Introduction to Parallel Computing

Iain Miller, Lucian Anton

ECMWF

© ECMWF June 15, 2026

Overview

- What is Parallel Computing
- Building a Supercomputer
- Supercomputing Performance Aspects
- Parallel Programming Paradigms
- Scaling Limitations
- Future Challenges
- Further Reading

What is Parallel Computing

The simultaneous use of more than one processor or computer to solve a problem in a shorter time.

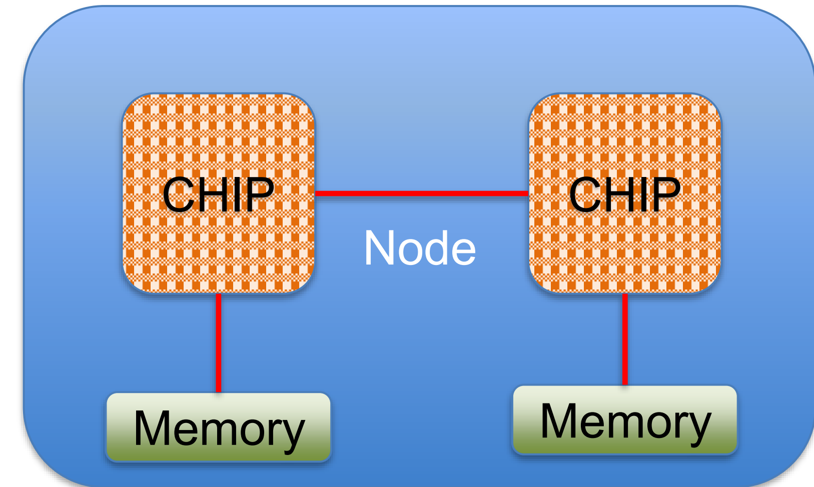
Why do we need Parallel Computing and how it can be achieved?

- Generally, it was found that models with more degrees of freedom and/or more data provides better solution for Science and Engineering numerical apps:
  - Finer grids for models described with PDEs
  - Larger ensembles for Monte Carlo type of models
  - Larger training sets and larger NN for machine learning models
- To cover the larger amount of work in an acceptable time on has to:
  - Find parallel algorithms for the problems at hand
  - Build hardware systems that can run in parallel the code that express the algorithms and handle the data
    - The route from algorithm to code requires programing languages and libraries that map the algorithmic parallelism to the hardware that works in parallel
    - Assembling serial processors in a working parallel systems requires specialized hardware, system software and system engineers to orchestrate the parallel data processing in an efficient and reliable way.

Building a Supercomputer

Supercomputer Building Blocks: Node

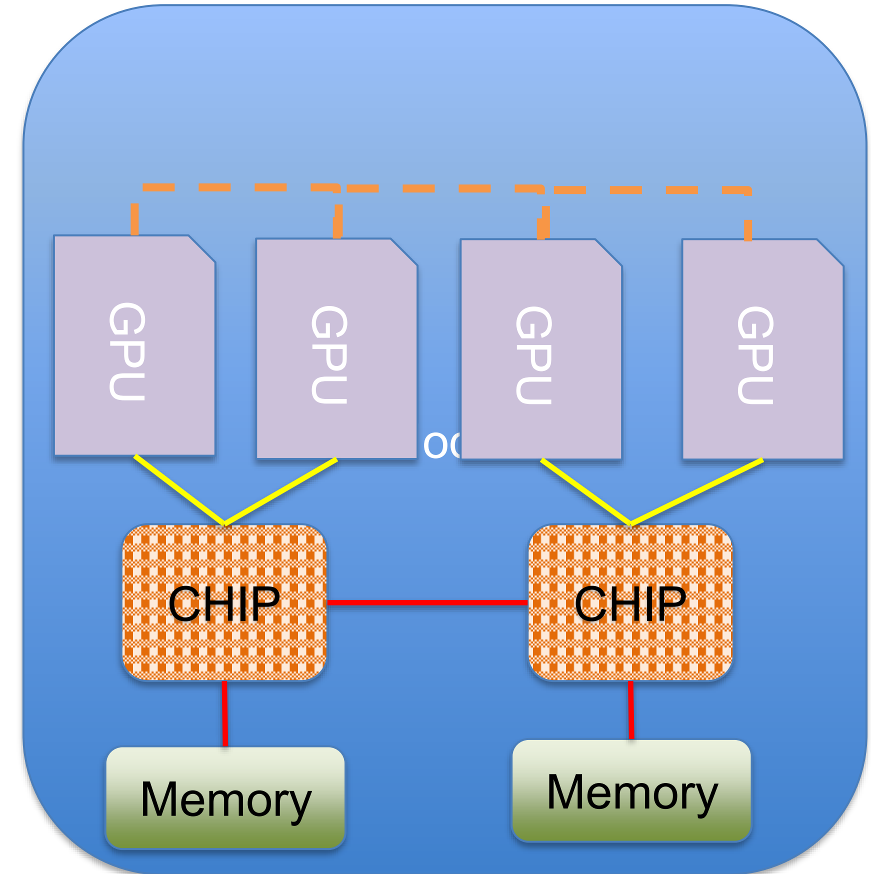
- Smallest building block is a node
  - Each node will have a number of sockets
  - Each socket will have a processor chip
  - Each processor chip will have a number of cores
  - Each core may or may not have a number of execution hardware threads
  - Each thread will have a vector width
- It is common for the lowest execution unit to be called a "Processing Element"



- Memory is attached in channels to each socket.
  - Slower access times than on chip memory (cache)
  - Usually accessible by all sockets
  - Will have variable access times depending on core location

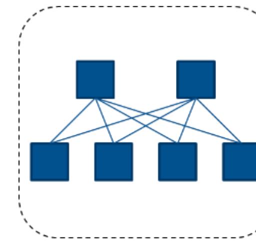
Supercomputer Building Blocks : Accelerators

- GPUs are the mostly used accelerators
  - Symmetric multiprocessor
    - Simple cores that are grouped in wraps to execute one instruction
    - Large number of registers
    - Cache and main memories
    - HBM
    - Tensor cores and reduced precision floating point representations
  - The main processor dispatches kernels to the accelerator
    - Kernels are defined in the source code using language extensions (CUDA, OpenACC, OpenMP 4+,…)
    - Data transfers between the CPU and accelerator memories might need application management
  –

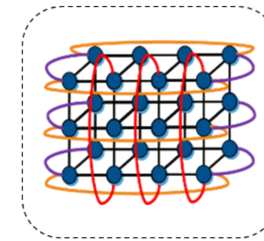


Supercomputing Building Blocks: Network

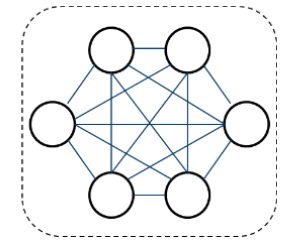
- Nodes exchange data through an interconnect
- Various Network Topologies can be used
  - Fat Tree is commonly used
    - Can be blocking or non-blocking, which determines the total available bandwidth available
  - Dragonfly is becoming more popular
    - Uses less cables, particularly on long links
    - But less connection between groups



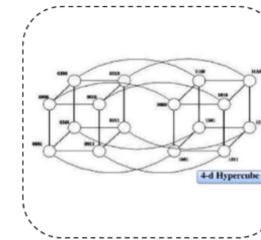
Fat Tree



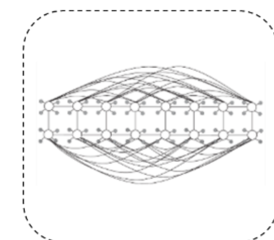
Torus



Dragonfly



Hypercube



HyperX

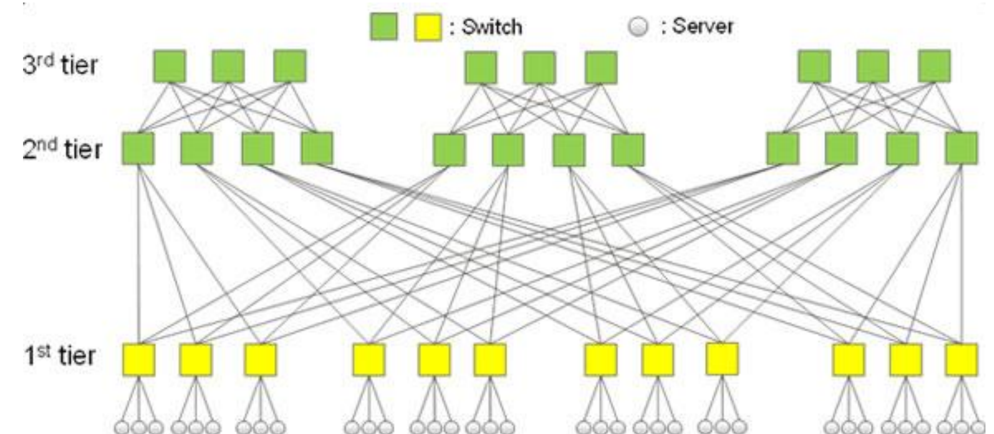
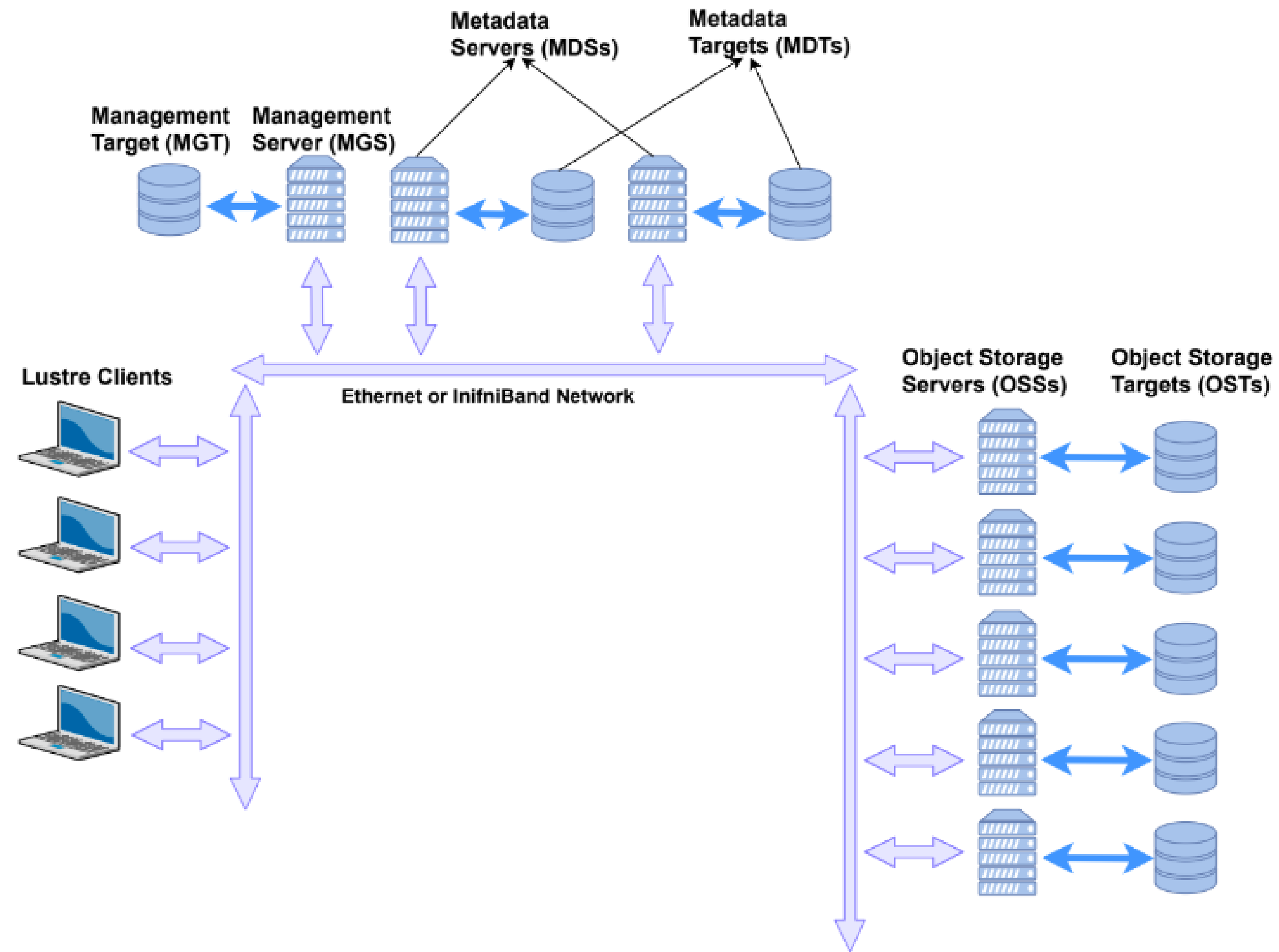
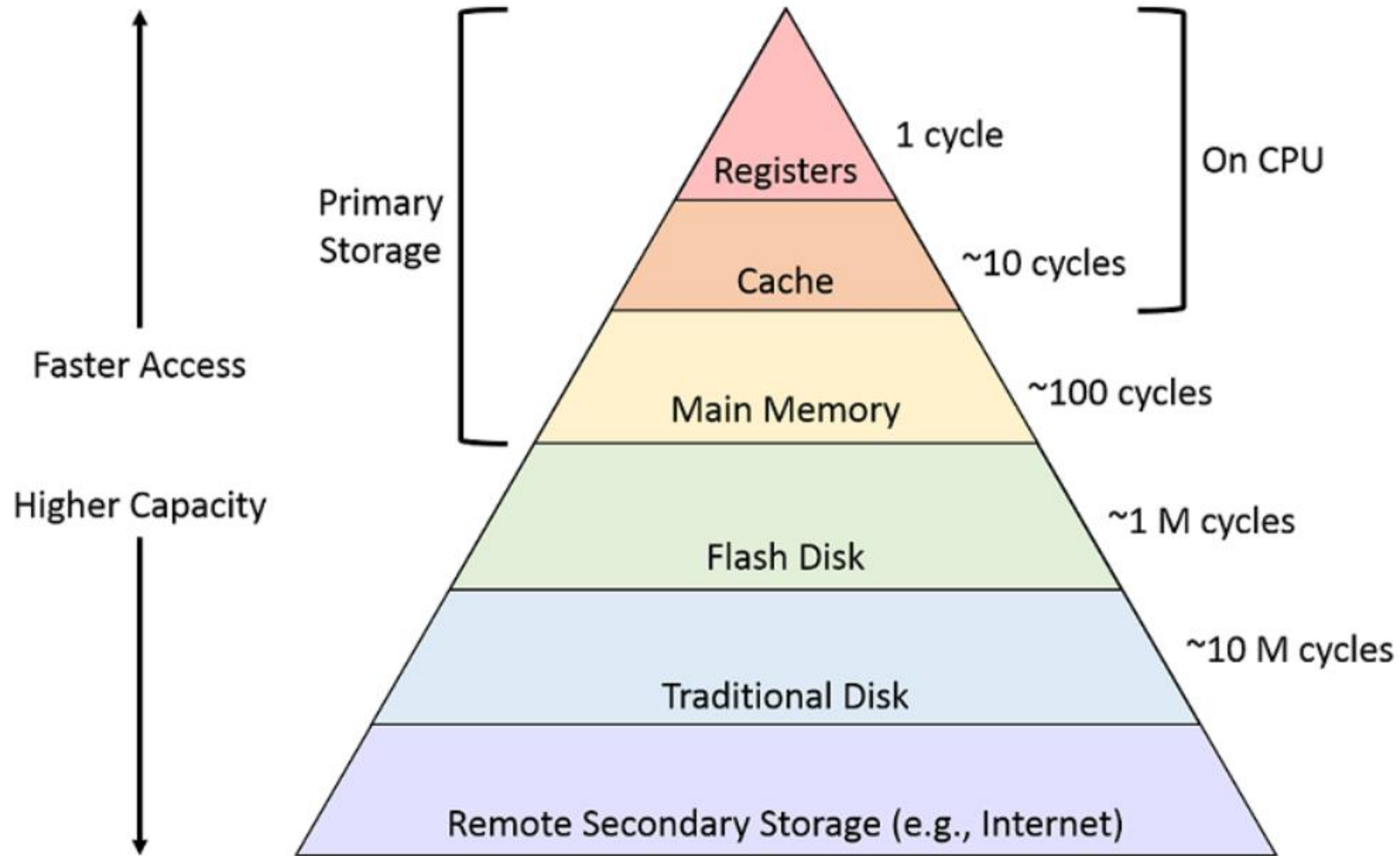


Image from https://www.hpcwire.com/2019/07/15/super-connecting-the-supercomputers-innovations-through-network-topologies/

Supercomputing Building Blocks: Parallel file system



Memory hierarchy



From https://markski.ar/blog/principle-of-locality

Supercomputing Performance Aspects

Supercomputer performance

- Traditionally a supercomputer "compute power" is expressed in Flop rate or Flops
  - 1 Flops = 1 double precision floating-point operation per second
  - Double precision uses 64-bits to store a value
  - **THEORETICAL** peak Flops of a supercomputer is Number of Floating-point operations per core per cycle multiplied by the number of cycles per second multiplied by the number of cores
- The world's top supercomputers are ranked in the Top 500 (www.top500.org), which measures the **SUSTAINED** peak Flops managed by the LINPACK benchmark
  - Solves a dense system of linear equations using LU factorization with partial pivoting
  - Scales with the size of supercomputer and memory available
  - Not representative of scientific codes which are memory bound
- HPCG benchmark
  - A good measurement tool for memory bound applications
- Energy efficiency becomes more important as the system size keep growing

Code performance main features

- Most codes (or subcomponents) will either be compute or memory bound:
  - Compute bound codes are limited by the clock speed of the processor and hardware features: number of FP units, vector width.
  - Memory bound codes are limited by the memory access bandwidth (β)
  - Operational Intensity is the amount of processing work completed per byte of memory accesses (**I**)

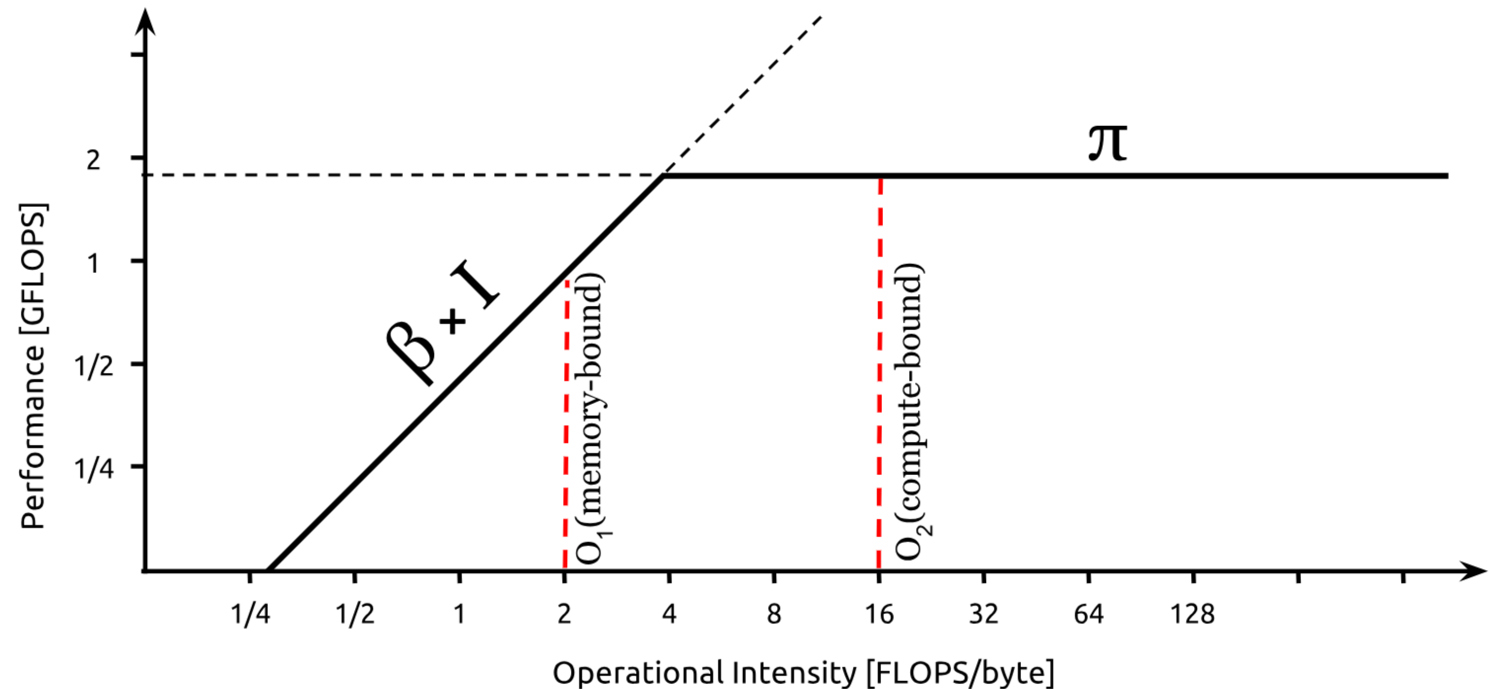
C(1:N) = A(1:N) * B(1:N)

1 FP / 1 store, 2 load

R = MATMUL (T,S)

$$r_{ij} = \sum_k t_{ik} s_{kj}$$

N^3 FP / [3 N^2 load/store]



Compute bound vs memory bound Top500 (I)

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,340,000	1,809.00	2,821.10	29,685
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	**JUPITER Booster** - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, RedHat Enterprise Linux, EVIDEN EuroHPC/FZJ Germany	4,801,344	1,000.00	1,226.28	15,794

Compute bound vs memory bound Top500 (II)

Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	HPCG (TFlop/s)
1	1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,340,000	1,809.00	17406.00
2	7	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	16004.50
3	2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	14054.00
4	3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	5612.60
5	9	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	4586.95

Energy efficiency

Green500 Data

Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	Power (kW)	Energy Efficiency (GFlops/watts)
1	420	**KAIROS** - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, RedHat Enterprise Linux, EVIDEN CALMIP / University of Toulouse - CNRS France	13,056	3.05	46	73.282
2	171	**ROMEO-2025** - BullSequana XH3000, Grace Hopper Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, Red Hat Enterprise Linux, EVIDEN ROMEO HPC Center - Champagne-Ardenne France	47,328	9.86	160	70.912
3	225	**Levante GPU extension** - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, RedHat Enterprise Linux, EVIDEN DKRZ - Deutsches Klimarechenzentrum Germany	35,904	6.75	110	69.426
4	213	**Isambard-AI phase 1** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE University of Bristol United Kingdom	34,272	7.42	117	68.835

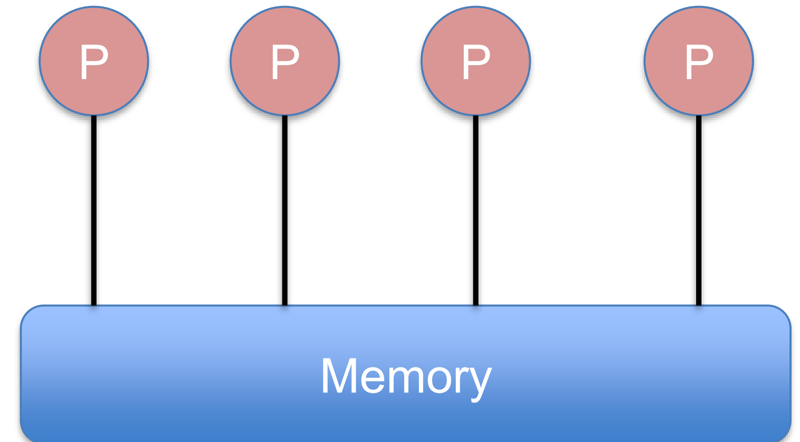
Machine learning supercomputers

- Recently “classical” HPC system were surpassed by the ML dedicated systems:
  - Colossus has O(100k) GPUs (~ 10 x El Capitan)
  - Consuming O(100 MW) of power going for GW in the near future
- The roofline model is a good tool in this case
  - Main differences
    - The ML model number of parameters is much larger that classical computational models
    - One can do batching (processing in parallel independent streams of tokens)
  - A good discussion of details can be found in presentation by Reiner Pope
    - https://www.youtube.com/watch?v=xmkSf5IS-zw&t=1012s
  -

Parallel Programming Paradigms

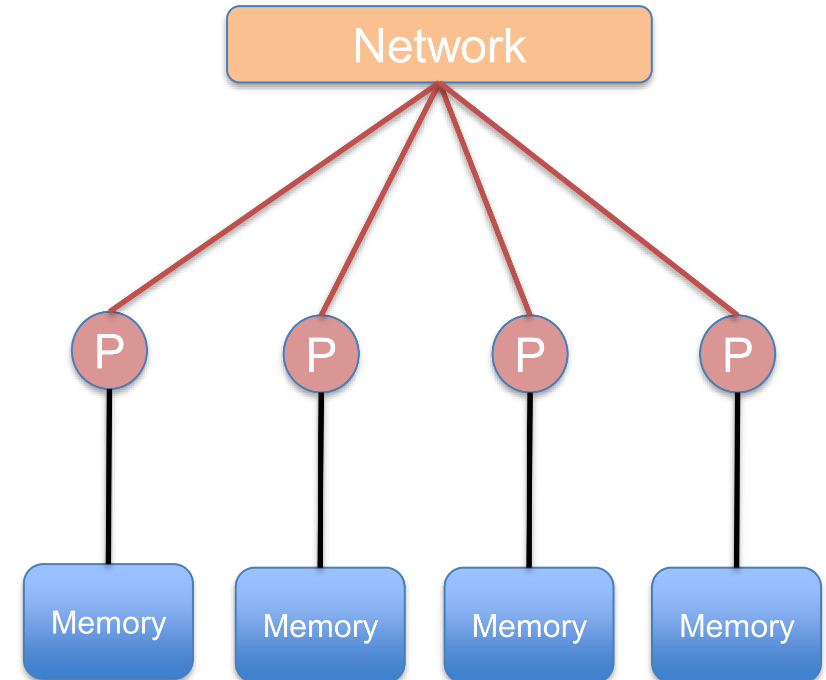
Shared Memory Parallelism

- All processors can see all the memory
  - Bandwidth may not be equal
- Entire domain within the memory
- Execution unit is commonly called a thread
- Need to explicitly protect some variables from being overwritten by other threads
- Most common programming paradigm is via OpenMP
  - Pragma based programming
  - Support is via the compiler
  - Control via environment variables

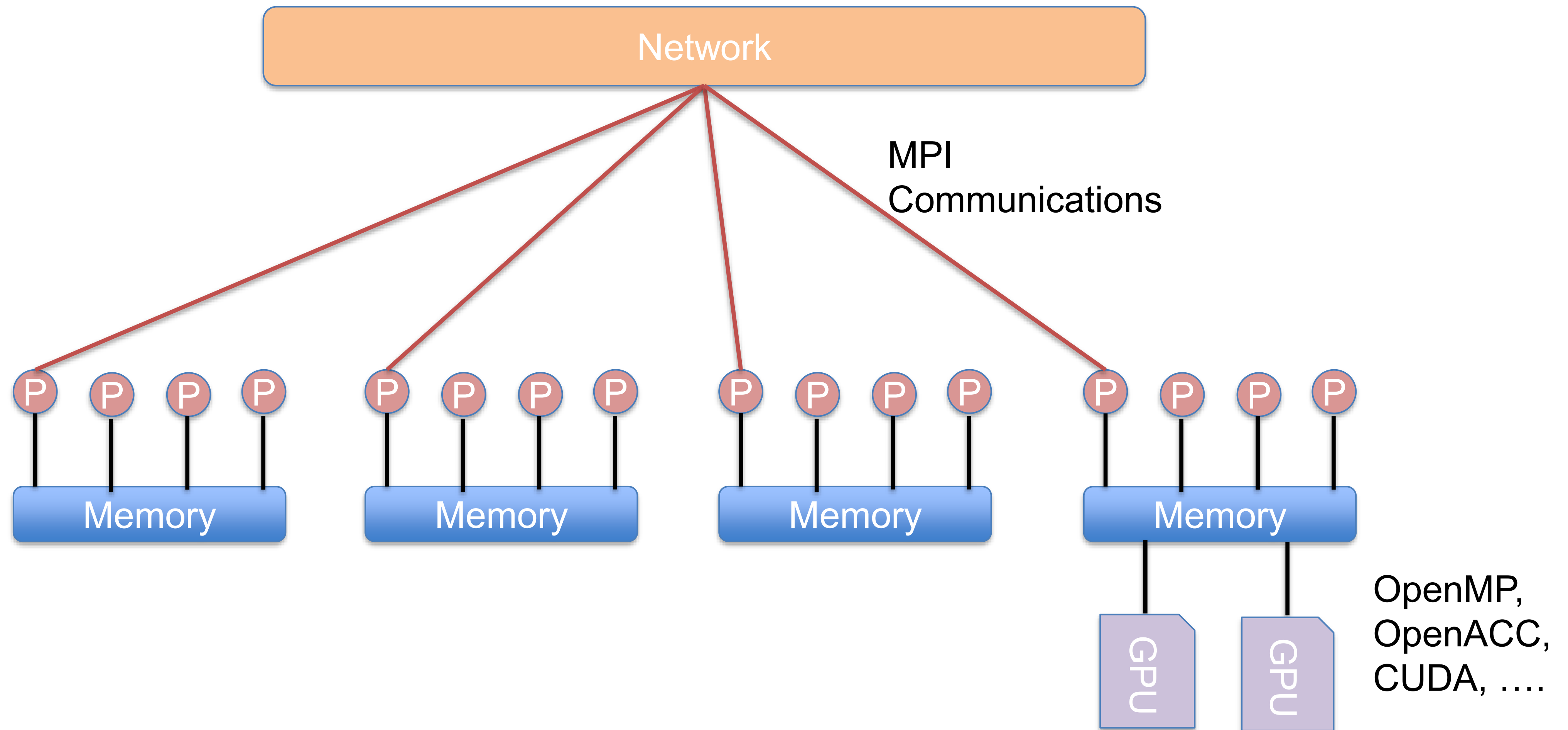


Distributed Memory Parallelism

- Each processor can only see its own memory
- Domain decomposed across the different memories
- Execution unit is commonly called a Rank
- Data exchange has to be explicitly coded and managed through external library
  - Often needed to store and transfer Halo information
- The most common programming paradigm is using MPI
  - Standardised API
  - Several major implementation libraries
  - Subtle differences between them
  - Control through job launchers



Hybrid Parallelism



Domain partition and comms for local operators - Haloes

D	D	D	D	
D	D	D	D	
D	D	D	D	
D	D	D	D	

	D	D	D	D
	D	D	D	D
	D	D	D	D
	D	D	D	D

D	D	D	D	
D	D	D	D	
D	D	D	D	
D	D	D	D	

	D	D	D	D
	D	D	D	D
	D	D	D	D
	D	D	D	D

- Cells labelled “D” are actual Domain Data for that processor
  - what the processor applies its algorithms to
- The different colours indicate what data needs to be shared with neighbouring processors
  - May be needed for algorithms to work
- After each step data in the “haloes” needs to be exchanged to update each processor on changes calculated.

Domain partition and comms for 3D FFT

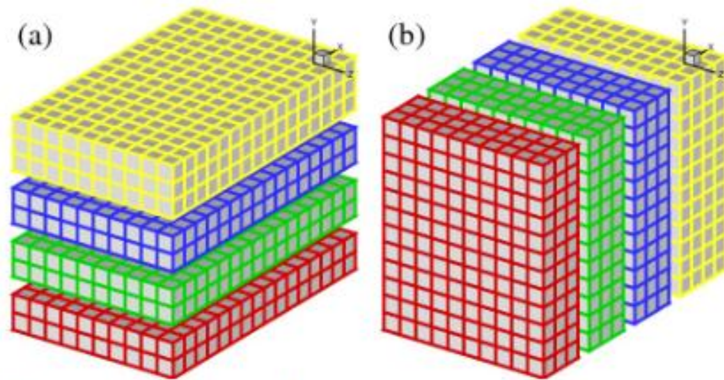


Figure 1: 1D domain decomposition using 4 processors:
(a) decomposed in Y; (b) decomposed in X.

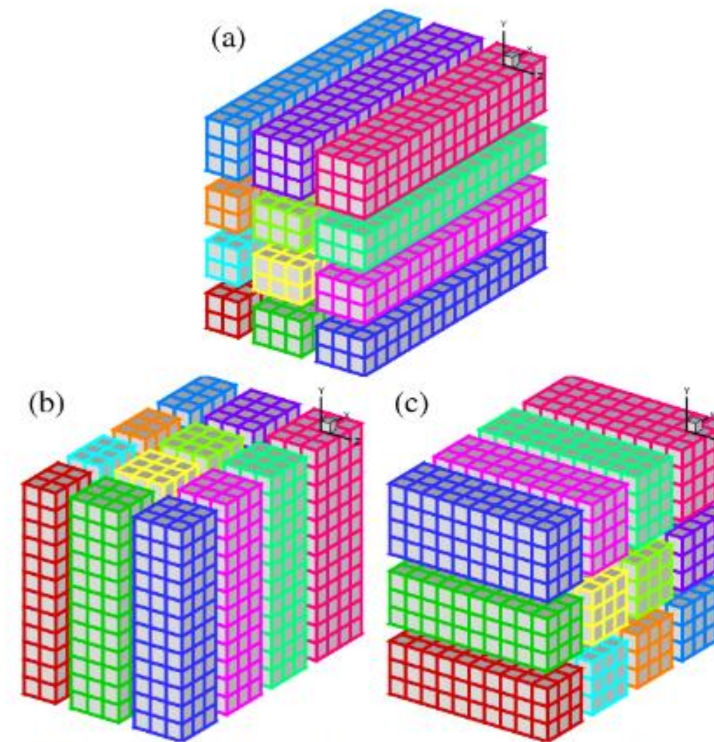


Figure 2: 2D domain decomposition using a 4 by 3 processor grid.

2DECOMP&FFT - A Highly Scalable 2D Decomposition Library and FFT Interface

Ning Li, *the Numerical Algorithms Group (NAG) and*
Sylvain Laizet, *Imperial College London*

Hybrid Parallelism

- Most supercomputers now consist of a series of nodes linked together by a network
  - Each node then consists of a number of processors with access to one or more banks of memory
- It is possible to run MPI across all the available processors
  - But processors compete for access to memory and network
  - Halo exchange becomes expensive
- Therefore, hybrid methods have been developed that
  - decompose the domain across memory regions on the nodes
  - Intra-domain calculations use shared memory paradigms
  - Inter-domain exchanges use distributed memory paradigms

Scaling Limitations

- There are two types of scaling
  - Weak Scaling
    - The amount of work per processor remains the same, i.e. Problem size is a factor of the number of processors
    - Expectation is that the amount of runtime required stays constant as the number of processors increases
  - Strong Scaling
    - The overall size of the problem remains the same but the work per processor reduces as the number of processors increases
    - Expectation is that the runtime decreases in proportion to the number of processors
- However, neither expectation is realized
- Speedup is limited by Amdahl's Law
  - The theoretical maximum is inversely proportional to portion of the code that cannot be parallelized
  - $T = T_{serial} + \frac{T_{par}}{N_{proc}} + T_{comm}(N_{proc}, \ldots)$

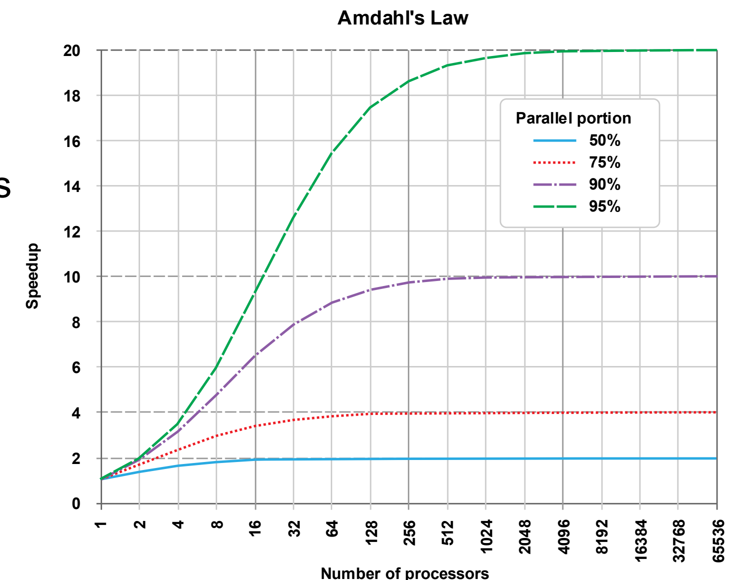


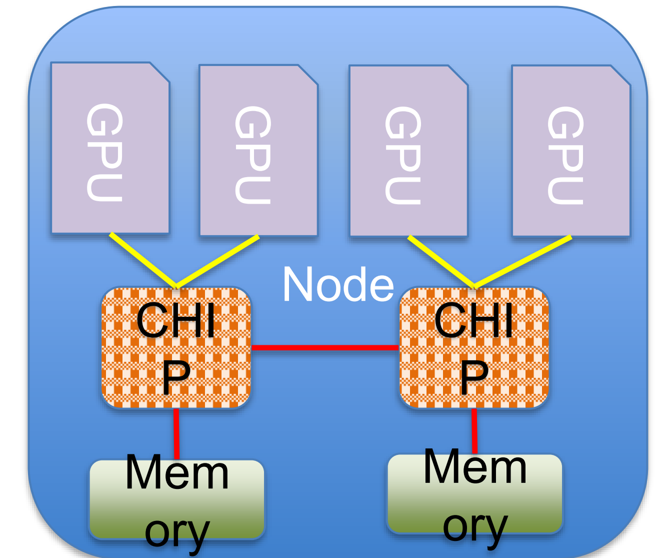
Image from Wikipedia under creative commons
https://en.wikipedia.org/wiki/Amdahl%27s_law

Scaling Limitations

- Some factors that affect scaling:
  - Serial portions of code
  - Load imbalance
    - Not all processors are doing the same amount of work during the same period of time
  - Synchronisation
  - Network bandwidth and latency
  - Algorithmic limitations
    - Running out of parallelism

Future Challenges

- Heterogeneity in many ways
  - Processor – complex compute modes with scalar and vector, prefetch, etc.
  - Many (but not all) include separate accelerators (GPUs and others)
  - Memory – Cache was bad enough; now HBM, other
  - I/O – Burst buffers (often violating POSIX semantics), on node, central, remote (cloud)
- For a high level of performance system programming elements will be needed in the app code to guide code generation.
- Data management
- Results validation (Bit-reproducibility)
- In NWP what is the right system design to balance DL with physics based models.



Further Reading

- OpenMP Standards Community: https://www.openmp.org/
- Basic OpenMP Tutorial: https://hpc-tutorials.llnl.gov/openmp/
- MPI Standards Website: https://www.mpi-forum.org/docs/
- Basic MPI Tutorial: https://mpitutorial.com/tutorials/
- Jülich do online and in-person courses, next years to be posted: https://www.fz-juelich.de/en/ias/jsc/news/events/training-courses

OpenMP Example

```
!$OMP PARALLEL DO SCHEDULE(STATIC,1)&
!$OMP& PRIVATE(JMLOCF,IM,ISTA,IEND)
      DO JMLOCF=NPTRMF(MYSETN),NPTRMF(MYSETN+1)-1
        IM=MYMS(JMLOCF)
        ISTA=NSPSTAF(IM)
        IEND=ISTA+2*(NSMAX+1-IM)-1
        CALL SPCSI(CDCONF,IM,ISTA,IEND,LLONEM,ISPEC2V,&
         &ZSPVORG,ZSPDIVG,ZSPTG,ZSPSPG)
      ENDDO
!$OMP END PARALLEL DO
```

MPI Examples

```
int ping_pong_count = 0;
int partner_rank = (world_rank + 1) % 2;

while (ping_pong_count < PING_PONG_LIMIT) {
     if (world_rank == ping_pong_count % 2) {
          ping_pong_count++;

          MPI_Send(&ping_pong_count, 1, MPI_INT,
          partner_rank, 0, MPI_COMM_WORLD);

          printf("%d sent and incremented
          ping_pong_count " "%d to %d\n”,
          world_rank, ping_pong_count,
          partner_rank);
     } else {
          MPI_Recv(&ping_pong_count, 1, MPI_INT,
          partner_rank, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);

          printf("%d received ping_pong_count %d
          from %d\n", world_rank, ping_pong_count,
          partner_rank);
     }
}
```

```
if (world_rank == 0) {
     rand_nums = create_rand_nums(elements_per_proc
     * world_size);
}

float *sub_rand_nums = malloc(sizeof(float) *
elements_per_proc);

MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT,
sub_rand_nums, elements_per_proc, MPI_FLOAT, 0,
MPI_COMM_WORLD);

float sub_avg = compute_avg(sub_rand_nums,
elements_per_proc);

float *sub_avgs = NULL;
if (world_rank == 0) {
     sub_avgs = malloc(sizeof(float) * world_size);
}

MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1,
MPI_FLOAT, 0, MPI_COMM_WORLD);

if (world_rank == 0) {
     float avg = compute_avg(sub_avgs, world_size);
}
```


Examples from www.mpitutorial.com