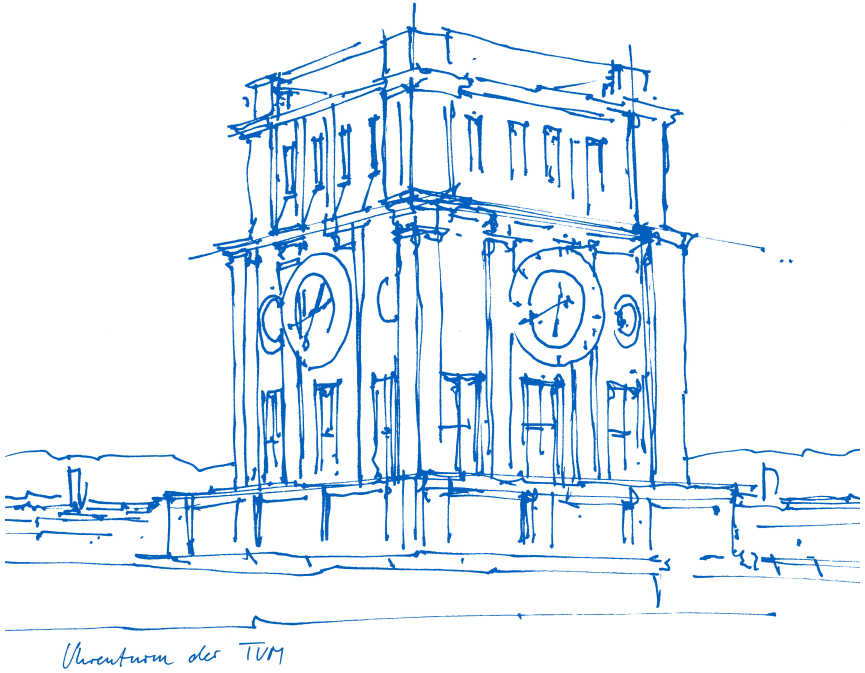
Lecture IN-2147
Parallel Programming

SoSe 2018

Martin Schulz
Technische Universität München
Fakultät für Informatik

Exercises:
Amir Raoofy



Parallelism

Multiple activities to solve one problem together

- Multiple threads, processes on multiple cores, nodes, …
- Work divided up by ...
  - Splitting up data
  - Assigning pipeline stages
  - Requesting work assignments
- Where needed synchronization and communication

Target: any compute intensive calculation

- Typical example: physics simulations (CFD, MD, …)
- Scheduling problems, discrete event simulations
- Many more

Contrast with concurrency

- Independent tasks to allow overlap
- Example: GUI threads to wait for input

Goals for this Lecture

Basic introduction into parallelism

- The need and purpose of using parallelism
- Challenges and how to overcome them
- Architectures and application areas
- Metrics

Parallel programming APIs

- Message Passing Interface (MPI)
- OpenMP
- Pthreads
- New models: CUDA, Task-based programming, …

Optimization and tuning

- Typical bottlenecks and pitfalls

Contents & Schedule (tentative!)

Lectures/Exercises: Monday 16:15, IHS-2

Exercises/Lectures: Wednesday 8:15, MW-0350

	Monday	Wednesday
9./11.4.	Basics / Introduction	Threading / Pthread
16./18.4.	**Exercise**	**Exercise**
23./25.4.	OpenMP Basics	Shared Memory / Dependencies
30.4./2.5.	**Exercise**	**Exercise**
7./9.5.	OpenMP Advanced	**Exercise**
14./16.5.	HPC Architectures and Concepts	**Exercise**
21./23.5.		**Exercise**
28./30.5.	MPI Basics	**Exercise**
4./6.6.	Distributed Memory / Networks	MPI Advanced
11./13.6.	**Exercise**	Tuning and Tools
18./20.6.	Scaling / Mapping	**Exercise**
25./27.6.	**Exercise**	**Exercise**
2./4.7.	Accelerator/GPU Programming	Tasks/PGAS/Future Trends
9./11.7.	**Exercise**	**Exercise**

Practical Information Lecture

Lecture slides will be available after the lectures at

http://parprog.lrr.in.tum.de/ (for now)

Moodle (soon)

Exam

- Final exam will cover lecture and exercises
- 24.07.2017, 16:00 (Please check the final date in TUM-Online)
- Repetition exam tbd.

Do not forget to register for the exam in TUM-Online

Please be interactive and ask questions!

Feel free to send me comments as we go along: schulzm@in.tum.de

Exercises are important!!!

Practical Information Exercises

Exercise slides and assignments will be available at

http://parprog.lrr.in.tum.de/

Need to register until the end of next week!

Assignments

- 11 assignments on parallel programming techniques in C/C++
- Automatic submission, grading, plagiarism, speedup and memory leak checks
- 0.3 bonus for successful submission of at least **80%** of the assignments
  - Successful = passes submission checks (correctness & Speedup)
- Q&A sessions with student tutors

Required knowledge

- Knowledge of C/C++
- Experience with Linux Command Line
- Knowing compilers/toolchain (e.g., GCC)

Keep in touch, send comments: amir.raoofy@tum.de

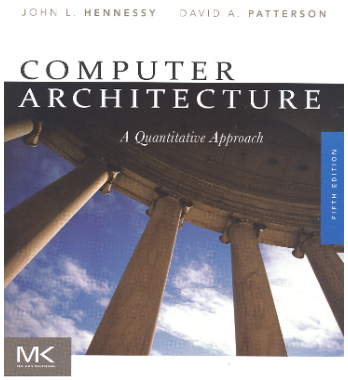
Contents & Schedule (tentative!)

Lectures/Exercises: Monday 16:15, IHS-2

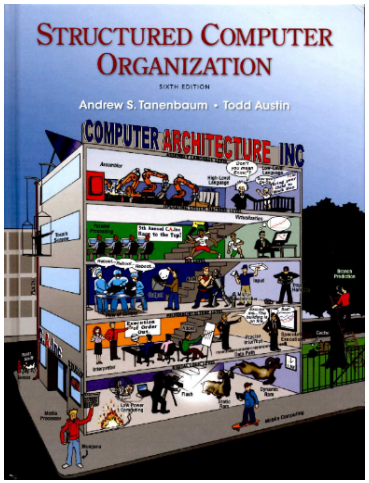
Exercises/Lectures: Wednesday 8:15, MW-0350

	Monday	Wednesday
9./11.4.	Basics / Introduction	Threading / Pthread
16./18.4.	**Exercise**	**Exercise**
23./25.4.	OpenMP Basics	Shared Memory / Dependencies
30.4./2.5.	**Exercise**	**Exercise**
7./9.5.	OpenMP Advanced	**Exercise**
14./16.5.	HPC Architectures and Concepts	**Exercise**
21./23.5.		**Exercise**
28./30.5.	MPI Basics	**Exercise**
4./6.6.	Distributed Memory / Networks	MPI Advanced
11./13.6.	**Exercise**	Tuning and Tools
18./20.6.	Scaling / Mapping	**Exercise**
25./27.6.	**Exercise**	**Exercise**
2./4.7.	Accelerator/GPU Programming	Tasks/PGAS/Future Trends
9./11.7.	**Exercise**	**Exercise**

Books on Computer Architecture

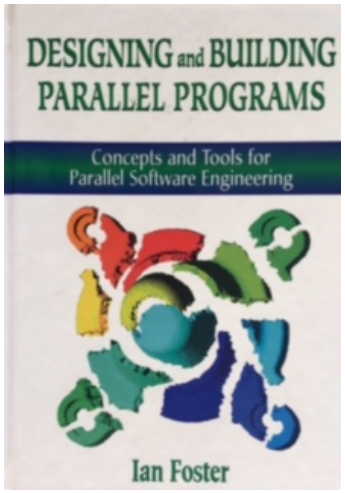


Computer Architecture - A quantitative Approach.
John Hennessy, David Patterson
5th Edition, September 2011

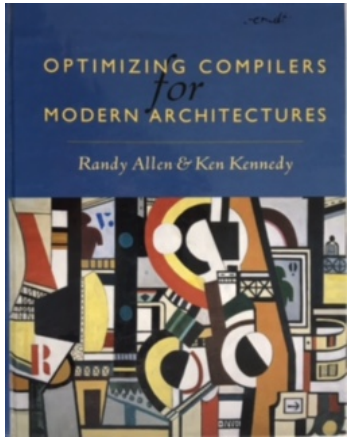


Tanenbaum: Structured Computer Organization
Pearson Studium, 2013, 6. Auflage,
Standardwerk

Books on Programming Models



Ian Foster: Designing and Building Parallel Programs.
http://www.mcs.anl.gov/~itf/dbpp/



Randy Allen, Ken Kennedy: Optimizing Compilers for Modern Architectures: A Dependence-based Approach

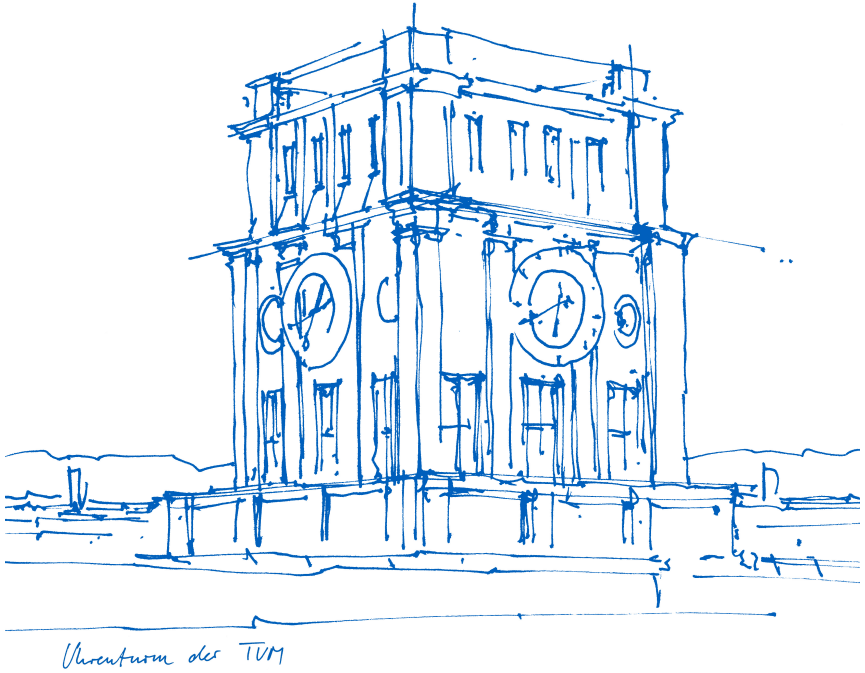
- MPI, OpenMP, CUDA, OpenACC Standards
- https://computing.llnl.gov/tutorials/parallel_comp/

Lecture IN-2147
Parallel Programming

SoSe 2018

Martin Schulz
Technische Universität München
Fakultät für Informatik

Lecture 1:
Basic Concepts,
Architectures & Metrics



The World is Parallel – Think Parallel

Parallelism is Nothing New

Hardware is by default parallel

Higher-level parallelism often unnoted / hidden in the processor

- Instruction-level parallelism (ILP)
- Many instructions active at the same time

Today: parallelism is everywhere

- Multi/many-core chips
- Multi-socket boards
- Cluster systems
- GPUs

Same in software

- Several dedicated parallel programming models that are widely used
- Active area of research
- Integration into many languages (Java, C++, HPF)

A bit of History: Illiac-IV

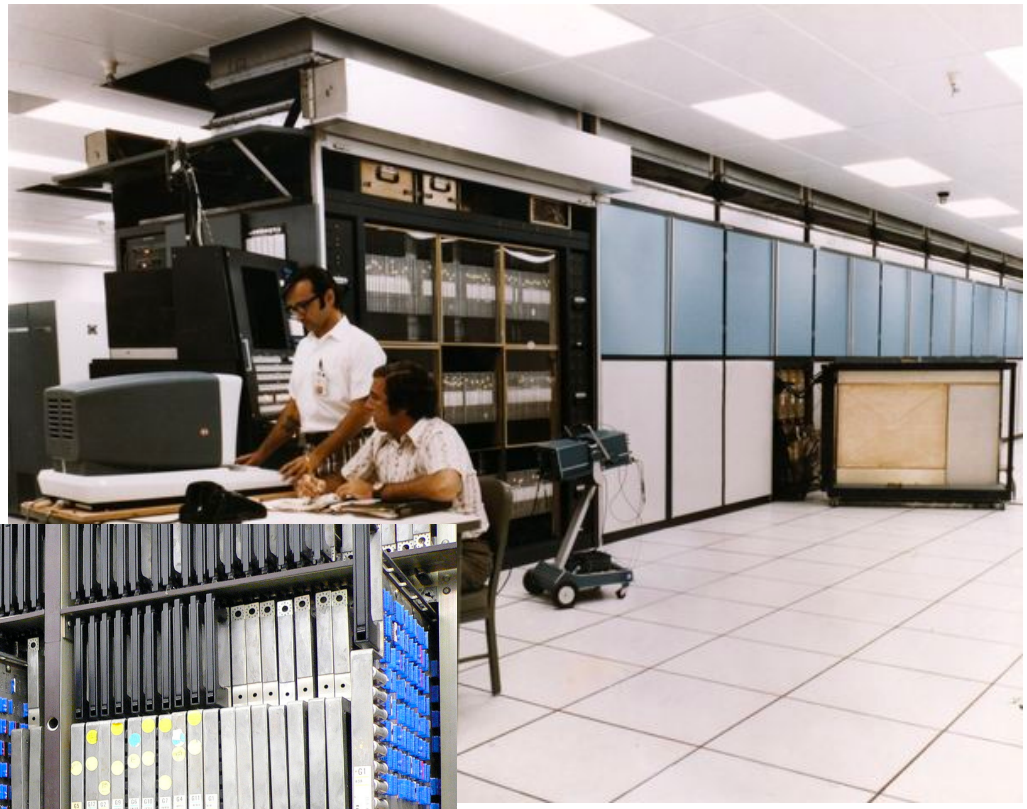
First massively parallel computer

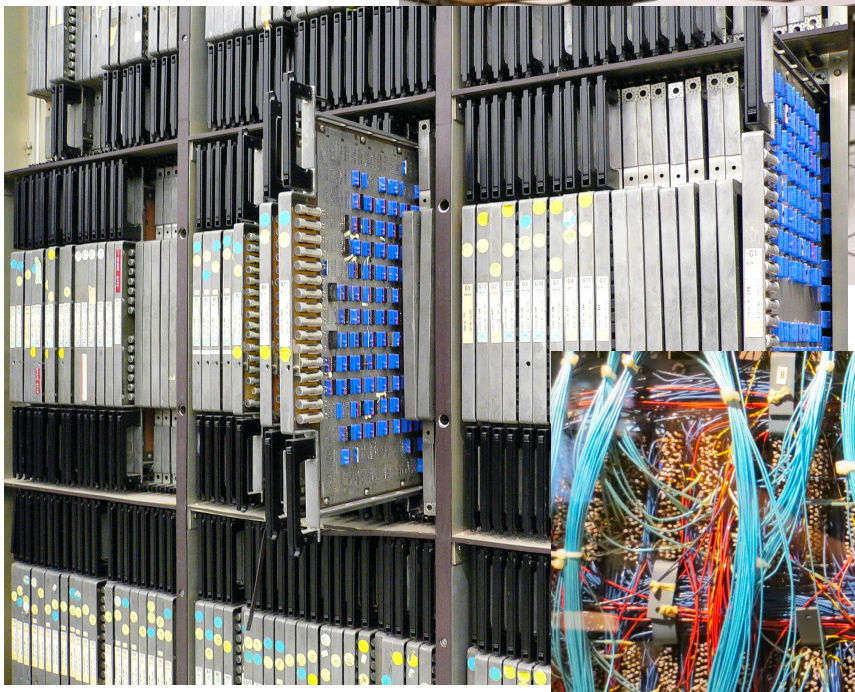
- Concept started in 1952
- Build started in 1966
- Delivered in 1972
  at NASA Aims

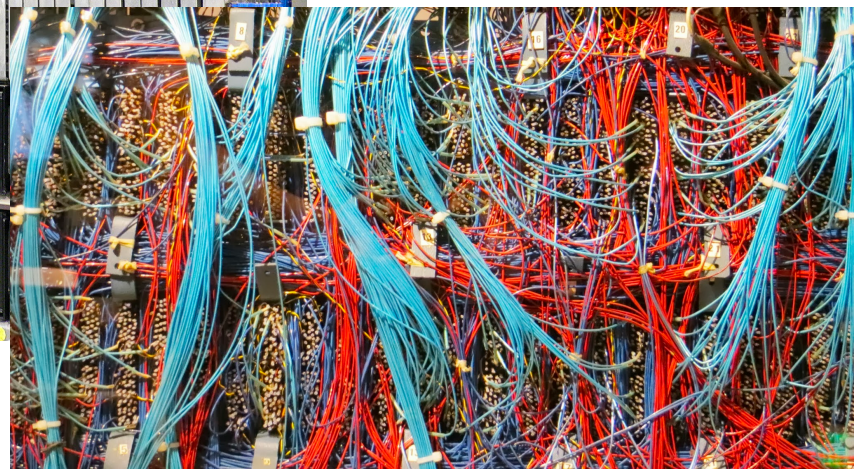
Designed with 4 cores

- Each with 64 Pes
- Built: 1 core

200 MFlop/s







Quelle: Wikipedia

Vector Processing

One control unit, but many processing units

- Follows ideas from Illiac-IV

First commercial viable machine: Cray-1

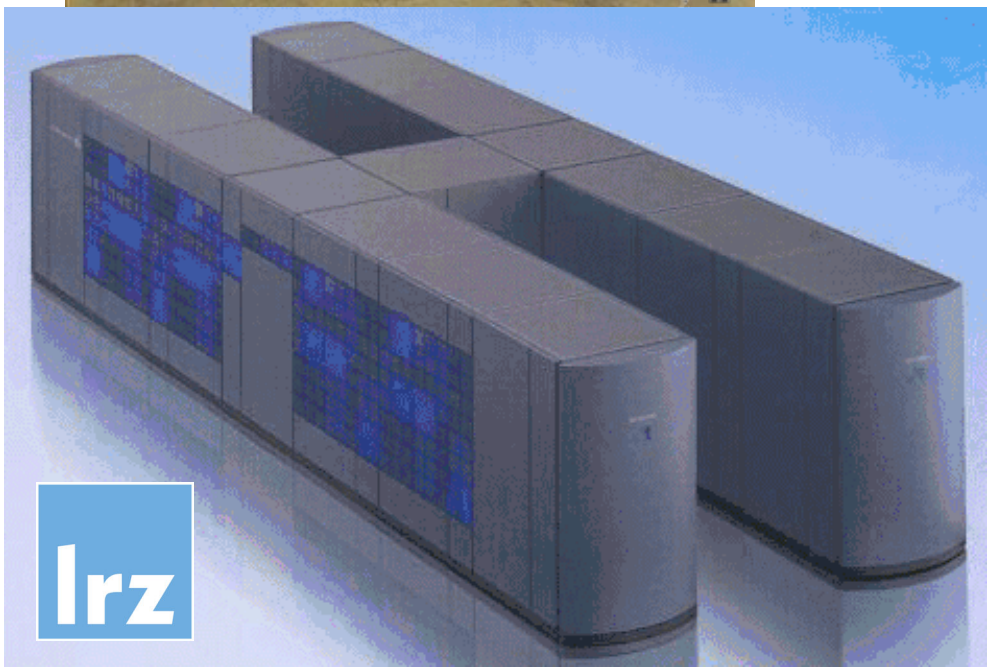
- Eight vector registers
- 64 words of 64 bits each
- Peak 240 Mflop/s

Many other models followed

- Cray series of machines
- Hitachi, Fujitsu, …
- Heavy impact on software

Very relevant (again) today

- Intel's AVX-512
- ARM's SVE
- GPU programming



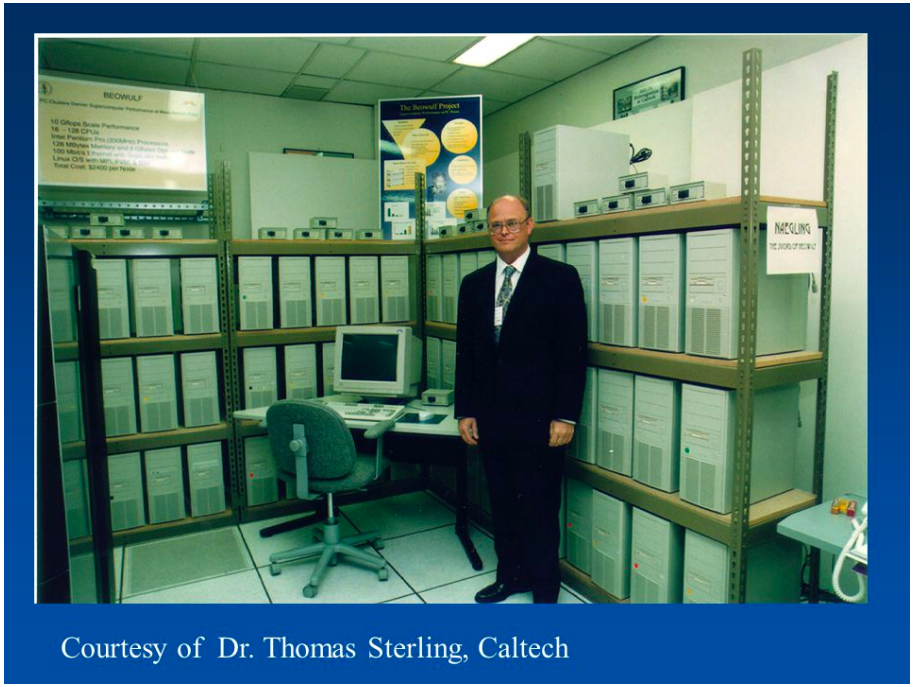
Cluster Computing

Started as “Cheap Parallel Processing”
- Several connected PCs
- Simple communication HW/SW
- “Beowulf” Systems

Use of standard SW stacks
- Linux played a big role

Today’s system
- More professional
- 1U nodes
- Blade designs
- Targeted cooling systems

Difference to HPC systems is getting less and less



Courtesy of Dr. Thomas Sterling, Caltech

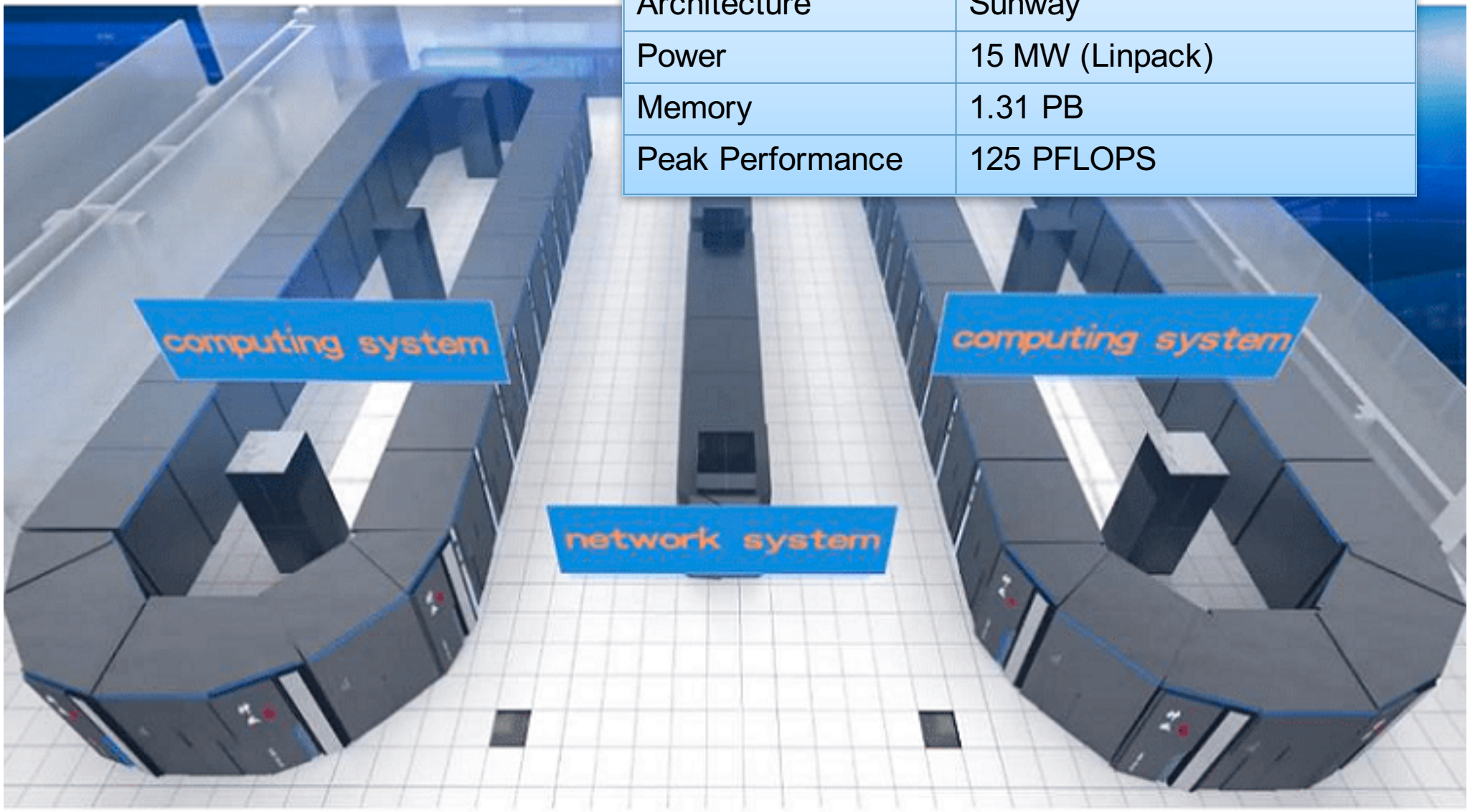
Top500 List as of November 2017

#	Site	Manufacturer	Computer	Country	Cores	Rmax [Pflops]	Power [MW]
1	National Supercomputing Center in Wuxi	NRCPC	Sunway TaihuLight NRCPC Sunway SW26010, 260C 1.45GHz	China	10,649,600	93.0	15.4
2	National University of Defense Technology	NUDT	Tianhe-2 NUDT TH-IVB-FEP, Xeon 12C 2.2GHz, IntelXeon Phi	China	3,120,000	33.9	17.8
3	Swiss National Supercomputing Centre (CSCS)	Cray	Piz Daint Cray XC50, Xeon E5 12C 2.6GHz, Aries, NVIDIA Tesla P100	Switzerland	361,760	19.6	2.27
4	Japan Agency for Marine-Earth Science and Technology	ExaScaler	Gyoukou ZettaScaler-2.2 HPC System, Xeon 16C 1.3GHz, IB-EDR, PEZY-SC2 700Mhz	Japan	19,860,000	19.1	1.35
5	Oak Ridge National Laboratory	Cray	Titan Cray XK7, Opteron 16C 2.2GHz, Gemini, NVIDIA K20x	USA	560,640	17.6	8.21
6	Lawrence Livermore National Laboratory	IBM	Sequoia BlueGene/Q, Power BQC 16C 1.6GHz, Custom	USA	1,572,864	17.2	7.89
7	Los Alamos NL / Sandia NL	Cray	Trinity Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries	USA	979,968	14.1	3.84
8	Lawrence Berkeley National Laboratory	Cray	Cori Cray XC40, Intel Xeons Phi 7250 68C 1.4 GHz, Aries	USA	622,336	14.0	3.94
9	JCAHPC Joint Center for Advanced HPC	Fujitsu	Oakforest-PACS PRIMERGY CX1640 M1, Intel Xeons Phi 7250 68C 1.4 GHz, OmniPath	Japan	556,104	13.6	2.72
10	RIKEN Advanced Institute for Computational Science	Fujitsu	K Computer SPARC64 VIIIfx 2.0GHz, Tofu Interconnect	Japan	795,024	10.5	12.7

Source: Top500

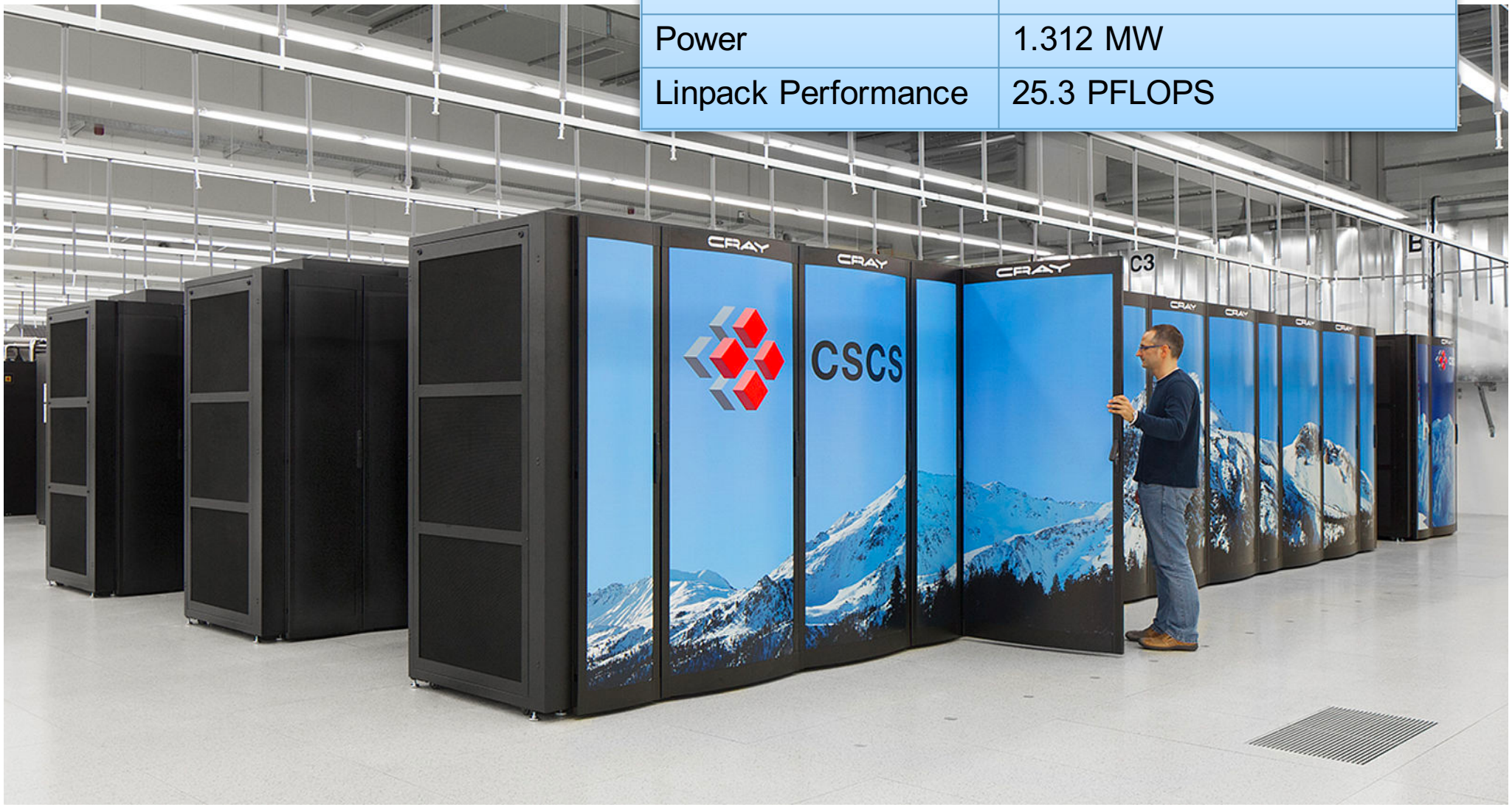
Sunway Taihu Light

Location	National Supercomputer Center, Wuxi, Jiangsu, China
Architecture	Sunway
Power	15 MW (Linpack)
Memory	1.31 PB
Peak Performance	125 PFLOPS



CSCS's Piz Daint

Location	Swiss National Supercomputing Centre
Architecture	Intel Xeon / Nvidia P100
Power	1.312 MW
Linpack Performance	25.3 PFLOPS



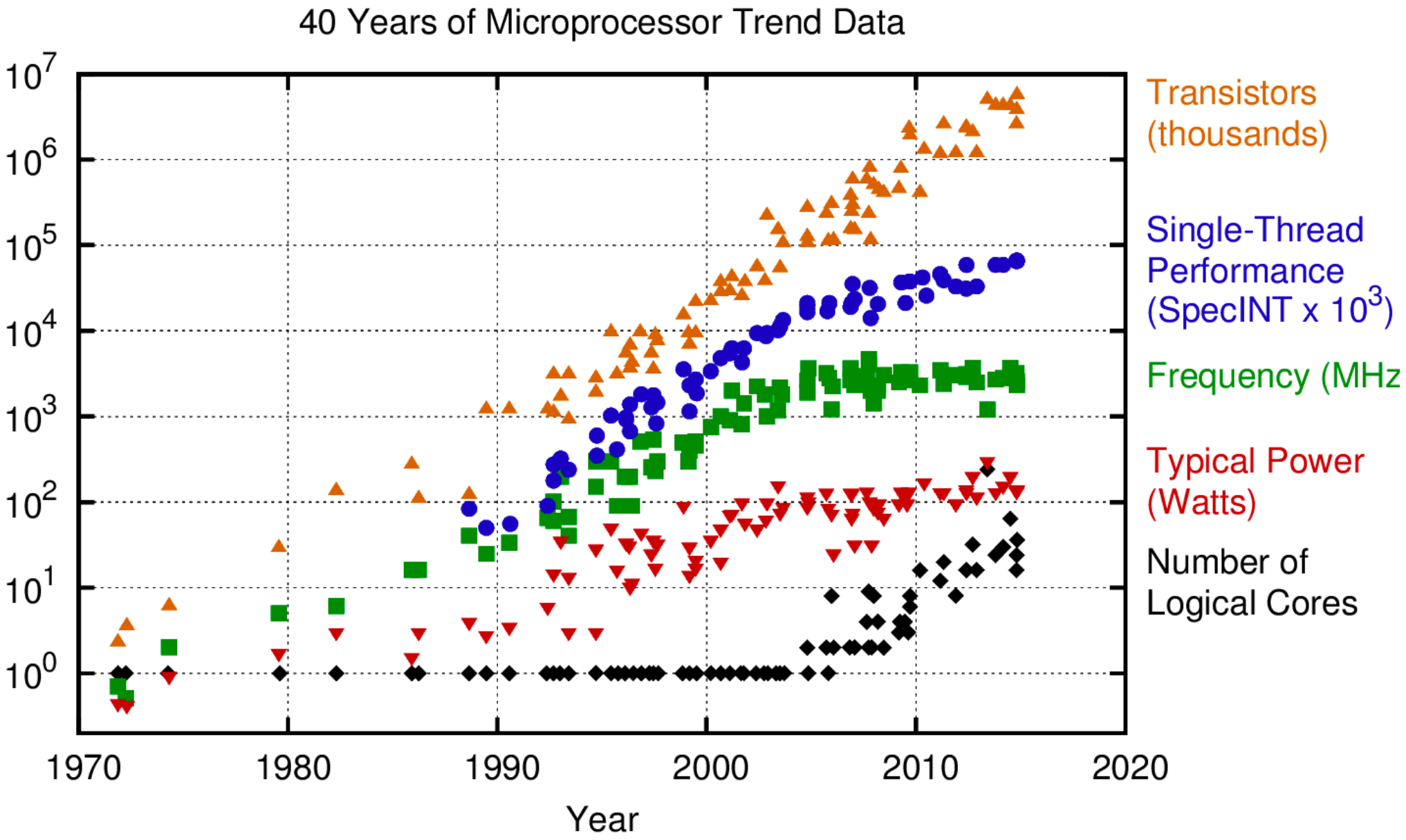
LLNL's Sequoia

Location	Lawrence Livermore National Laboratory, CA, USA
Architecture	IBM Power
Power	7.9 MW (Linpack)
Memory	1.5 PB
Peak Performance	20.13 PFLOPS

LRZ's SuperMUC

44	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM/Lenovo	147,456	2,897.0	3,185.1	3,423
45	Leibniz Rechenzentrum Germany	**SuperMUC Phase 2** - NeXtScale nx360M5, Xeon E5-2697v3 14C 2.6GHz, Infiniband FDR14 Lenovo/IBM	86,016	2,813.6	3,578.3	1,481

Trend towards Multi-core Chips



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Why Parallelism?

Sequential computing has come to its limits

Parallel processing to solve problems faster
- Cooperating cores/nodes
- Gaining scientific results faster
- Achieving results under a deadline (e.g., wheather forecasting)

Parallel processing to solve larger problems
- More nodes = more space to store data
- Higher resolution, new physical techniques

In both cases, we need efficiency
- Efficient utilization of computational resources
- Efficient utilization of available memory
- New topic: efficient utilization of energy

Performance Metrics

Speedup

$$speedup(p\ processors) = \frac{performance(p\ processors)}{performance(1\ processor)}$$

Scientific computing: performance=work/time

$$speedup(p\ processors) = \frac{time(1\ processor)}{time(p\ processors)}$$

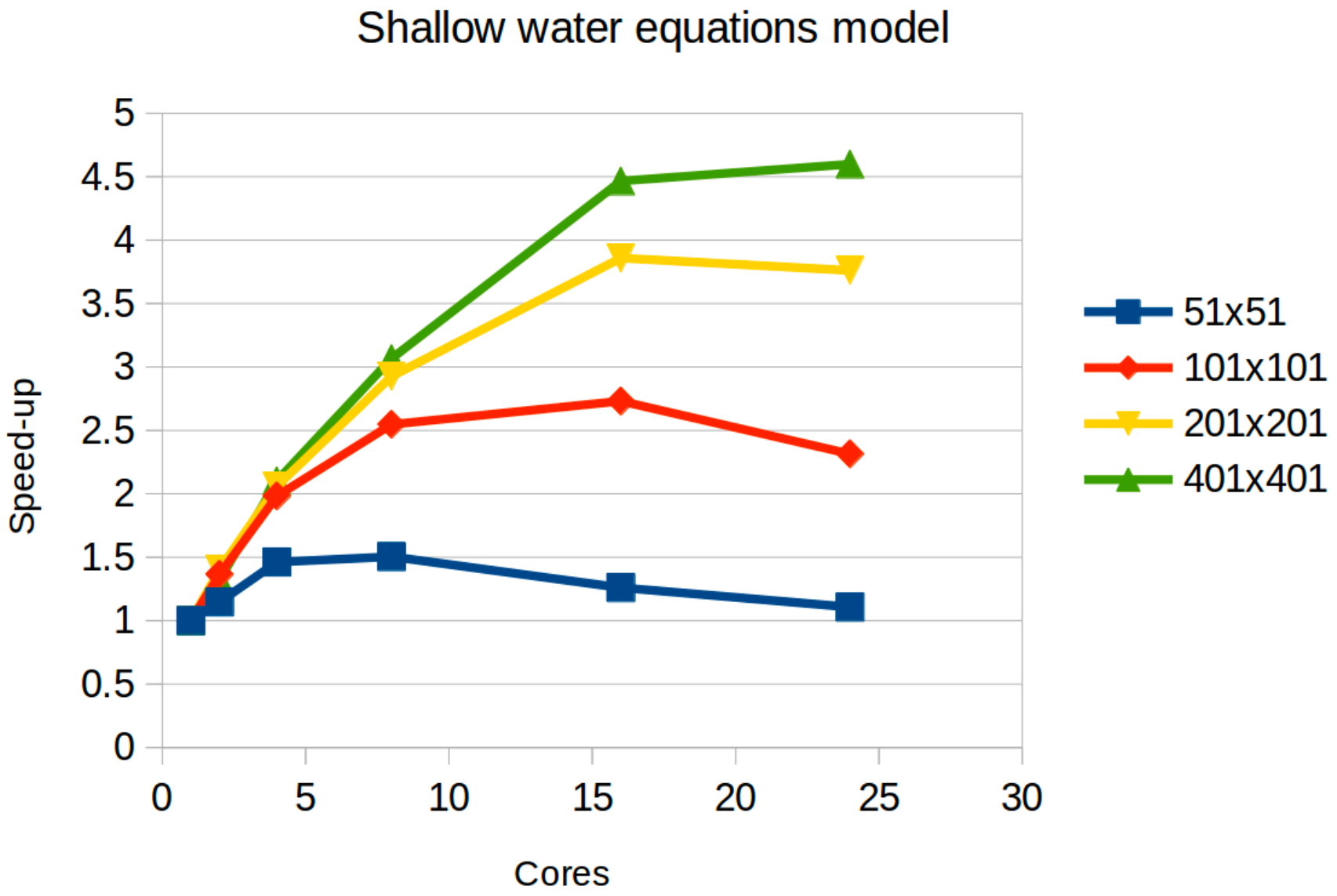
Speedup based on Troughput

$$speedup(p\ processors) = \frac{tpm(p\ processor)}{tpm(1\ processors)}$$

Parallel Efficiency

$$efficiency(p\ processors) = \frac{speedup(p\ processors)}{p}$$

Speed-Up/Efficiency Curves



https://summerofhpc.prace-ri.eu/shallow-waters-the-waves-start-to-spread/

How (Not) to Improve your Results

1. Quote only 32-bit performance results, not 64-bit results.
2. Present performance figures for an inner kernel, but claim for entire application.
3. Quietly employ assembly code and other low-level language constructs.
4. Scale problem size with the number of processors, but omit any mention of this fact.
5. Quote performance results projected to a full system.
6. Compare your results against scalar, unoptimized code (on Crays).
7. Compare with old code on an obsolete system.
8. Base operation count on the parallel code, not on the best sequential one.
9. Quote performance as processor utilization, speedups or MFLOPS/$$$.
10. Mutilate the algorithm used in the parallel implementation to match the architecture.
11. Measure parallel runtimes on a dedicated system, but seq. runtimes in a busy one.
12. If all else fails, show pretty pictures/videos, and don't talk about performance.

D. Bailey, Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers Slightly paraphrased

Amdahl's Law

Gene Amdahl

1922-2015

Formulation for maximal speedup

Parameters:

f = fraction of parallel execution

p = number of parallel tasks/threads/processes

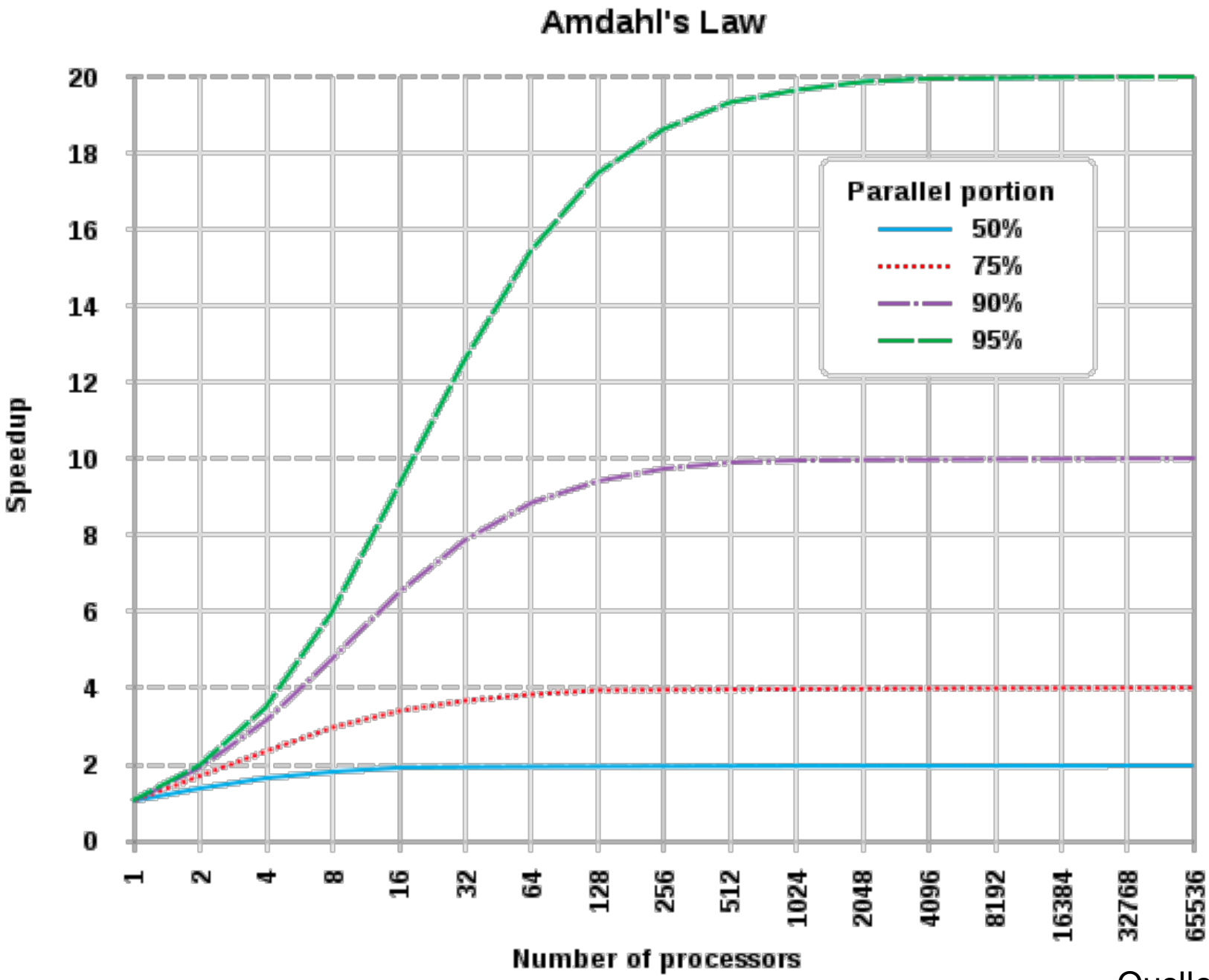
Execution time (assuming parallel regions have full speedup)

$$T(p) = (1 - f) * T + \frac{f * T}{p}$$

Maximal Speedup:

$$SU(p) = \frac{T}{T(p)} = \frac{T}{(1 - f) * T + \frac{f*T}{p}} = \frac{1}{1 - f + \frac{f}{p}}$$

Amdahl’s Law - Examples

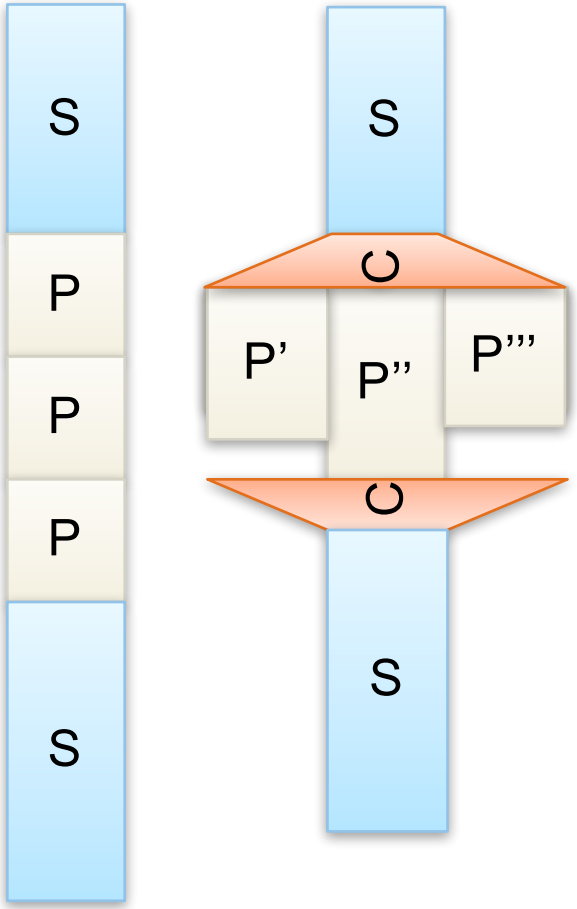


Quelle: Wikipedia

Consequences of Amdahl's Law

Even small portions of sequential work impacts scalability
- Speedup of 1000x –> 99.9% must be (perfectly) parallel
- Applies to all codes, algorithms, …

But: Amdahl's Law is only one challenge, since perfect scaling within a parallel piece of code is hard
- Communication and Synchronization
- Resource bottlenecks on a processor
  - Memory bandwidth
  - Hyperthreading, SMT, ...
- Contention in the network
- Multiple processes on one node
- Load imbalance

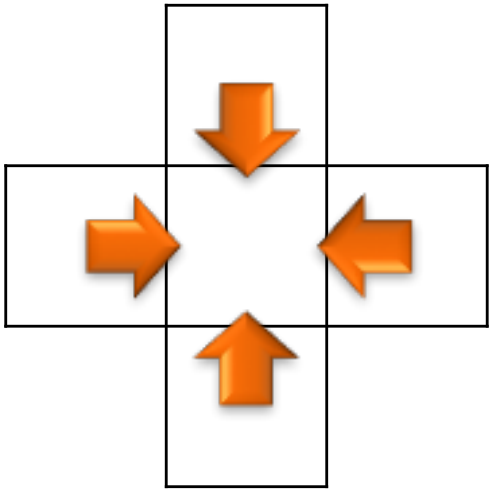


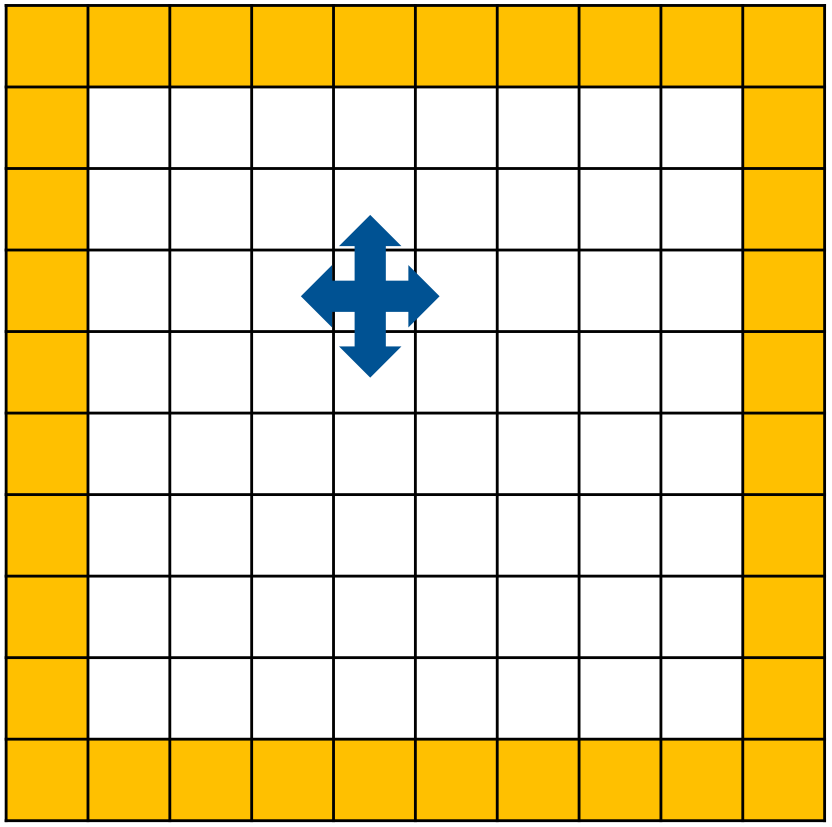
But: we can also use parallelism to our advantage
- More nodes = more memory = more memory bandwidth
- More parallelism = may be less work per process/task = may fit in cache
- These effects could lead to „superlinear speedup"

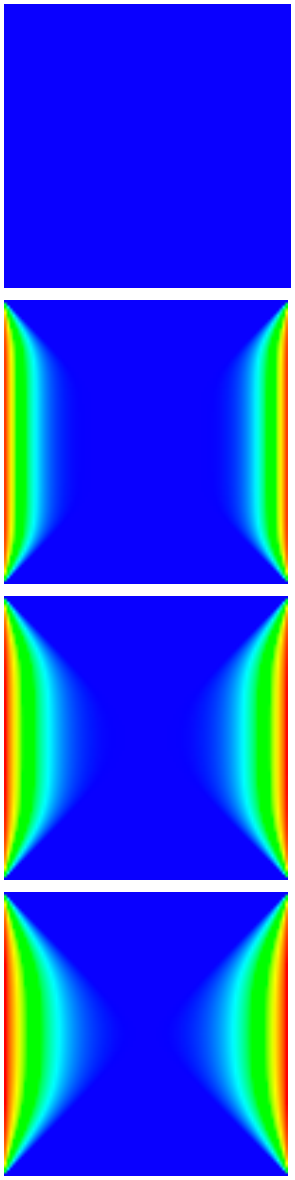
How to Write a Parallel Program?

Example: Heat Diffusion

- Calculate heat distribution
- Solved using a stencil
- Example in 2D
- Dense matrix

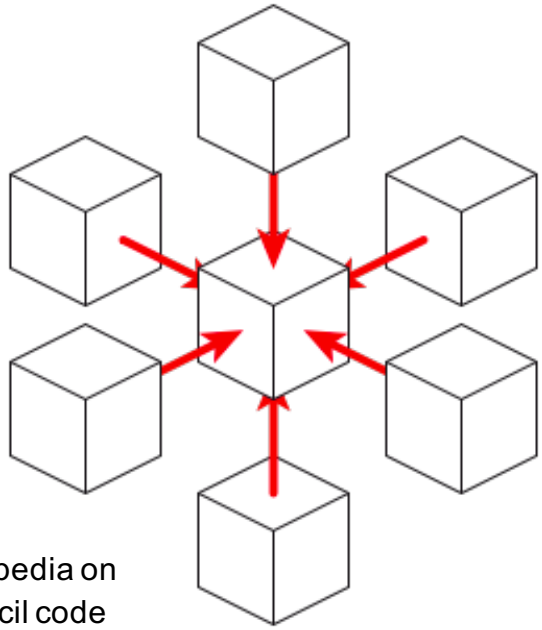




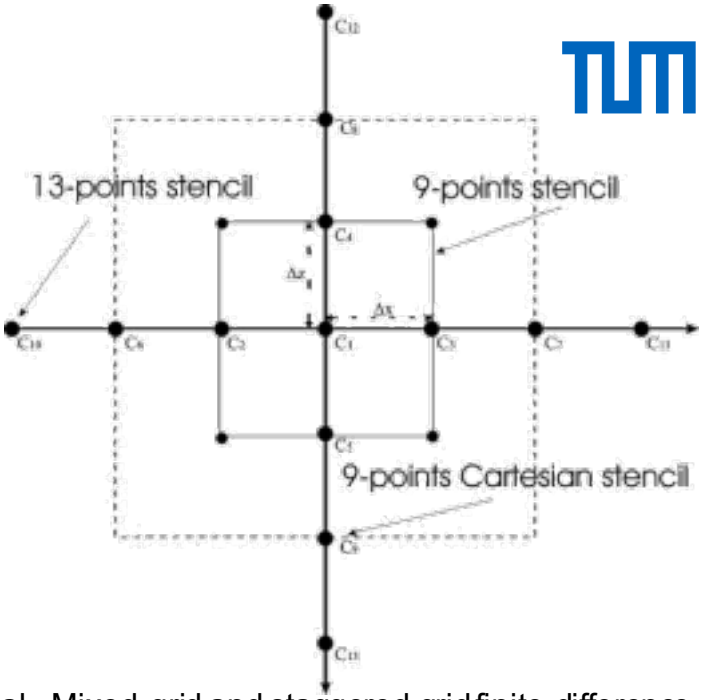


$$M(x,y) = \frac{M(x-1,y) + M(x+1,y) + M(x,y-1) + M(x,y+1)}{4}$$

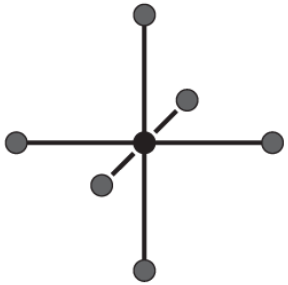
Stencils



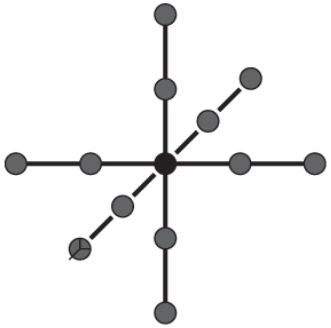
Wikipedia on Stencil code



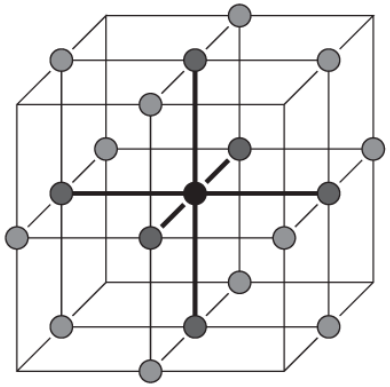
Hustedt et al., Mixed-grid and staggered-grid finite-difference methods for frequency-domain acoustic modeling
Geophysical Journal International 157(3):1269 - 1296 · June 2004



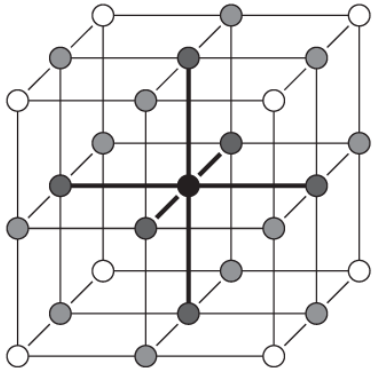
(a) 7-Point



(b) 13-Point



(c) 19-Point



(d) 27-Point

Zhang et al., Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters
IEEE TPDS, Issue No. 03 - March (2013 vol. 24)

A Simple 4/5 Point 2D Stencil Code

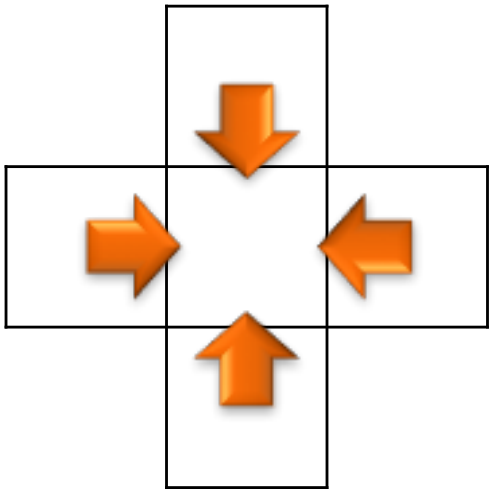
```
#define N 100
#define ITER 100

double M[N+2][N+2];
int i,j,k;

/* Set initial conditions */

/* Outer loop */
for (i=0; i<ITER; i++) {

   /* Inner Stencil Loops */
   for (j=1; j<=N; j++) {
      for (k=1; k<=N; k++) {
         M[j][k] = (M[j-1][k]+M[j+1][k]+M[j][k-1]+M[j][k+1])/4.0;
      }
   }
}
```

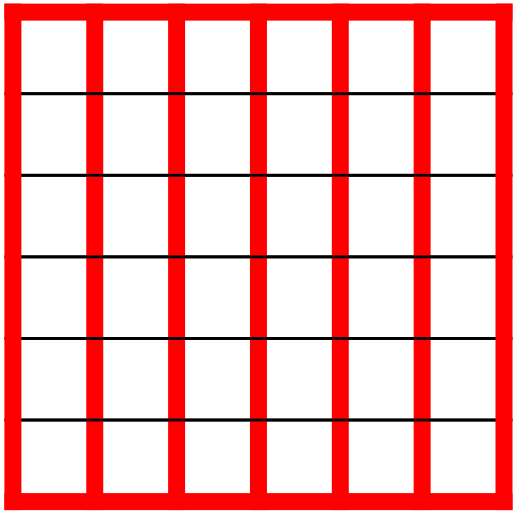


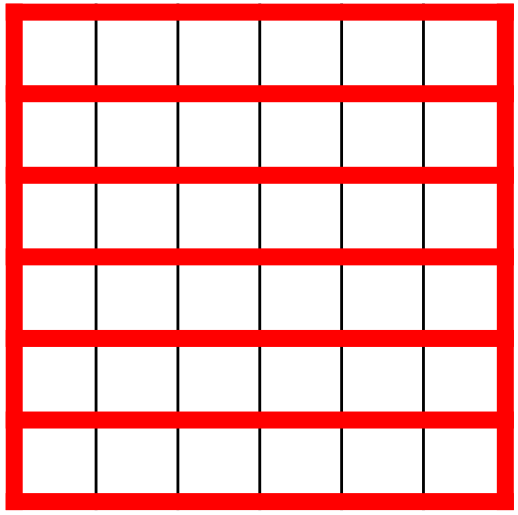
How to split up work?

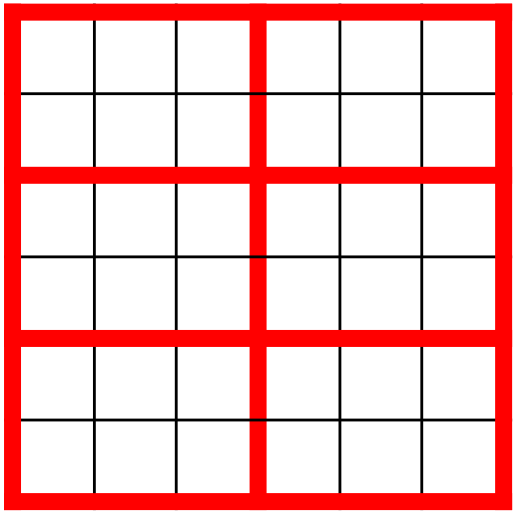
In this example
- Data is in the matrix
- Work is tied to the data
- Need to divide matrix

Multiple options
- 1D vs. 2D
- Irregular division possible

Considerations
- Number of partitions
- Surface exposure
- Easy of access
- Locality
- Granularity







How to distribute and coordinate work?

How to distribute work?

- Need to hand partitions to processing elements

How to communicate data?

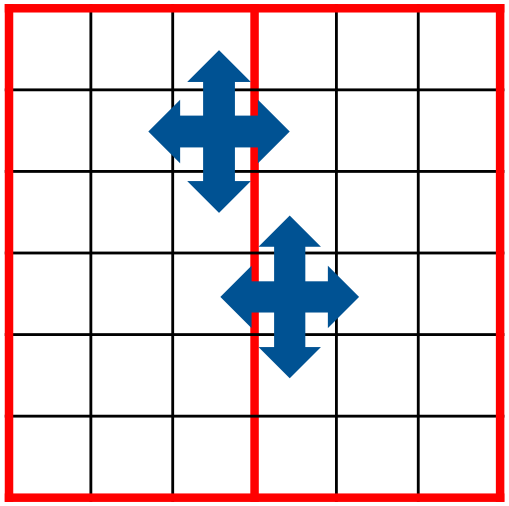
- Creation of halo zones
- Need to be communication after each step

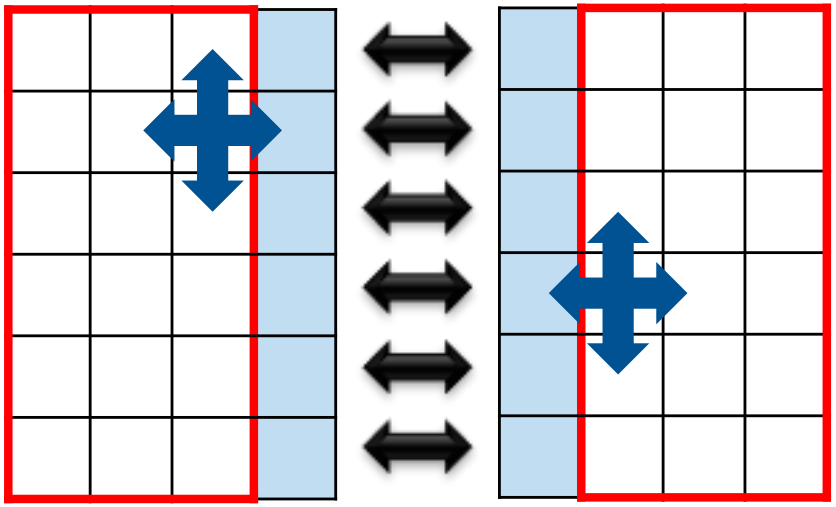
Critical question: Ordering

- In case of updates in place, ordering will change
- Is this critical?
- If so, how can we fix it?

How to coordinate work?

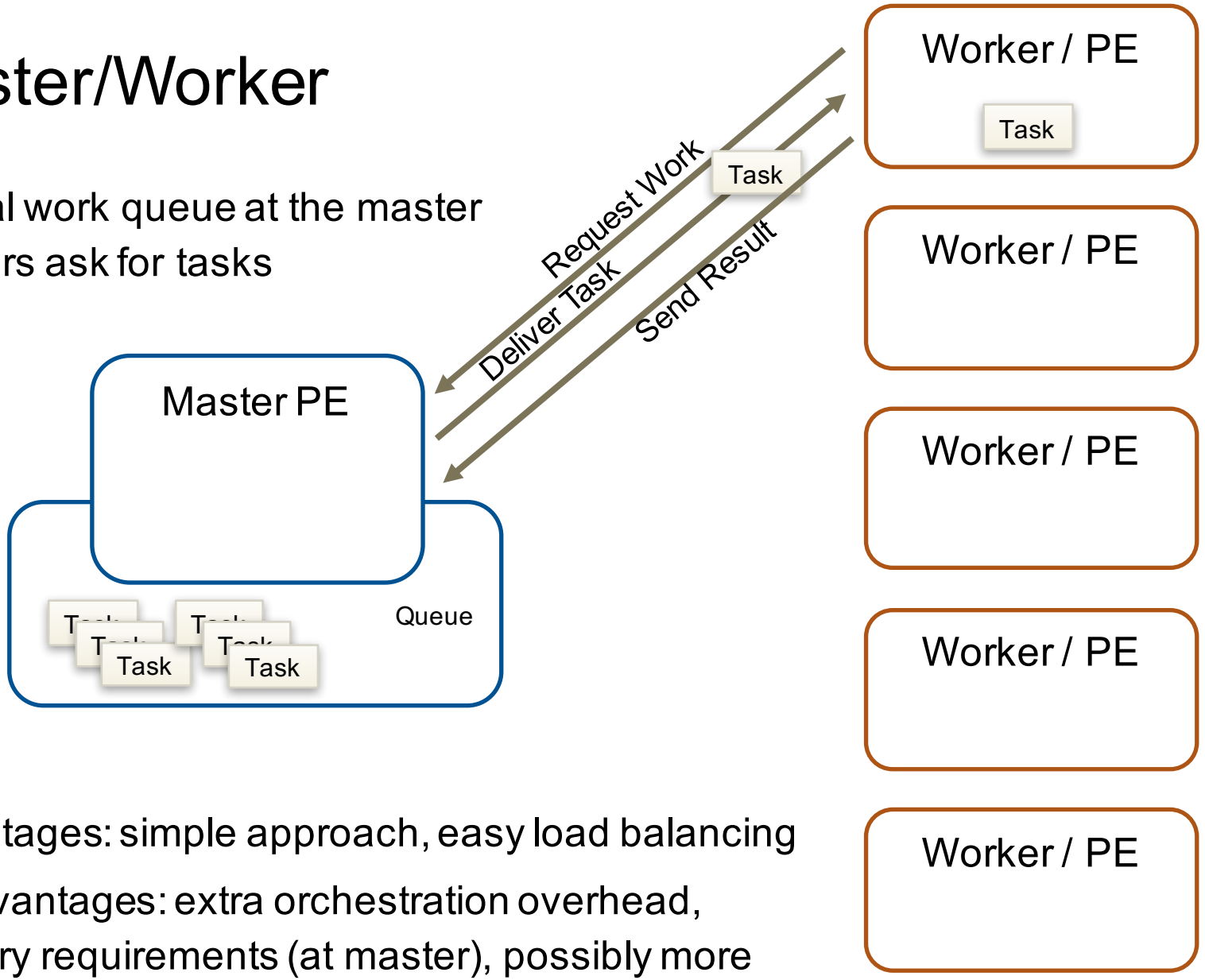
- Synchronization at iteration boundary





Master/Worker

Central work queue at the master
Workers ask for tasks

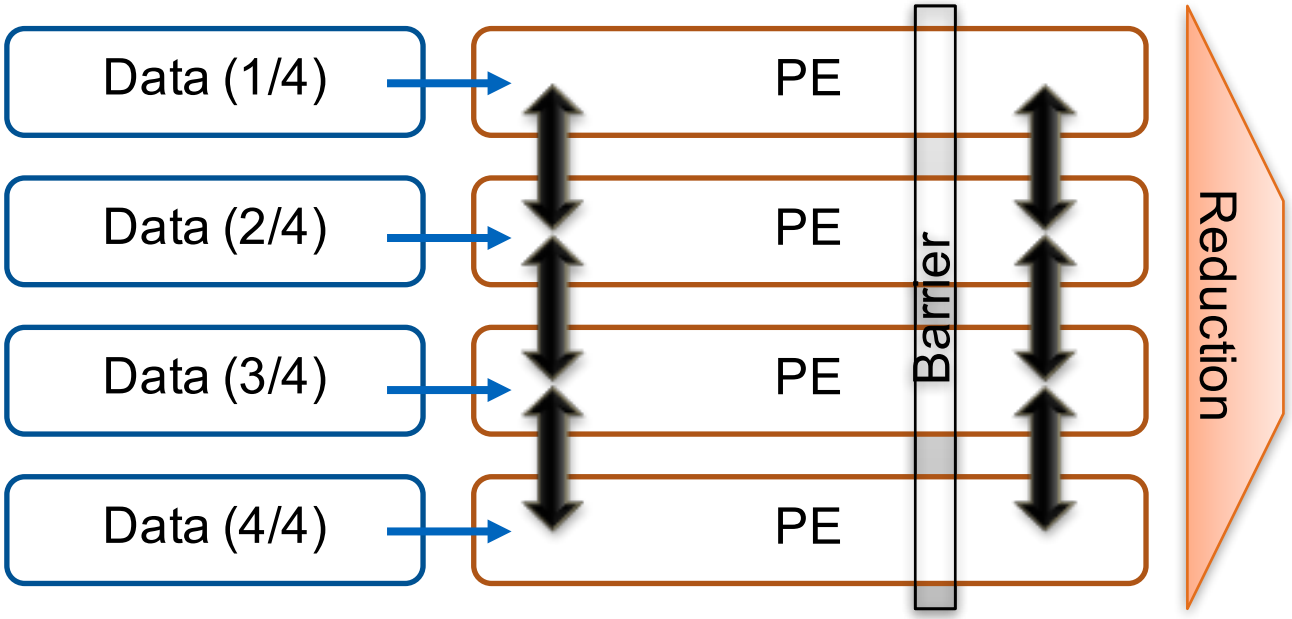


Advantages: simple approach, easy load balancing

Disadvantages: extra orchestration overhead, memory requirements (at master), possibly more communication, scalability

Multiple Concurrent Executions aka. Single Program Multiple Data (SPMD)

Multiple concurrent PEs
Distributed data
Communication between PEs

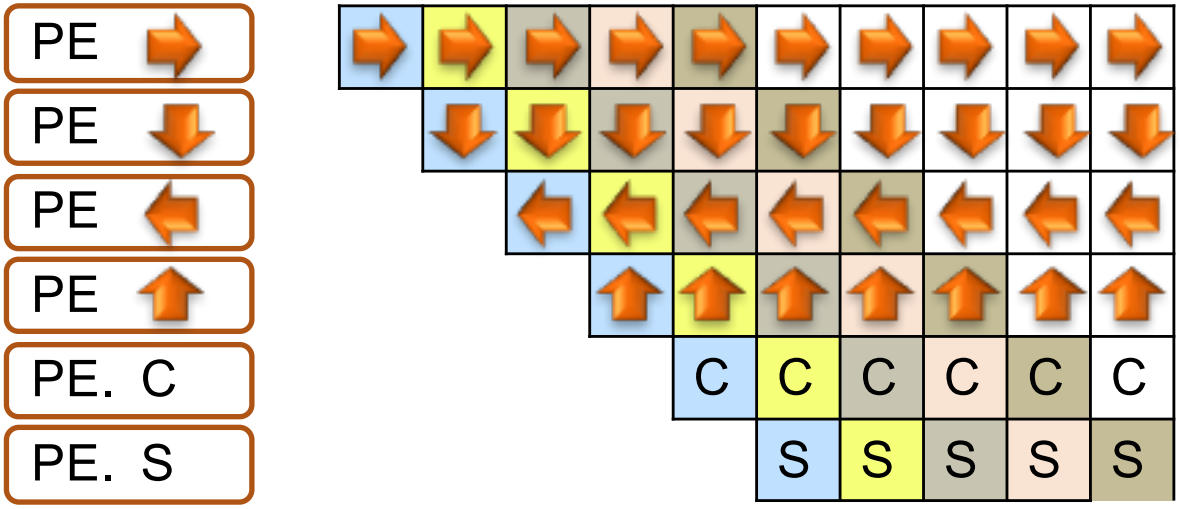


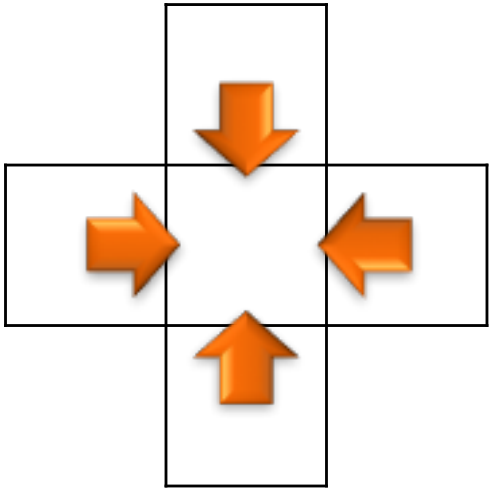
Advantages: limited orchestration overhead, explicit mapping of problem
Disadvantages: need to explicitly split data, possibility of load imbalance

Pipelining

Split functionality among PEs

Pass “task” on once functionality is done



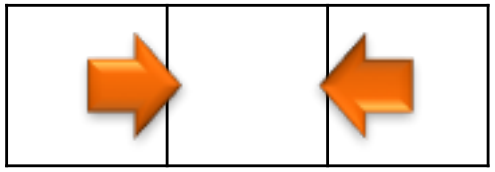


Advantages: specialized units

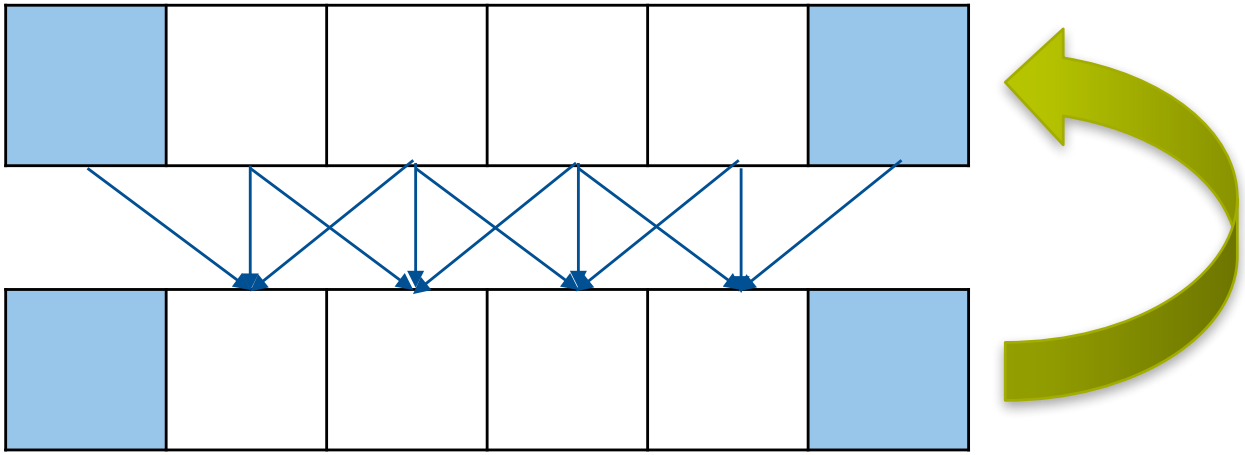
Disadvantages: more communication, limited parallelism

Arbitrary Task Dependencies

Based on dependencies for individual operations



Example in 1D



Advantages: most parallelism exposed

Disadvantages: overhead, complicated dependencies

Data Parallelism vs Functional Parallelism

Data parallelism (e.g., SMPD)

- The same operations are executed in parallel for the elements of large data structures, e.g. arrays.
- Tasks are the operations on each individual element or on subsets of the elements.
- Whether tasks are of same length or variable length depends on the application. Quite some applications have tasks of same length.

Functional parallelism (e.g., Pipelining)

- Entirely different calculations can be performed concurrently on either the same or different data.
- The tasks are usually specified via different functions or different code regions.
- The degree of available functional parallelism is usually modest.
- Tasks are of different length in most cases.

Summary: Parallel Programming Patterns

Need to decide how to distribute and communicate work and data

- Master/worker operates from a central queue
- Multiple execution streams working on separate data elements (SPMD)
- Pipelining exploits specialized units
- Task-based computing just looks at basic task dependencies

List certainly not complete

- Custom distributions can help
- Hybrid versions often useful

Has to fit to algorithm

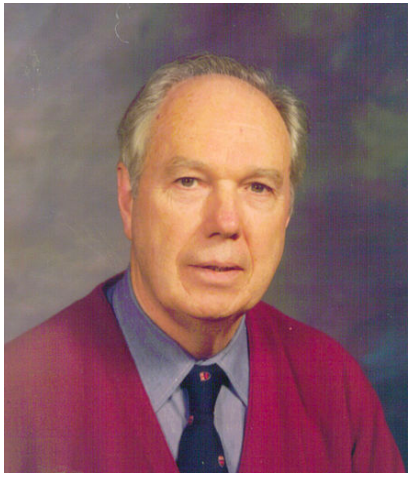
- Communication demands//requirements
- Expressability
- Granularity of tasks
- Synchronization requirements

For efficiency, need to understand mapping to hardware

Flynn's Classification

M. Flynn, Very High-Speed Computing Systems, Proceedings of the IEEE, 54, 1966

	Single Data	Multiple Data
Single Instruction	**SISD** Sequential Processing	**SIMD** Pipelines, Vectors, GPUs
Multiple Instruction	**MISD** ??? / Systolic Arrays	**MIMD** MPP Systems Clusters



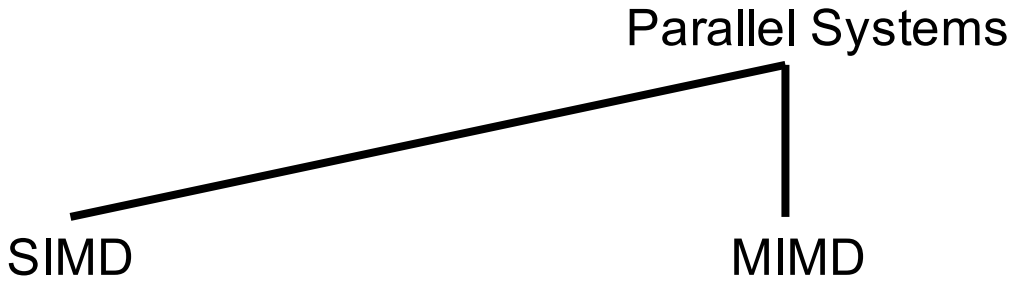
Michael Flynn
Born 1938

Relevant items covering parallel systems

- SIMD (Single Instruction Multiple Data):
  Synchronized execution of the same instruction on a set of data

- MIMD (Multiple Instruction Multiple Data):
  Asynchronous execution of different instructions

Classification

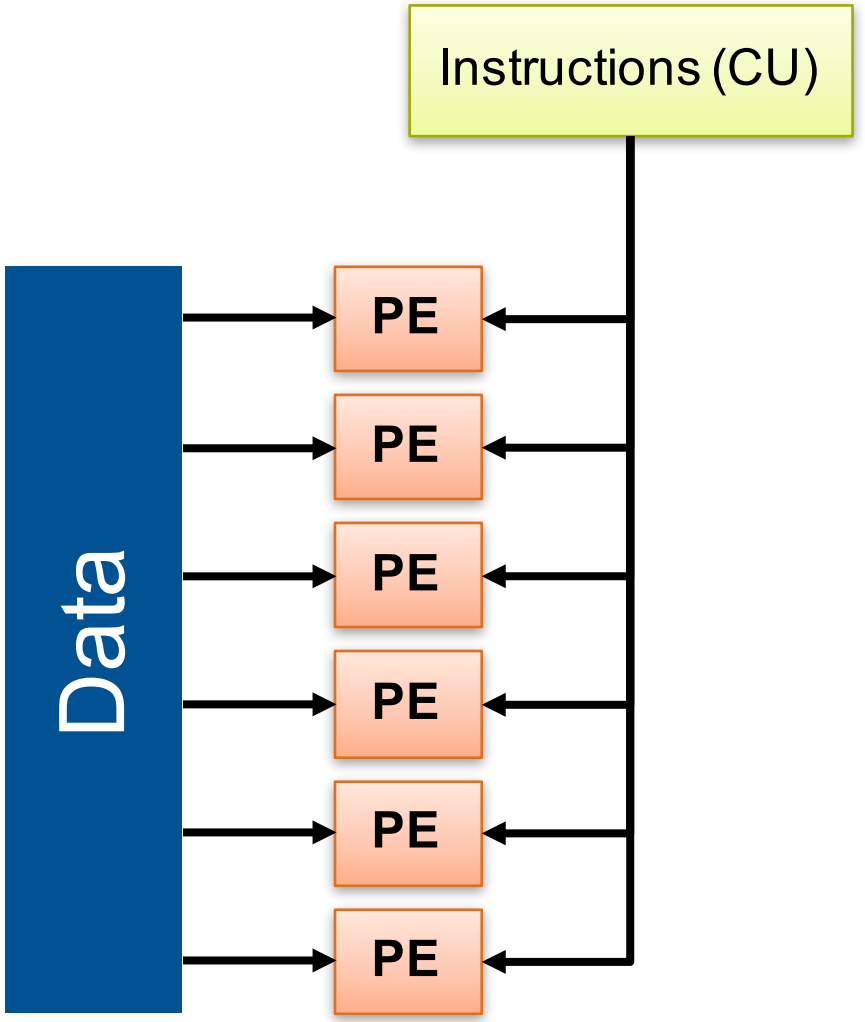


SIMD Systems

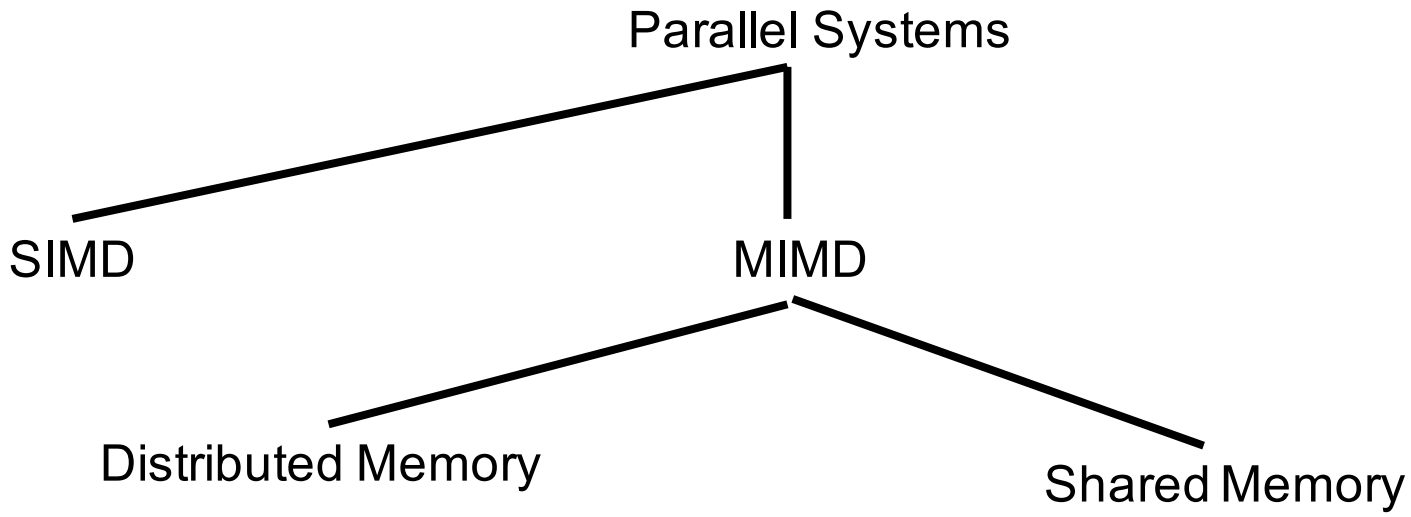
One instructions operates on a many data streams

Vector processing (see above)

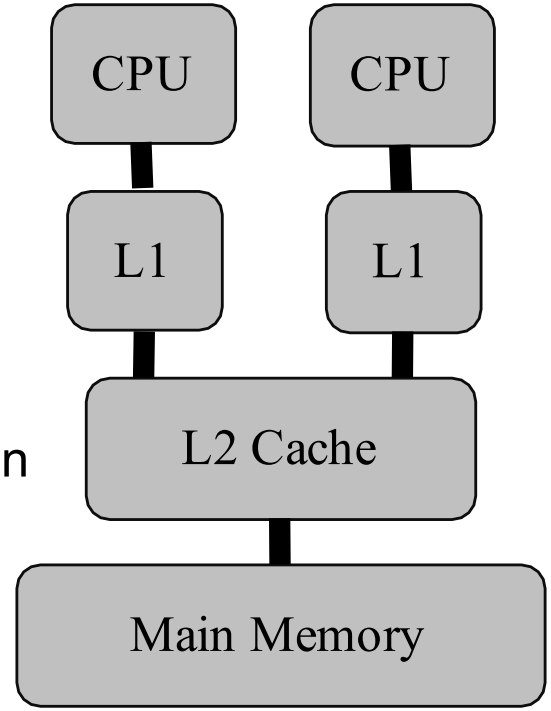
GPGPU processing fits the same model



Classification



MIMD Systems

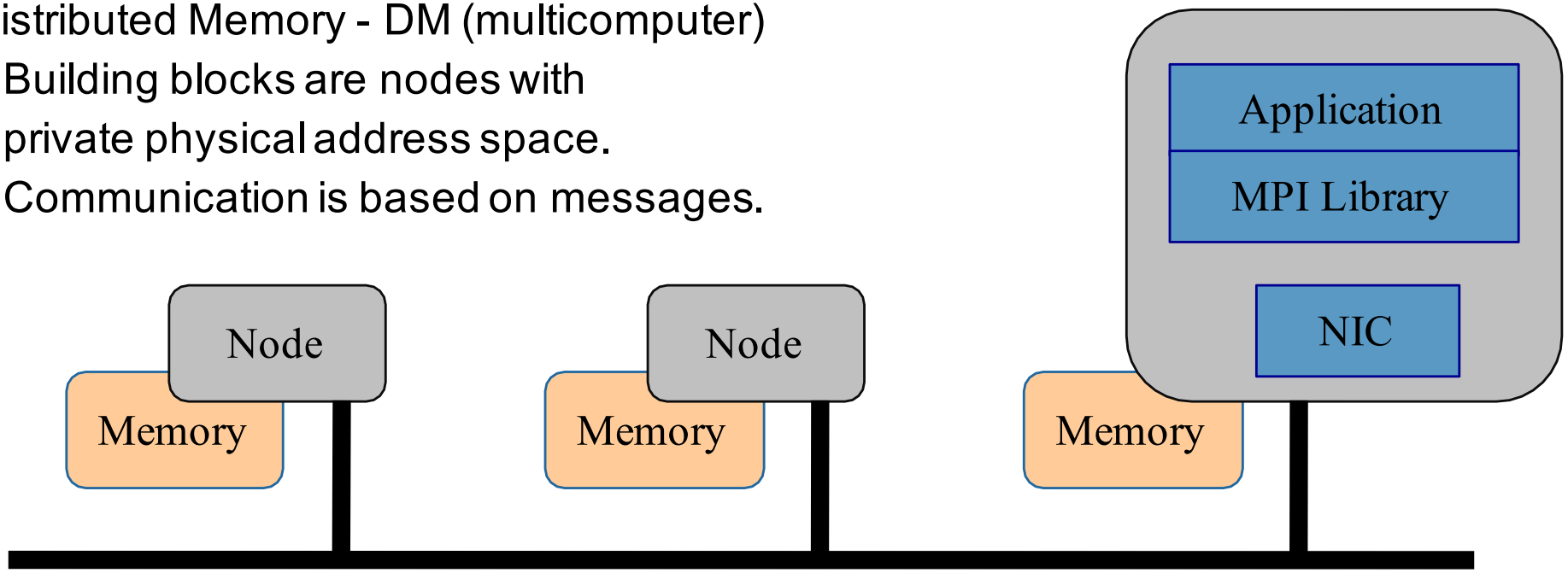


Shared Memory - SM (multiprocessor)

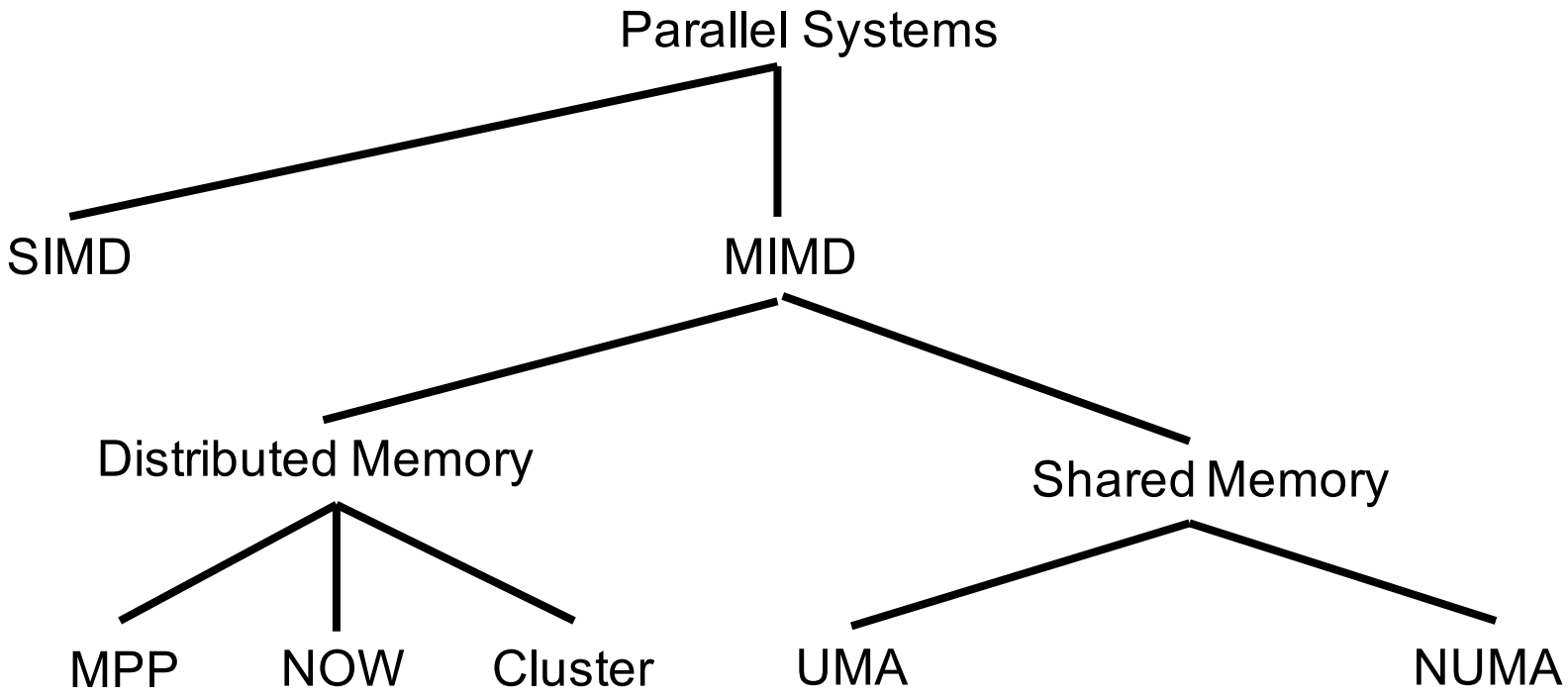
- System provides a shared address space. Communication is based on read/write operation to global addresses.

Distributed Memory - DM (multicomputer)

- Building blocks are nodes with private physical address space. Communication is based on messages.



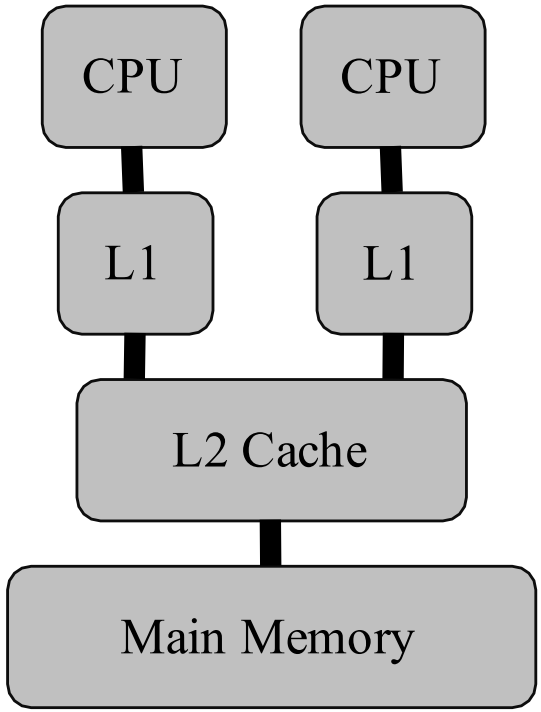
Classification



Shared Memory

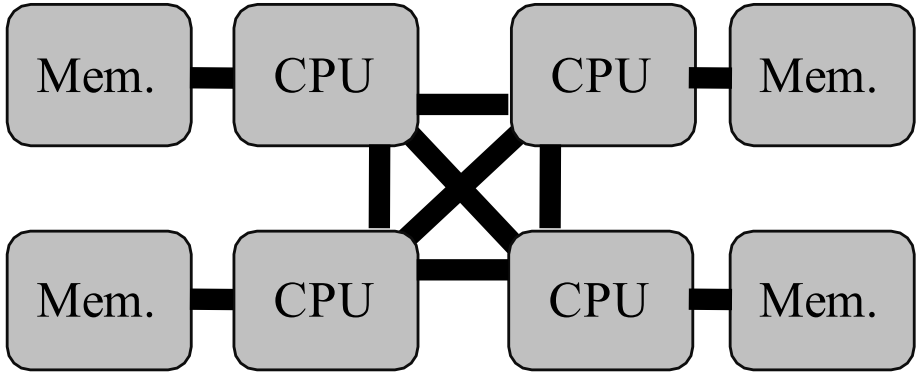
Uniform Memory Access – UMA :
(symmetric multiprocessors - SMP):

- Centralized shared memory
- Accesses to global memory from all processors have “same” latency.
- Transition from bus to crossbars



Non-uniform Memory Access Systems - NUMA
(Distributed Shared Memory Systems – HW-DSM):

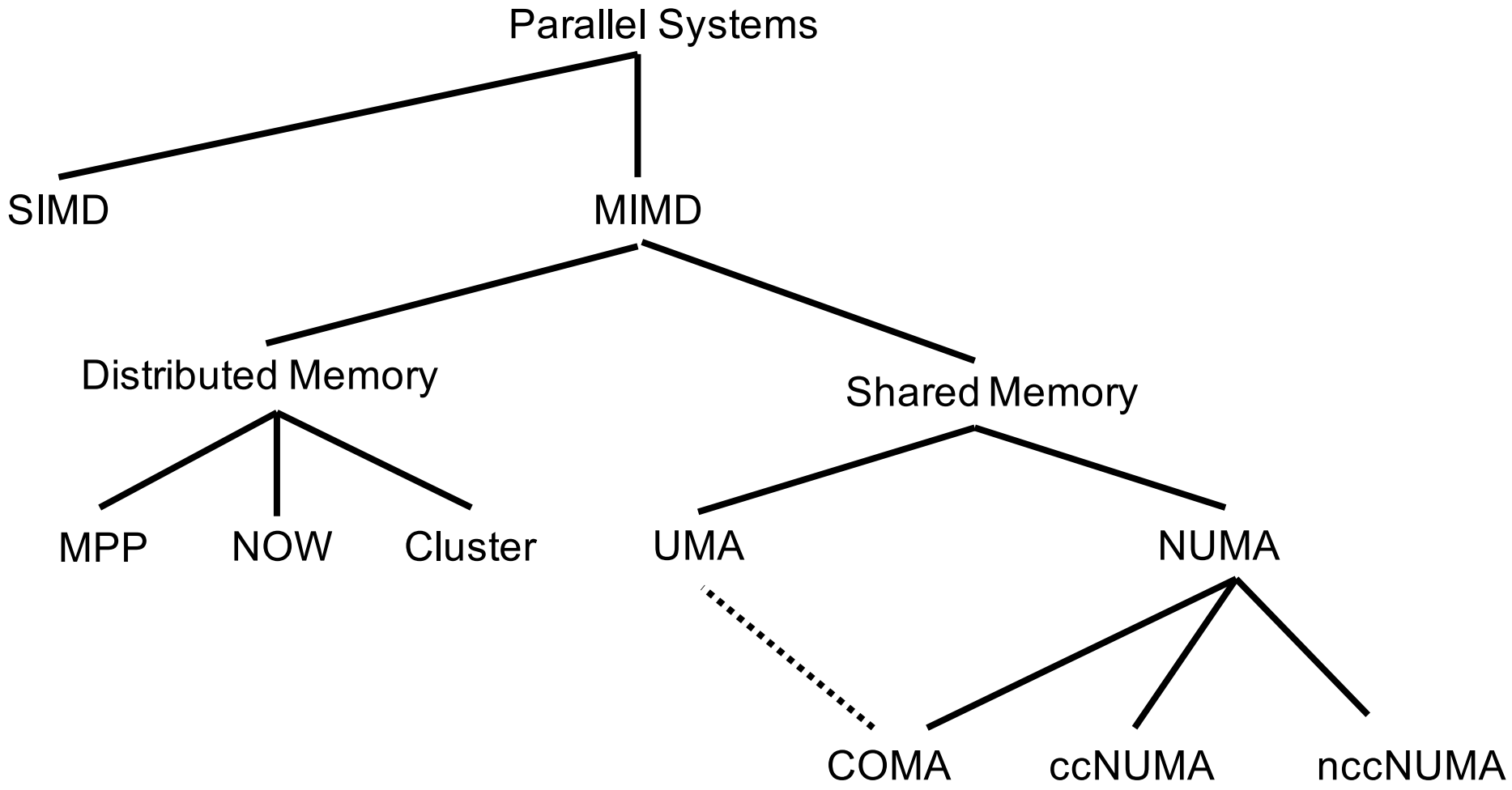
- Memory is distributed among the nodes
- Local accesses much faster than remote accesses.



More exotic

- COMA (Cacho Only)
- NCC-NUMA (non cache coherent)

Classification



Diversity in Parallel Programming Models

Driven by architecture developments (at least traditional and at the low level)

Most attached to an existing sequential programming language
- Most common: C, C++, Fortran
- Scripting languages are becoming more relevant
- APIs or language extensions

SIMD or Vector Programming
- Often in the form of pragmas (many vectorizing compilers)
- CUDA, OpenCL, ... as separate languages (but again built on base language)

Shared Memory Programming Models
- MIMD models that match shared memory architectures

Message Passing Programming Models
- MIMD models that match distributed memory architectures

Shared Memory Models Match Shared Memory

Assume a global address space with random access

- Any read/write can reach any memory cell
- This is also for NUMA systems, but locality gets tricky
- Most models assume cache coherency

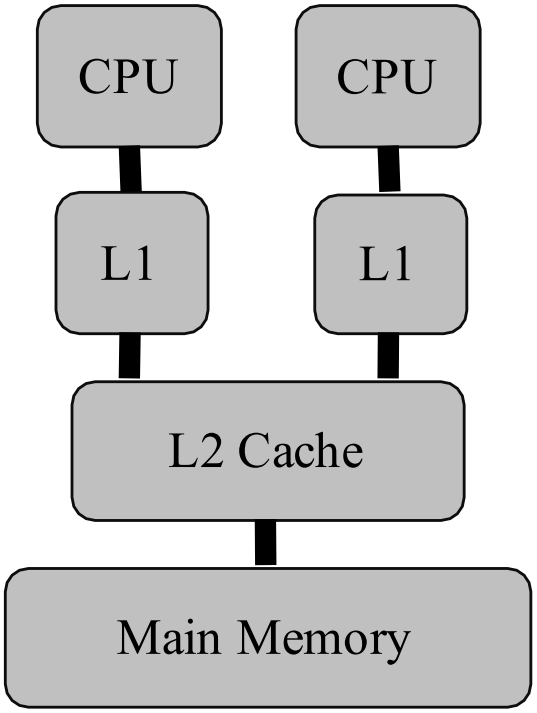
Communication through memory accesses

- Load/Store operations to arbitrary addresses
- Pass data from PE to the next

Synchronization constructs to coordinate accesses

- Need to ensure consistency
  - Data synchronization
- Need to ensure control flow
  - Control synchronization

Examples: POSIX threads, OpenMP, …



Message Passing Match Distributed Memory

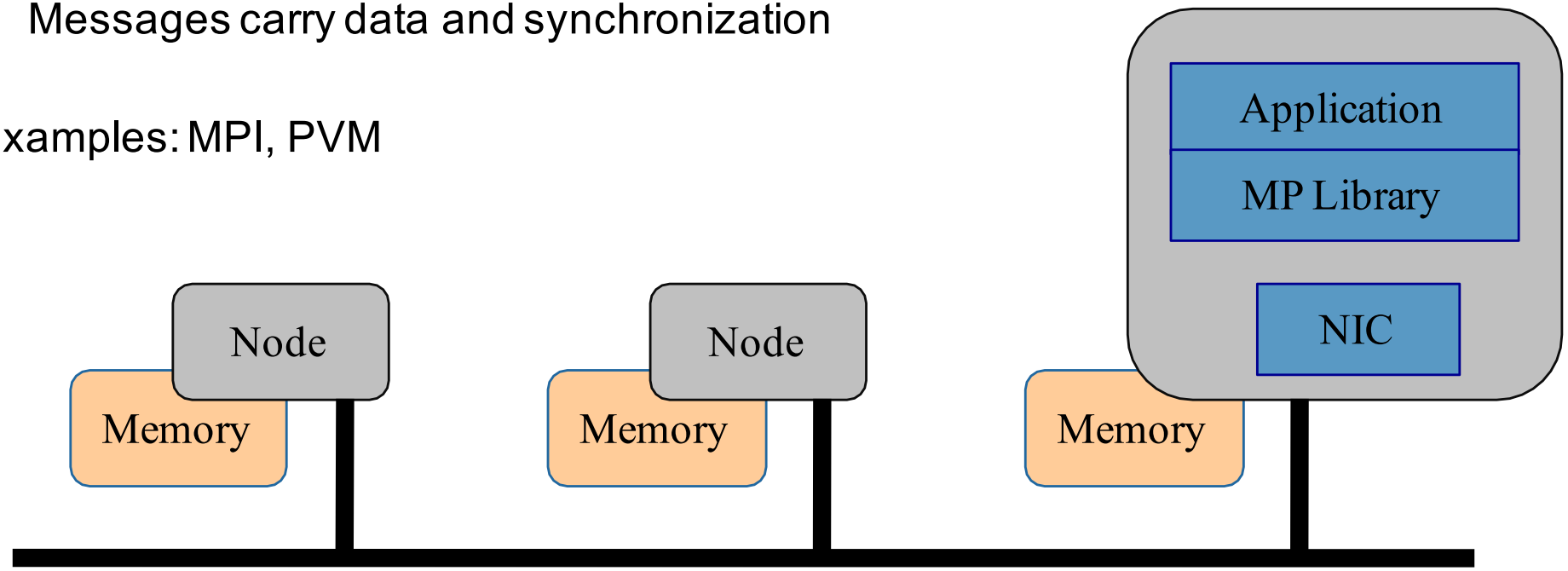
Assumes no global address space

- Independent nodes with their own memory connected via a network
- No direct visibility of data

All data communicated via messages

- Explicit Send/Recv Pairs
- Remote Memory Access (put/get) also an option
- Messages carry data and synchronization

Examples: MPI, PVM



Can we Deal with Mismatches?

Message Passing on Shared Memory Architectures

- Very common: simply not using the shared memory
- Internal implementation if message passing library uses shared memory
- Why can this be useful?
  - Portability of existing MPI programs, locality abstraction if NUMA domains

Shared Memory on Message Passing Architectures

- Problematic: fine grained accesses have to be mapped to messages
- Distributed Shared Memory
  - Hardware: hardware extensions to enable remote memory accesses
  - Software: detect accesses and forward them
  - Both have severe performance implications
  - Why can this be useful? Portability and (perceived) ease of use
- PGAS = Partitioned Global Address Space
  - Make distinction between local and remote memory visible
  - Why can this be useful: easy of use coupled with explicit locality for performance

Hybrid Programming

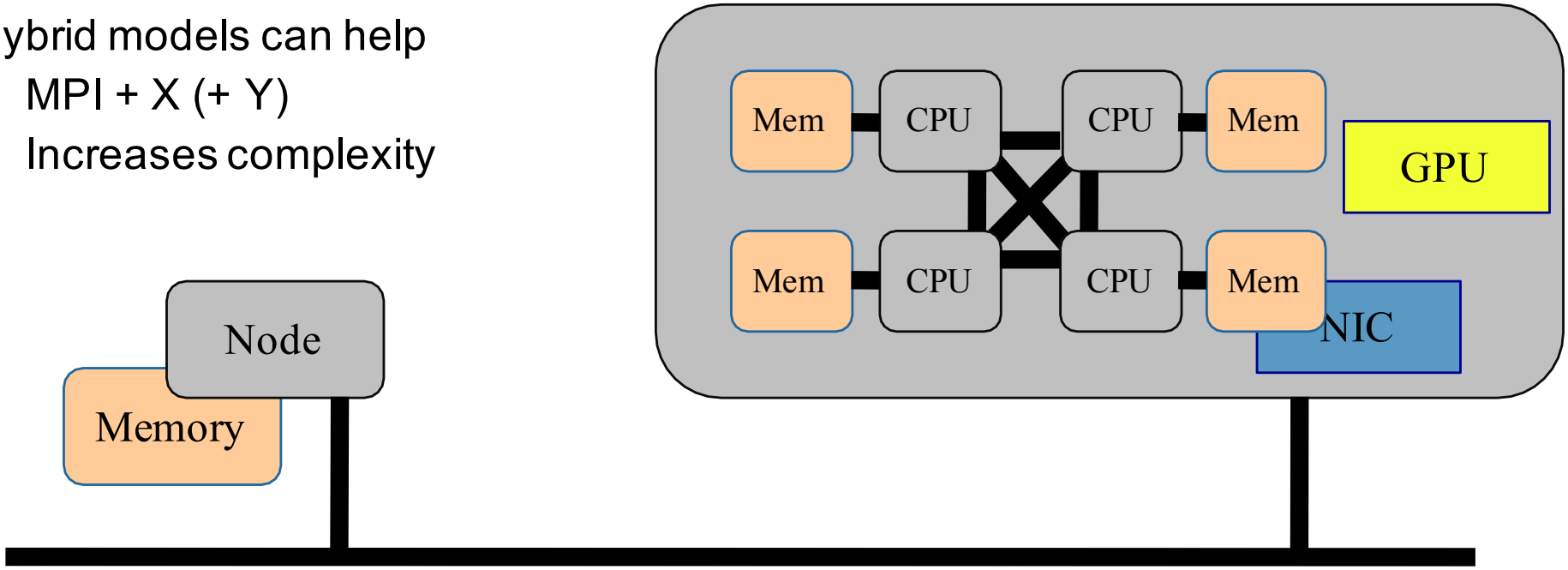
All architectures are hybrid (see multi/many core discussion earlier)

Reflected in programming models as

- Pure shared memory models don't scale beyond node
- Pure message passing models create on-node performance issues
  - Longer latency than necessary, too many message endpoints, memory. …

Hybrid models can help

- MPI + X (+ Y)
- Increases complexity



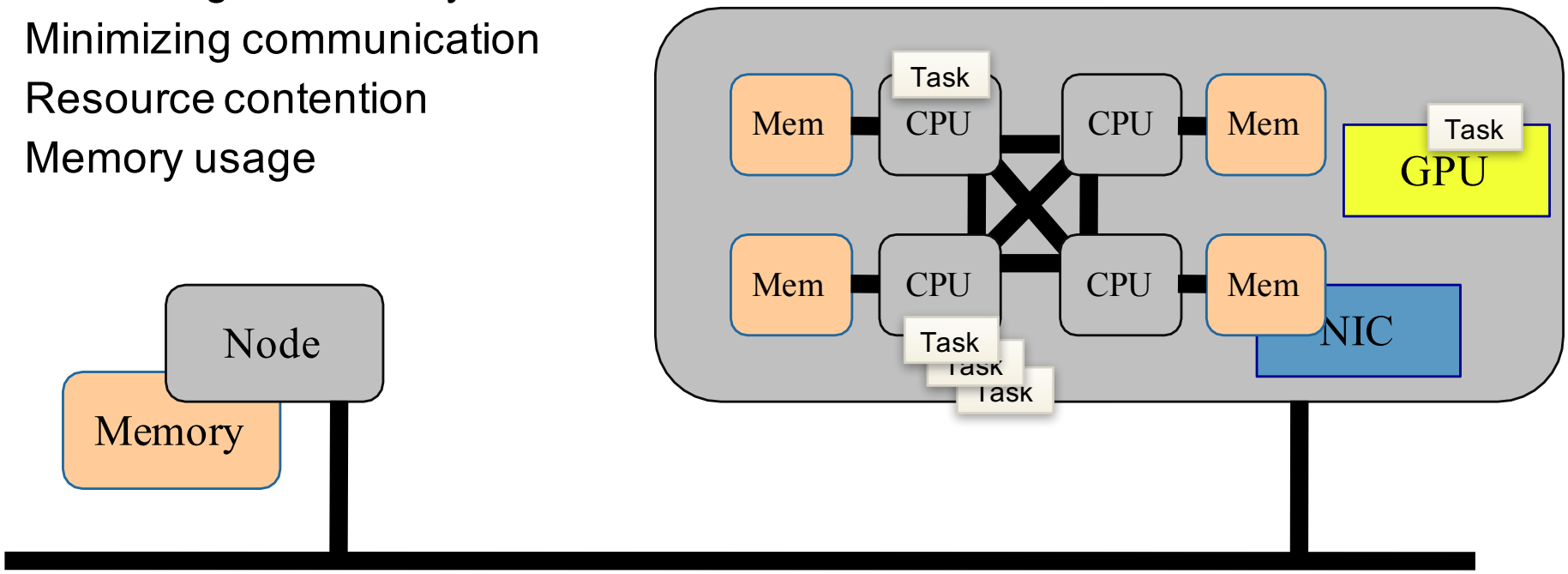
Critical Step: Mapping

Which task to execute where?

- What to express in shared memory and what in message passing?
- What to map to processes and threads?
- Where to locate processes and threads (statically or dynamically)?

Issues to consider (partial list)

- Increasing data locality
- Minimizing communication
- Resource contention
- Memory usage



Higher Level Parallel Programming Models

Automatic parallelization

- Sequential code turned by compiler into parallel program
- “The Dream of Parallel Computimg”
- Works for some examples, but generally hard to do efficiently

Loop abstractions

- Examples: RAJA, Kokkos
- Decouple loop body and loop scheduling
- Cleaner code and options for auto-tuning

Domain Specific Languages (DSLs)

- Abstract data and code structures for one domain
- Enables high-level programming with efficiency
- Limited applicability
- Typically applications specific frameworks

Summary

Parallel processing
- Multiple tasks working together to finish a (a) larger problem (b) faster
- Goal has to be efficiency

Parallelism is becoming more and more common place
- No longer niche HPC
- Multi-/Many-core developments catapult this to every system

Programming in parallel
- Decomposition of work and data using choice of best fitting pattern
- Mapping to architectures critical

Wide choice of parallel programming models
- Strong connection to underlying architectures, but not 1:1 match
- Hybrid programming is becoming the norm

Think parallel!