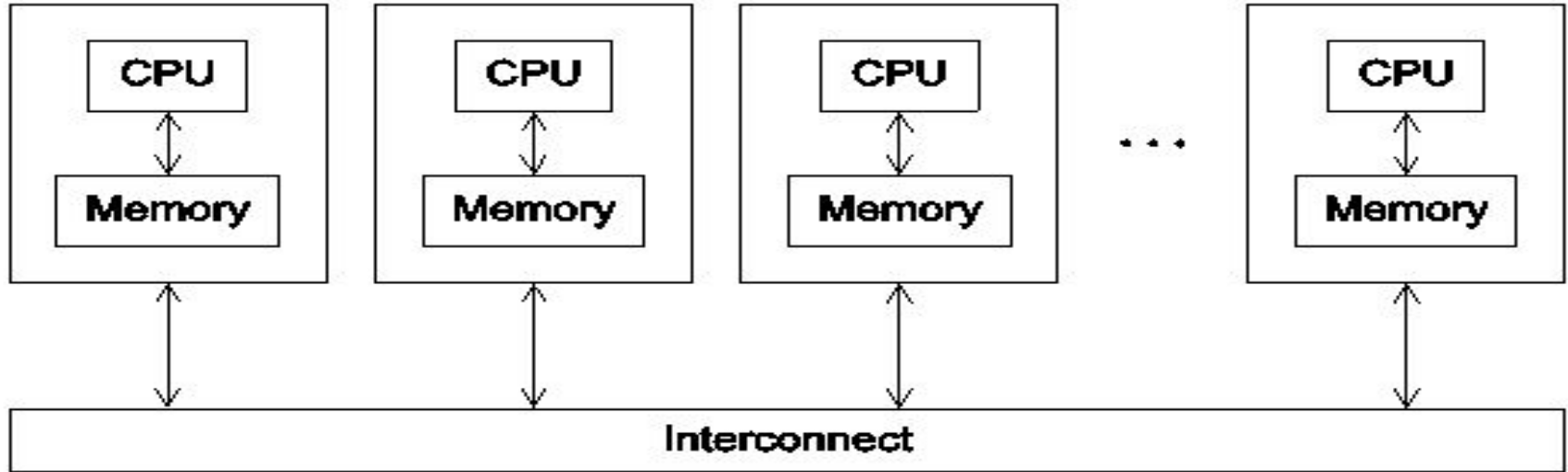
$i = \text{blockId} * (\text{blockDim.x} * \text{blockDim.y}) + (\text{threadIdx.y} * \text{blockDim.x}) + \text{threadIdx.x};$

# Diagram of benchmarks

Time



# Message Passing Interface (MPI) architecture



# Implementation of MPI

- Standards: <https://www.mpi-forum.org/>
- MPICH: <https://www.mpich.org/>
- Headers: `#include <mpi.h>`
- Functions: `MPI_Init(&argc, &argv);`
- Constants: `MPI_COMM_WORLD`
- Compilation: `mpicc -o prog prog.c`
- Launch: `mpirun -n 5 ./prog`

# MPI essentials

- Single program, multiple data (SPMD)
- Scalability: processes run in same way on a single computer and on a cluster
- During execution processes synchronize and exchange by data
- Synchronization and data exchange decrease performance



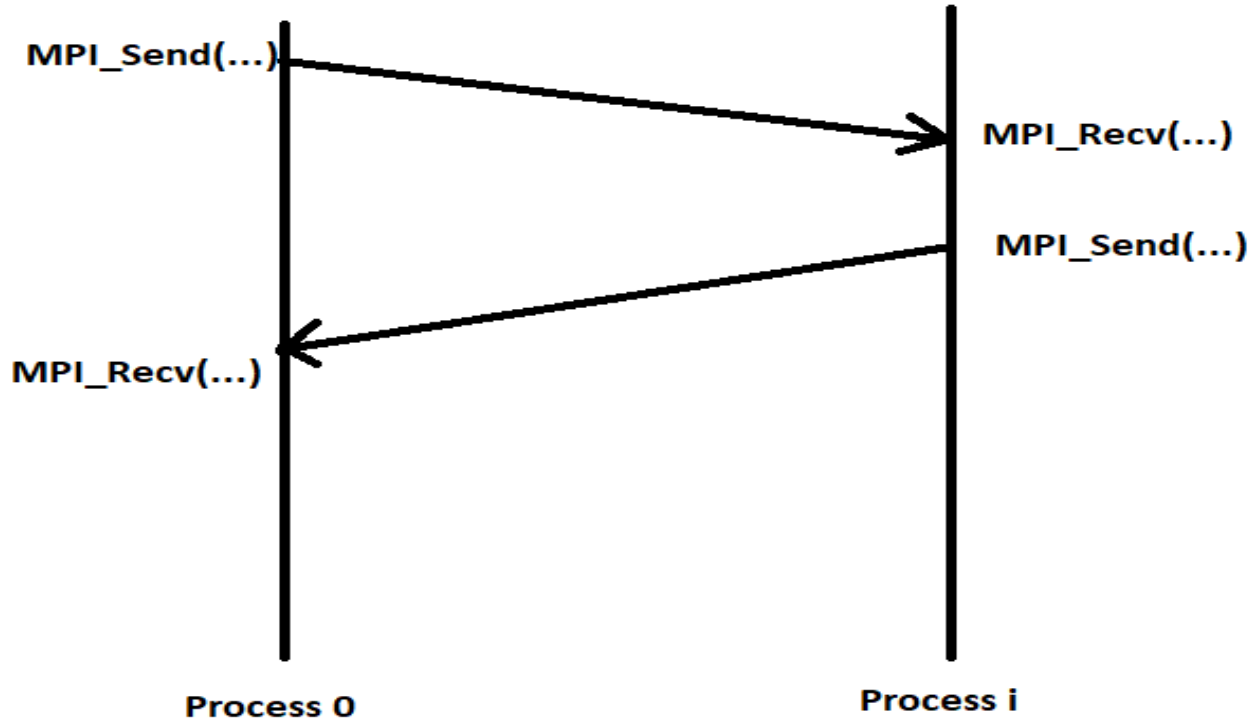
# Send message

- `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `buf` адрес буфера
- `count` количество элементов в буфере
- `datatype` тип данных элемента
- `dest` номер процесса назначения
- `tag` номер типа сообщения
- `comm` номер коммутатора
- Пример: `MPI_Send(&b, 1, MPI_FLOAT, 1, 5, MPI_COMM_WORLD);`

# Receive message

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status * status)`
- Типы данных: `MPI_CHAR`, `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`
- status: `MPI_Source`, `MPI_Tag`, *`MPI_Error`*
- `MPI_ANY_SOURCE`, `MPI_ANY_TAG`
- Код возврата: `MPI_SUCCESS`
- Пример: `MPI_Recv(&b, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD, &status);`

# Typical message exchange



# Numerical example, $n=17$ , $p=5$

- Simple division in parts, lazy master  
0, 5, 5, 5, 2
- Simple division in parts, laborious master  
4, 4, 4, 4, 1
- Fine balancing (laborious master)  
4, 4, 3, 3, 3

# Master delivers jobs

```
for(j=0;j<n;j++) { // init arrays
    a[j]= (float)( rand() & 0xFF ) / 10.0f;
    b[j]= (float)( rand() & 0xFF ) / 10.0f;
}
t1=magma_wtime();
k=n/size; extra=n%size; beg=0;
for(i=1;i<size;i++) { // send jobs
    kk=(extra>0)?k+1:k; extra--;
    rc = MPI_Send(&kk, 1, MPI_INT, i, 10, MPI_COMM_WORLD);
    rc = MPI_Send(a+beg, kk, MPI_FLOAT, i, 11, MPI_COMM_WORLD);
    rc = MPI_Send(b+beg, kk, MPI_FLOAT, i, 12, MPI_COMM_WORLD);
    beg+=kk;
}
```

# Master collects results

```
for(j=0;j<k;j++) // master part of job
    c[beg+j]=a[beg+j]+b[beg+j];

beg=0;
for(i=1;i<size;i++) { // receive results
    rc = MPI_Recv(&kk, 1, MPI_INT, i, 13, MPI_COMM_WORLD, &stat);
    rc = MPI_Recv(c+beg, kk, MPI_FLOAT, i, 14,
                  MPI_COMM_WORLD, &stat);
    beg+=kk;
}

t2=magma_wtime(); printf("time = %le s.\n", t2-t1);
```

# Worker

```
// receive actual length
rc = MPI_Recv(&kk, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &stat);

// receive arrays
rc = MPI_Recv(a1, kk, MPI_FLOAT, 0, 11, MPI_COMM_WORLD, &stat);
rc = MPI_Recv(b1, kk, MPI_FLOAT, 0, 12, MPI_COMM_WORLD, &stat);

// sum arrays
for(j=0;j<kk;j++)
    c1[j]=a1[j]+b1[j];

// send resulting vector
rc = MPI_Send(&kk, 1, MPI_INT, 0, 13, MPI_COMM_WORLD);
rc = MPI_Send(c1, kk, MPI_FLOAT, 0, 14, MPI_COMM_WORLD);
```

# Actual benchmarks for MPI on a single computer

daze@hare:~/c\$ mpirun -n 1 ./mpi\_sv\_b  
time = 5.199909e-04 s.

daze@hare:~/c\$ mpirun -n 2 ./mpi\_sv\_b  
time = 7.140636e-04 s.

daze@hare:~/c\$ mpirun -n 3 ./mpi\_sv\_b  
time = 1.447916e-03 s.

daze@hare:~/c\$ mpirun -n 4 ./mpi\_sv\_b  
time = 9.531975e-04 s.

daze@hare:~/c\$ mpirun -n 10 ./mpi\_sv\_b  
time = 2.343178e-02 s.

daze@hare:~/c\$ mpirun -n 2 ./mpi\_sv\_b  
time = 6.828308e-04 s.

daze@hare:~/c\$ ./vsum 100000 1  
time = 2.930164e-04 s.

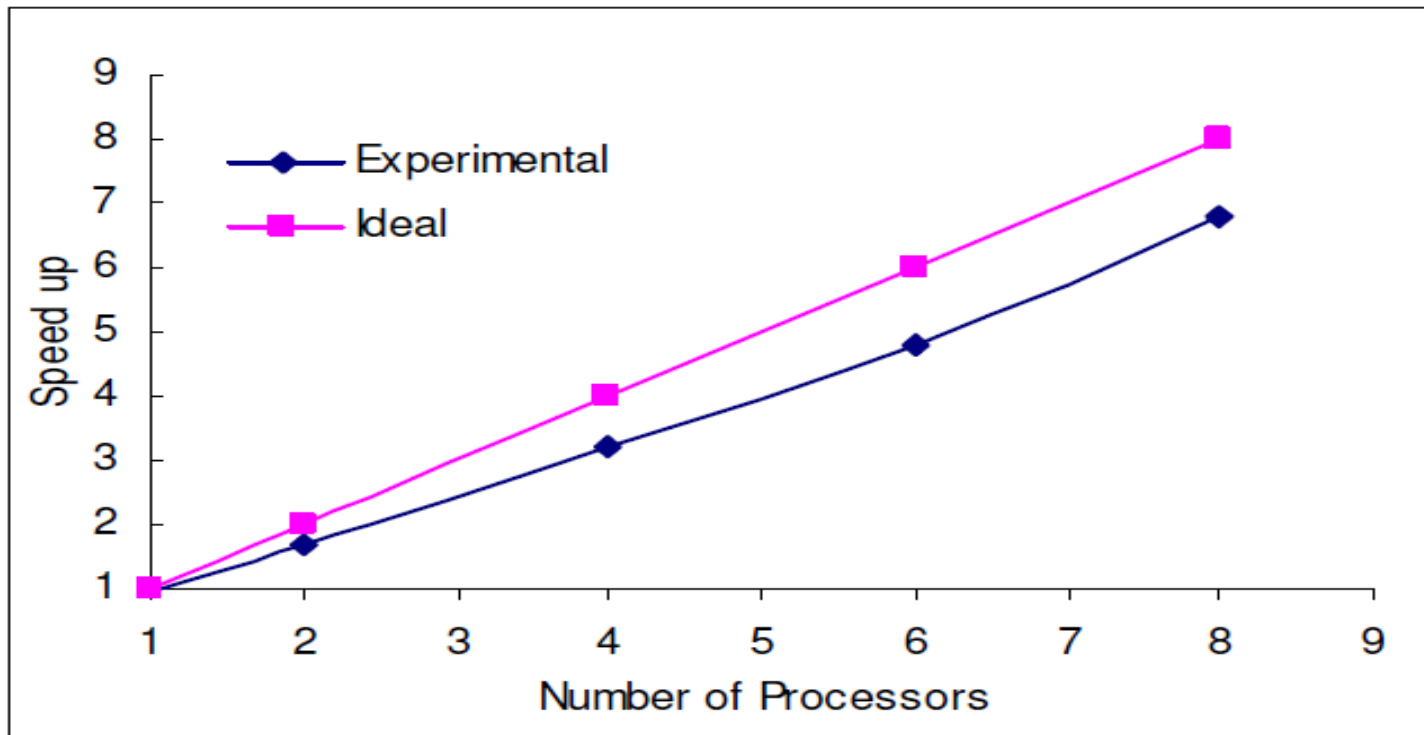
daze@hare:~/c\$ ./vsum 100000 2  
time = 2.379417e-04 s.

daze@hare:~/c\$ ./vsum 100000 3  
time = 2.219677e-04 s.

daze@hare:~/c\$ ./vsum 100000 4  
time = 2.198219e-04 s.



# Prime number generator benchmarks for MPI



# Disadvantages of traditional computer architecture

- Bottlenecks: memory-processor etc
- Heterogeneous: necessity of integral use of a set of technologies (OpenMP, CUDA, MPI)
- Explicit specification of control flow (parallel parts)