



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Prof. Dr. Felix Wolf

LARGE-SCALE PARALLEL COMPUTING



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Large-Scale Parallel Computing

Prof. Dr. Felix Wolf

ORGANIZATION

Team



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Lecturer



Prof. Dr.
Felix Wolf

Organization



Sebastian
Rinke

Exercises



Aamer
Shah

Email: <last name>@cs.tu-darmstadt.de

General information



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Please register for this class via TUCaN

- Makes it easier for us to notify you of changes
- Gives you access to the course material in Moodle

Schedule

- Lecture - Thursday, 9:50h - 11:30h
- Exercise - Tuesday, 13:30h - 15:20h (biweekly, starting Oct 27)

Questions regarding organization

- Sebastian Rinke (rinke@cs.tu-darmstadt.de)

General information (2)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Slides will be available in PDF in Moodle

- Only for the purpose of this class
- Redistribution not permitted

Requirements

- Knowledge of C

Exercises



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Task sheets available in Moodle
- Solutions will be presented during the exercise
- No grading of exercises
- Students are encouraged to collaborate
- Contact for questions provided on exercise sheet
- First exercise sheet already available
- Exercise platform: Lichtenberg cluster
 - More details soon

Contents of this course



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Parallel programming for distributed memory architectures

- Distributed-memory architectures
- Foundations of message passing
- Collective operations
- Data types
- Remote memory access
- Hybrid programming
- Parallel I/O
- Partitioned global address space

Coverage as far
as we get...

C refresher exercise



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Dates

- Tuesday, October 27

Learning objectives



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Understand, design, implement, and optimize parallel
programs for distributed-memory architectures

Philosophy



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Interactive style – physical presence strongly recommended but not monitored
- No full coverage of programming standards – rather in-depth study of key concepts
- Sound track not always mirrored on slides – please take notes or rely on books for reference

Exam



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Written or oral test

- Depending on number of registrations

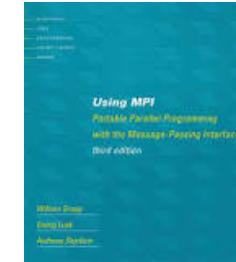
Q & A session during last week of lecture

Literature



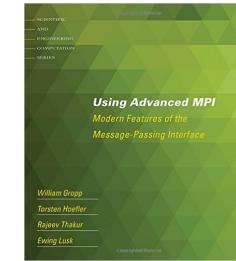
Using MPI

- William Gropp, Ewing Lusk, Anthony Skjellum, 3rd edition, MIT Press, 2014



Using Advanced MPI

- William Gropp, Torsten Hoefler, Rajeev Thakur, MIT Press, 2014

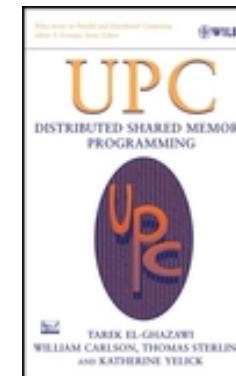


MPI standard

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

UPC: Distributed Shared Memory Programming

- Tarek El-Ghazawi, William Carlson, Thomas Sterling, Katherine Yellick, Wiley, 2005



Appointments



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Upon request <wolf@cs.tu-darmstadt.de>



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Prof. Dr. Felix Wolf

LABORATORY FOR PARALLEL PROGRAMMING

Introduction



- Recently moved from Aachen to Darmstadt
- Research statement

“Develop methods, tools, and infrastructure to exploit massive parallelism on modern computer architectures”



www.parallel.informatik.tu-darmstadt.de

Team



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Prof. Dr.
Felix Wolf



Alexandru
Calotoiu



David
Giessing



Dr. Daniel
Lorenz



Suraj
Prabhakaran



Sebastian
Rinke



Laura
von Rüden



Aamer
Shah



Sergei
Shudler

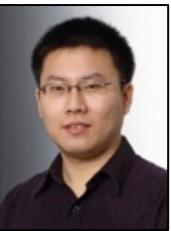
Multicore Programming Group



Dr. Ali
Jannesari



Rohit
Atre



Zhen
Li



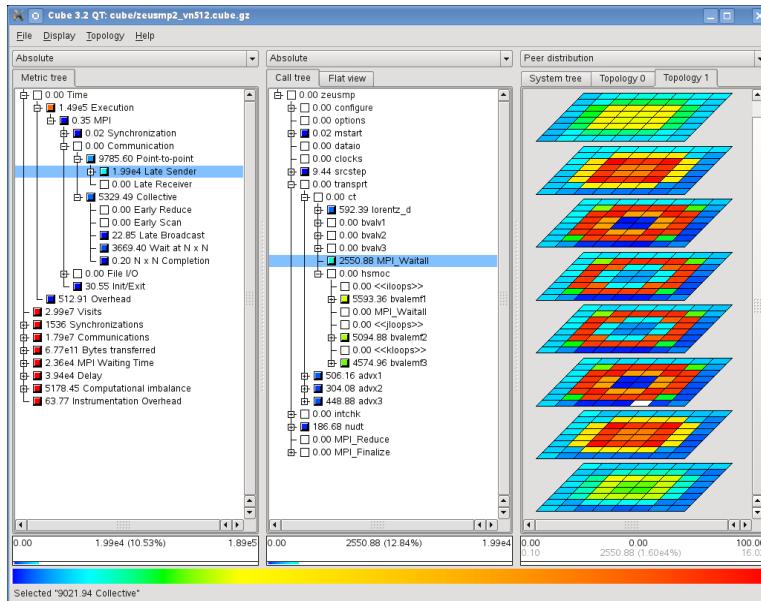
Zia
Ul Huda



Mohammad
Norouzi



Arya
Mazaheri

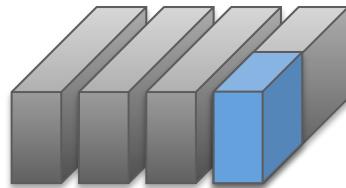


COURTESY: FORSCHUNGZENTRUM JÜLICH

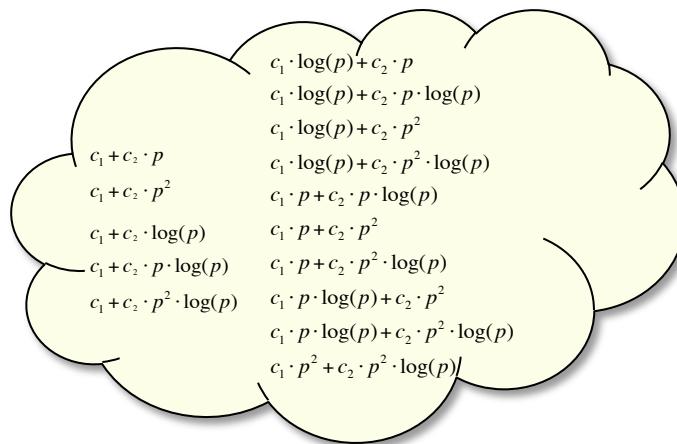
IBM Blue Gene/Q in Jülich

- Performance analysis tool for HPC applications
- Collaboration with Jülich Supercomputing Centre
- Our focus: automated performance modeling & visual analytics
- www.scalasca.org

Automatic empirical modeling



Small-scale measurements



Generation of candidate models
and selection of best fit

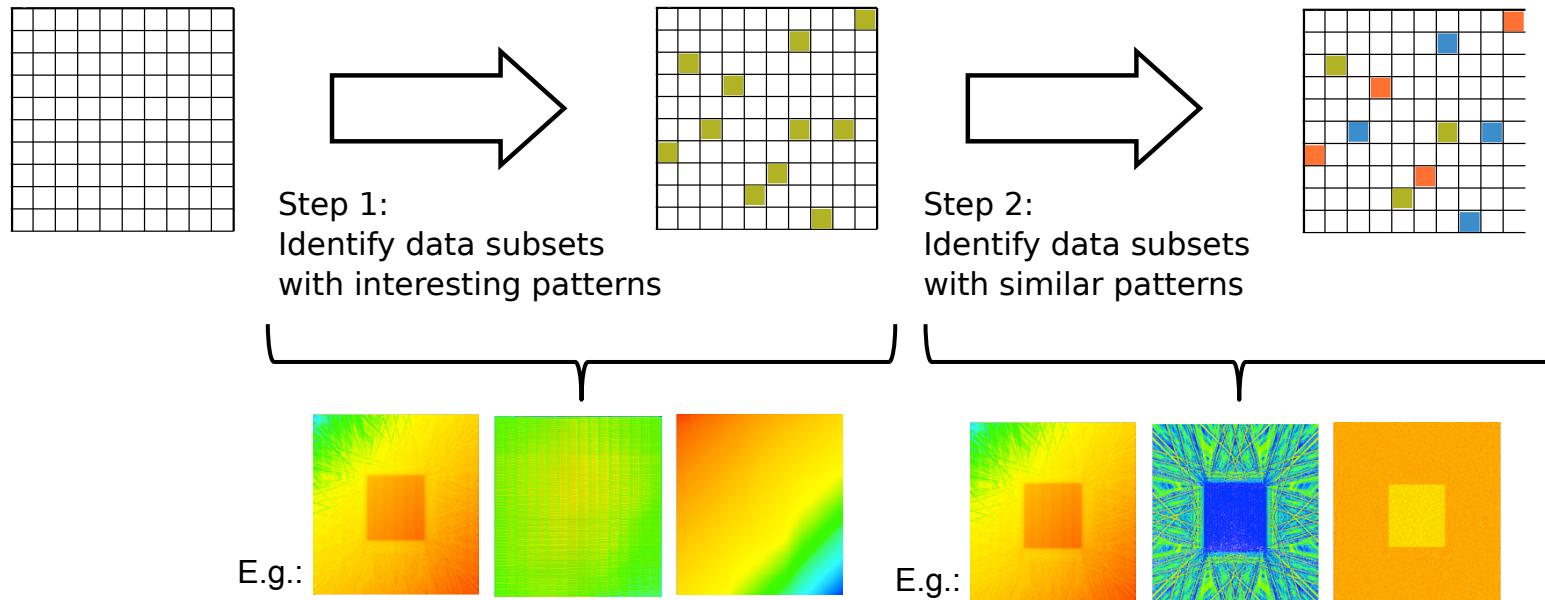
$$f(p) = \sum_{k=1}^n c_k \cdot p^{i_k} \cdot \log_2^{j_k}(p)$$

Performance model normal form (PMNF)

Kernel [2 of 40]	Model [s] $t = f(p)$	Error [%] $p_t=262k$
sweep → MPI_Recv	$4.03\sqrt{p}$	5.10
sweep	582.19	0.01

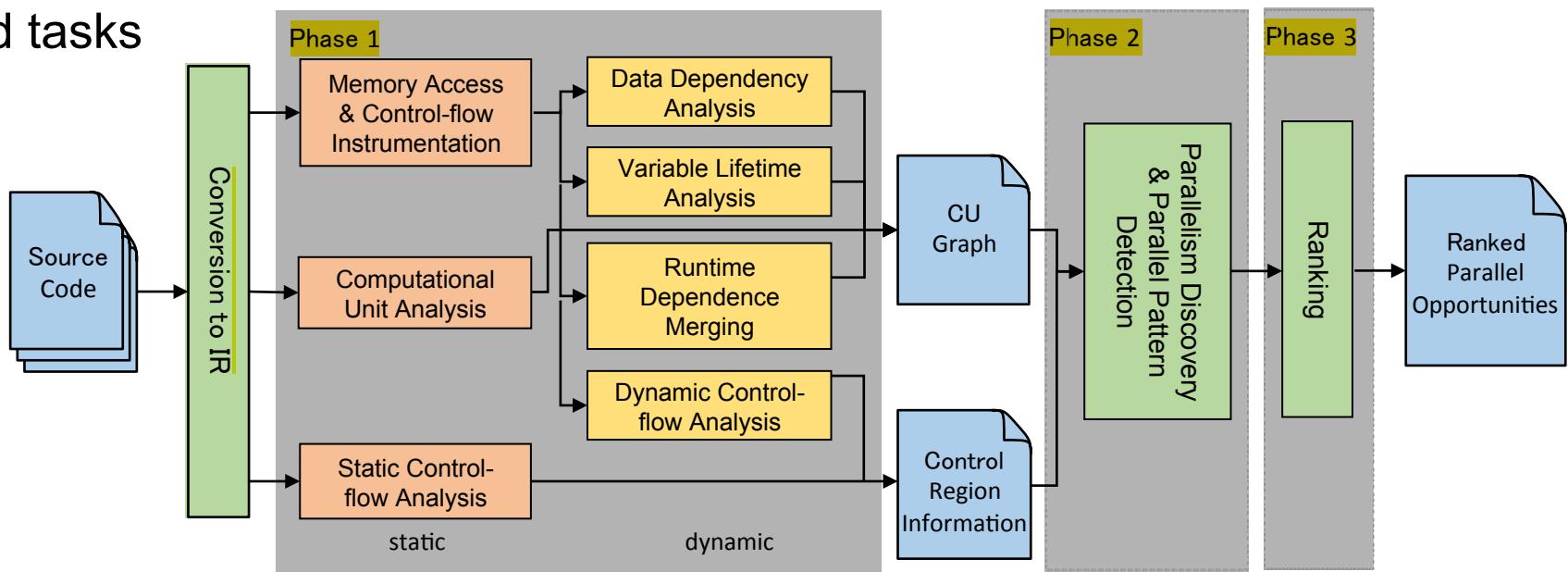
Visual analytics of performance data

Visual analytics: Visual data exploration + automatic data analysis¹



1. Daniel Keim, Jörn Kohlhammer, Geoffrey Ellis, and Florian Mansmann: Mastering the Information: Age Solving Problems with Visual Analytics. Eurographics Association, 2010.

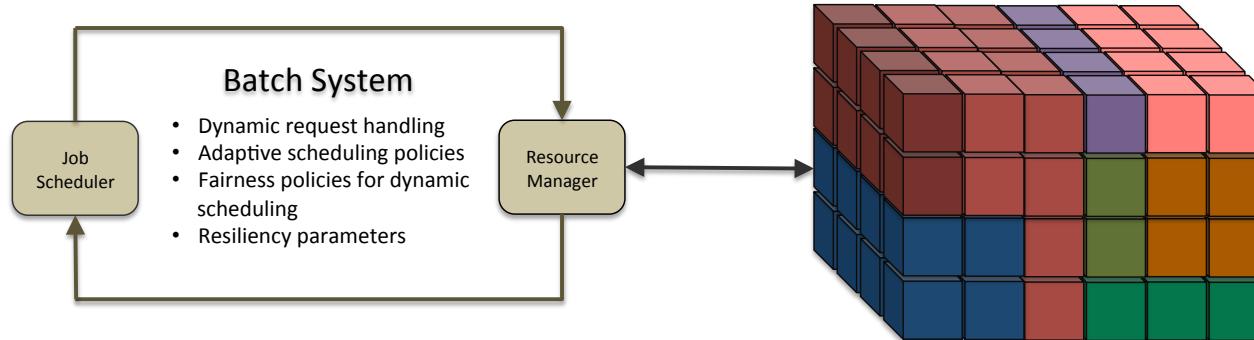
- Discovers potential parallelism in sequential programs
- Targets DOALL loops, pipelines, and tasks
- Reveals data dependences that prevent parallelization
- Efficient in time and space



Scheduling

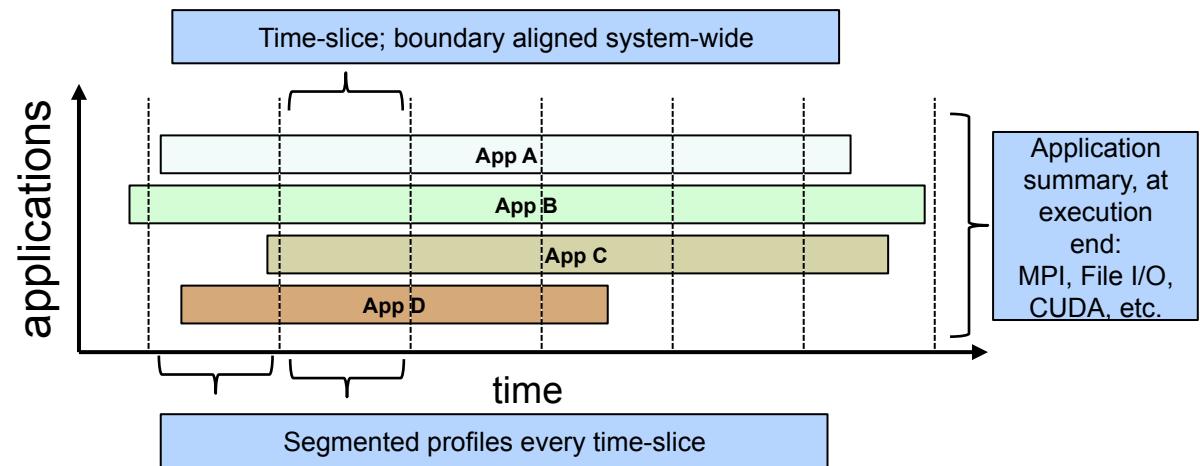


- Dynamic resource management and job scheduling in HPC
- Support for more classes of applications - moldable, malleable and evolving
- Adaptive scheduling with enhanced fairness for high throughput
- Fault tolerance with dynamic node replacement



- Light-Weight Monitoring Module
- Profiles: MPI, File I/O, OpenMP and CUDA
- Easy to use: No code recompilation or relinking
- Generates simple command line output and data files
- Geared towards identifying performance interference between jobs

- Synchronized, time-stamped, periodic profiles across multiple applications



Scalable brain simulation

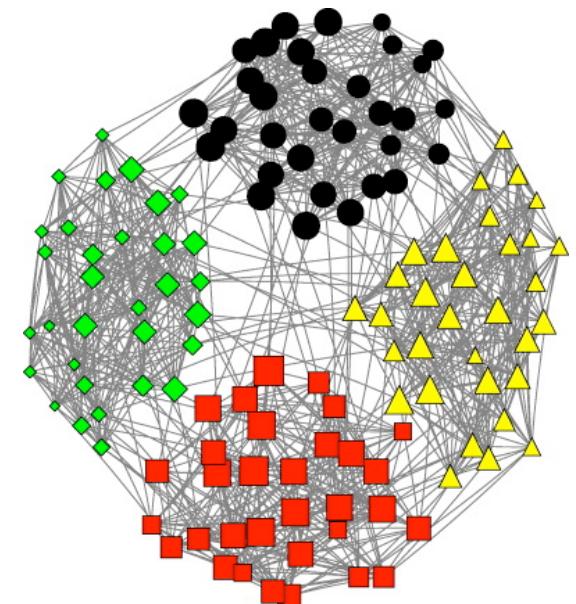


- Collaboration with SimLab Neuroscience at Forschungszentrum Jülich
- Neuronal network of human brain not “hard-wired”
- Lesions, e.g. stroke, cause reorganization of connections
- Goal:
 - Develop biologically realistic full-scale network model of the brain
 - Better understand the dynamics of the network
- Algorithmic challenge:
 - How to handle 10^{11} neurons in the simulation



Large-scale word sense disambiguation

- Collaboration with Ubiquitous Knowledge Processing Lab at TU Darmstadt
- A word can have many different meanings depending on its context
- Goal:
 - Use supercomputer to disambiguate all words in large text collections
 - Create basis for further semantic analysis
- Algorithmic challenge:
 - Processing large texts requires much file I/O
 - Minimize slow file I/O through efficient use of main memory and network communication



Student assistant positions



TECHNISCHE
UNIVERSITÄT
DARMSTADT

The position offers:

- Research and development in the area of programming tools for parallel computing
- Experience in parallel programming
- Working in an international team
- The option to prepare for a master's thesis
- Negotiable number of hours per week

The ideal candidate will have:

- A bachelor's degree in computer science or a related discipline
- Programming practice in C/C++
- Familiarity with UNIX-like system environments
- Good command of English
- High motivation and the ability to work effectively with others

Additional qualifications:

- Knowledge of parallel programming

If you are interested,
please contact Prof. Wolf
[<wolf@cs.tu-darmstadt.de>](mailto:wolf@cs.tu-darmstadt.de)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

THANK YOU!



Large-Scale Parallel Computing

Prof. Dr. Felix Wolf

INTRODUCTION

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Why parallel computing?
- Why parallel programming?
- Types of parallelism
- Developing parallel software
- Obstacles to parallelism
- Preview of course contents

Why (large-scale) parallel computing?



Problems that cannot be solved fast enough sequentially

- Example: simulation of physical phenomena
- Based on the idea that larger problems can be divided into subproblems to be solved concurrently
- Two dimensions
 - Problem size
 - Time to solution
- Sometimes real-time constraints
(e.g., weather forecast)
- Usually requires parallel computer / multiprocessor

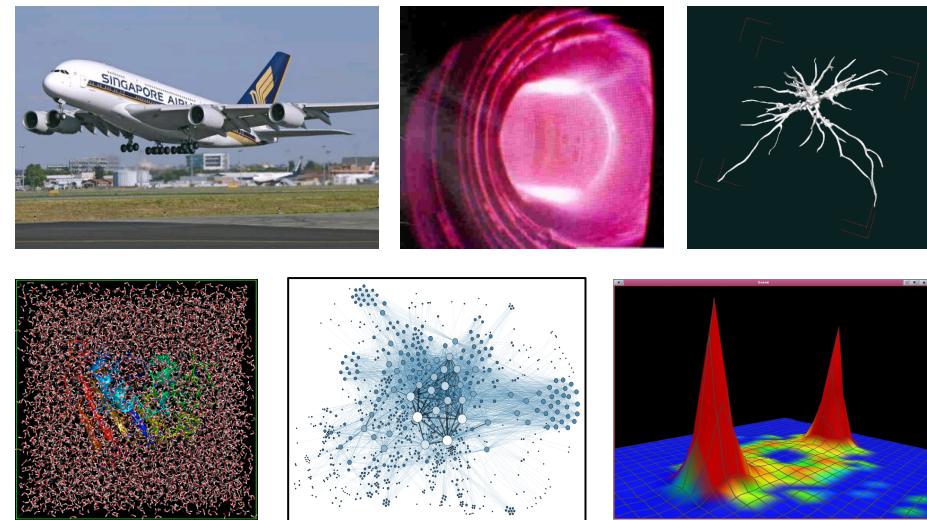


Hurricane Katrina

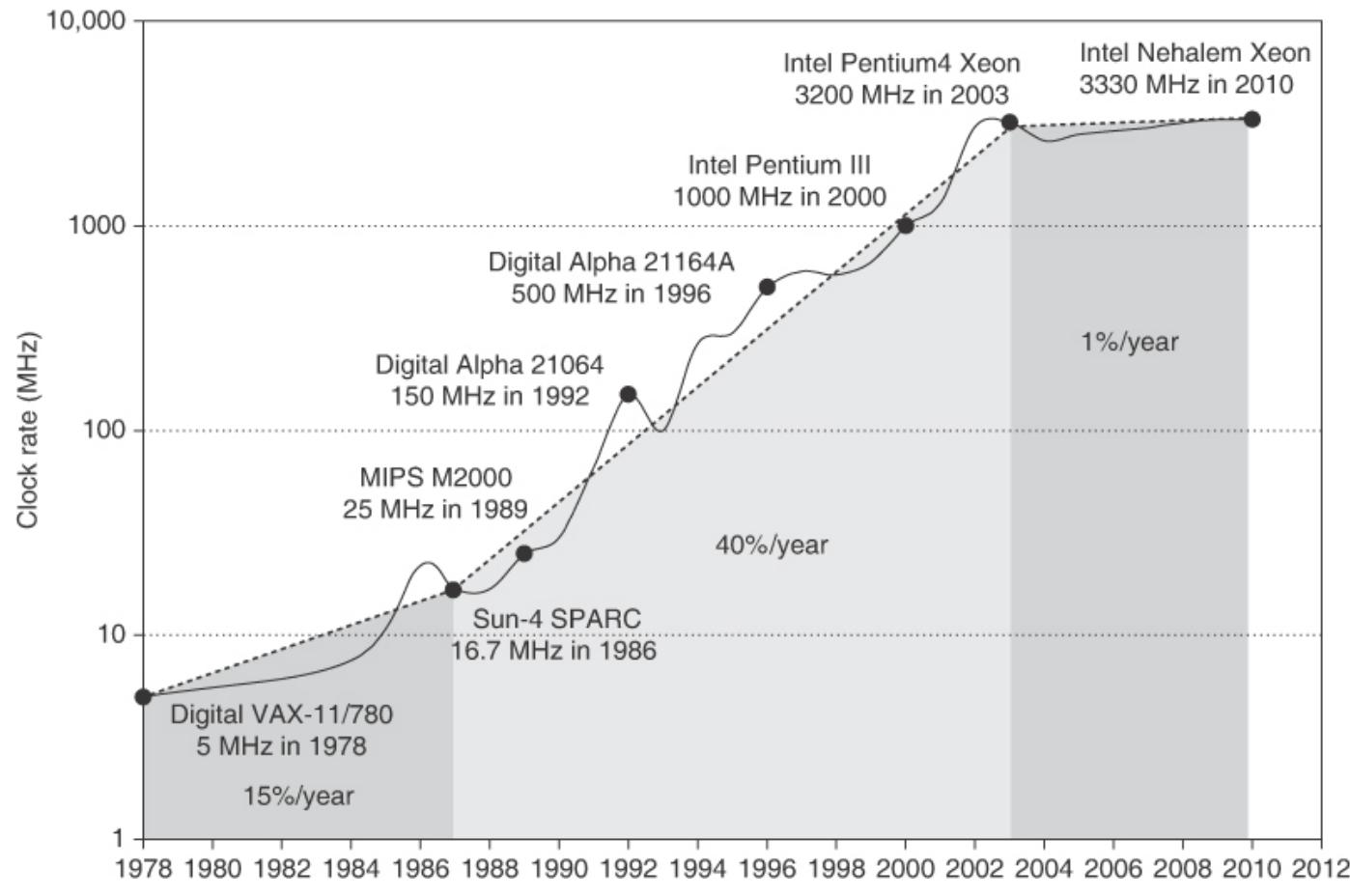
Further examples of compute-intensive application



- Simulations
 - Natural sciences: molecular dynamics, materials science
 - Engineering: crash, aerodynamics, fluid dynamics, combustion
- BigData
 - Graph analysis
 - Sorting
- Multimedia
 - Stream processing
- Finance
 - Valuation of options and financial securities



Why can't we just create faster uni-processors?



Source: Hennessy, Patterson: Computer Architecture, 5th edition, Morgan Kaufmann

Moore's law



The number of transistors per chip doubles roughly every **two years**

Exponential growth of processor performance

VISUALIZING PROGRESS

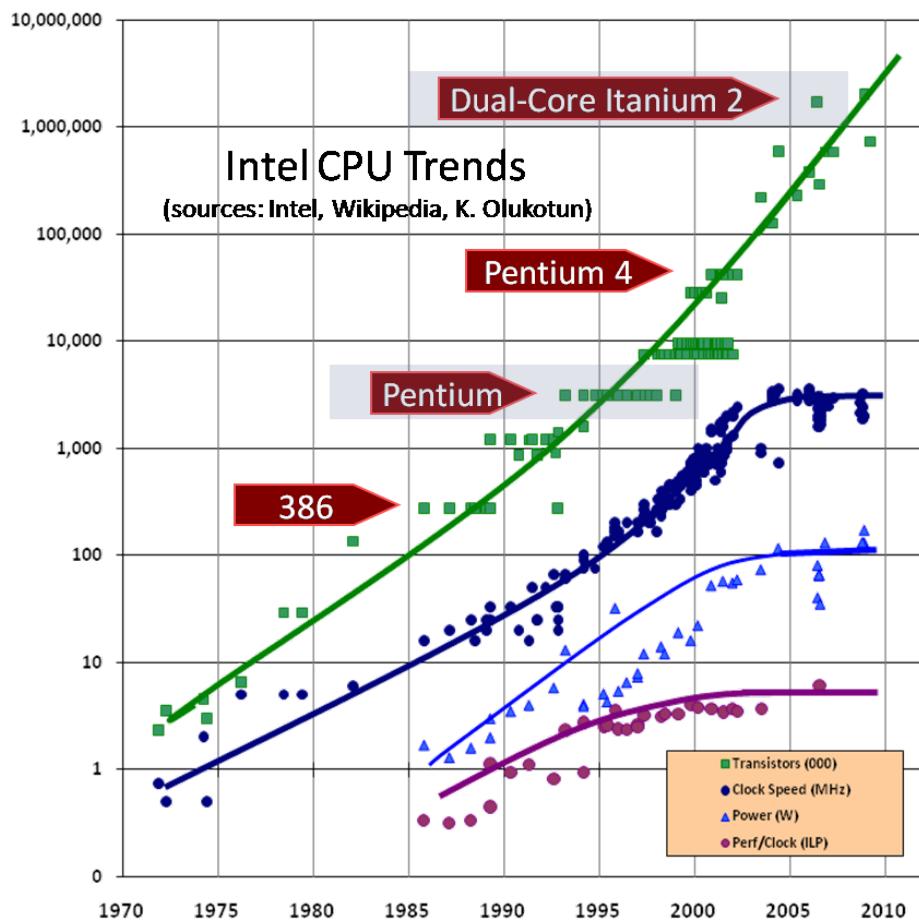
If transistors were people

If the transistors in a microprocessor were represented by people, the following timeline gives an idea of the pace of Moore's Law.



Now imagine that those 1.3 billion people could fit onstage in the original music hall. That's the scale of Moore's Law.

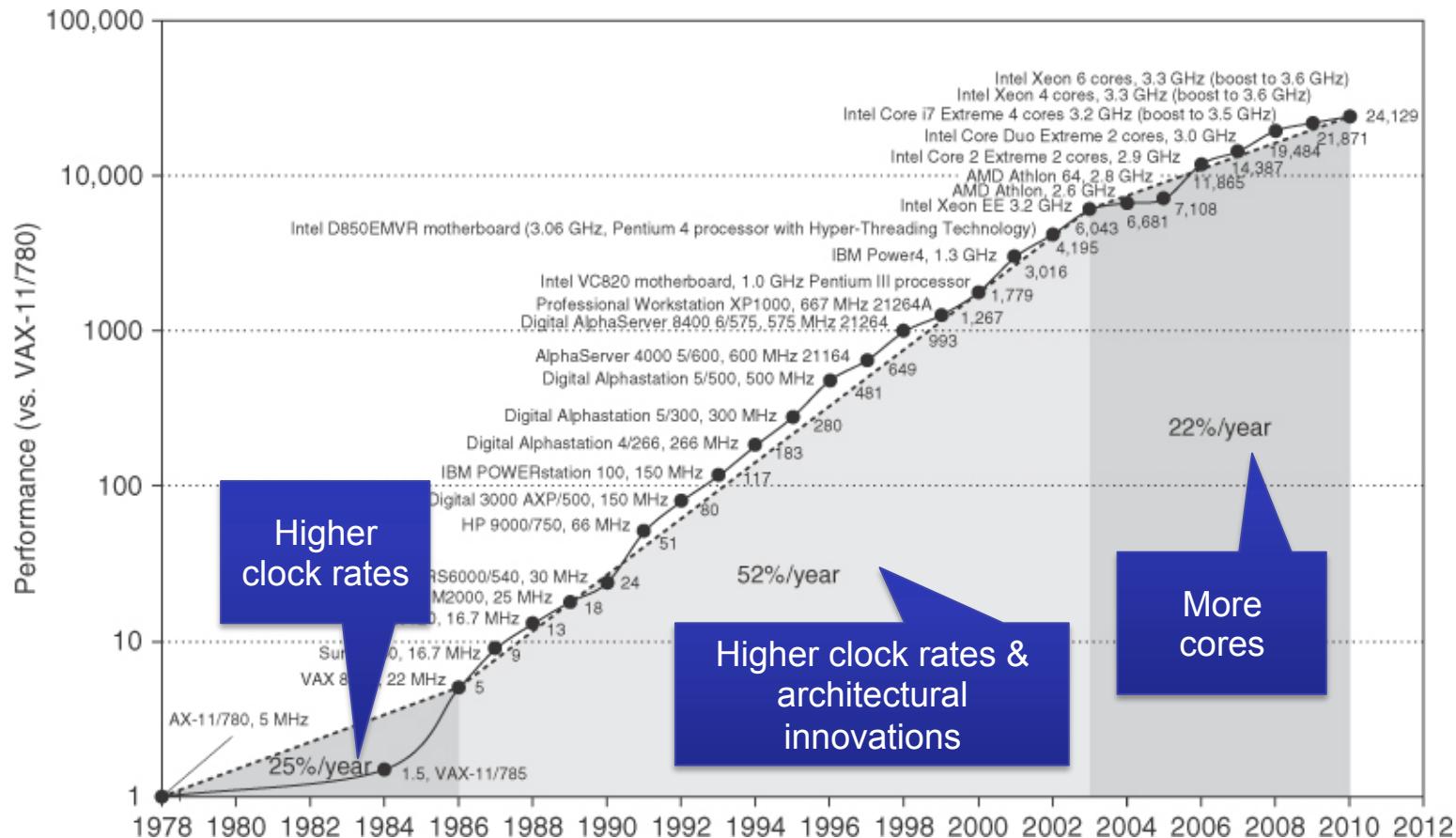
Moore's law (2)



- Reduction of feature size won't last forever
- May continue up to 5 nm
- End maybe ~ mid 2020s

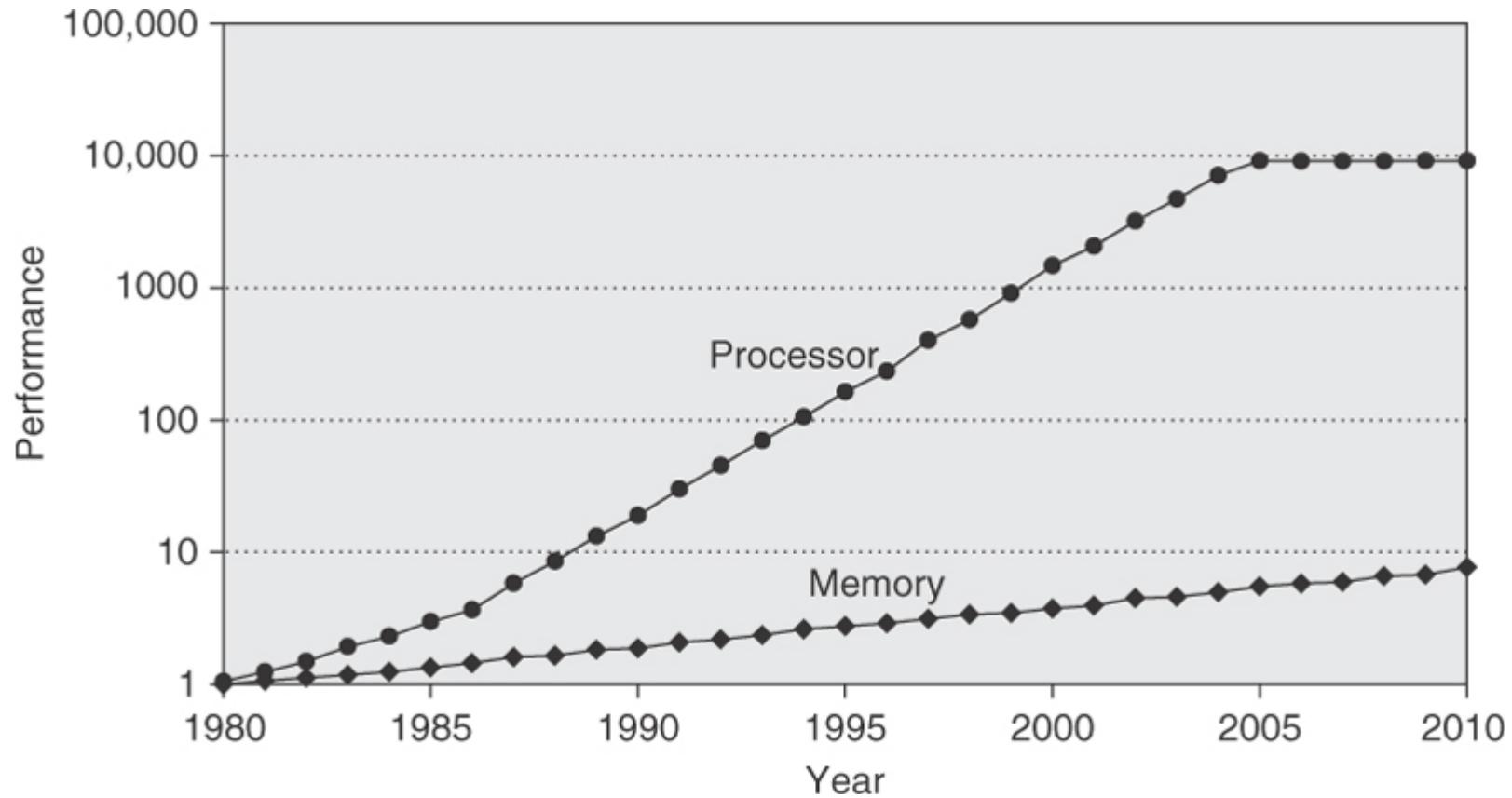
Source: Herb Sutter: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, 2009.

Growth in processor performance



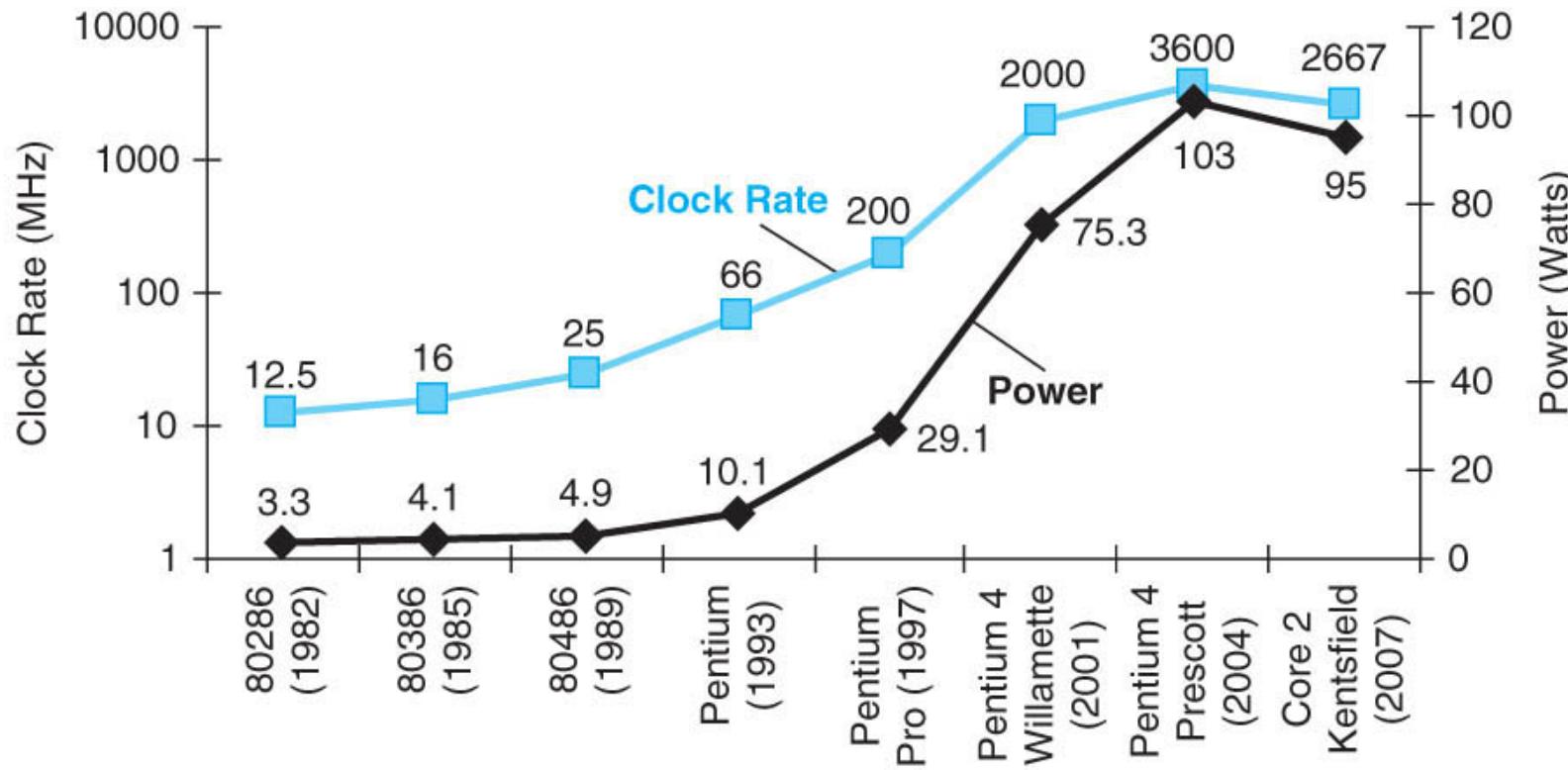
Source: Hennessy, Patterson: Computer Architecture, 5th edition, Morgan Kaufmann

CPU and memory performance



Source: Hennessy, Patterson: Computer Architecture, 5th edition, Morgan Kaufmann

Clock rate vs. power



Source: Patterson, Hennessy: Computer Organization & Design, 4th edition, Morgan Kaufmann



Dennard scaling

- Why haven't clock speeds increased, even though transistors have continued to shrink?
- Dennard (1974) observed that voltage and current should be proportional to the linear dimensions of a transistor
 - Thus, as transistors shrank, so did necessary voltage and current; power is proportional to the area of the transistor

Courtesy of Bill Gropp



Dennard scaling

Dynamic power = $\alpha * CFV^2$

- α = percent time switched
- C = capacitance
- F = frequency
- V = voltage

Capacitance is related to area

- So, as the size of the transistors shrunk, and the voltage was reduced, circuits could operate at higher frequencies at the same power

Courtesy of Bill Gropp

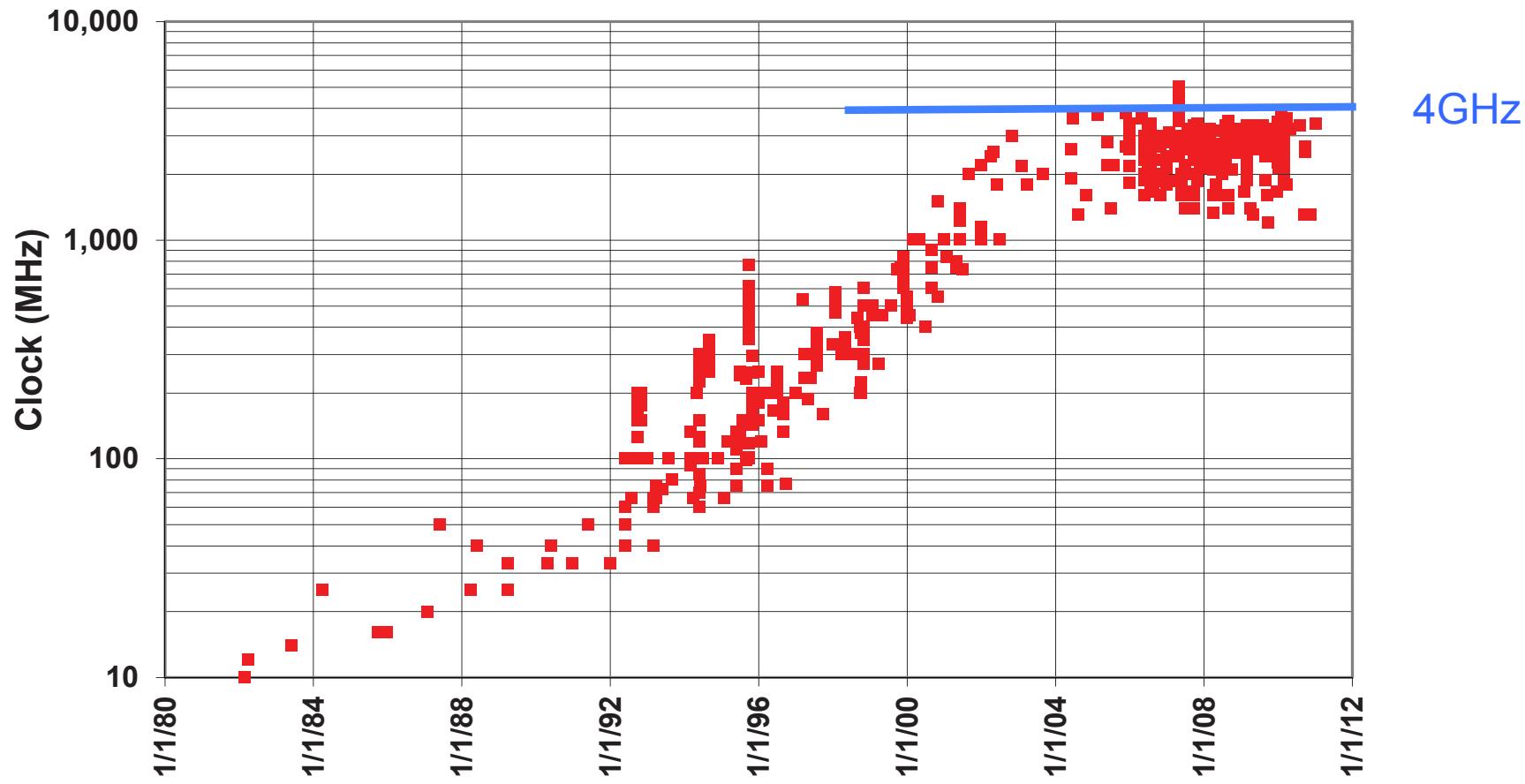
End of Dennard scaling



- Dennard scaling ignored the “leakage current” and “threshold voltage”, which establish a baseline of power per transistor
- As transistors get smaller, power density increases because these don’t scale with size
- These created a “Power Wall” that has limited practical processor frequency to around 4 GHz since 2006

Courtesy of Bill Gropp

Historical clock rates



Courtesy of Bill Gropp

Road maps



TECHNISCHE
UNIVERSITÄT
DARMSTADT

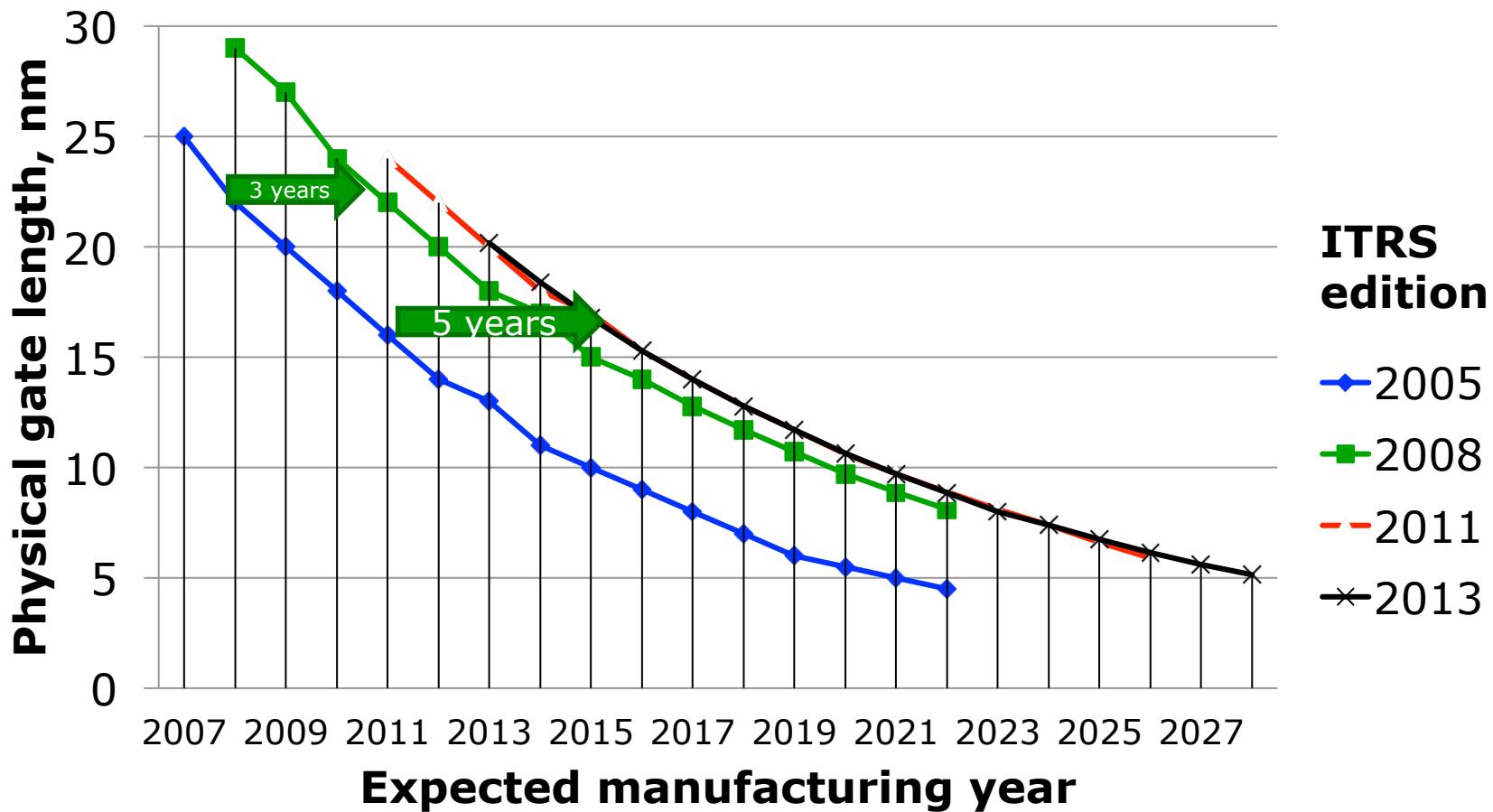
- The Semiconductor industry has produced a roadmap of future trends and requirements
- Semiconductor Industry Association (~1977, roadmaps from early '90s)
- International Technology Roadmap for Semiconductors (~1998)
 - <http://www.itrs.net/>

Courtesy of Bill Gropp

ITRS projections for gate lengths (nm) for 2005, 2008 and 2011 editions



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Courtesy of Bill Gropp

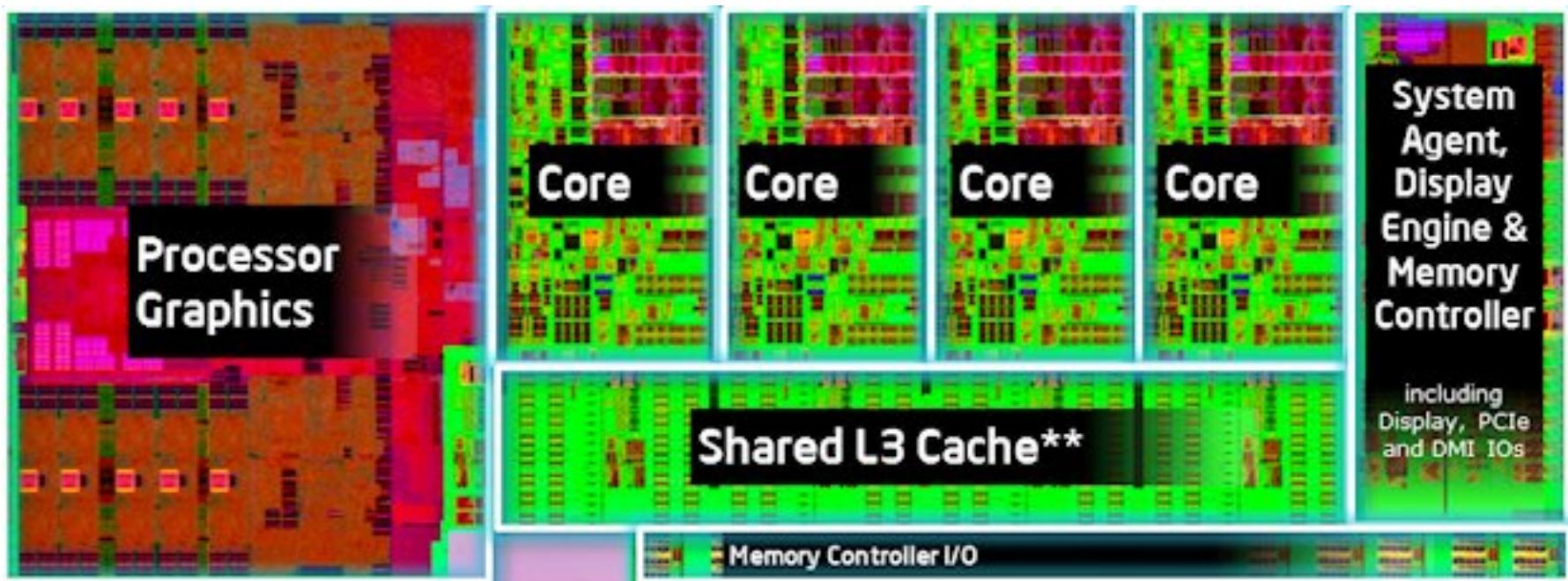


- Since 2002, uni-processor performance improvement has dropped
 - Power dissipation
 - Little instruction-level parallelism left to exploit efficiently
 - Almost unchanged memory latency
- Further performance improvements by placing multiple processors on a single die (multi-core architecture)
 - Initially called on-chip or single-chip multiprocessing
 - Cores often share resources (e.g., L1, L2 cache, I/O bus)
 - Does not solve the memory wall problem
(memory bandwidth)
- Leverages design investment by replicating it

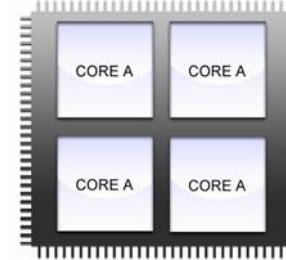
Intel Haswell



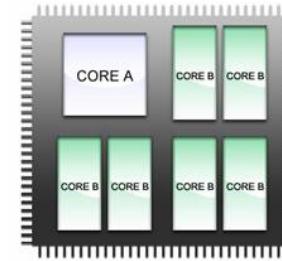
TECHNISCHE
UNIVERSITÄT
DARMSTADT



- New version of Moore's law
 - The **number of cores** will double every two years
 - Recall that today's GPUs feature 100s and 1000s of cores
- Heterogeneity
 - Not all cores necessarily uniform
 - Cores for specific functions
 - Control vs. computation
 - Video
 - Graphics
 - Cryptography
 - Digital signal processing
 - Vector processing



Homogeneous
design

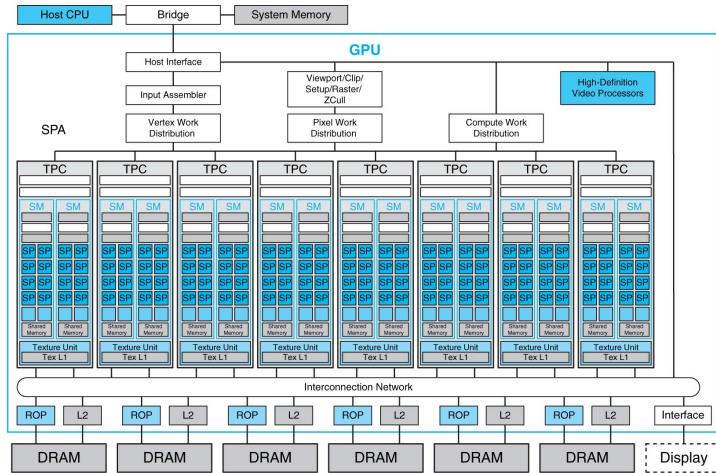


Heterogeneous
design

Graphics processing units (GPUs)

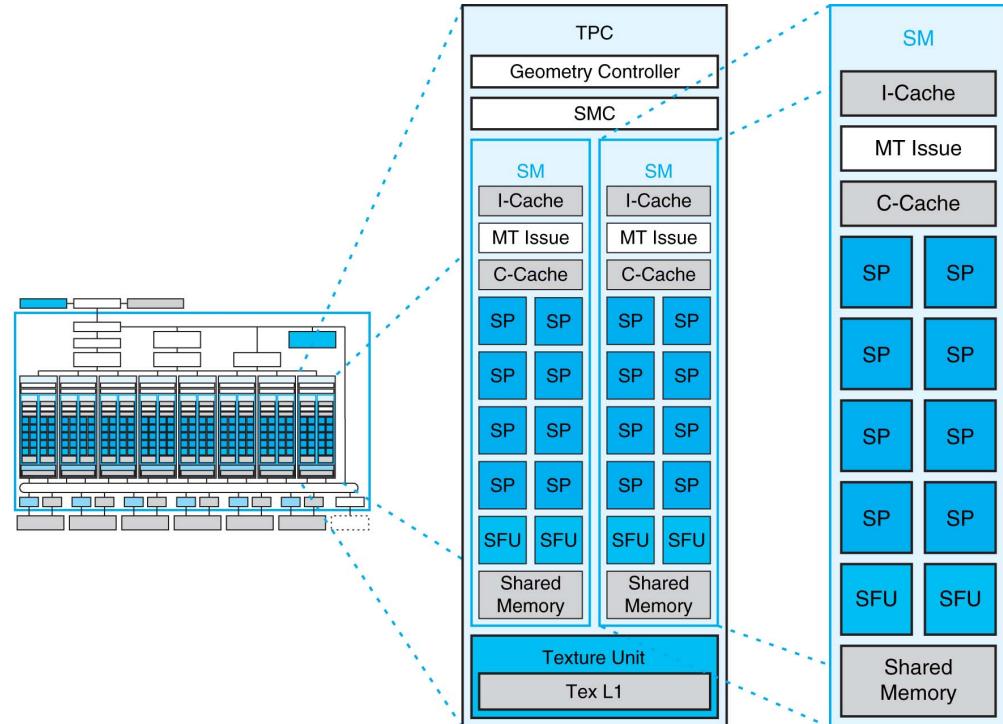
- Processors optimized for 2D and 3D graphics and video
- Became more programmable over time
 - Dedicated logic replaced by programmable processors
- New paradigm at the intersection of graphics processing and parallel computing: **visual computing**
 - Enables new graphics algorithms
- **GPU computing**
 - Using a GPU for computing via a parallel programming language and API (e.g., CUDA, OpenCL)

Example: NVIDIA G80

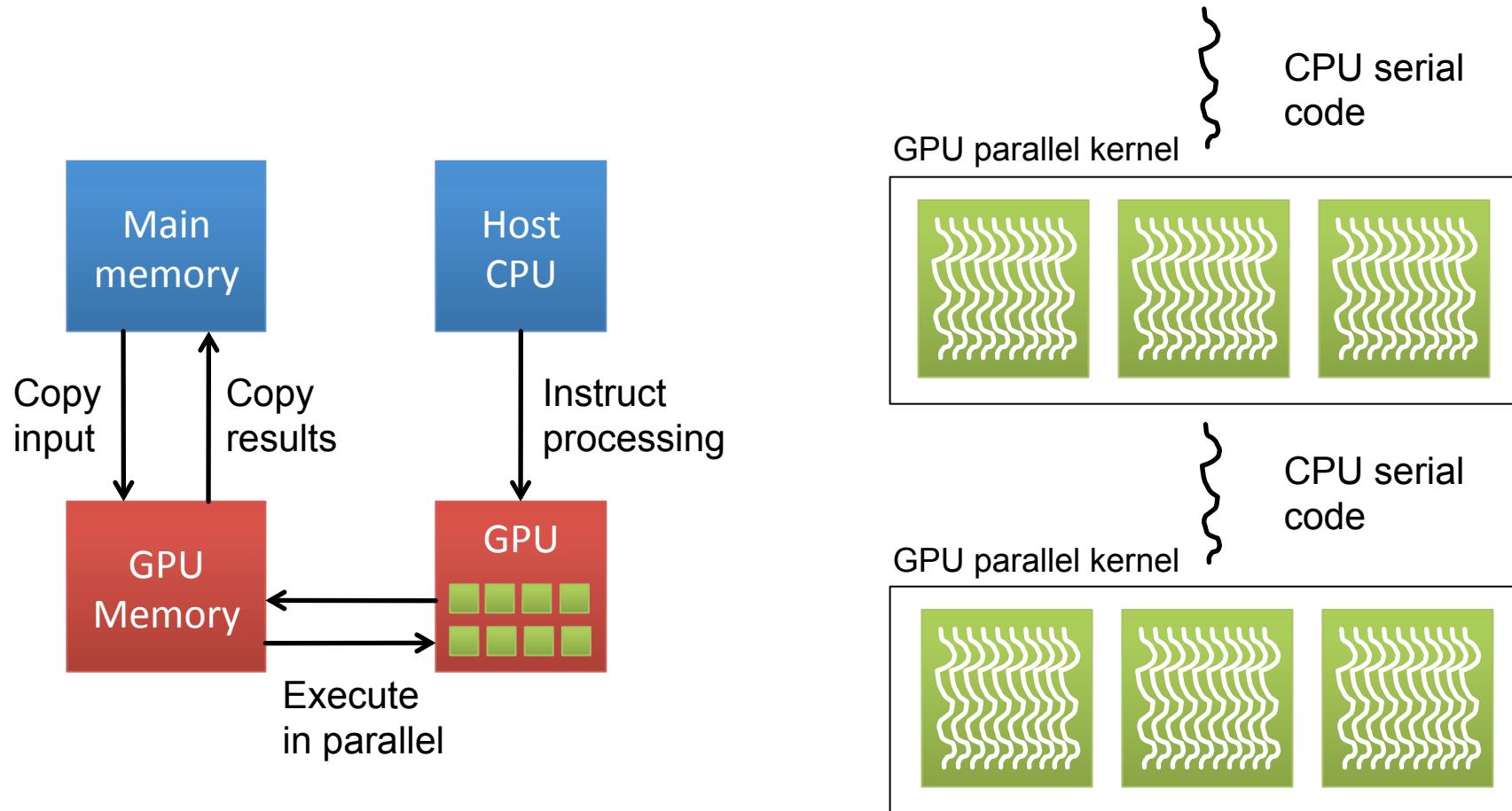


Source: Patterson, Hennessy:
Computer Organization & Design, 4th
edition, Morgan Kaufmann

GPU forms
heterogeneous system
with general-purpose
CPU



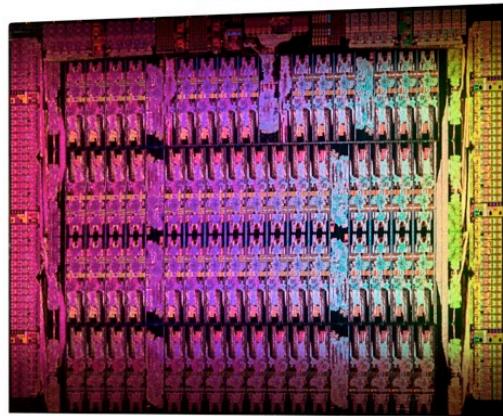
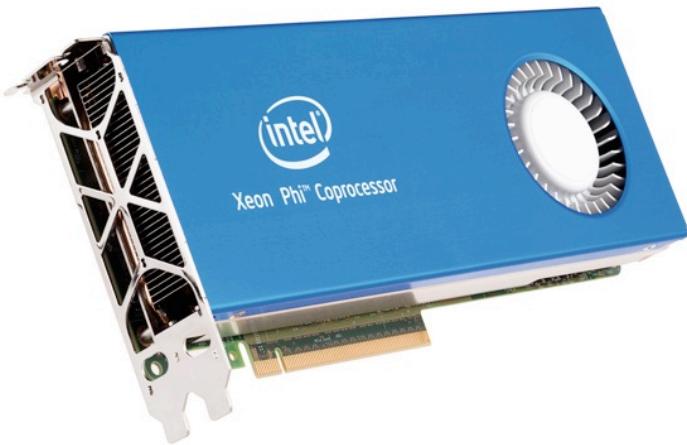
GPU computing



Intel Xeon Phi Coprocessor



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Model 7120

- 61 x86-based cores
- Core frequency 1.238 GHz
- 4 hardware threads per core
- 32 KB L1 cache per core
- 512 KB L2 cache per core
- Cache coherence across entire coprocessor
- 244 hardware threads in total
- 512-bit wide SIMD instructions
- 16 GB GDDR5 memory

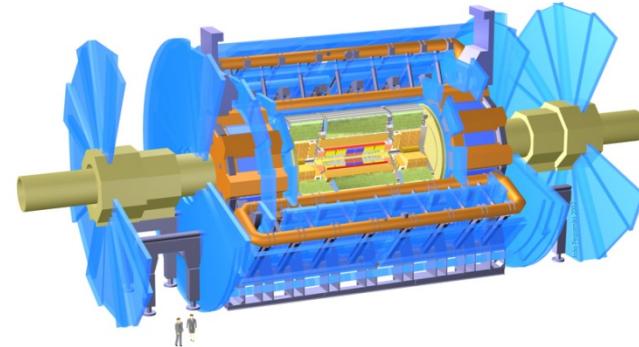
Distributed computing



- Aggregation of dispersed computing resources
 - Idle workstations or dedicated workstation farms
 - Example: SETI@home
- Usually more loosely coupled than parallel computing
 - Most suitable for embarrassingly-parallel problems
- Emphasis on high throughput instead of high performance of individual jobs
 - Larger number of smaller jobs



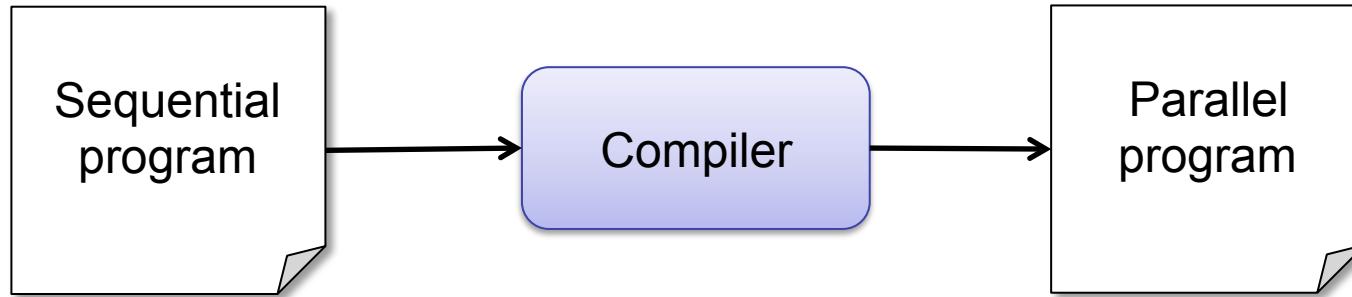
PC farms at CERN



Detector ATLAS

Why parallel programming?

Why don't we have auto-parallelizing compilers?



- Not all parallelization opportunities statically visible
- Would result in very conservative parallelization
- Practical only for certain types of loops (e.g., Intel compiler)
- Also auto-vectorization of suitable code possible



Functional parallelism

- Views problem as a stream of instructions that can be broken down into functions to be executed simultaneously
- Each processing element performs a different function
- Sometimes also called task parallelism

Data parallelism

- Views problem as an amount of data that can be broken down into chunks to be independently operated upon (e.g., array)
- Each processing element performs the same function but on different pieces of data

Example

Several tutors grade a test

- The task sheet contains several tasks



Functional parallelism

Each tutor grades a different subset of the tasks

Data parallelism

Each tutor grades a subset of the students

Another example



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Functional parallelism



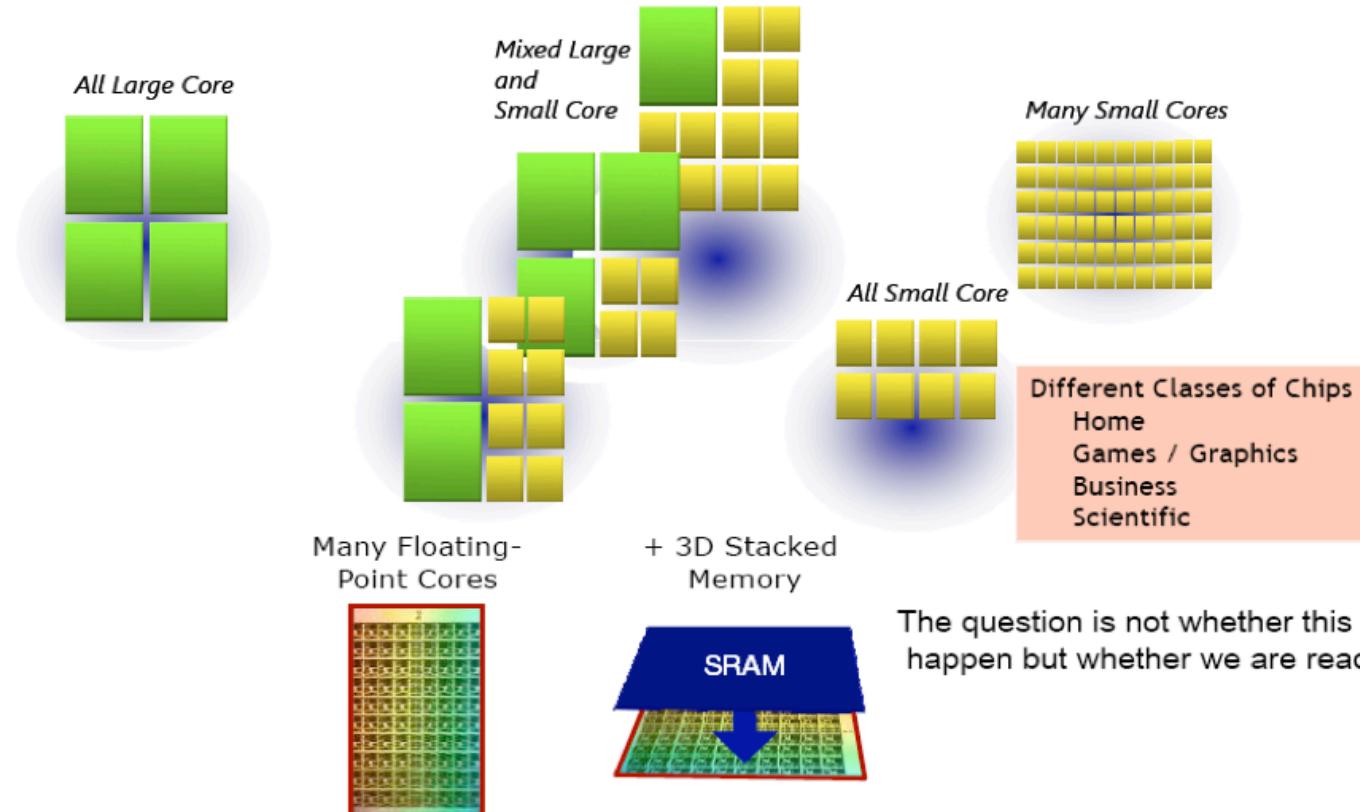
Construction workers

Data parallelism



Hollow square formation

Heterogeneity generalized



Source: Jack Dongarra, ISC 2008



Three scenarios

- Writing parallel code from scratch
 - Parallelizing a sequential program
 - Modifying a parallel program
- } Modifying existing software

Redesign is normal, and software design “de novo” is the exception

Ralph E. Johnson

Cost and benefits of parallelization



Benefit

- Speedup
- Sometimes improved interactivity and cleaner design (separation of concerns)
- More aggregate memory (distributed memory parallelization)

Cost

- Programming effort
- Program complexity
- Overhead (communication & synchronization)
- Bugs
- Potentially non-determinism
- Extra dependencies (library, compiler)



Parallelization strategy

Sequential program with 150 loops

- Where to look for potential parallelism?
- Which loops should be parallelized?
- Which loops cannot be parallelized?

Dependences



Two types

- Control dependences

```
if (condition) then  
    do_work();
```

- Data dependences

```
for ( i = 1; i <= 2; i++ )  
    a[i] = a[i] + a[i-1];
```

Dependences may
prevent parallelization

Summary



- No more improvements of scalar performance
 - Frequency wall
 - Memory wall
 - Power wall
- Data parallelism usually more scalable than functional parallelism
- Development of parallel software occurs rarely from scratch
- When parallelizing a program, pay attention to
 - Correctness
 - Performance
 - Cost

Preview



- Large-scale parallel computing
- Distributed-memory architectures
- Foundations of message passing
- Collective operations
- Data types
- Remote memory access
- Hybrid programming
- Parallel I/O
- Partitioned global address space

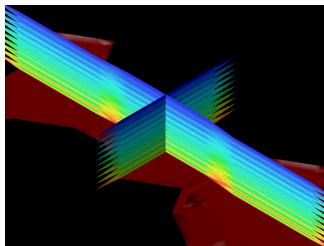
Coverage in course
as far as we get...

Large-scale parallel computing

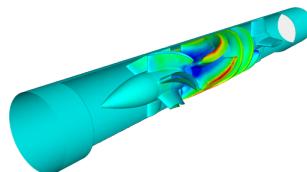


TECHNISCHE
UNIVERSITÄT
DARMSTADT

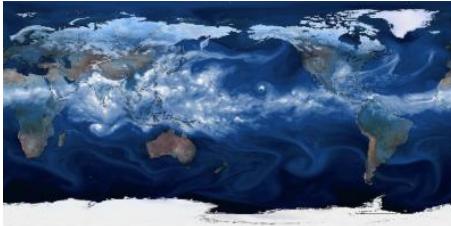
- Solution of very big problems / processing of very big data sets using very big machines



Life sciences



Engineering



Climate prediction

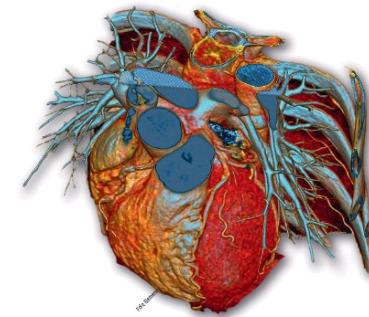


Astrophysics

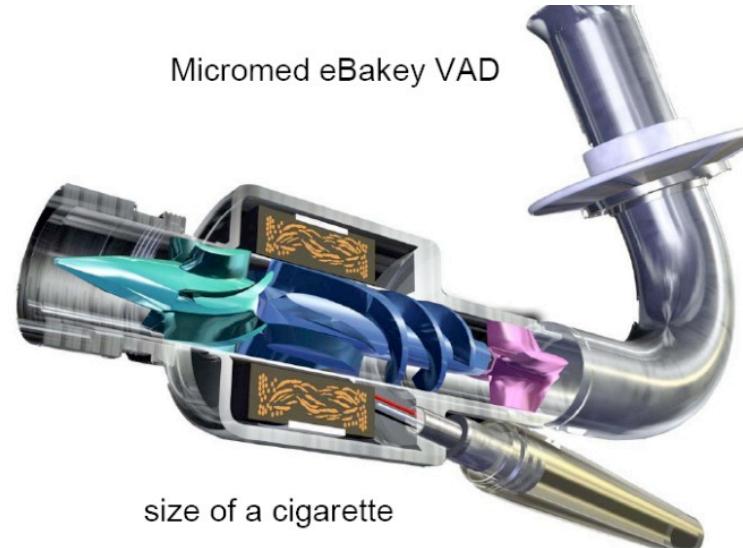


Example: blood pump

- Heart disease is a major cause of death in industrialized nations
- Alternative: ventricular assist device
- Impeller drives blood circulation and disburdens the heart
- Complications
 - Damage of blood cells (hemolysis)
 - Thrombosis



Micromed eBakey VAD



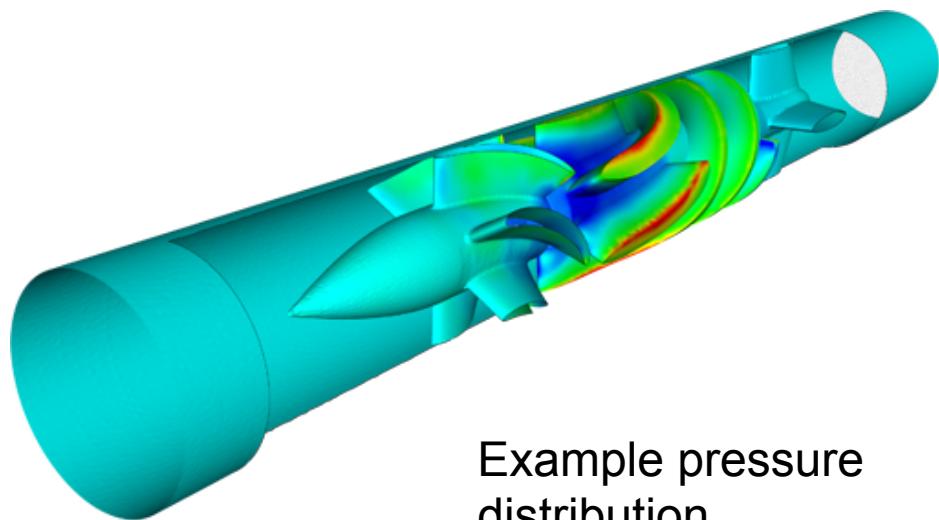
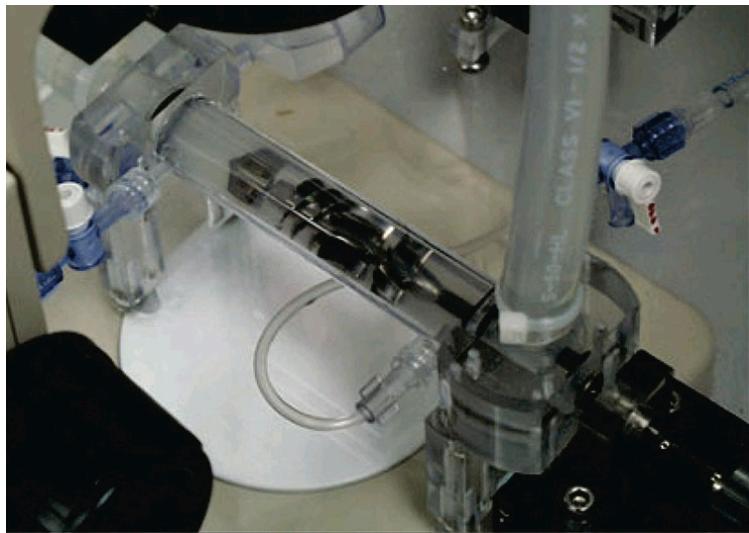
Source: Computer Assisted Analysis of Technical Systems, RWTH Aachen, Prof. Marek Behr

Optimization



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Trial and error on the basis of experiments
 - Expensive prototypes
 - Little insight

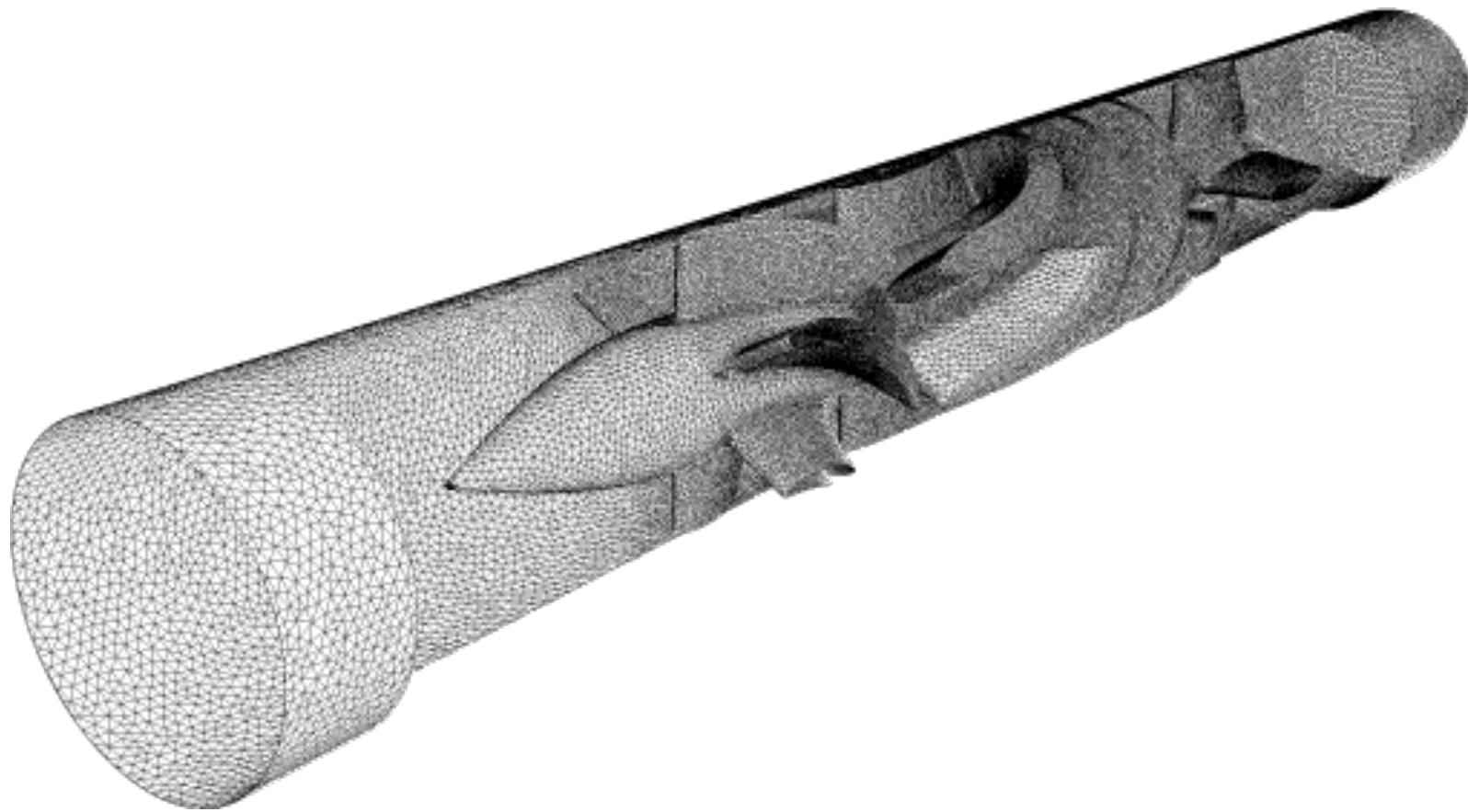


Example pressure distribution

Discretization



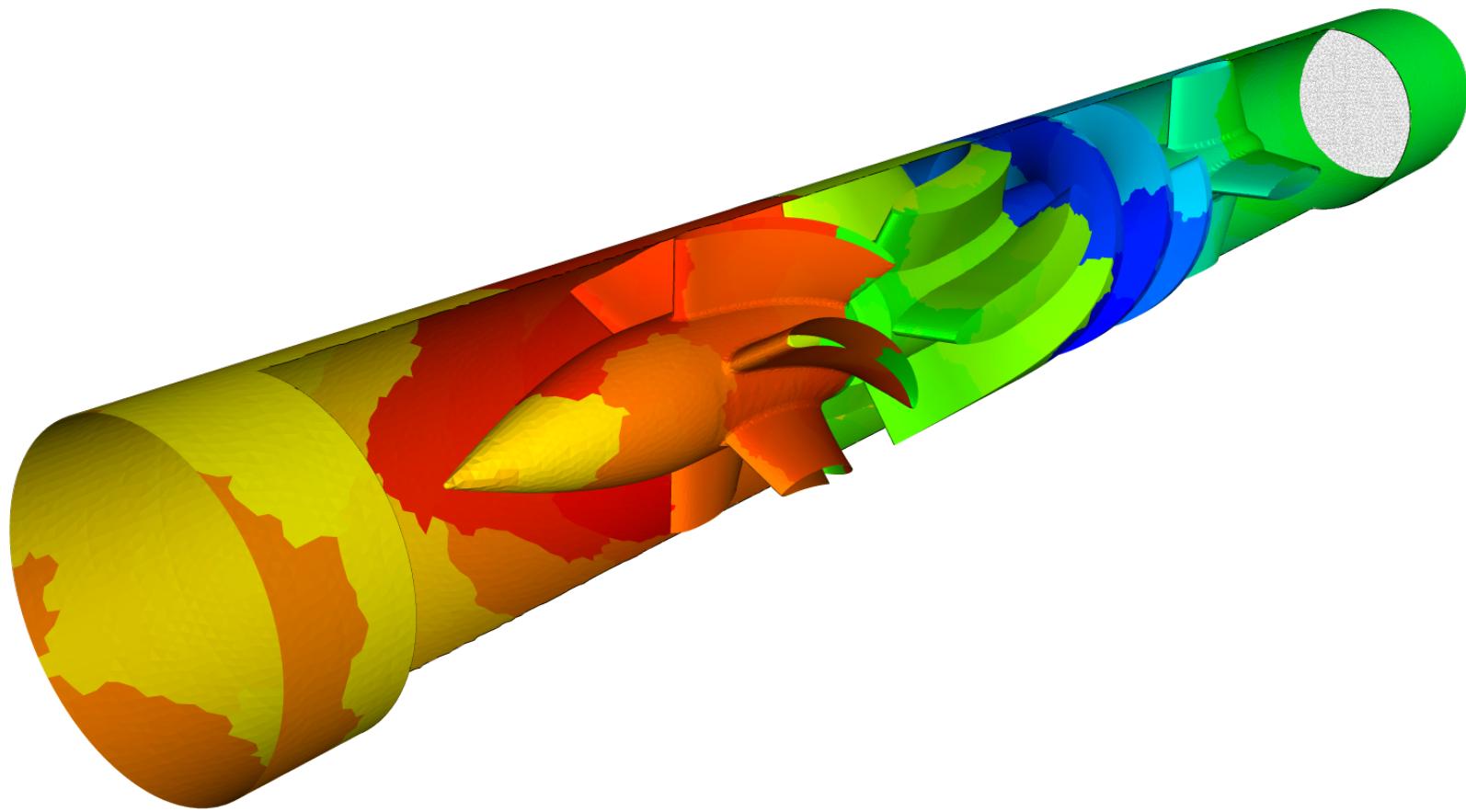
TECHNISCHE
UNIVERSITÄT
DARMSTADT



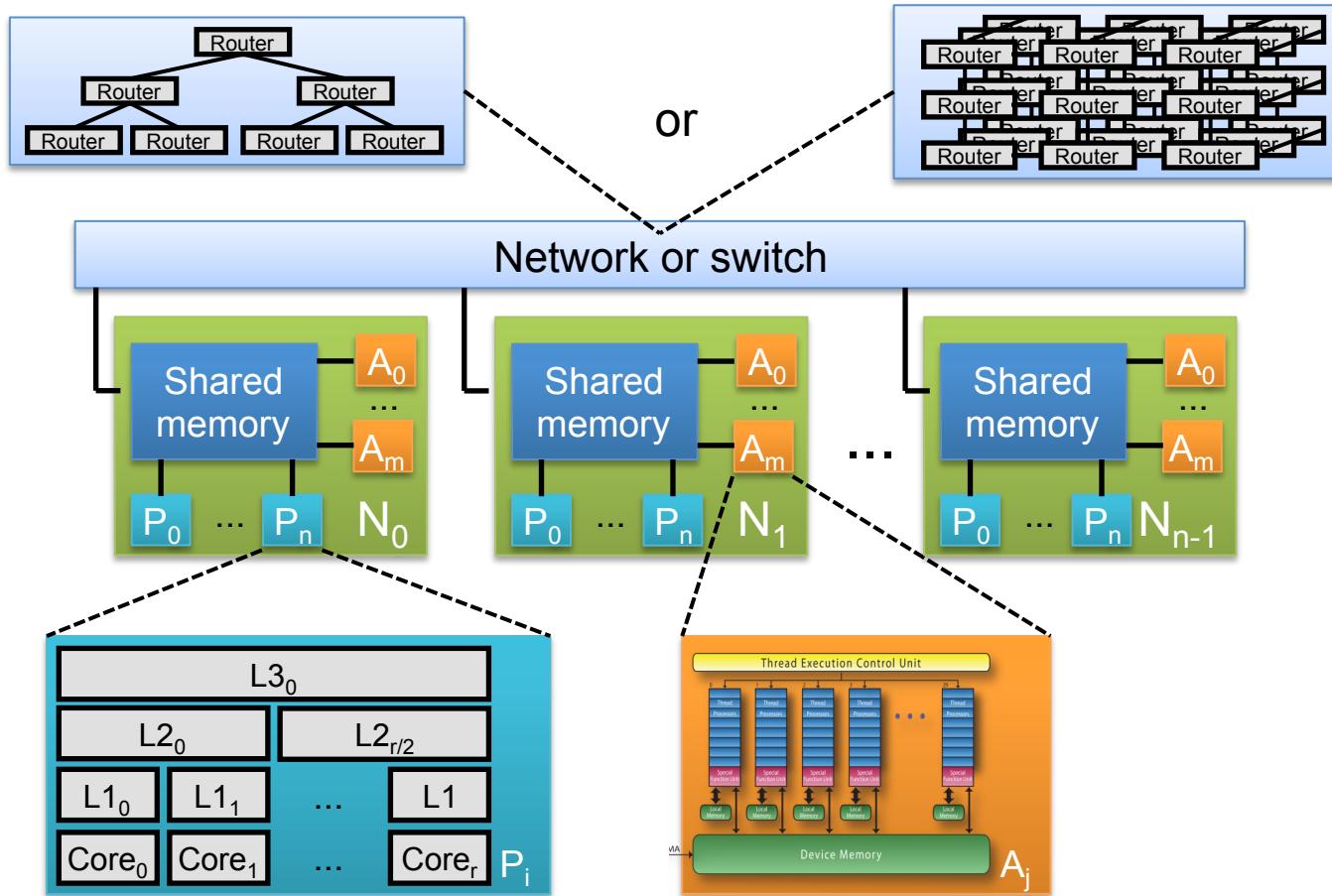
Partitioning



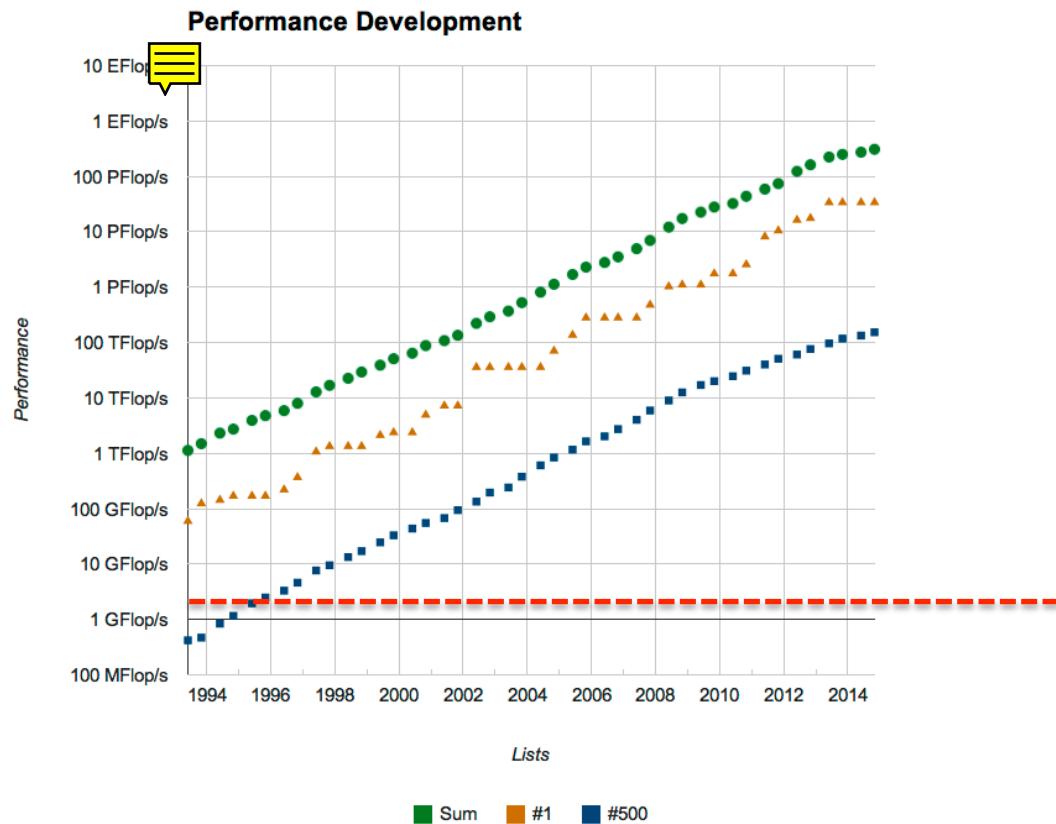
TECHNISCHE
UNIVERSITÄT
DARMSTADT



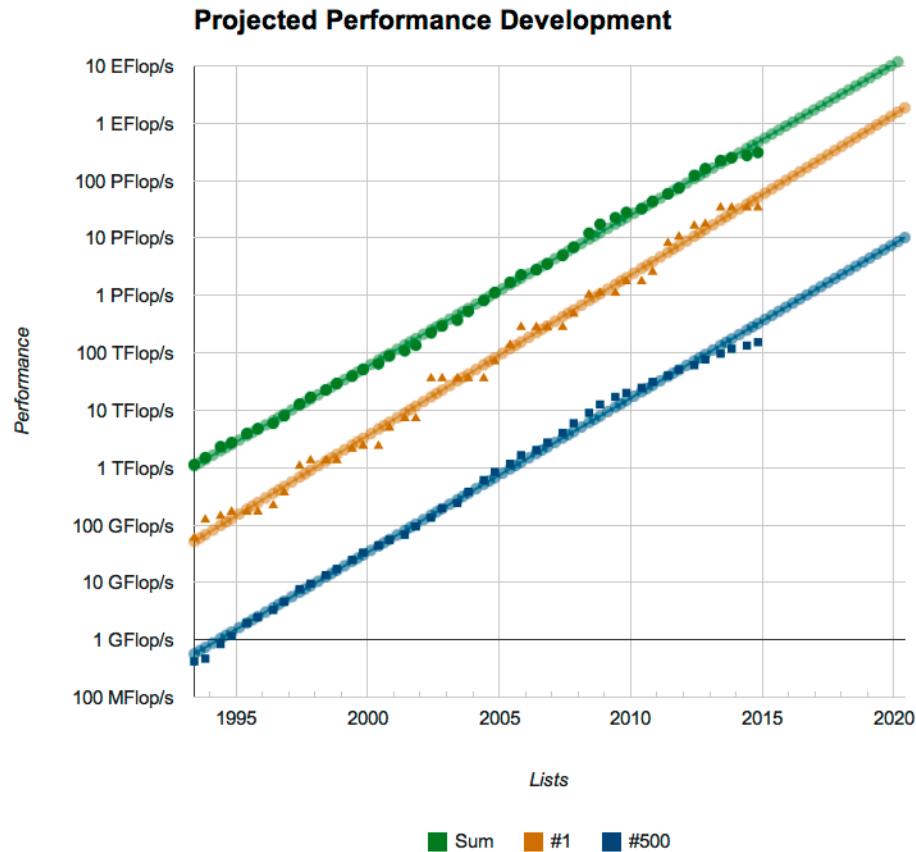
Typical supercomputer architecture



Top 500 supercomputers



Top 500 supercomputers - projection



Exascale: 10^{18} FLOP/s



Systems	2015 Tianhe-2	2020-2023	Difference Today & Exa
System peak	55 Pflop/s	1 Eflop/s	~20x
Power	18 MW (3 Gflops/W)	~20 MW (50 Gflops/W)	O(1) ~15x
System memory	1.4 PB (1.024 PB CPU + .384 PB CoP)	32 - 64 PB	~50x
Node performance	3.43 TF/s (.4 CPU +3 CoP)	1.2 or 15TF/s	O(1)
Node concurrency	24 cores CPU + 171 cores CoProc	O(1k) or 10k	~5x - ~50x
Node Interconnect BW	6.36 GB/s	200-400 GB/s	~40x
System size (nodes)	16,000	O(100,000) or O(1M)	~6x - ~60x
Total concurrency	3.12 M 12.48M threads (4/core)	O(billion)	~100x
MTTF 	Few / day	Many / day	O(?)

Tianhe-2 (MilkyWay-2)

National University of Defense Technology

Processor

- Intel Xeon
- Intel Xeon Phi

Cores: 3,120,000

Linpack performance: 33.9 PF

Theoretical peak: 54.9 PF

Power: 17.8 MW

Memory: 1,024 TB

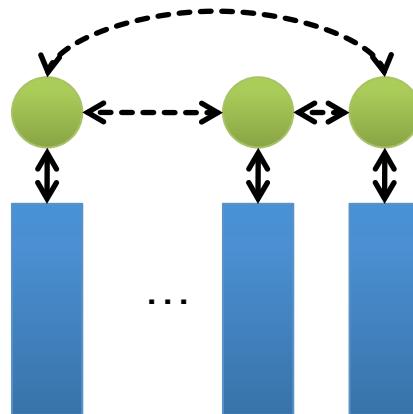


Tianhe-2

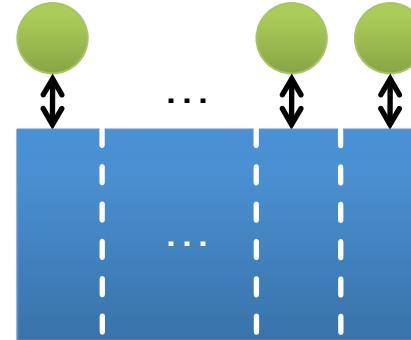
Focus: inter-node parallelism



- Distributed (private) memory & message passing
- Alternative: partitioned global address space (PGAS)
- Scalability (1000s of nodes)



Message passing



Partitioned global address space

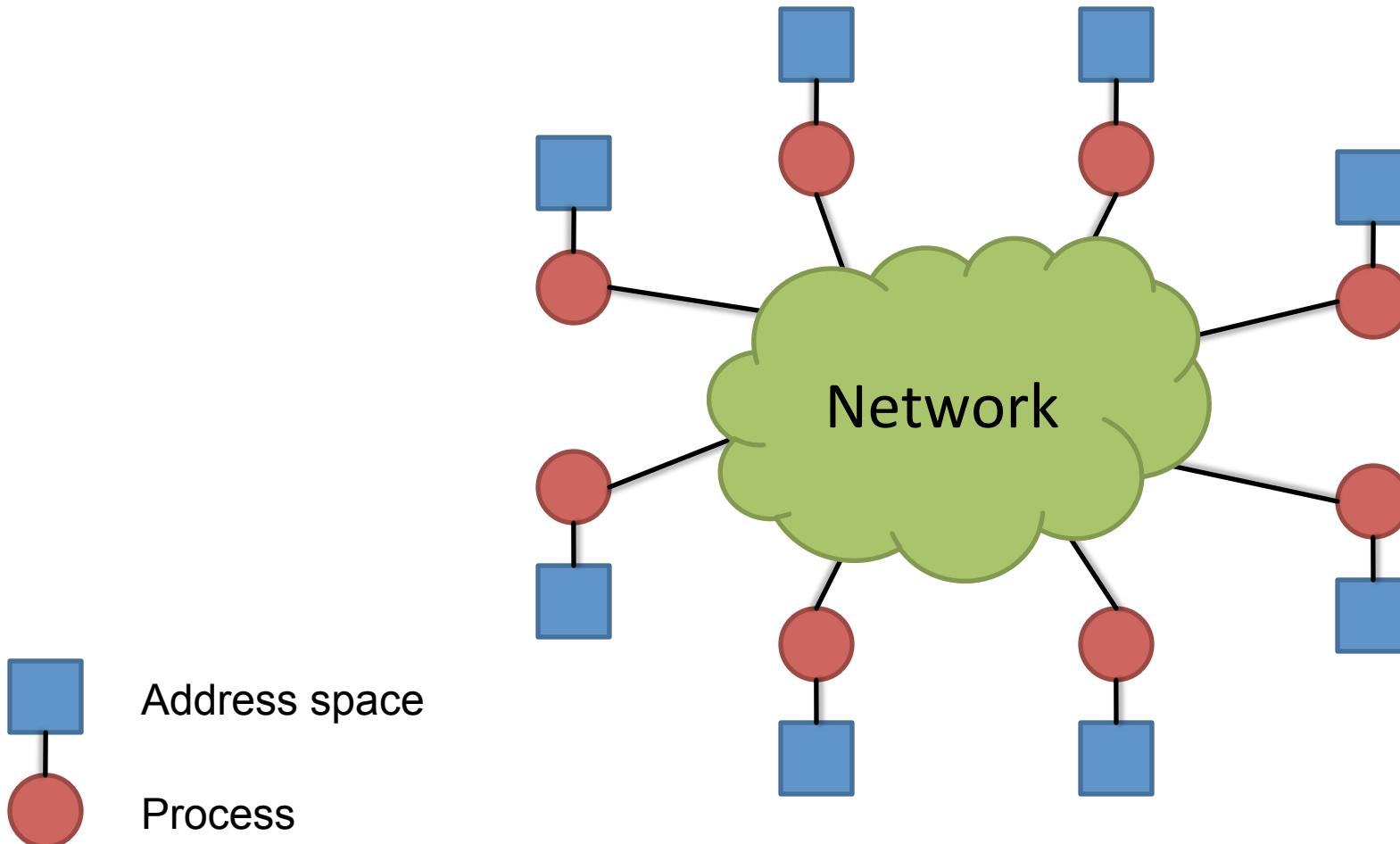
Single Program Multiple Data (SPMD)



- Execution model underlying most parallel programming models
- The same program is executed on multiple processors
- Processes or threads are enumerated; each process or thread “knows” its number (ID)
- Case distinctions based on the process or thread number lead to different control flows
(i.e., multiple instruction streams)

```
if (process_id == 42) then
    call do_something()
else
    call do_something_else()
endif
```

Message passing



Message passing (2)



- Suitable for distributed memory
- Multiple processes each having their own private address space
- Access to (remote) data of other processes via sending and receiving messages (explicit communication)

- Sender invokes send routine
- Receiver invokes receive routine

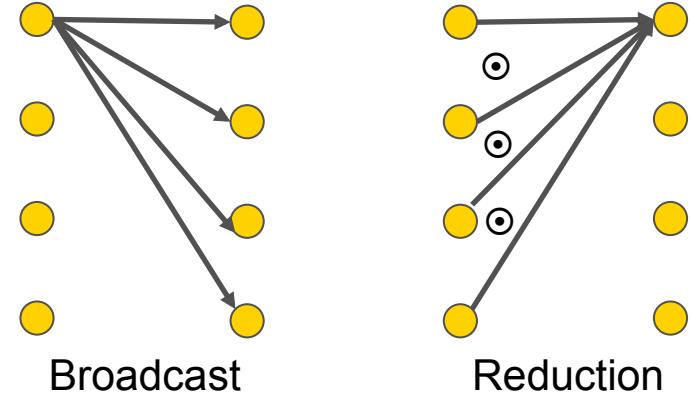
```
if (my_id == SENDER)
    send_to(RECEIVER, data);

if (my_id == RECEIVER)
    recv_from(SENDER, data)
```

- De-facto standard MPI: www mpi-forum.org

Group communication & computation

- Example: broadcast, reduction
- Different flavors: $1 \rightarrow n$, $n \rightarrow 1$, $n \rightarrow n$



- **Manual** implementation via point-to-point messages
cumbersome and often suboptimal in terms of performance
- MPI offers a range of predefined **collective operations**
 - Embody recurring parallel communication / design patterns
 - Will study how to use them and their performance characteristics

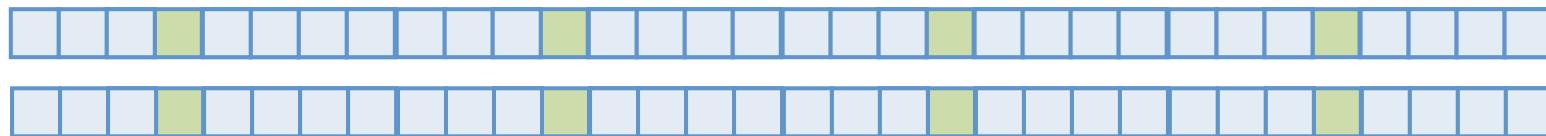
Non-contiguous data

- Certain data types may lack contiguous memory representation
- Sending them across the network requires tedious packing and unpacking
- MPI data type concept avoids to do this manually

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Matrix

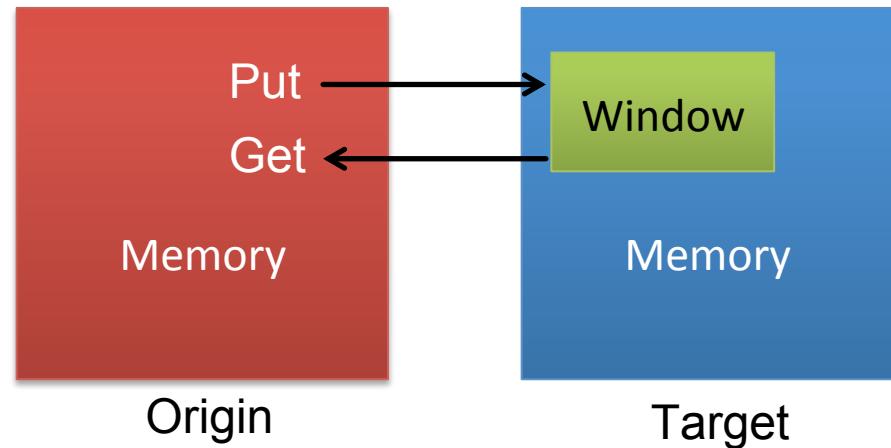
Memory layout in C (row major)



Remote memory access

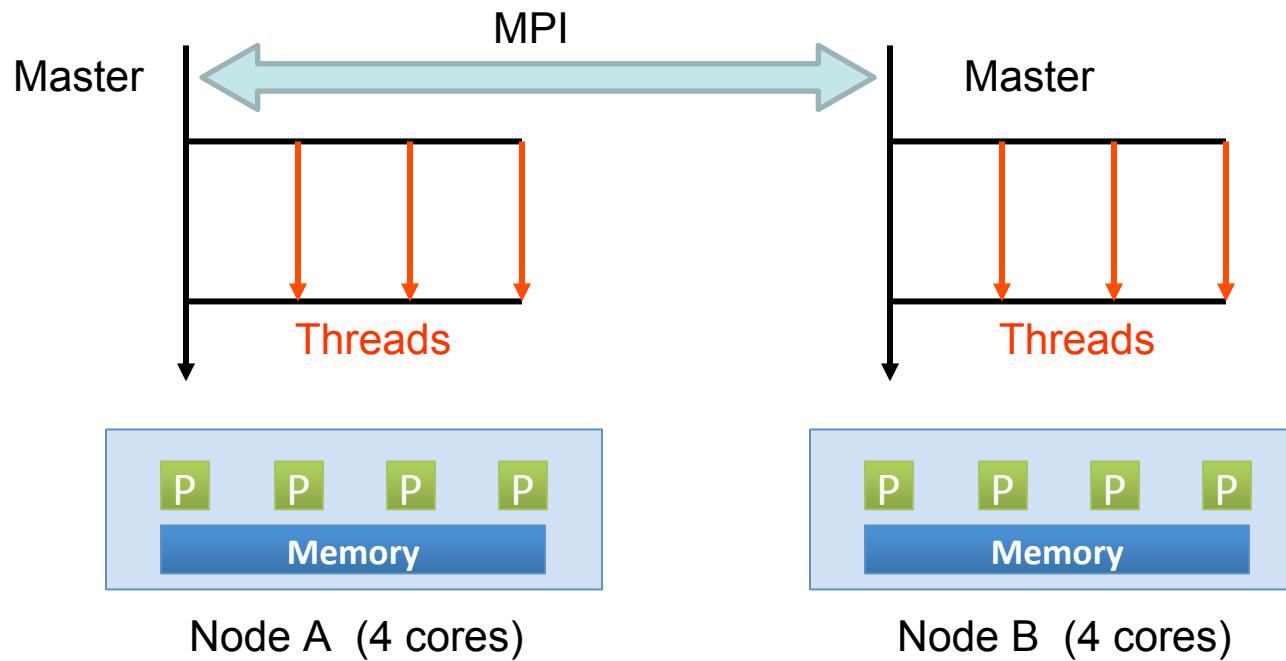


- Conventional message passing is two-sided
 - Send to destination process / receive from source process
 - Both processes specify message parameters
- Remote memory access
 - Also called one-sided communication
 - Parameters of data transfer are determined by one process only
 - Typically expressed through
get and put operations
- Can be used to build powerful distributed data structures
- Underlying communication substrate for PGAS languages



Hybrid programming

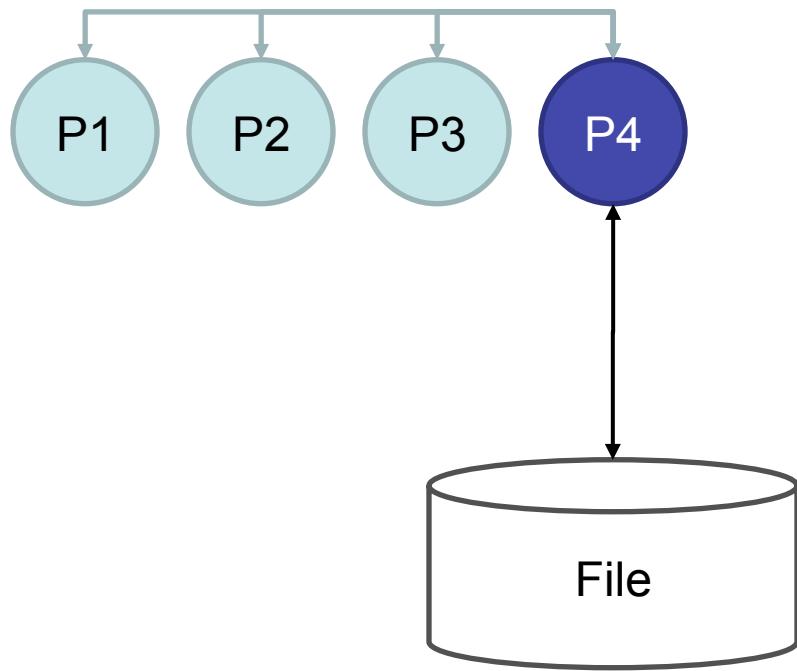
- Between processes parallelism via MPI
- Thread parallelism inside a process (e.g., via OpenMP)



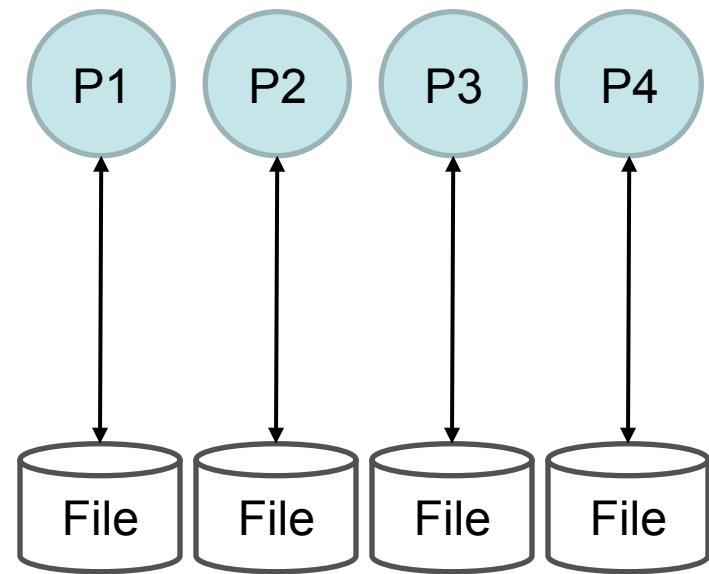
File access in parallel applications



Two traditional models



Sequential file access – all file
accesses through a single process.



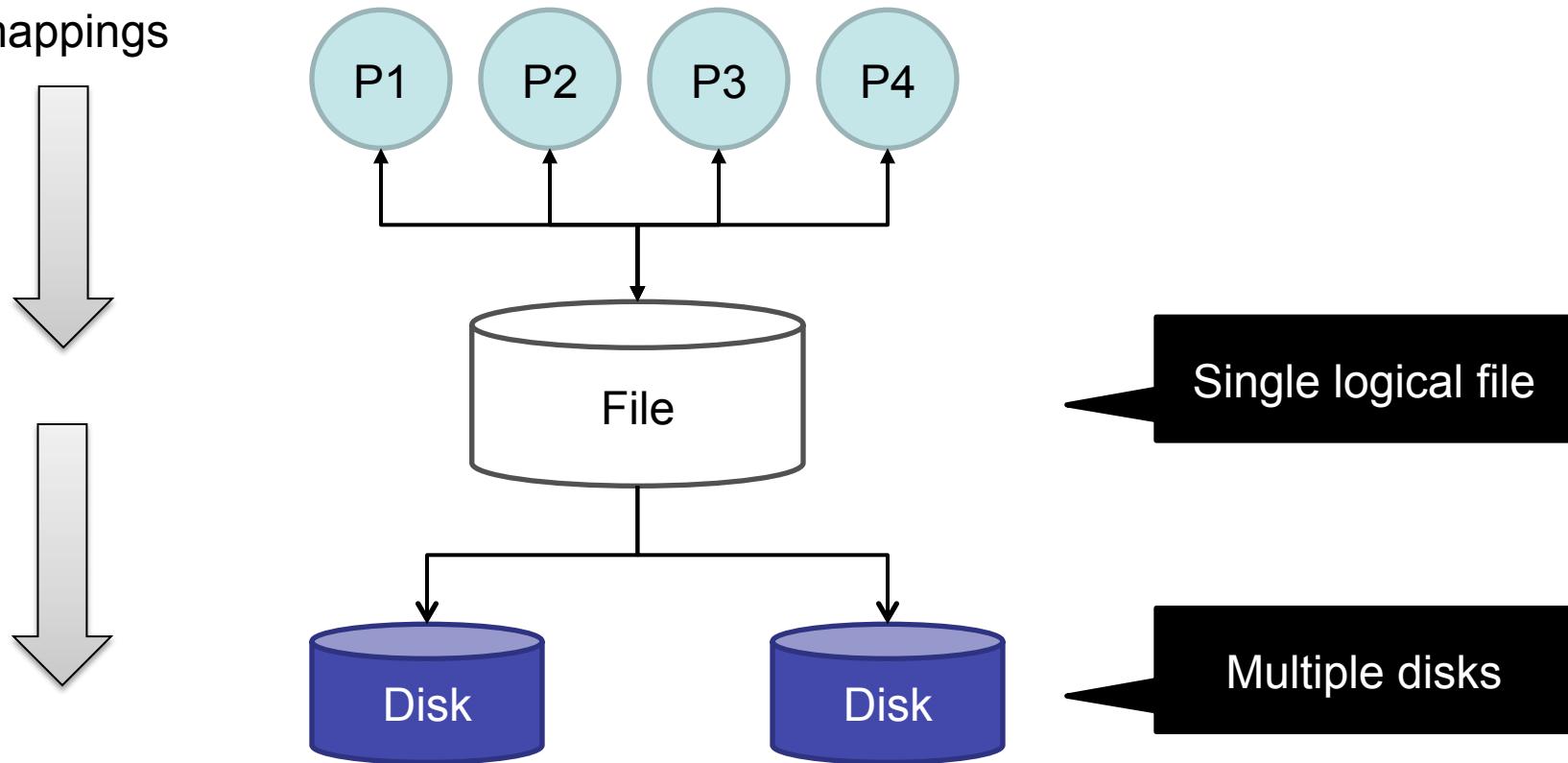
Multiple file access – each
process writes its own file.

Parallel file I/O

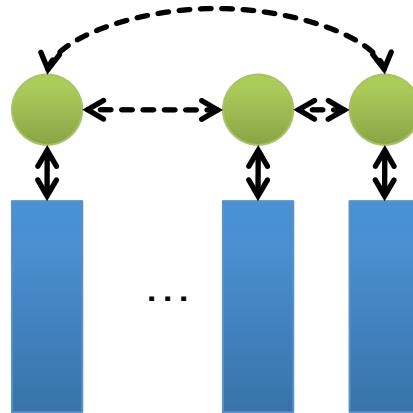


Multiple processes access the same file

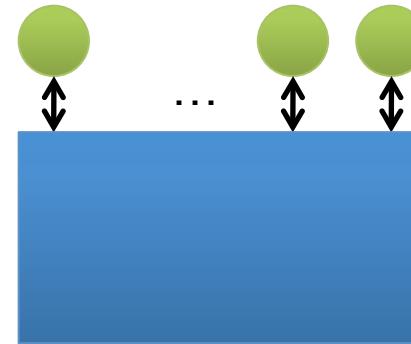
2 mappings



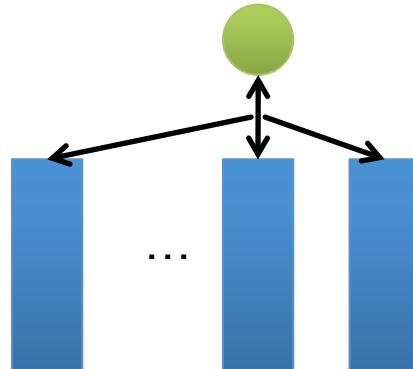
Parallel programming models



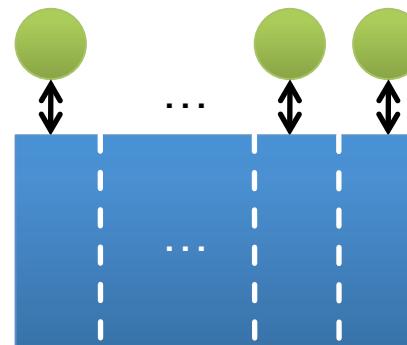
Message passing



Shared memory



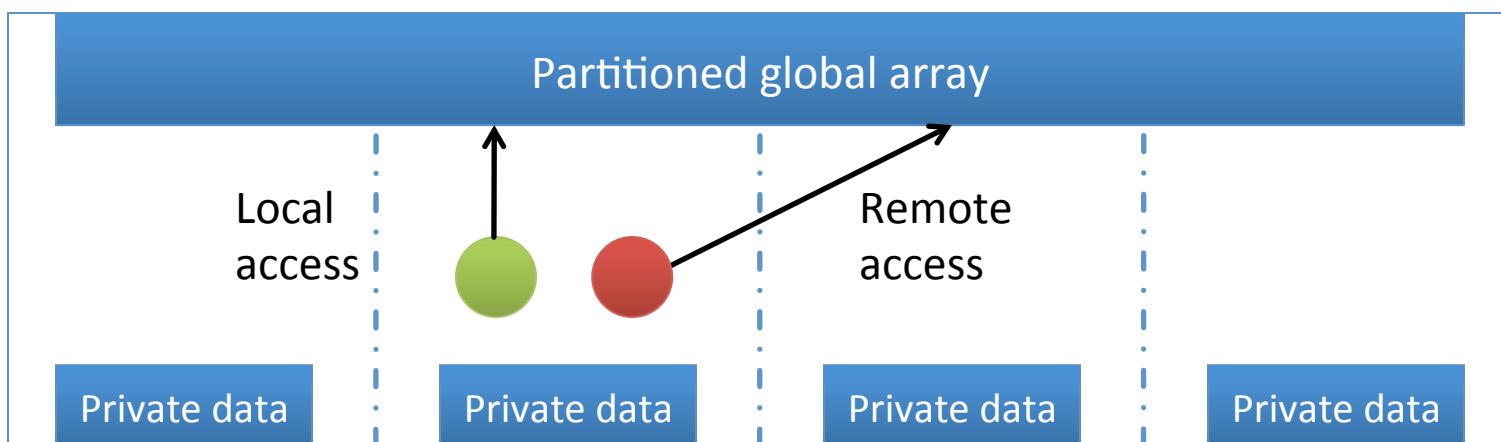
Data parallel



Partitioned global address space (PGAS)

Partitioned global address space (PGAS)

- The PGAS memory model allows any thread to read or write memory anywhere in the system
- It is partitioned to indicate that some data is local, whereas other data is further away (slower to access)



Unified Parallel C (UPC)



- Extension of the C programming language
- Designed for high performance computing on large-scale parallel machines
- Provides uniform programming model for both shared and distributed memory hardware
- The programmer is presented with a single shared, partitioned address space
 - Variables may be directly read and written by any processor
 - But each variable is physically associated with a single processor
- SPMD model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per processor
- Implementation Berkeley UPC <http://upc.lbl.gov/>



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Advanced Parallel Programming

Prof. Dr. Felix Wolf

FUNDAMENTALS

Outline



- Performance metrics
 - Locality
 - Amdahl's law
 - Speedup & parallel efficiency
 - Scaling
 - Law of Gustafson
 - Abstract models of parallel machines
 - Asymptotic complexity
 - Processes & threads
-

Primary performance metrics

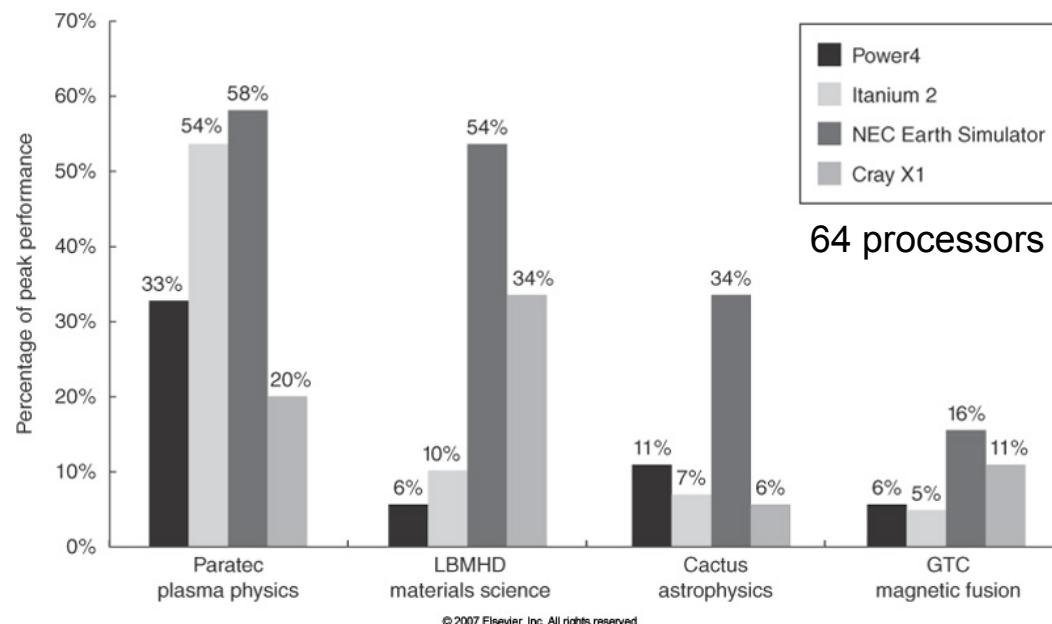
- Response time, execution time
 - Time between start and completion of an event or program
 - Throughput
 - Total amount of work done in a given time
 - Energy (to solution)
- Performance = $\frac{1}{\text{Resources to solution}}$



Peak performance

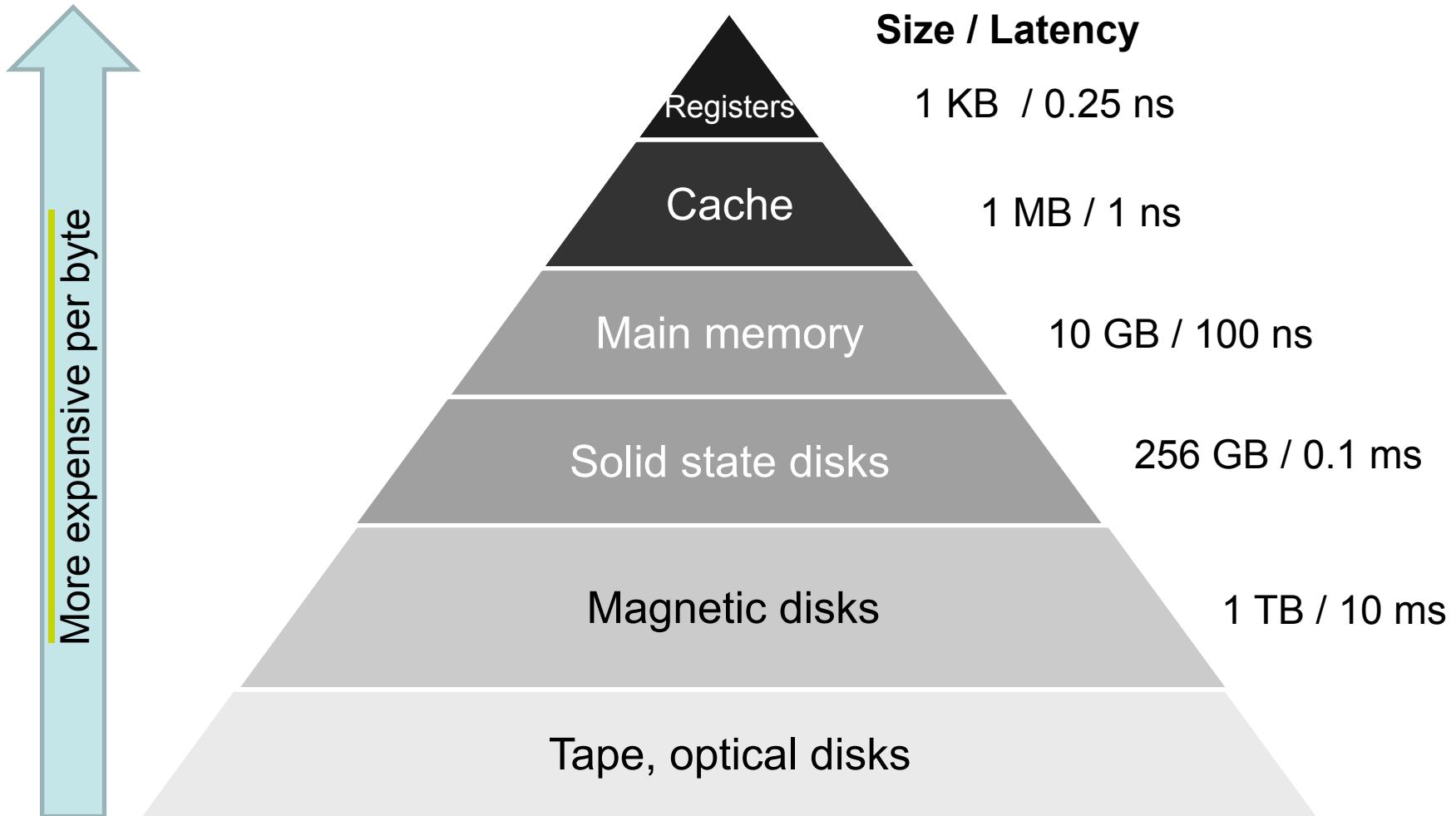


- Peak performance is the performance a computer is guaranteed not to exceed



Source: Hennessy, Patterson: Computer Architecture, 4th edition, Morgan Kaufmann

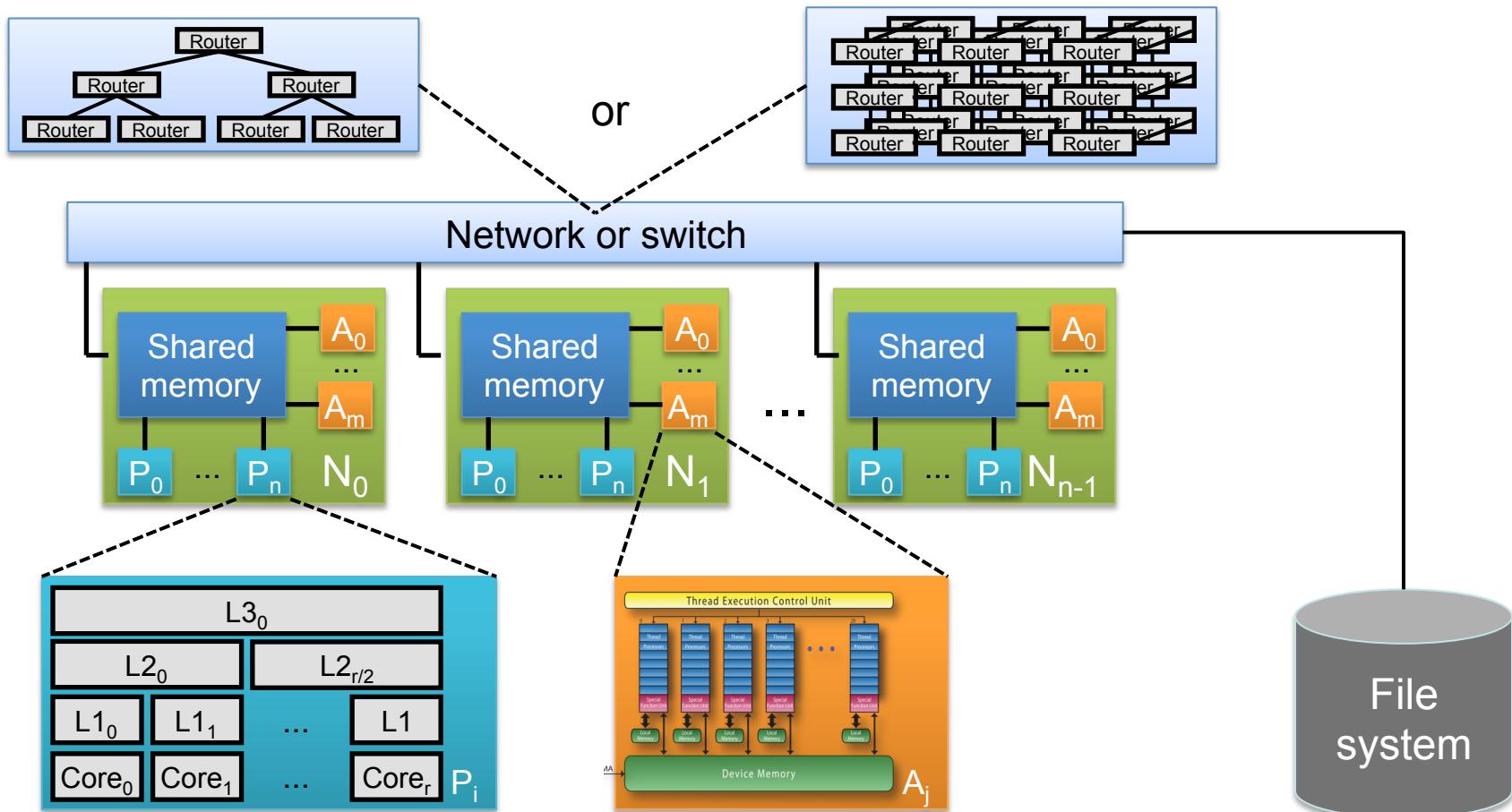
Memory hierarchy



- Programs tend to reuse data and instructions they have used recently
 - Allows to estimate what instructions and data a program will use in the near future
- A program typically spends 90% of the time in 10% of its code
- Locality also applies to data accesses, but usually not as much as to code accesses

Temporal locality	Spatial locality
Recently accessed items are likely to be accessed in the near future	Items with addresses close to each other tend to be referenced close together in time

Typical supercomputer architecture



Amdahl's law



- The improvement in execution time to be gained from using some faster mode of execution is limited by the fraction of the time that faster mode can be used
- Speedup =
$$\frac{\text{Execution time for the entire task without using the enhancement}}{\text{Execution time for the entire task using the enhancement when possible}}$$
- Depends on two factors
 - Fraction of the original execution that can benefit from the enhancement
 - Improvement gained through enhancement during that fraction



Amdahl's law (2)

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} < \frac{1}{(1 - \text{Fraction}_{\text{enhanced}})}$$

Example

- Function `foo()` of a program takes 20 % of the overall time
- How is the speedup if the time needed for `foo()` can be halved?

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - 0.2) + \frac{0.2}{2}} = \frac{10}{9}$$

- How is the speedup if the time needed for `foo()` can almost be eliminated?

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - 0.2) + \frac{0.2}{\infty}} = \frac{10}{8}$$



System / application

- Server throughput can be improved by spreading workload across multiple processors or disks
- Ability to add memory, processors, and disks is called scalability

Processor

- Instruction-level parallelism (e.g., pipelining)
- Depends on the fact that many instructions do not depend on the results of their predecessor

Detailed digital design

- Set-associative caches use multiple banks of memory
- Carry-lookahead or prefix adder
- Parallelism exploited to speed up the calculation of sums from linear to logarithmic in the number of bits



Amdahl's law for parallelism

- Assumption – program can be parallelized on p processors except for a sequential fraction f with

$$0 \leq f \leq 1$$

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} < \frac{1}{(1 - \text{Fraction}_{\text{enhanced}})}$$

$$\text{Speedup}(p) = \frac{1}{(1 - (1 - f)) + \frac{1-f}{p}} = \frac{1}{f + \frac{1-f}{p}} < \frac{1}{f}$$

- Speedup limited by sequential fraction



Available parallelism

Amdahl's Law

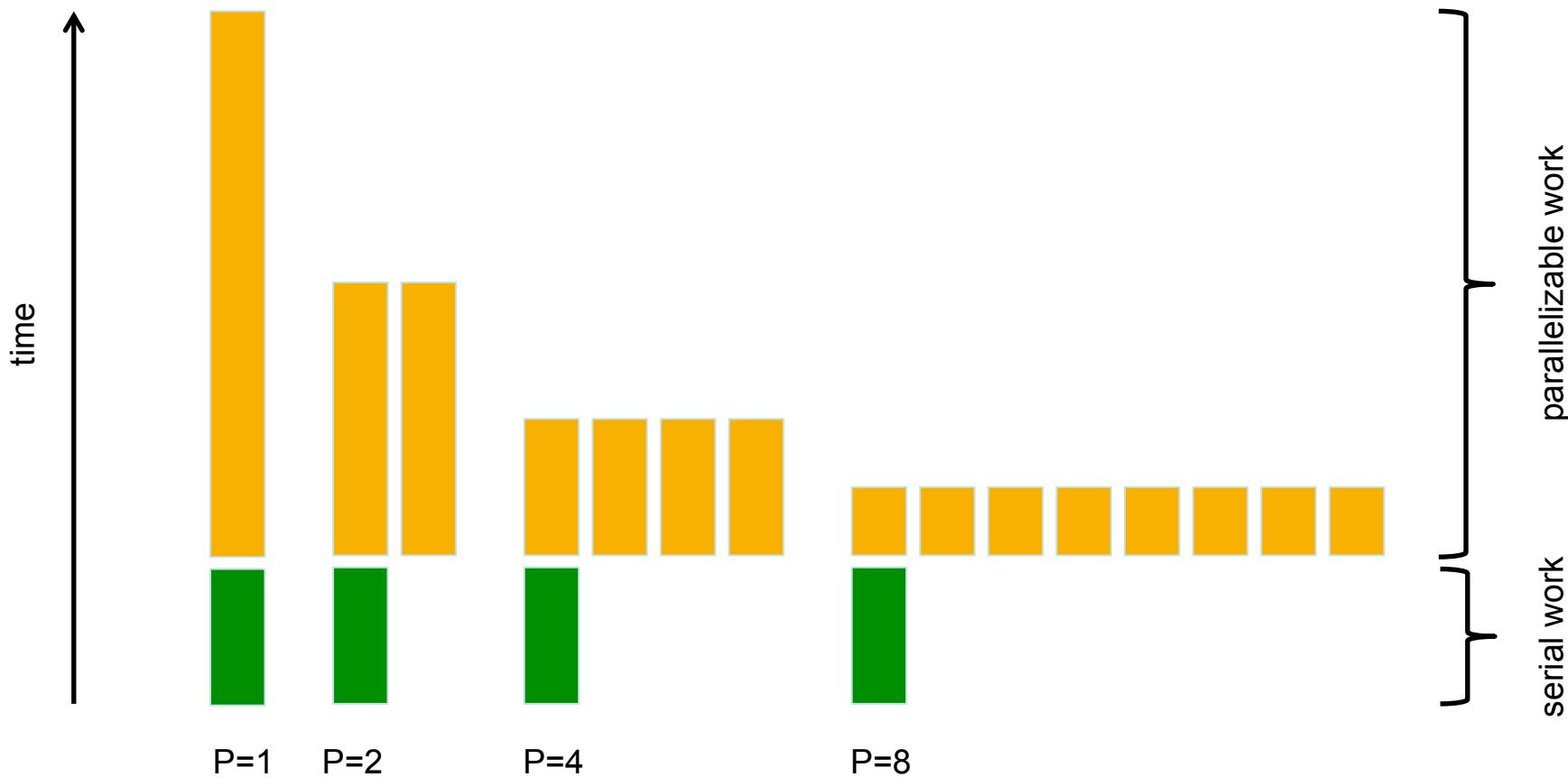
$$\text{Speedup}(p) = \frac{1}{f + \frac{1-f}{p}}$$

Overall speedup of 80 on 100 processors

$$80 = \frac{1}{f + \frac{1-f}{100}}$$

$$f = 0.00\overline{25}$$

Amdahl's law (3)



Parallel efficiency



$$\text{Efficiency}(p) = \frac{\text{Speedup}(p)}{p}$$

- Metric for cost of parallelization (e.g., communication)
- Without super-linear speedup

$$\text{Efficiency}(p) \leq 1$$

Super-linear speedup possible

- Critical data structures may fit into the aggregate cache

Law of Gustafson

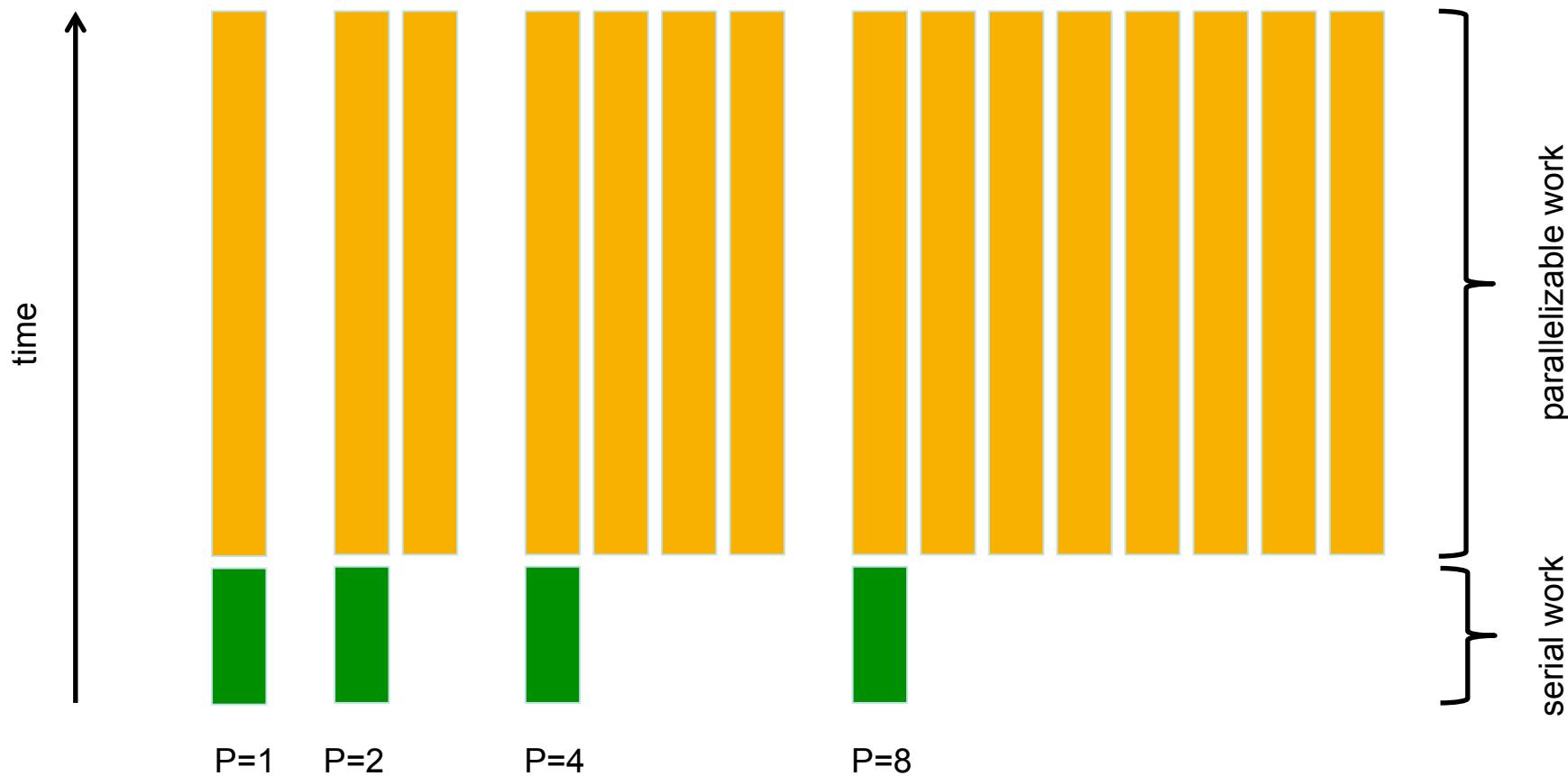


- Amdahl's Law ignores increasing problem size
 - Parallelism often applied to calculate bigger problems instead of calculating a given problem faster
- Fraction of sequential part may be function of problem size
- Assumption
 - Sequential part has constant runtime τ_f
 - Parallel part has runtime $\tau_v(n,p)$
- Speedup

If parallel part can be perfectly parallelized

$$\text{Speedup}(n,p) = \frac{\tau_f + \tau_v(n,1)}{\tau_f + \tau_v(n,p)}$$

Law of Gustafson (2)





Weak scaling

- Ability to solve a larger input problem by using more resources (here: processors)
- Problem size per processor remains constant
- Example: larger domain, more particles, higher resolution

Strong scaling

- Ability to solve the same input problem faster as more resources are used
- Usually more challenging
- Limited by Amdahl's law and communication demand

Bandwidth and latency



Bandwidth or throughput

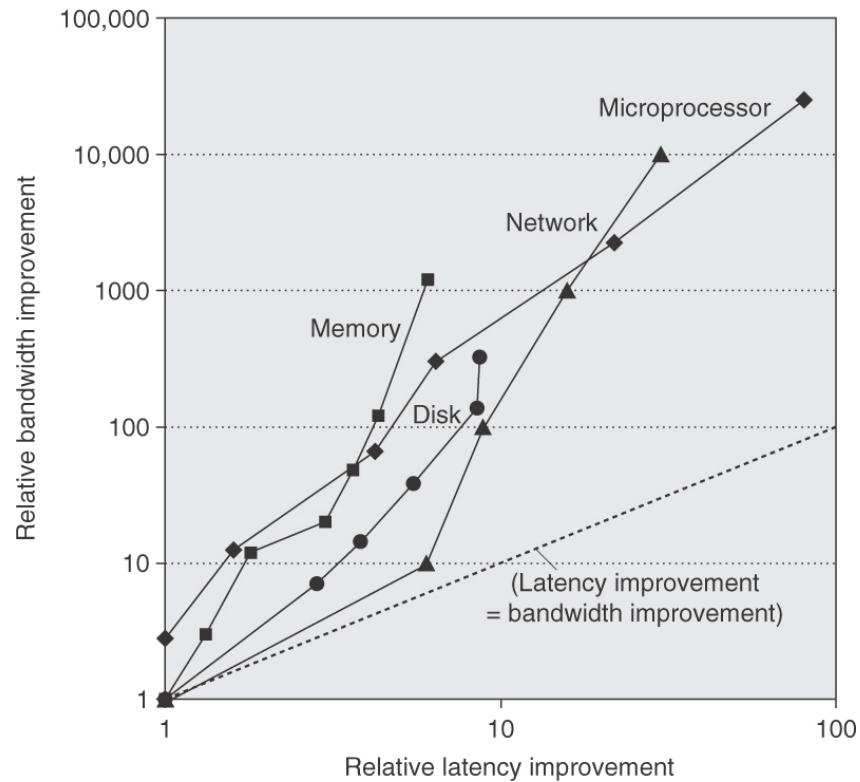
- Total amount of work done in a given time (e.g., disk transfer rate)

Latency or response time

- Time between start and completion of an event (e.g., disk access time)

Rule of thumb

- Bandwidth grows by at least the square of the improvement in latency



Source: Hennessy, Patterson: Computer Architecture, 5th edition, Morgan Kaufmann

Abstract models of parallel machines

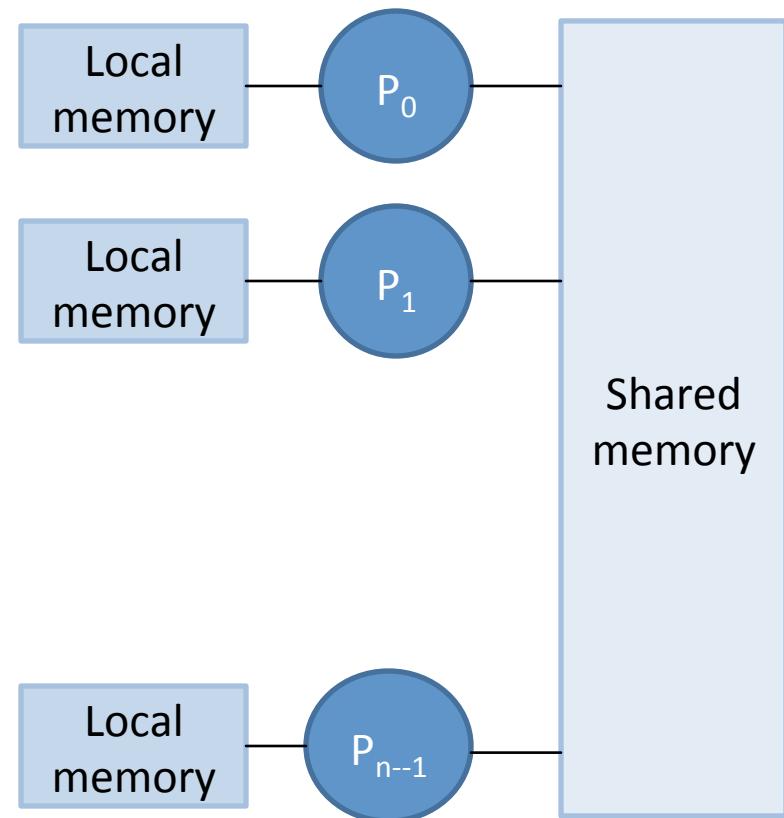


- Used by parallel algorithms designers to estimate algorithm performance independent of specific
 - Hardware
 - Programming language
- Define on abstract level
 - Basic operations
 - When the corresponding actions take place
 - How to access data
 - How to store data
- Objective
 - Abstract from unnecessary details
 - But cover essential characteristics of a broad class of systems

Parallel Random Access Machine (PRAM)



- Any (problem-size dependent) number of processors
- Access to shared memory in unit time
- Input and output stored in shared memory
- Processors execute instructions synchronously
- Cost of an algorithm specified in terms of the number of required PRAM instructions
 - Usually asymptotic behavior as function of problem size
 - Example: $O(n^2)$



Parallel Random Access Machine (2)



- Suitable for the theoretical analysis of algorithms
- Impractical for prediction of actual runtimes on given machine because of too many simplifying assumptions
 - No limit on the number of processors
 - Any memory location uniformly accessible from any processor
 - There is no limit on the amount of shared memory
 - Resource contention is absent



- Alternative machine model for parallel computation
 - Arbitrary number of processing units with distributed memory
 - Processing units are connected through an abstract communication medium which allows point-to-point communication
 - Still simple enough to allow the creation of performance models
- Defined by four parameters

L = latency of the communication medium

o = overhead of sending and receiving a message

g = the gap required between two send/receive operations

P = the number of processing units

Asymptotic complexity



- Example: dot product of two vectors of length $N \geq P$
 - Split vector into pieces of length N/P
 - Calculate subproducts in parallel
 - Calculate global sum in tree-like fashion
- Asymptotic running time

$$\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} * \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

$$T(N, P) = \Theta(N / P + \lg P)$$

Asymptotic complexity notation



- Big **O** notation – denotes a set of functions with an upper bound.
 $O(f(N))$ = set of all functions $g(N)$ such that there exists positive constants c , N_0 with $|g(N)| \leq c * f(N)$ for $N \geq N_0$.
- Big **Omega** notation – denotes a set of functions with a lower bound.
 $\Omega(f(N))$ = set of all functions $g(N)$ such that there exists positive constants c , N_0 with $|g(N)| \geq c * f(N)$ for $N \geq N_0$.
- Big **Teta** notation – denotes a set of functions with both lower and upper bounds. $\Theta(f(N))$ = set of all functions $g(N)$ such that there exists positive constants c_1 , c_2 , N_0 with $c_1 * f(N) \leq |g(N)| \leq c_2 * f(N)$ for $N \geq N_0$.



Asymptotic speedup

- Asymptotic speedup

$$\frac{T_1}{T_P} = \frac{\Theta(N)}{\Theta(N/P + \lg P)} = \Theta\left(\frac{N}{N/P + \lg P}\right)$$

- Asymptotic efficiency

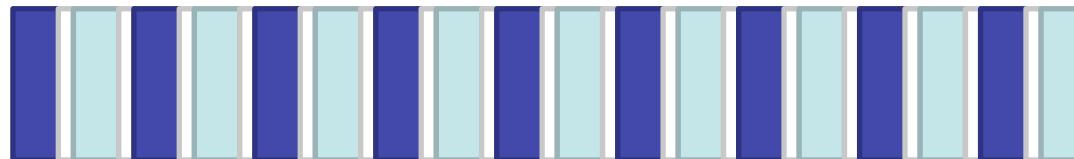
$$\frac{T_1}{P \cdot T_P} = \Theta\left(\frac{N}{N + P \lg P}\right)$$

Time sharing



- Also called **multitasking**
- Logical extension of multiprogramming
- CPU executes multiple processes by switching among them
- Switches occur frequently enough so that users can interact with each program while it is running
- On a multiprocessor, processes can also run truly concurrently, taking advantage of additional processors

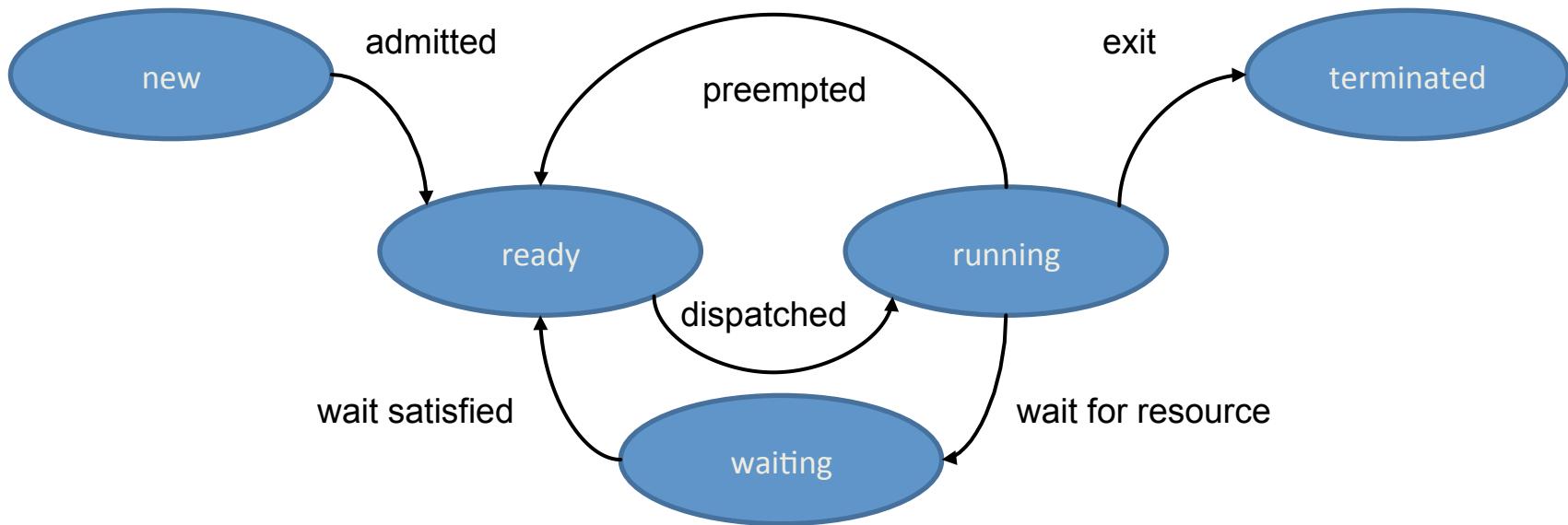
Single core



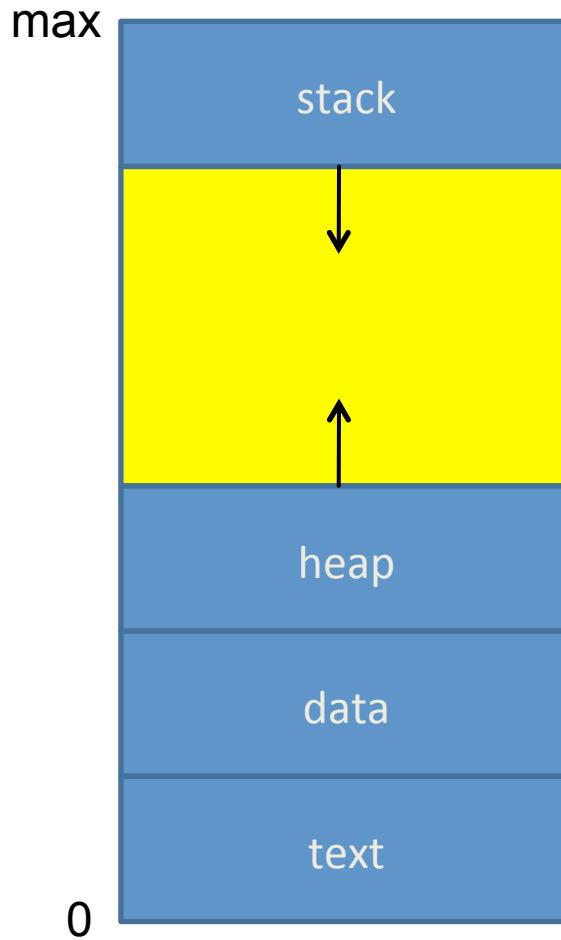
Process



- A process is a program in execution
- States of a process



Processes in memory



Temporary data: function parameters,
return addresses, local variables

Dynamically allocated memory

Global variables

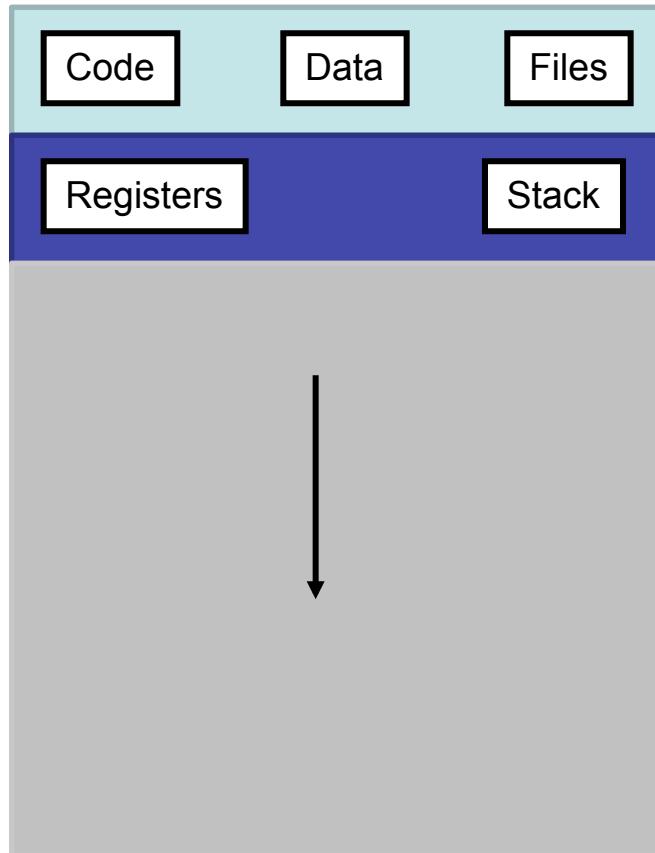
Program code

Thread

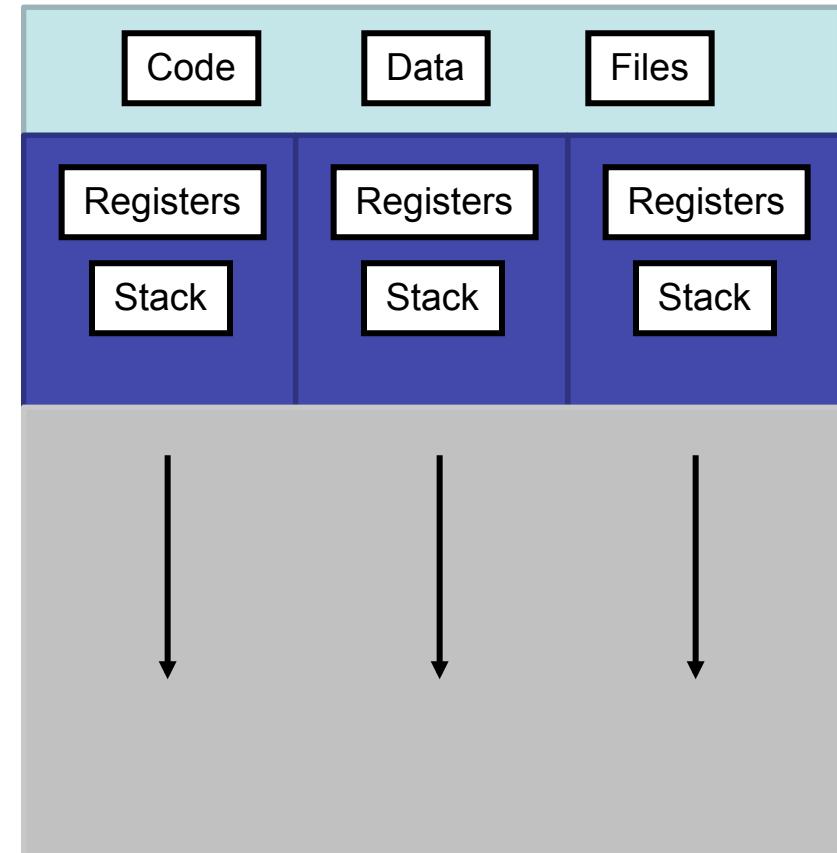


- Basic unit of CPU utilization
 - Flow of control within a process
- A thread includes
 - Thread ID
 - Program counter
 - Register set
 - Stack
- Shares resources with other threads belonging to the same process
 - Text (i.e., code) section
 - Data section (i.e., address space)
 - Other operating system resources (e.g., open files, signals)

Single-threaded vs. multi-threaded

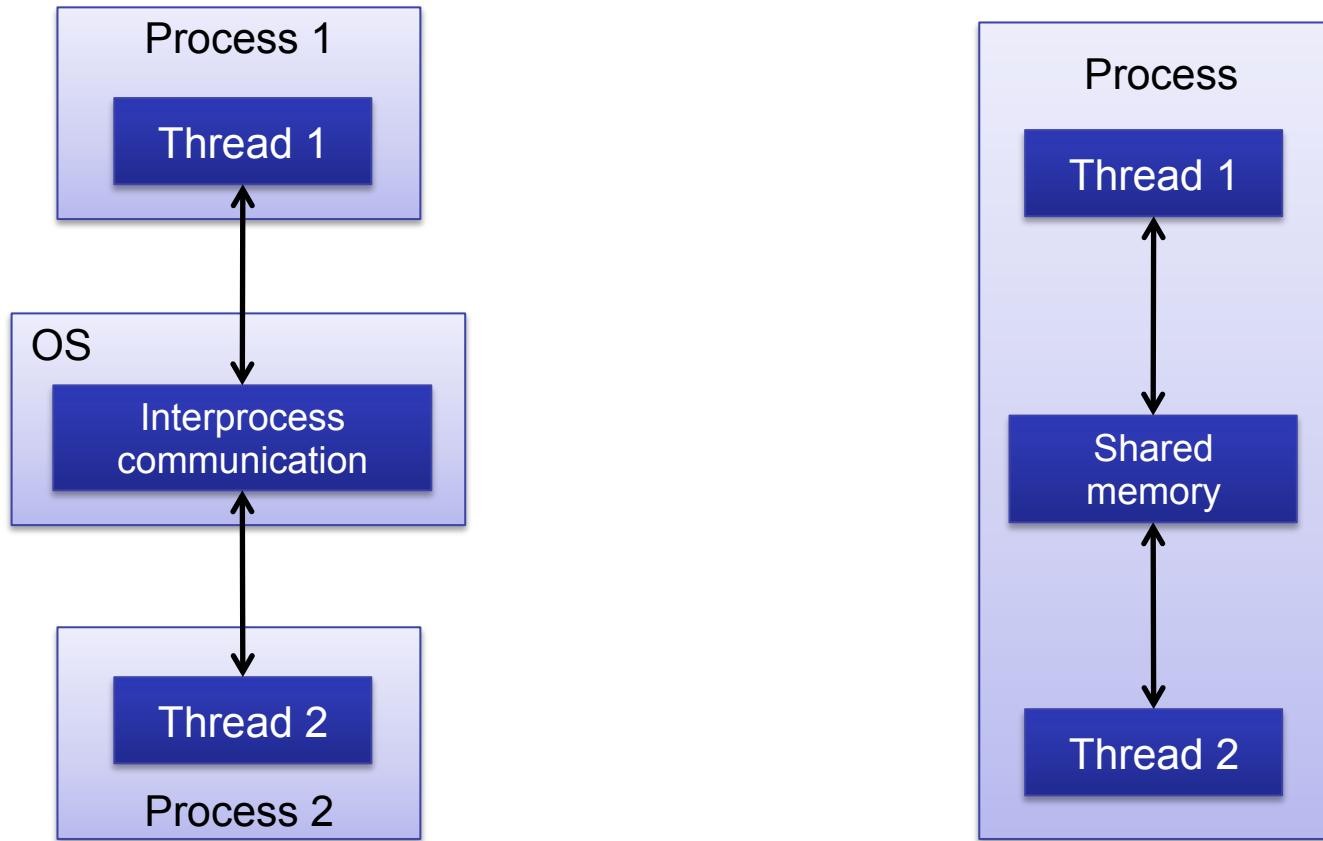


Single-threaded

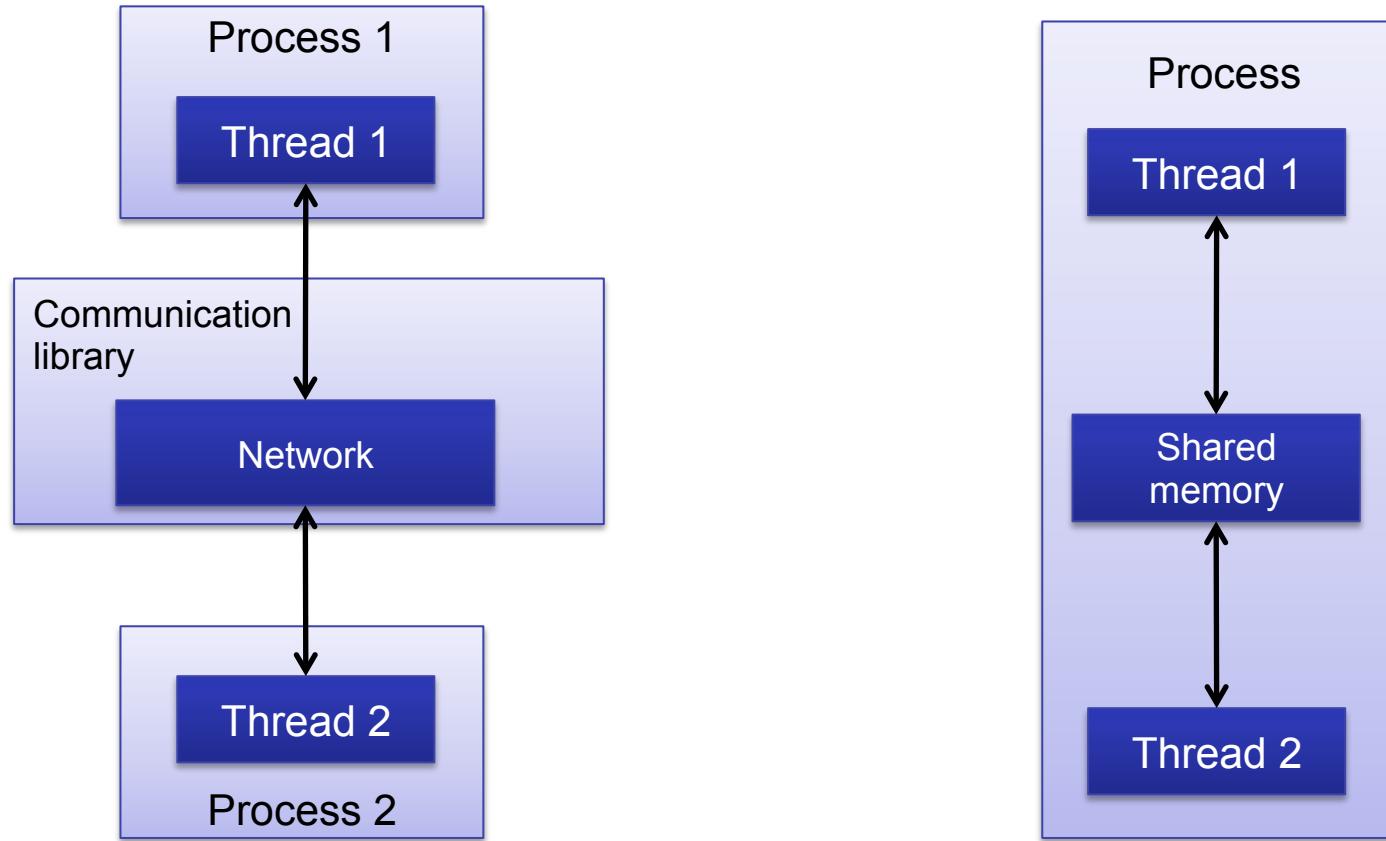


Multi-threaded

Concurrency – processes vs. threads



Concurrency – processes vs. threads (2)



Concurrency – processes vs. threads (3)



Processes (with distributed memory)

- Communication explicit
- Often requires replication of data
- Address spaces protected
- Parallelization usually implies profound redesign
- Writing debuggers is harder
- More scalable

Threads (with shared memory)

- Convenient communication via shared variables
- More space efficient - sharing of code and data
- Context switch cheaper
- Incremental parallelization easier
- Harder to debug – race conditions

Summary



- Peak performance usually never reached
- A memory hierarchy assumes locality of accesses
- Amdahl's law applies to strong scaling
- Law of Gustavson takes weak scaling into account
- LogP abstract machine model designed for distributed memory
- Asymptotic complexity is an effective instrument to reason about an algorithm's scalability



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Large-Scale Parallel Computing

Prof. Dr. Felix Wolf

PARALLEL ARCHITECTURES

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

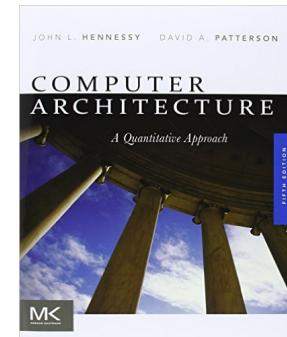
- Classification
- Memory architecture
- Interconnection networks
- Examples
 - Lichtenberg Cluster
 - IBM BlueGene/Q

Literature

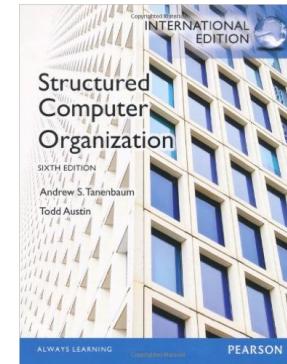


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- John L. Hennessy, David A. Patterson: Computer Architecture: A Quantitative Approach, 5th edition, Morgan Kaufmann, 2011



- Andrew S. Tanenbaum, Todd Austin: Structured Computer Organization, 6th edition, Pearson, 2013



Taxonomies



- Number of instruction streams vs. number of data streams
- Memory architecture
- Network architecture
- Degree of heterogeneity
- Degree of customization
- Size of nodes in relation to number of nodes

Flynn's classification [1966]



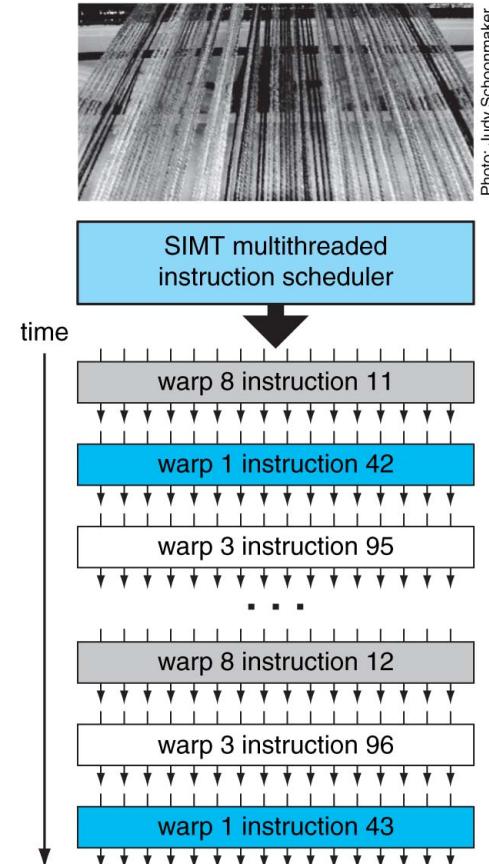
- Single instruction stream, single data stream
 - Classical uniprocessor
- Single instruction stream, multiple data streams
 - Same instruction is executed by multiple processors using different data streams
 - Data parallelism
 - Examples: SIMD extensions for multimedia, vector processors
- Multiple instruction streams, single data stream
 - No commercial multiprocessor of this type ever built
- Multiple instruction streams, multiple data streams
 - Each processor fetches its own instructions and operates on its own data
 - Thread-level parallelism

Single-instruction multiple threads (SIMT)

- Used on GPUs -

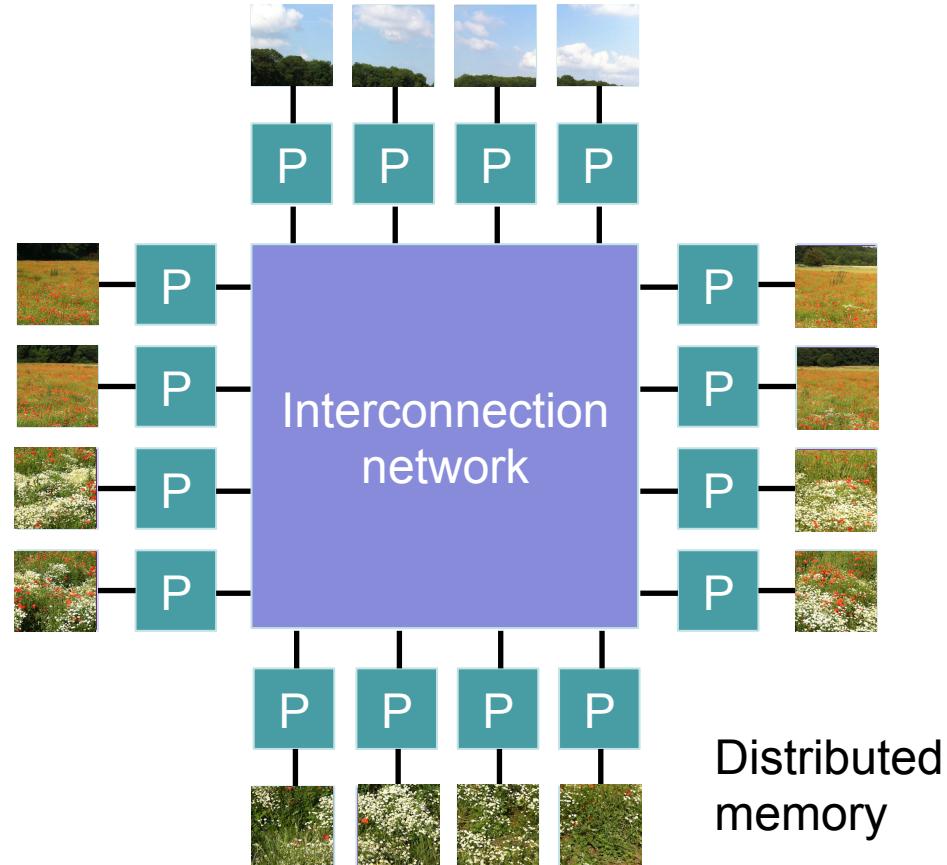
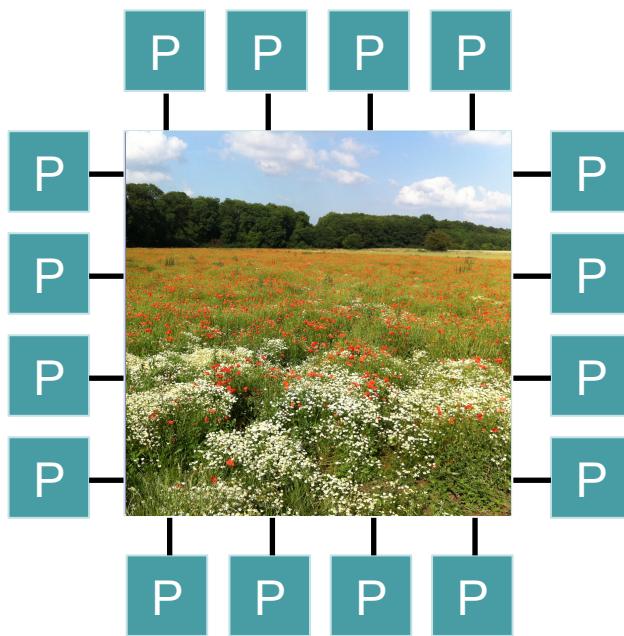


- Creates, manages, schedules, and executes threads in groups of parallel threads called **warps**
- At each instruction issue time, SIMT instruction unit
 - Selects warp that is ready to execute its next instruction
 - Broadcasts instruction to all active threads of that warp
- Individual threads may be inactive to do independent branching



- Architecture of choice for general-purpose multiprocessors
- Offers high degree of flexibility
 - High performance for one application or multi-programmed multiprocessor
- Can take advantage of off-the-shelf processors
- Popular execution model - Single Program Multiple Data (SPMD)
 - The same program is executed in parallel with each instance having a potentially different control flow

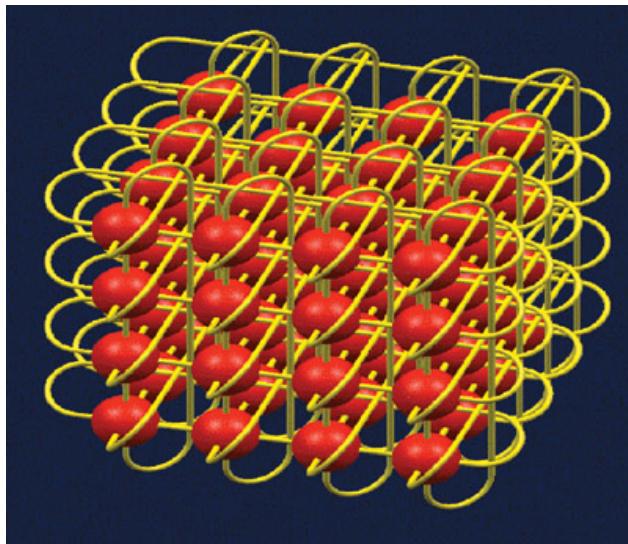
Memory architecture



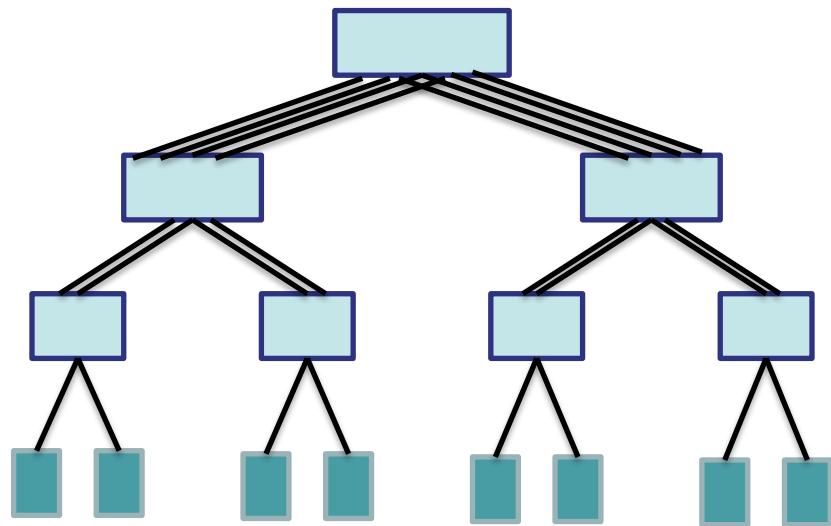
Popular network architectures for distributed memory systems



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Torus
(distributed switched network)

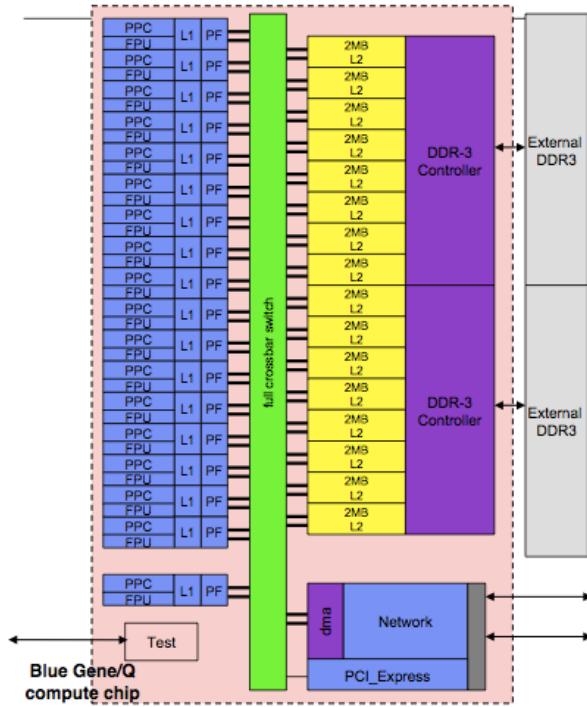


Fat tree
(centralized switched network)

Degree of heterogeneity

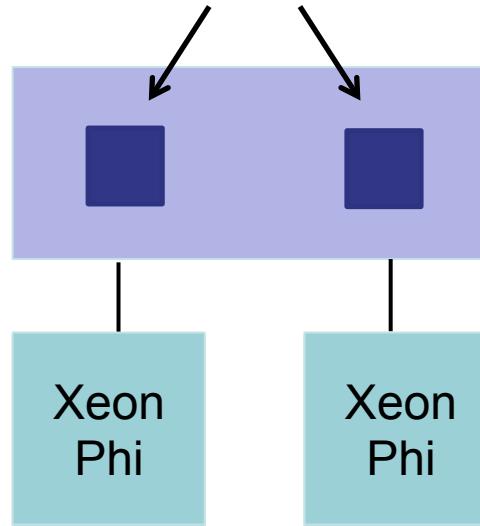


Homogeneous node architecture



BlueGene/Q
chip architecture

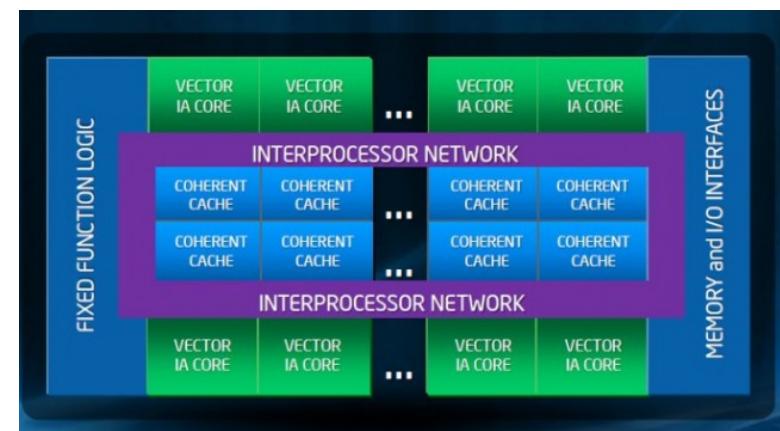
Classic server CPUs
(e.g. Intel Xeon)



Accelerators

Heterogeneous node architecture

Accelerators



Intel Xeon Phi

NVIDIA Kepler GPU

Degree of customization



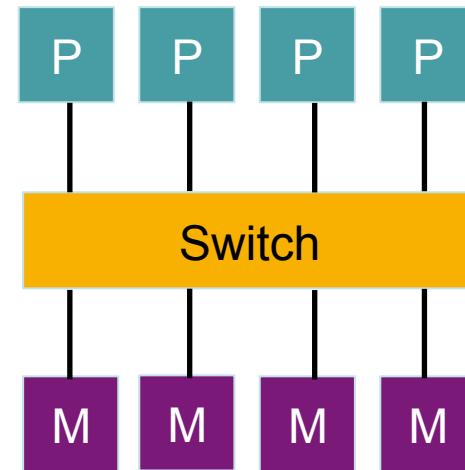
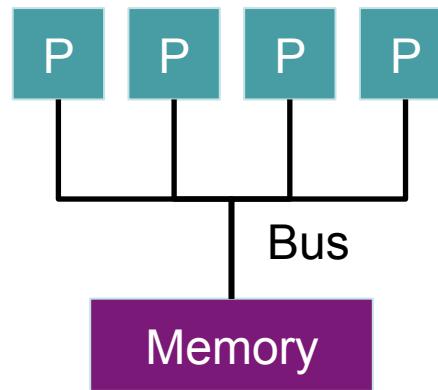
- **Commodity clusters** – standard nodes and standard network
 - Focus on applications with small communication requirements
 - Example: Beowulf cluster
- **Custom clusters** – custom nodes and custom network
 - Also called massively parallel processors
 - Focus on applications that exploit large amounts of parallelism on single problem
 - Example: IBM Blue Gene/Q
- Above classes are extremes of a broad spectrum

Shared memory



UMA (Uniform memory access)

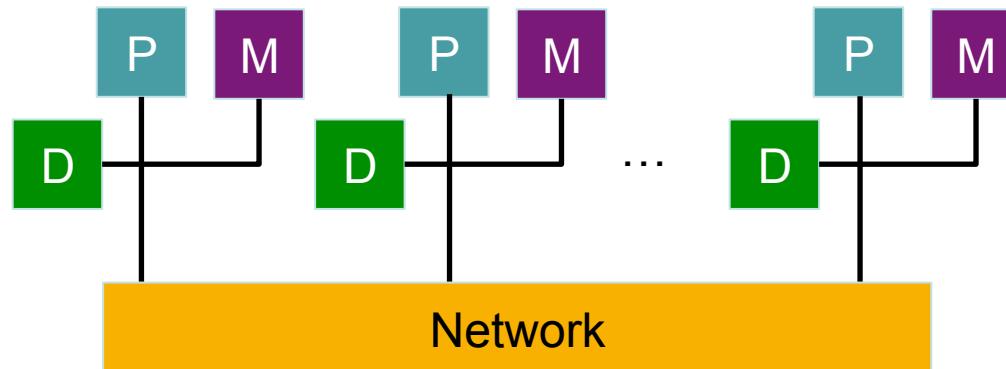
- Each CPU has same access time to each memory address
- Simple design but limited scalability (multicore or less)



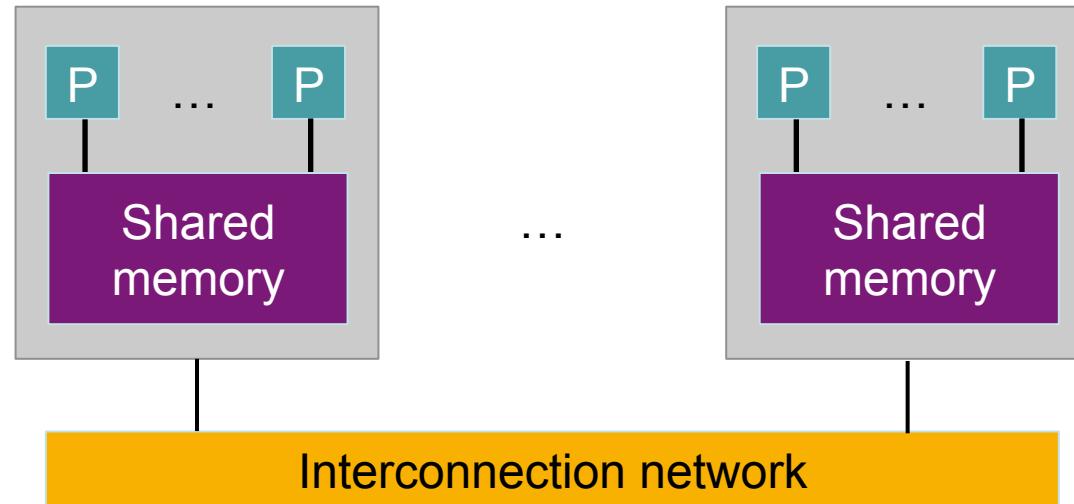
Shared memory (2)

NUMA (Non-uniform memory access)

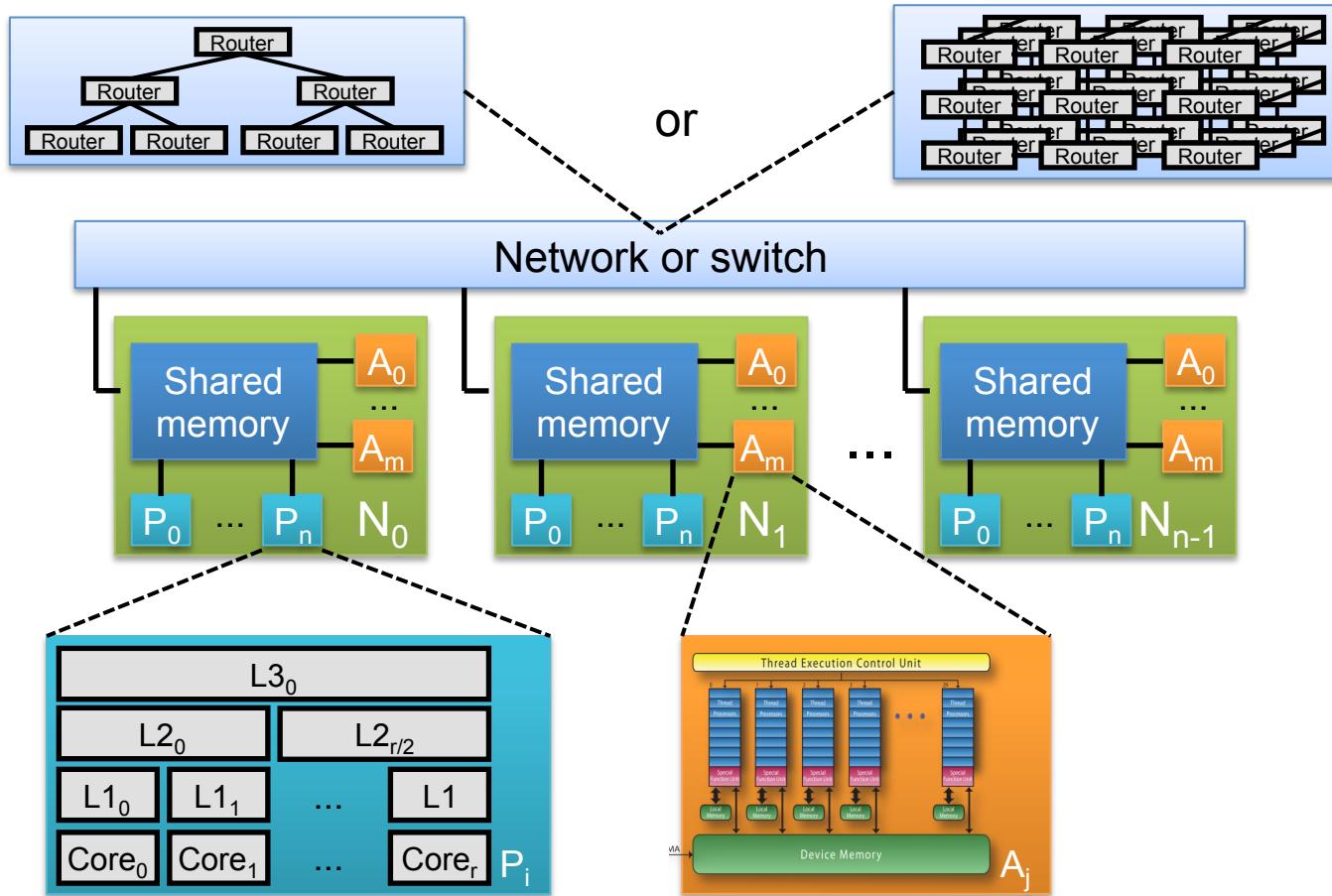
- Memory has affinity to a processor
- Access to local memory faster than to remote memory
- Harder to program but more scalable



Distributed memory (aka multicompiler)



Typical supercomputer architecture



Interconnection network



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Physical link between components of a parallel system

- Between processors and memory
- Between nodes

Communication via exchange of messages

- Example: intermediate results, memory requests

Design elements

- **Topology** – determines geometric layout of links and switches
- **Routing technique** – determines paths of messages through network

Bandwidth

- Maximum rate at which information can be transferred
- Aggregate bandwidth – total data bandwidth supplied by network
- Effective bandwidth or throughput – fraction of aggregate bandwidth delivered to an application

Latency

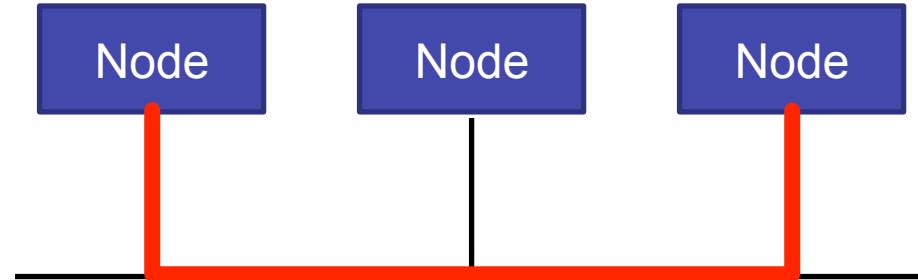
- Sending overhead + time of flight + receiving overhead



Shared-media networks



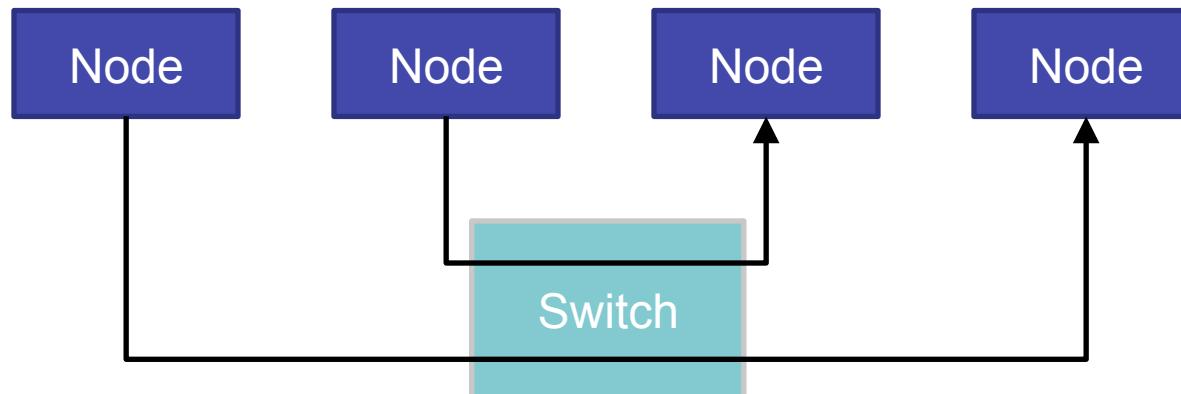
- Only one message at a time – processors broadcast their message over the medium
- Each processor “listens” to every message and receives the ones for which it is the destination
- Decentralized arbitration
 - Before sending a message, processors listen until medium is free
 - Message collision can degrade performance
- Low cost but not scalable
- Example – bus networks to connect processors to memory



Switched-media networks



- Support point-to-point messages between nodes
- Each node has its own communication path to the switch
- Advantages
 - Support concurrent transmission of multiple messages among different node pairs
 - Scale to very large numbers of nodes



Centralized switched networks

- Also called indirect or dynamic interconnection networks
- Connect processors / memory indirectly using several links and intermediate switches
- Examples: switching networks
- Used both for shared- and distributed-memory architectures

Crossbar switch

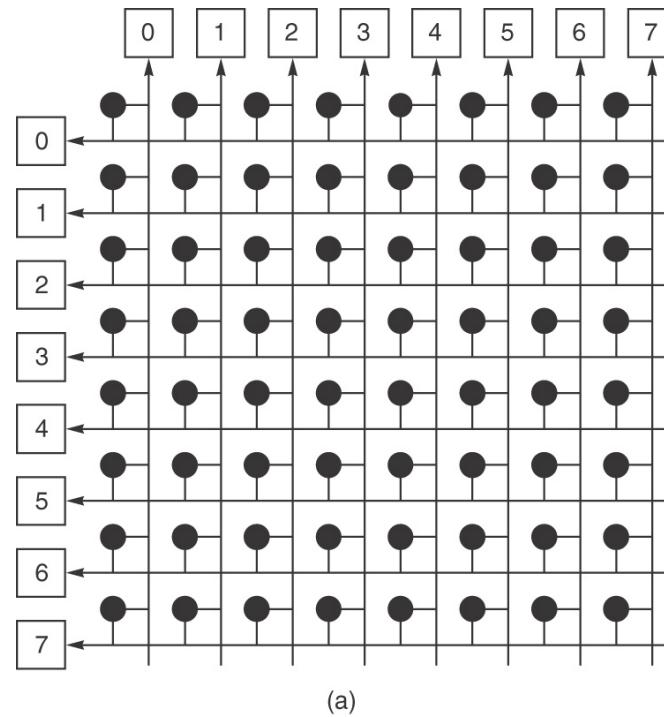


Non-blocking

- Links are not shared among paths to unique destinations

Requires N^2 crosspoint switches

- Limited scalability

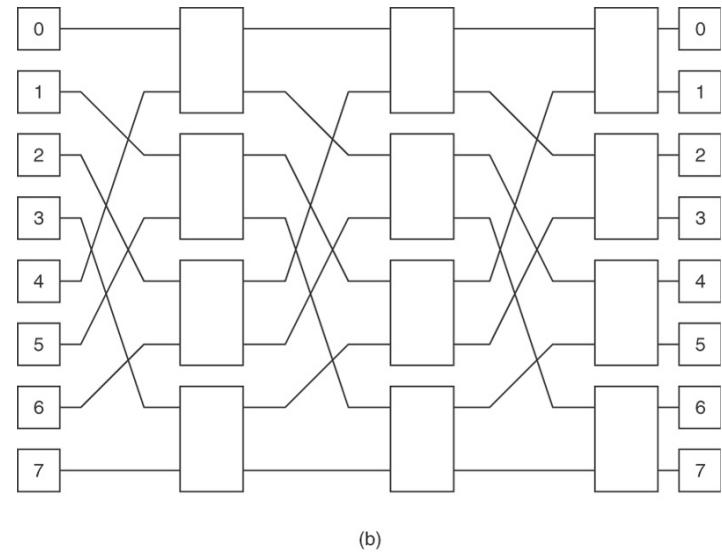


Source: Hennessy, Patterson: Computer Architecture, 4th edition, Morgan Kaufmann

Multistage interconnection network (MIN)

Example: Omega network

- Complexity $O(N \log N)$
- Perfect shuffle permutation at each stage
- Blocking due to paths between different sources and destinations simultaneously sharing network links
- Omega with $k \times k$ switches
 - $\log_k N$ stages ; $N/k \log_k N$ switches



(b)

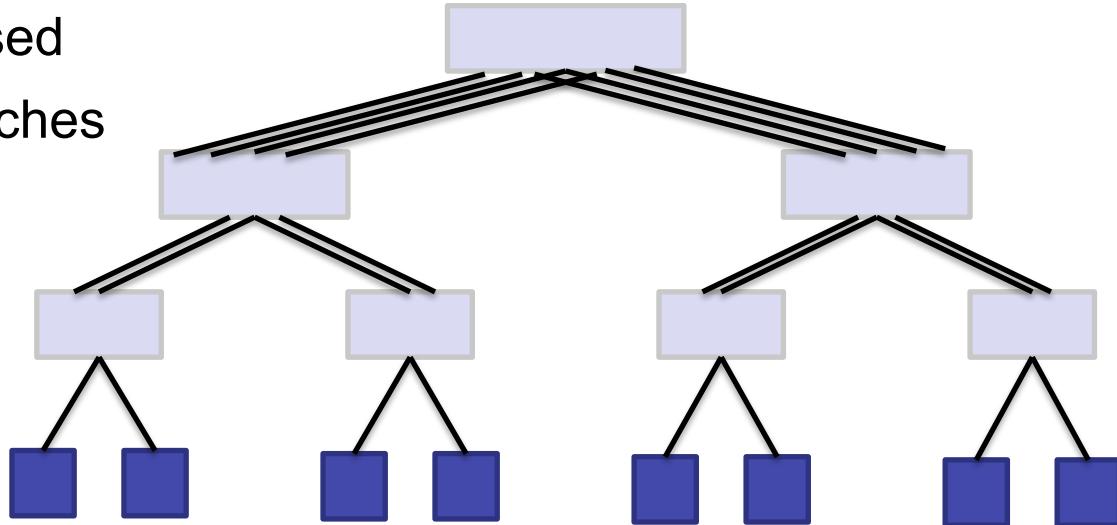
© 2007 Elsevier, Inc. All rights reserved.

Source: Hennessy, Patterson: Computer Architecture, 4th edition, Morgan Kaufmann

MINs can be extended to **rearrangeably** non-blocking topologies

Fat tree

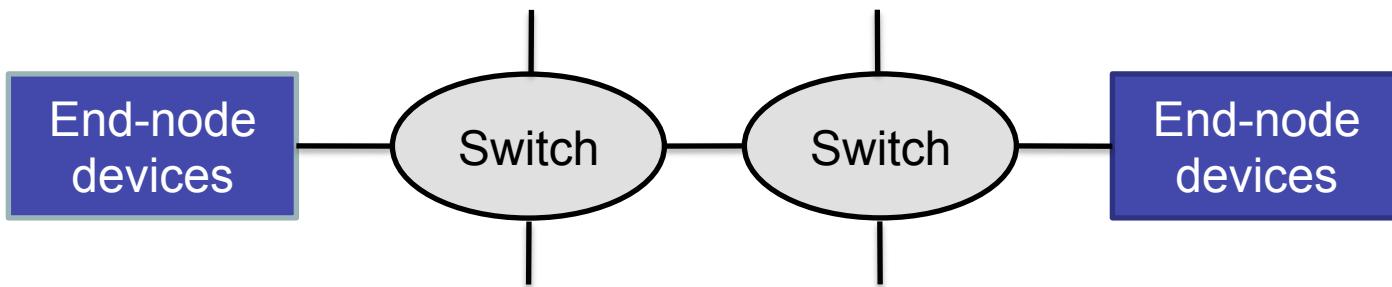
- Balanced tree where
 - Leaves = end node devices
 - Vertices = switches
- Total link bandwidth constant across all levels
- Switches often composed of multiple smaller switches
- Popular topology for cluster interconnects



Distributed switched networks



- Each network switch has one or more end node devices directly attached to it
- End node devices = processor(s) + memory
 - Directly connected to other nodes without going through external switches
 - Mostly used for distributed-memory architectures
- Also called direct or static interconnection networks
- Ratio of switches to nodes = 1:1



Evaluation criteria



Network degree

- Maximum node degree
- Node degree = number of adjacent nodes = (incoming + outgoing) edges

Diameter

- Largest distance between two nodes

Bisection width

- Minimum number of edges between nodes that must be removed to cut the network into two roughly equal halves

- Bisection bandwidth = bandwidth [bytes/s] between the two parts

Edge / node connectivity

- Minimum number of edges / nodes that need to be removed to render network disconnected

Embedding

- Mapping of one network onto another

Requirements

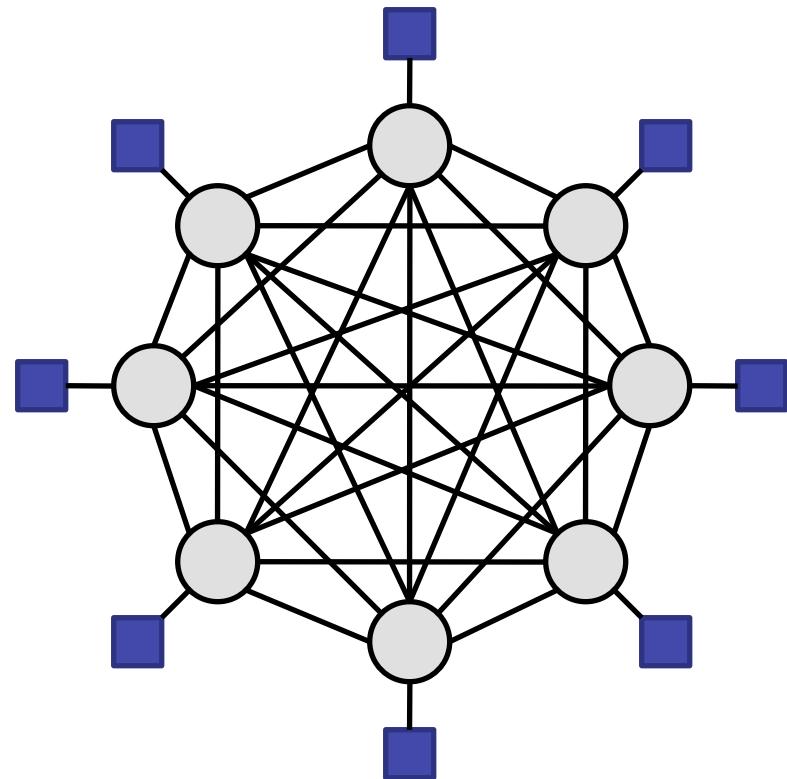


- Low network degree to reduce hardware costs
- Low diameter to ensure low distance (i.e., latency) for message transfer
- High bisection bandwidth to ensure high throughput
- High connectivity to ensure robustness
- Option to embed many other networks to ensure flexibility

Often conflicting goals

Fully connected topology

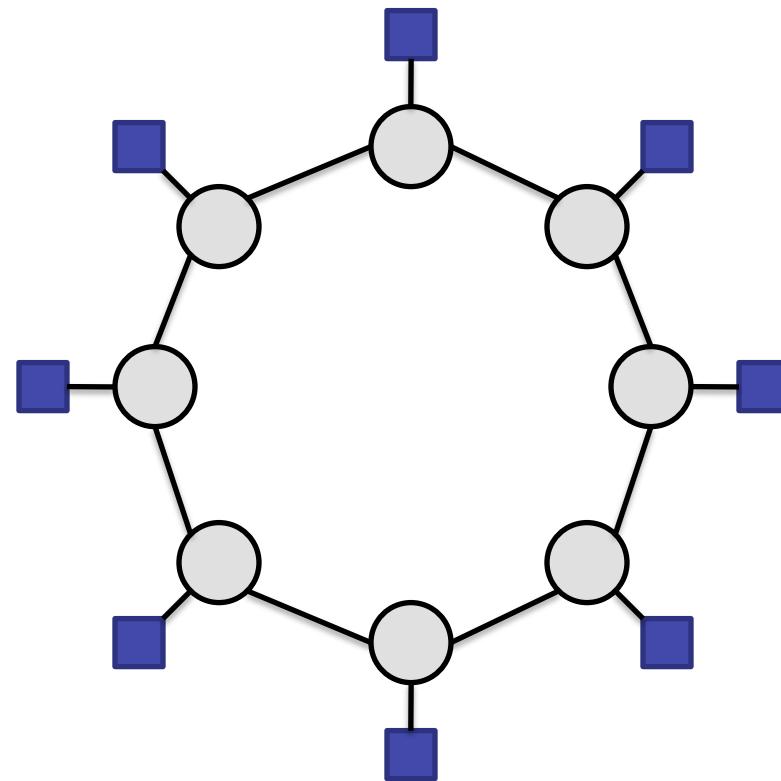
- Each node is directly connected to every other node
- Expensive for large numbers of nodes
- Dedicated link between each pair of nodes
- Cheaper alternative: crossbar topology



Ring topology

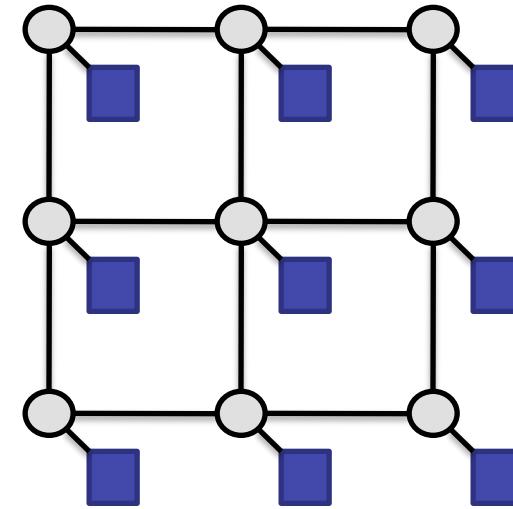


TECHNISCHE
UNIVERSITÄT
DARMSTADT



N-dimensional meshes

- Typically 2 or 3 dimensions
- Direct link to neighbors
- Each node has 1 or 2 neighbors per dimension
 - 2 in the center
 - Less for border or corner nodes
- Efficient nearest neighbor communication
- Suitable for large numbers of nodes

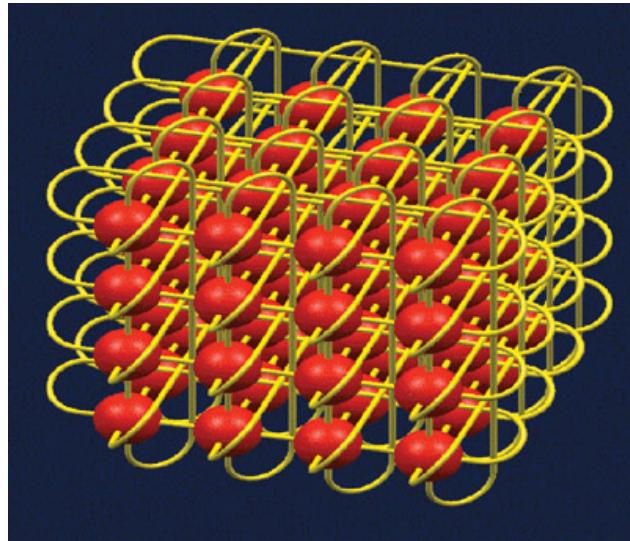


2D mesh

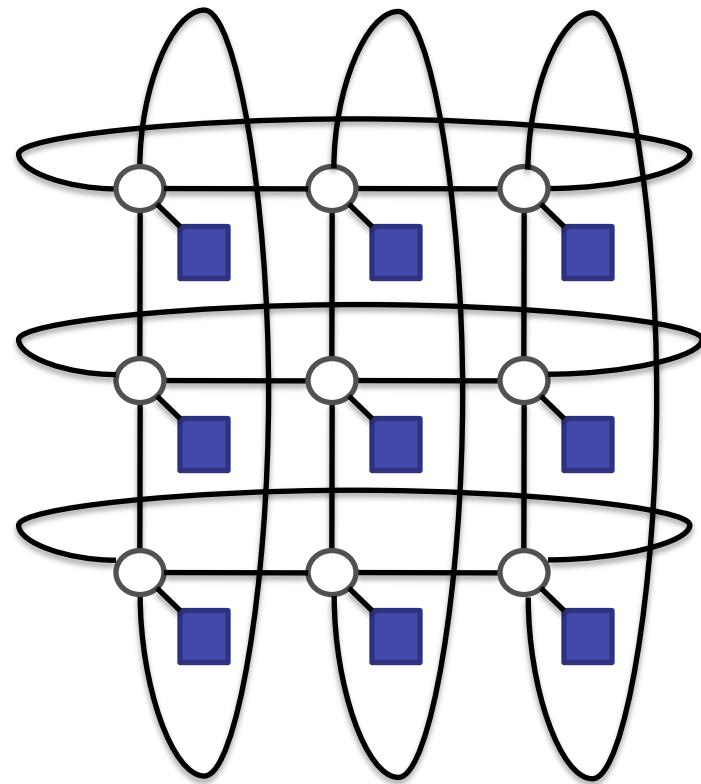
Torus



- Mesh with wrap-around connections
- Each node has exactly 2 neighbors per dimension



3D torus



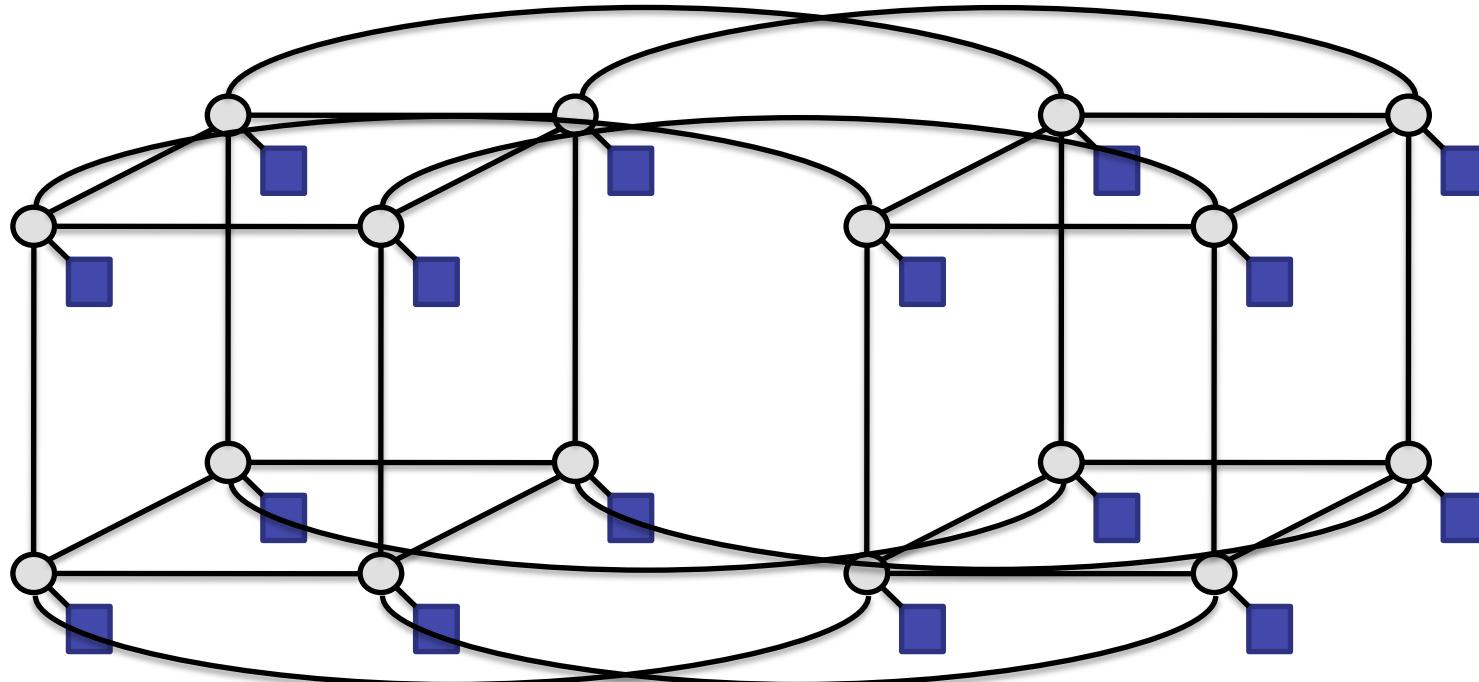
2D torus

Hypercube

16 nodes
 $(16 = 2^4 \text{ so } n = 4)$



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Each node has one connection along each dimension ($n = \# \text{dimensions}$)

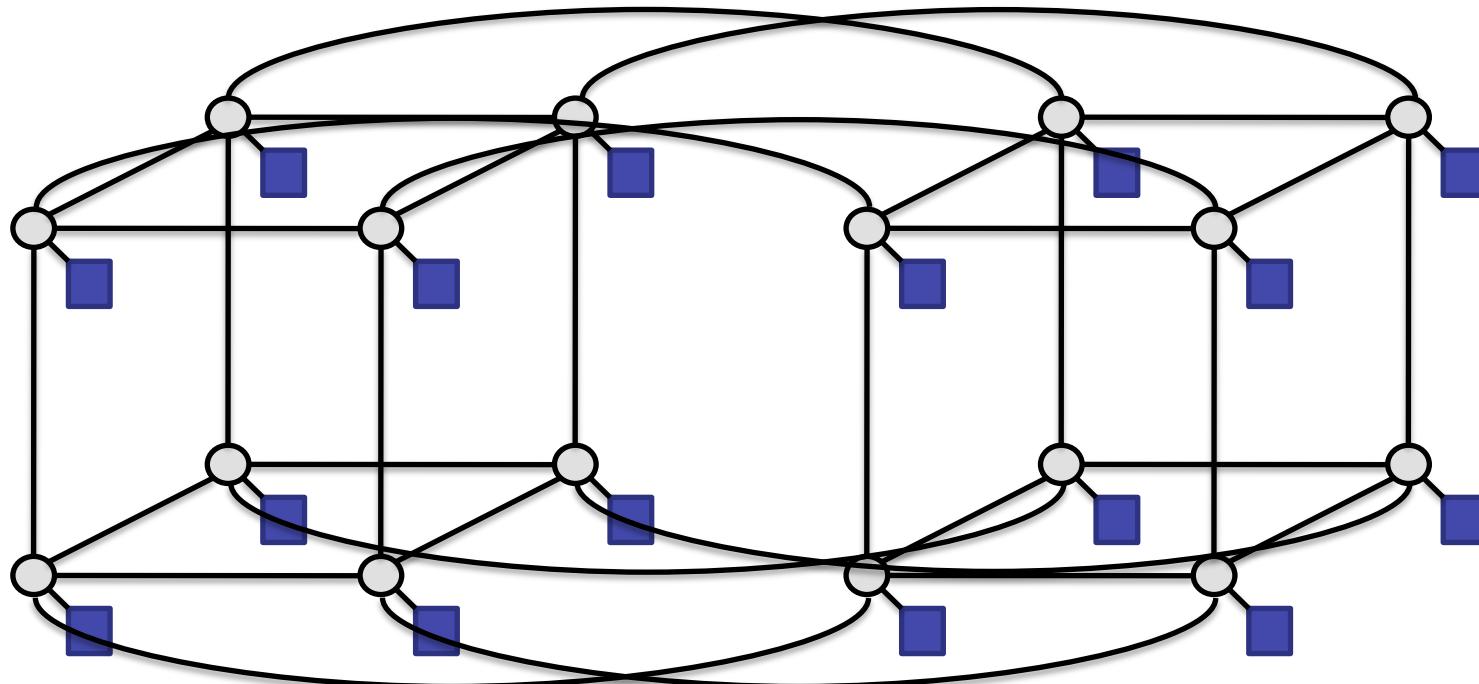
Usually better connectivity than tori at the expense of higher link and switch costs

Hypercube

16 nodes
 $(16 = 2^4 \text{ so } n = 4)$



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Each node has one connection along each dimension ($n = \# \text{dimensions}$)

Usually better connectivity than tori at the expense of higher link and switch costs

Performance and cost (64 nodes)



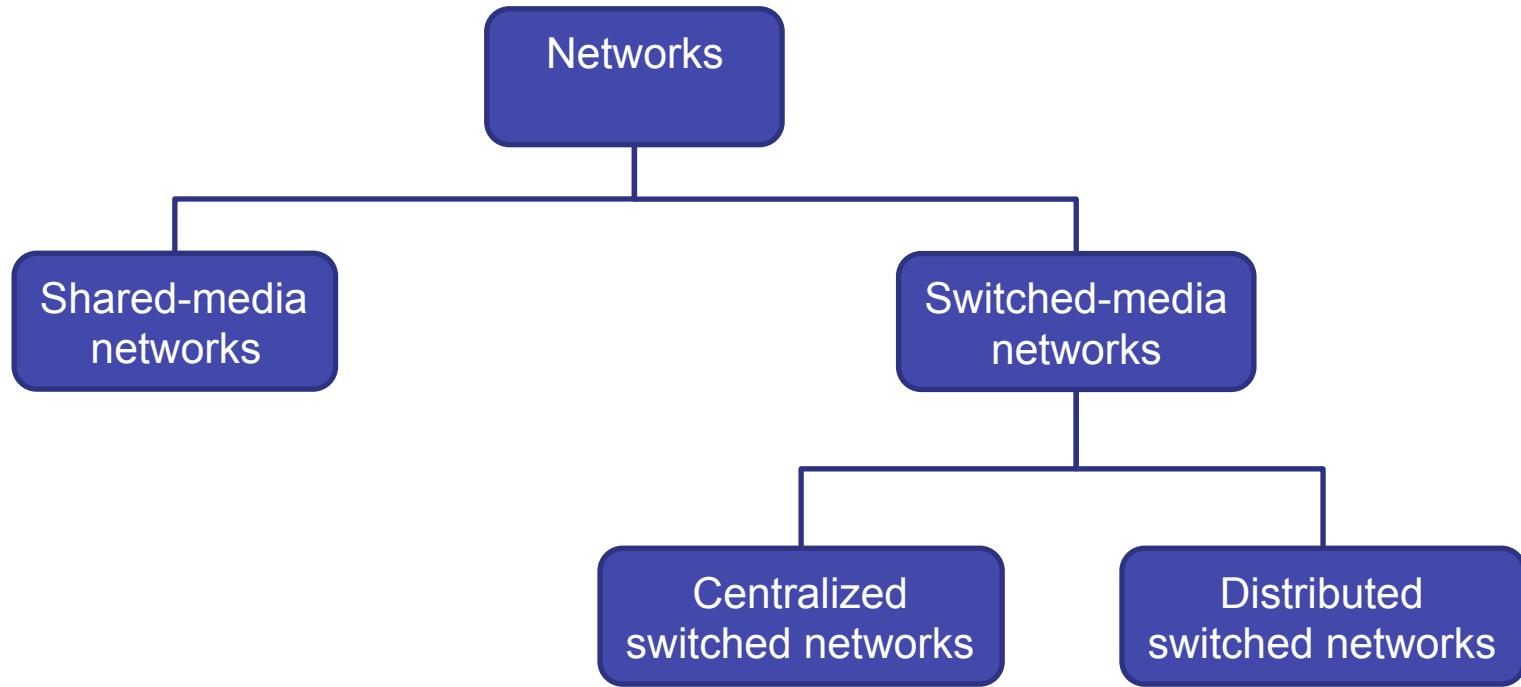
Evaluation category	Bus	Ring	2D mesh	2D torus	Hypercube	Fat tree	Fully connected
Performance							
BW bisection [#links]	1	2	8	16	32	32	1024
Max (avg.) hop count	1(1)	32(16)	14(7)	8(4)	6(3)	11(9)	1(1)
Cost							
I/O ports per switch	NA	3	5	5	7	4	64
#Switches	NA	64	64	64	64	192	64
#Network links	1	64	112	128	192	320	2016
Total #links	1	128	176	192	256	384	2080

Commercial HPC machines



Company	System name [network name]	Max. #nodes [x #CPUs]	Basic network topology
Intel	ASCI Red Paragon	4816 [x 2]	2D mesh 64 x 64
IBM	ASCI White SP Power3 [Colony]	512 [x 16]	Bidirectional MIN (fat tree or Omega)
Intel	Thunder Itanium2 Tiger 4 [QsNet ^{II}]	1024 [x 4]	Fat tree with 8-port bi-directional switches
Cray	XT3 [Seastar]	30,508 [x 1]	3D torus 40 x 32 x 24
Cray	X1E	1024 [x 1]	4-way bristled 3D torus (~23 x 11)
IBM	ASC Purple pSeries 575 [Federation]	> 1280 [x 8]	Bidirectional MIN (fat tree or Omega)
IBM	Blue Gene/L eServer Solution [Torus Network]	65,536 [x 2]	3D torus 32 x 32 x 64

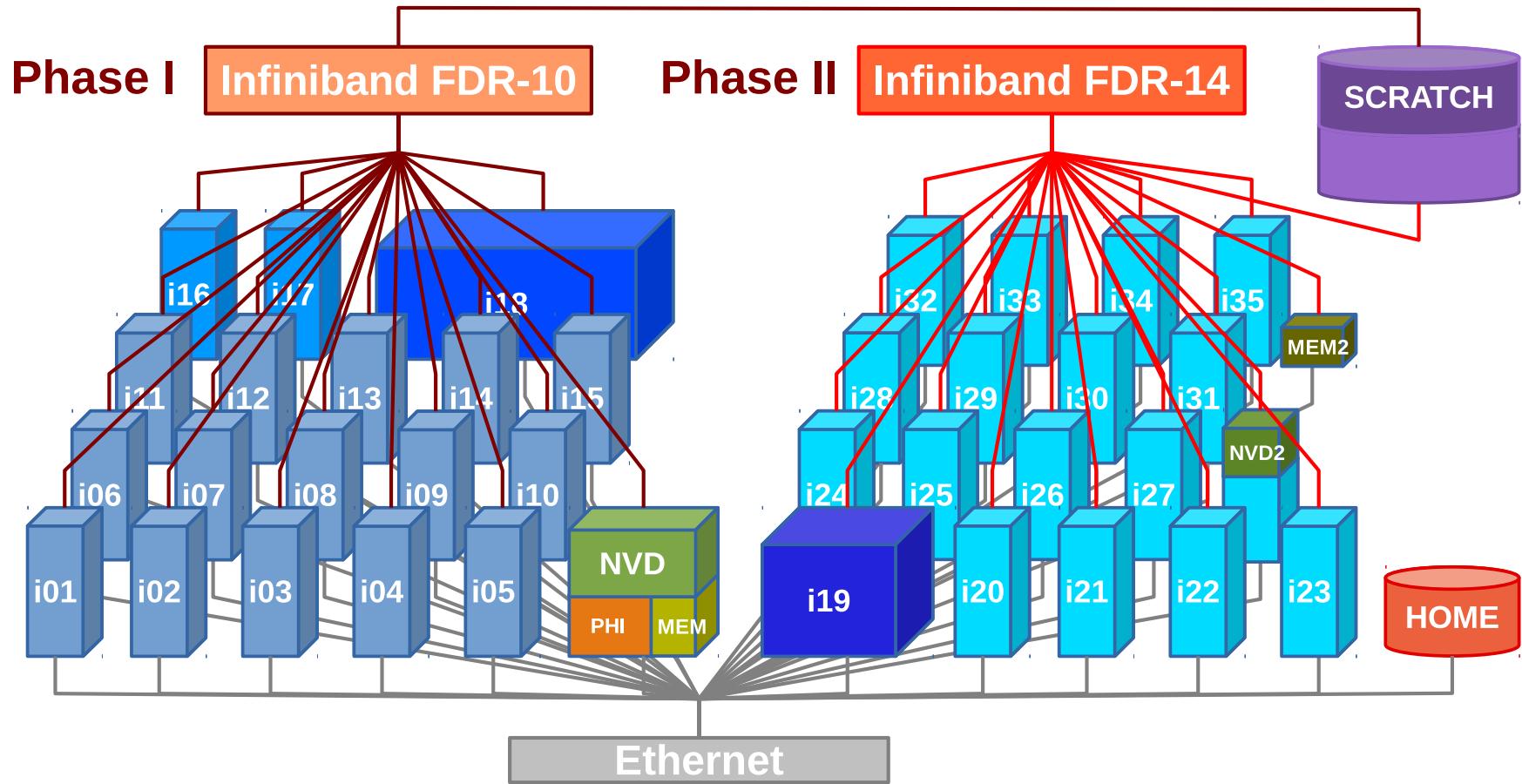
Network classes



Lichtenberg Cluster @ TU Darmstadt

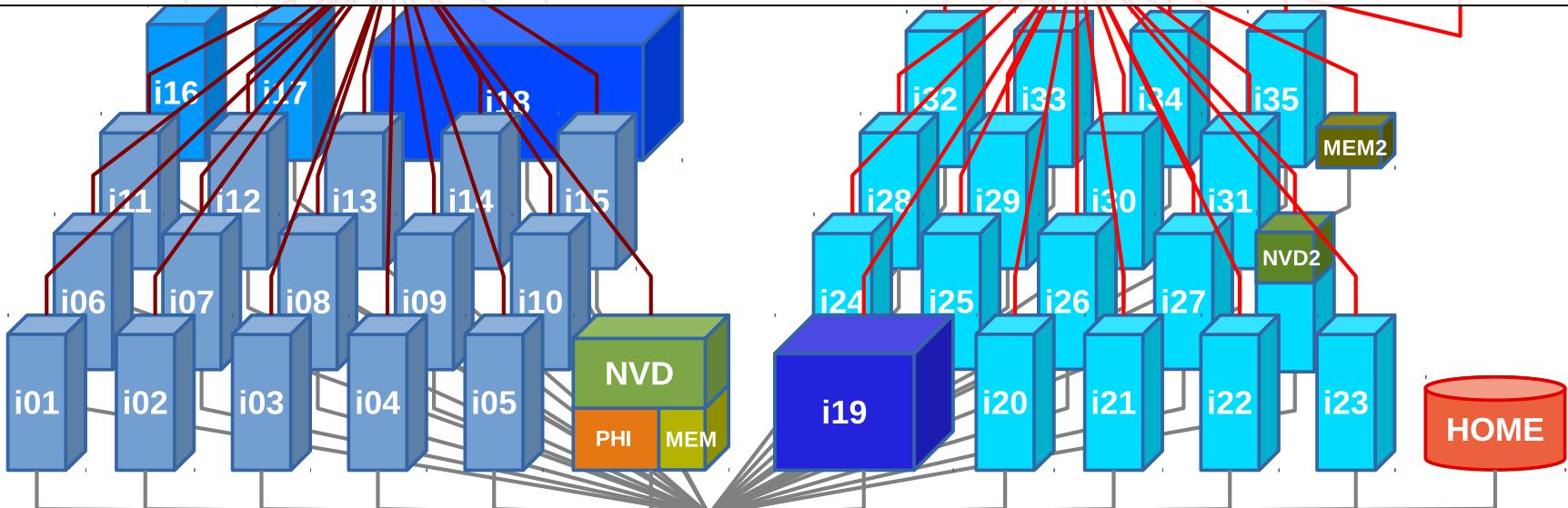


One cluster – multiple islands



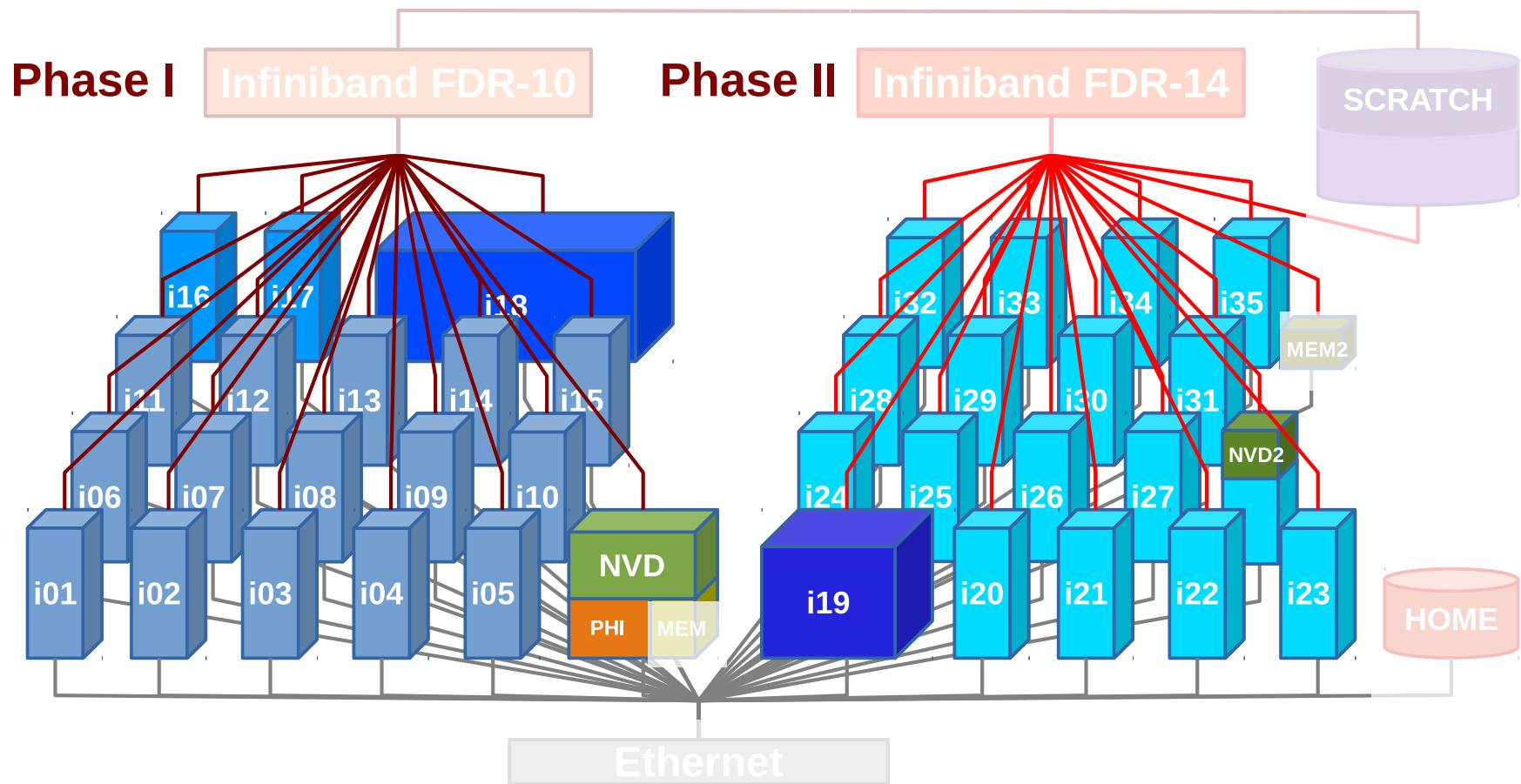
One cluster – multiple islands

- Cluster is divided into 2 phases
- Each phase is divided into several islands
- Rule of thumb: band FDR-10 $\text{Phase II Infiniband FDR-14}$
 $1 \text{ island} \hat{=} 32 \text{ compute nodes} \hat{=} 512 \text{ (ph. I) / } 768 \text{ (ph. 2) CPU cores}$
- For large computations, there are 2 islands with more than 2000 CPU cores



- Computation across more than one island is only possible on request, due to some technical limitations (across phases impossible).

Compute nodes (“mpi”, “nvd”, “phi”)



Compute nodes



Phase I (704+70 nodes):

Processors:

- 2 Intel Xeon E5-4650
(Sandy Bridge) processors
 $\triangleq 2 \cdot 8 = 16$ CPU cores

- 2.7 GHz
(up to 3.3 GHz in turbo mode)

Main Memory:

- 32 GB RAM (some have 64 GB)

Network:

- Gigabit Ethernet
- FDR-10 InfiniBand

Phase II (596+31 nodes):

Processors:

- 2 Intel Xeon E5-2680 v3
(Haswell) processors
 $\triangleq 2 \cdot 12 = 24$ CPU cores

- 2.5 GHz
(up to 3.3 GHz in turbo mode)

Main Memory:

- 64 GB RAM

Network:

- Gigabit Ethernet
- FDR-14 InfiniBand

Accelerator nodes



TECHNISCHE
UNIVERSITÄT
DARMSTADT

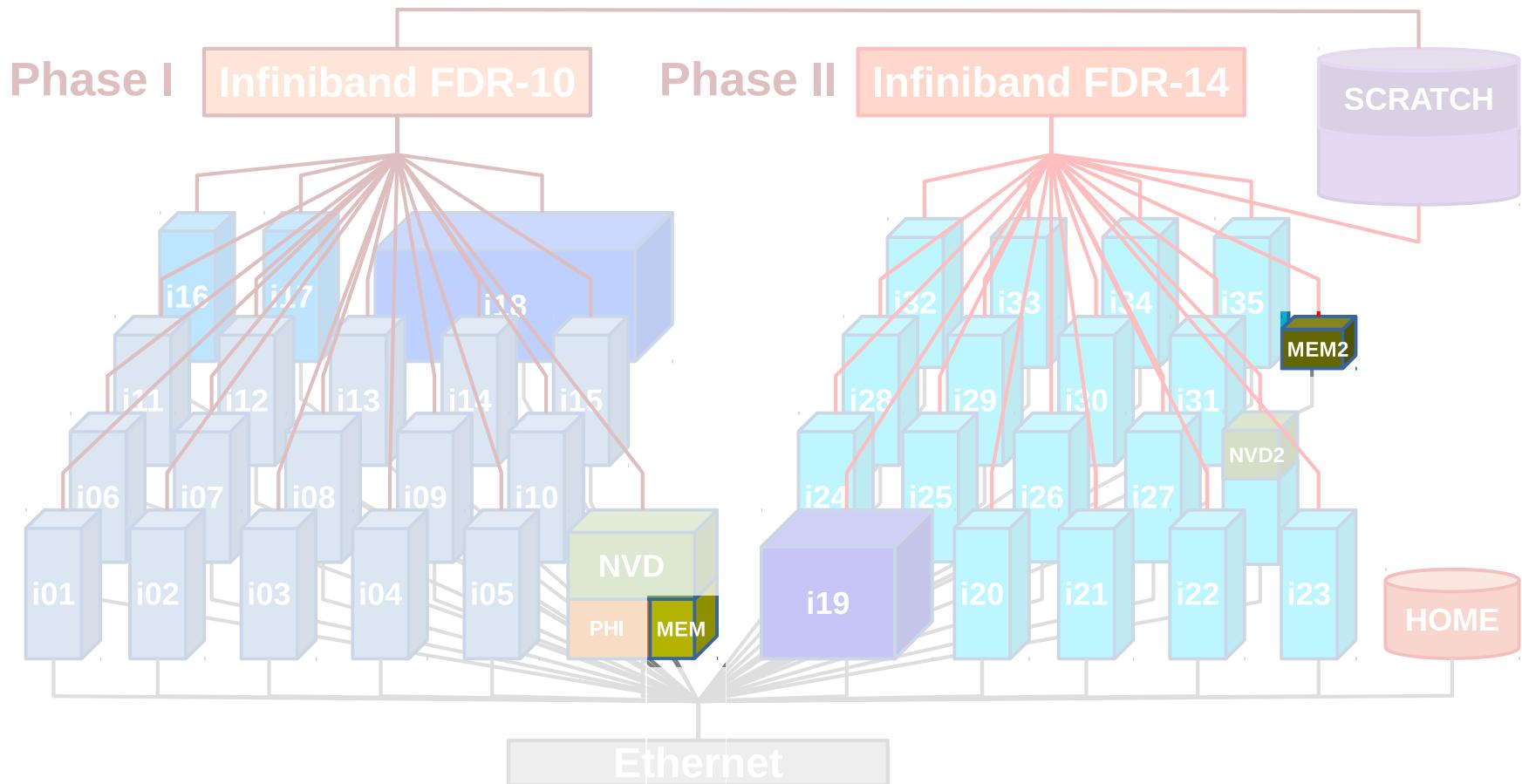
NVIDIA nodes

- 44 Sandy Bridge compute nodes have 2 **NVIDIA K20Xm** cards each
- 2 Haswell compute nodes have 2 **NVIDIA K40m** cards each
- 1 Haswell compute node has 2 **NVIDIA K80** cards

Xeon Phi Nodes

- 24 Sandy Bridge compute nodes have
2 Intel Xeon Phi 5110P cards each
- 2 Sandy Bridge compute nodes have
2 Intel Xeon Phi 7120P cards each

Big mem nodes (“mem”, “mem2”)



Mem nodes



Phase I (4 nodes):

Processors:

- 8 Intel Xeon E7-8837
(Westmere) processors
 $\triangleq 8 \cdot 8 = 64$ CPU cores

- 2.66 GHz
(up to 2.8 GHz in turbo mode)

Main Memory:

- 1 TB (1024 GB) RAM

Network:

- 10 Gigabit Ethernet
- 2 · FDR-10 InfiniBand

Phase II (4 nodes):

Processors:

- 4 Intel Xeon E7-4890 v2
(Ivy Bridge) processors
 $\triangleq 4 \cdot 15 = 60$ CPU cores

- 2.8 GHz
(up to 3.4 GHz in turbo mode)

Main Memory:

- 1 TB (1024 GB) RAM

Network:

- 10 Gigabit Ethernet
- 2 · FDR-14 InfiniBand

File systems



Mountpoint	/home	/work/scratch	/work/local
Size	> 300 TB	> 650 TB	> 100 GB per node
Access time	Normal (Ethernet)	Fast (InfiniBand)	Very fast (local HDD)
Accessibility	Global (cluster)	Global (cluster)	Local (node)
Data availability	permanent	≥ 1 month	Only during job runtime
Quota*	15 GB**	100 TB** 2 Mio. files**	none
Backup	Weekly + snapshots	none	none

* Use the command `cquota` to find out your current usage and quota.

** Can be increased on request.

Login nodes



4 nodes (hardware similar to Phase I):

Processors:

- 4 Intel Xeon E5-4650
(Sandy Bridge) processors
- $\triangleq 4 \cdot 8 = 32$ CPU cores

- 2.7 GHz
(up to 3.3 GHz in turbo mode)

Main Memory:

- 128 GB RAM

Network:

- 2 · 10 Gigabit Ethernet
- 2 · FDR-10 InfiniBand

8 nodes (hardware similar to Phase II):

Processors:

- 2 Intel Xeon E5-2680 v3
(Haswell) processors
- $\triangleq 2 \cdot 12 = 24$ CPU cores

- 2.5 GHz
(up to 3.3 GHz in turbo mode)

Main Memory:

- 128 GB RAM

Network:

- 2 · 10 Gigabit Ethernet
- FDR-14 InfiniBand

IBM BlueGene/Q



TECHNISCHE
UNIVERSITÄT
DARMSTADT



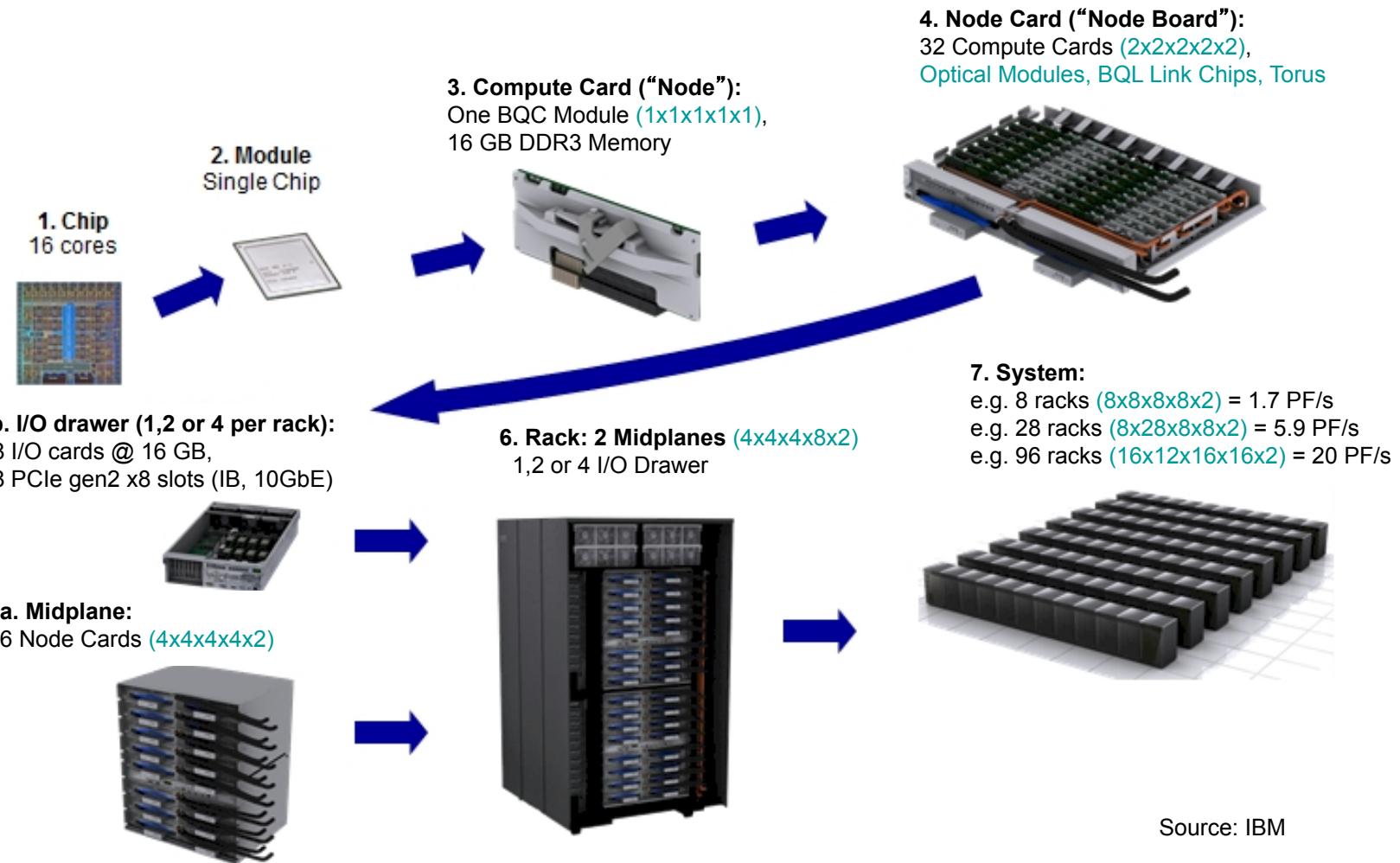
Blue Gene/Q JUQUEEN at Forschungszentrum Jülich

Blue Gene design goals



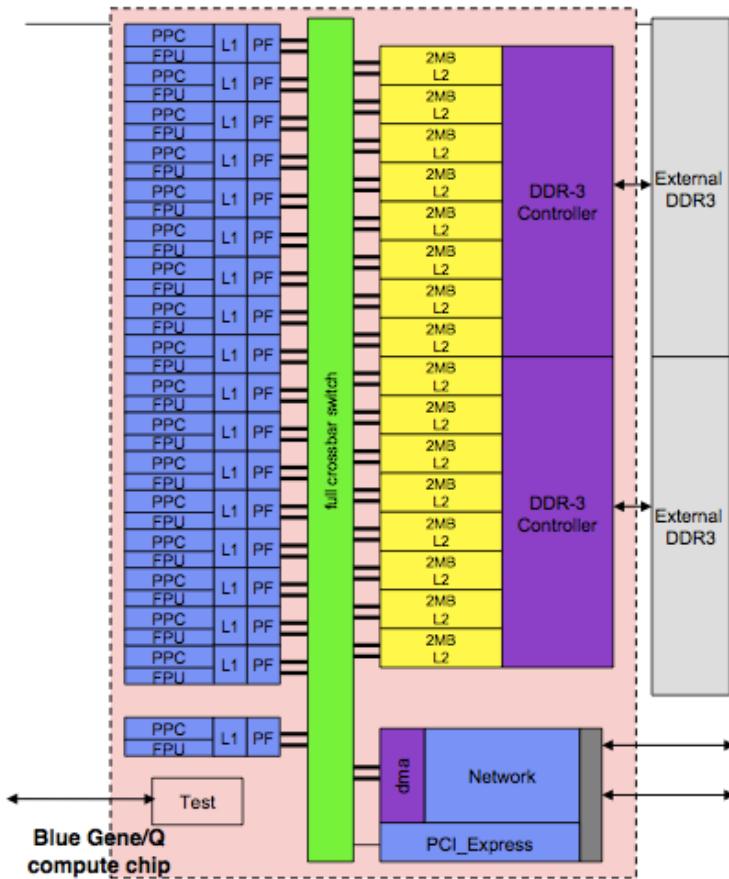
- System-on-Chip (SoC) design
- Processor comprises both processing cores and network
- Optimal performance / watt ratio
- Small foot print
- Transparent high-speed reliable network
- Easy programming based on standard message passing interface (MPI)
- Extreme scalability (> 1.5 M cores)
- High reliability

Blue Gene/Q design



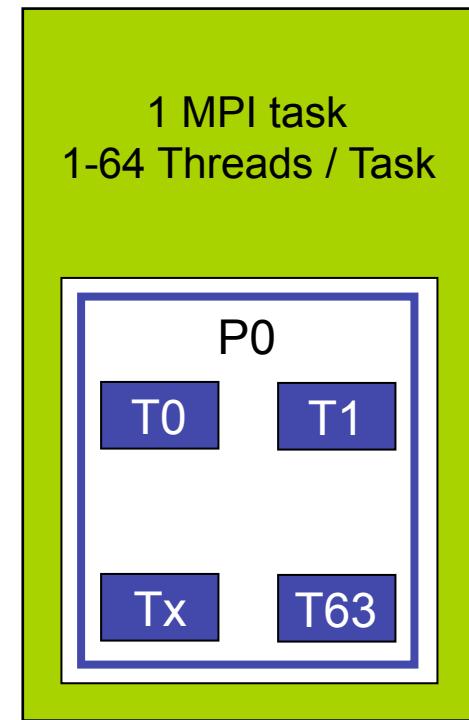
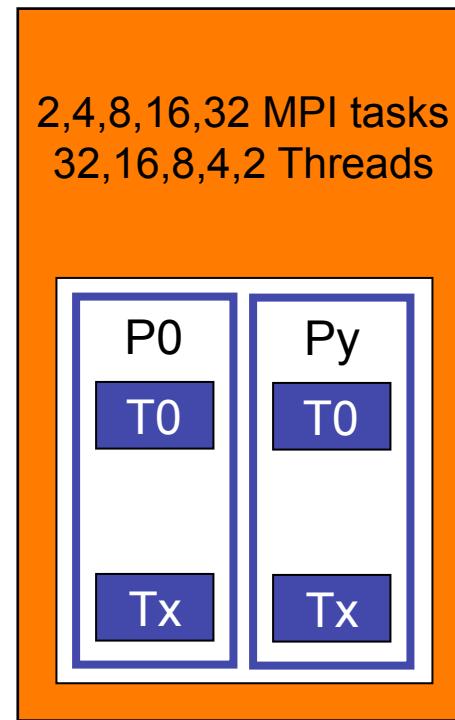
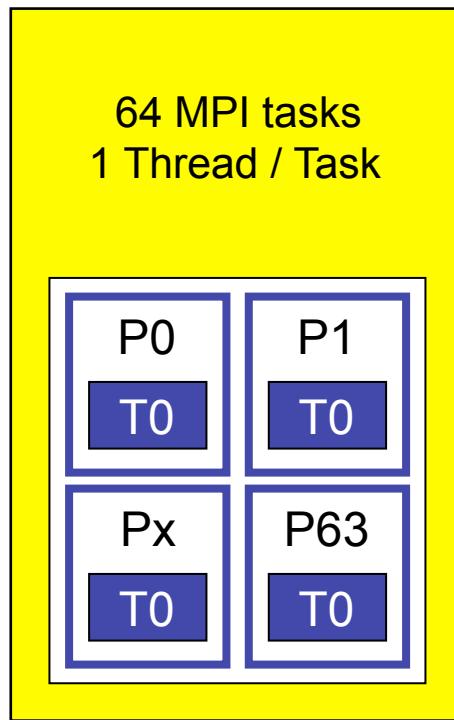
Source: IBM

Blue Gene/Q chip architecture



- 16+1+1 core SMP @ 1.6 GHz
 - Each core 4-way hardware threaded
 - 2-way concurrent issue
- Transactional memory and thread level speculation
- Quad floating point unit on each core
 - 204.8 GF peak per node
- 563 GB/s bisection bandwidth to shared L2
- 32 MB shared L2 cache
- 42.6 GB/s DDR3 bandwidth (1.333 GHz DDR3)
- 10 intra-rack inter-processor links each at 2.0GB/s (5D-Torus)
- One I/O link at 2.0 GB/s
- 16 GB memory/node
- ~60 watts max chip power

Execution Modes in BG/Q



BG/Q's new network architecture



- 11 bi-directional chip-to-chip links
 - 2 GB/s bandwidth, about 40 ns latency
- 5-dimensional torus topology
 - Dimension E limited to length 2
- Why d-dimensional torus with large d?
 - High bi-section bandwidth
 - Flexible partitioning in lower dimensions
- Deterministic/dynamic routing support
- Collective and barrier networks embedded in 5-D torus network
 - Floating point addition support in collective network
 - 11th port for auto-routing to IO fabric



Source: IBM

JUQUEEN Configuration



TECHNISCHE
UNIVERSITÄT
DARMSTADT

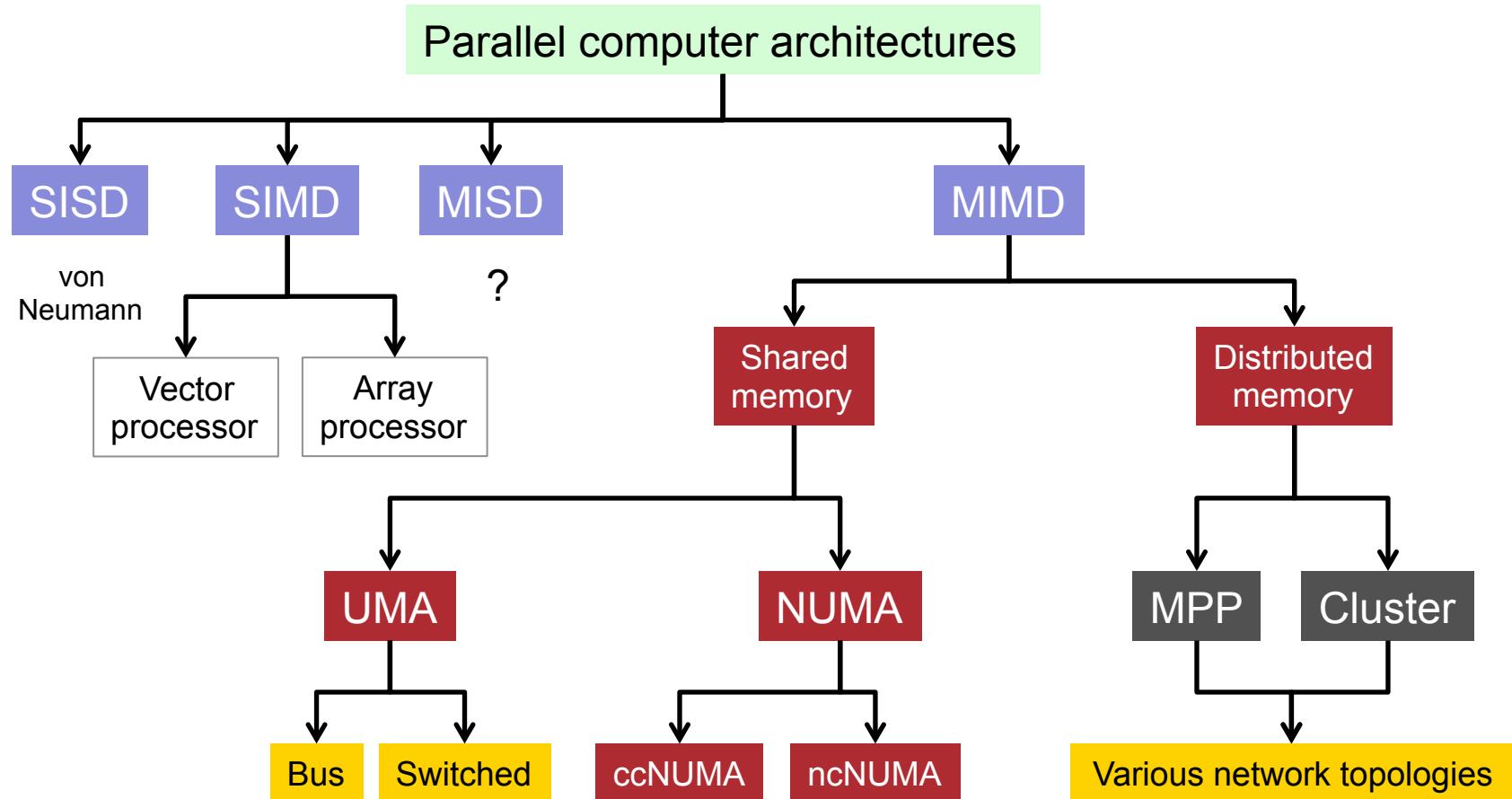
28 racks Blue Gene/Q

- 28672 compute nodes (16 cores, 16 GB memory)
- 458752 cores / 1.8M threads
- 5.88 PFlop/s peak performance
- 248 I/O nodes (10GigE) ← (1x32 + 27x8)
- ~2.2 MW power consumption (~80 kW per rack)

4 frontend nodes (user login) + 2 service nodes (system, database)

- IBM p7 740, 8 cores (3.55 GHz), 128 GB memory
- Local storage device DS5020 (16 TB)

Summary





Large-Scale Parallel Computing

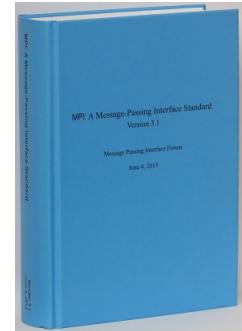
Prof. Dr. Felix Wolf

MESSAGE PASSING INTERFACE

PART 1

Literature

- William Gropp, Ewing Lusk, Anthony Skjellum: Using MPI, 3rd edition, MIT Press, 2014
- Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 3.1
 - <http://www mpi-forum org>

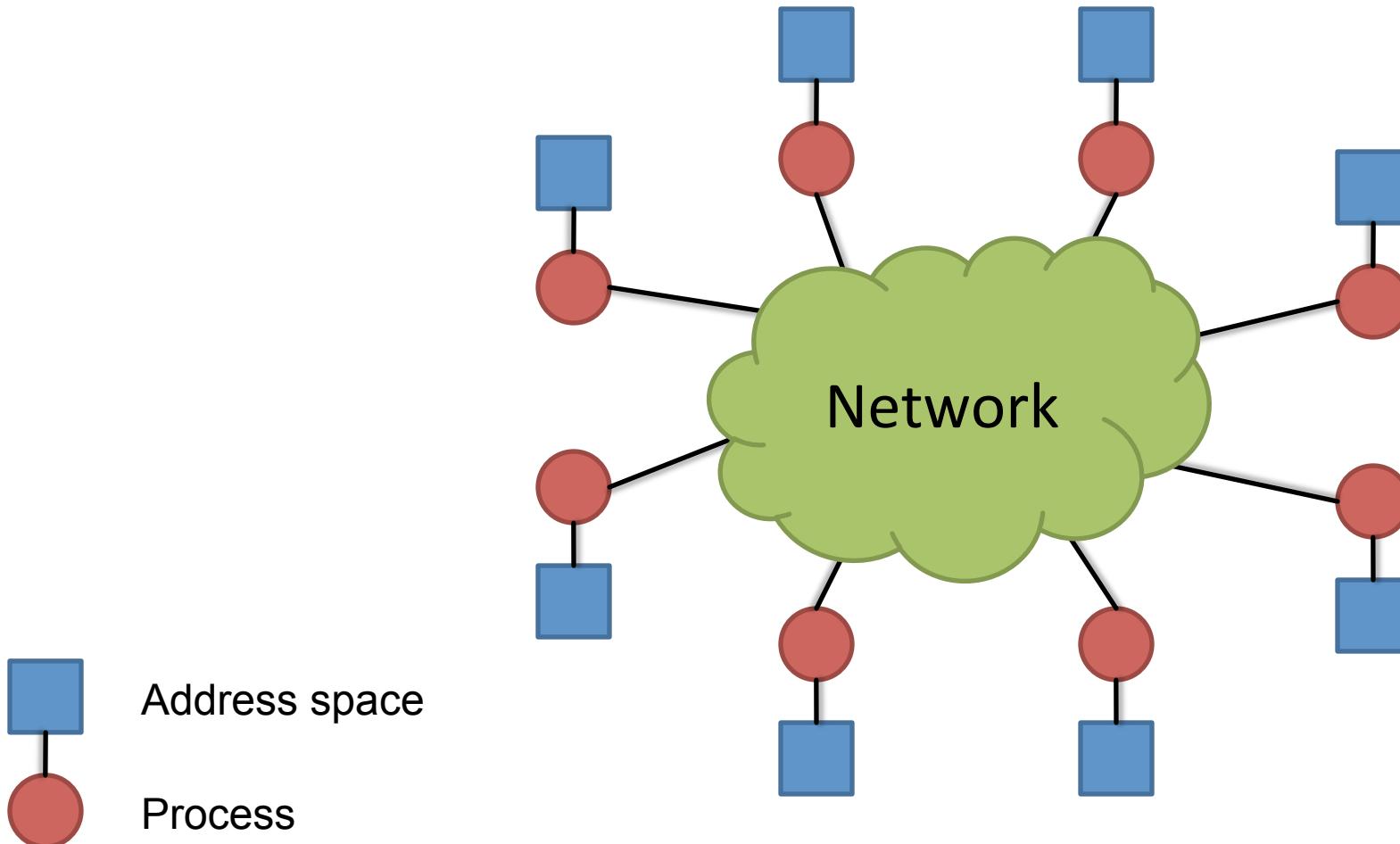


Outline



- Message-passing model
- Basic MPI concepts
- Essential MPI functions
- Simple MPI programs
- Virtual topologies
- Point-to-point communication
- Datatypes
- Collective communication

Message passing



Message passing



- Suitable for distributed memory
- Multiple processes each having their own private address space
- Access to (remote) data of other processes via sending and receiving messages (explicit communication)

- Sender invokes send routine
- Receiver invokes receive routine

```
if (my_id == SENDER)
    send_to(RECEIVER, data);

if (my_id == RECEIVER)
    recv_from(SENDER, data)
```

- De-facto standard MPI: [www.mpi-forum.org](http://www mpi-forum.org)

Advantages



- **Universality** - Works with both distributed and shared memory
- **Expressivity** - Intuitive (anthropomorphic) and complete model to express parallelism
- **Ease of debugging** - Debugging message-passing programs easier than debugging shared-memory programs
 - Although writing debuggers for message-passing might be harder
- **Performance & scalability** - Better control of data locality. Distributed memory machines provide more memory and cache
 - Can enable super-linear speedups



Disadvantages

- Incremental parallelization hard - Parallelizing a sequential program using MPI often requires a complete redesign
- Message-passing primitives relatively low-level - Much programmer attention is diverted from the application problem to an efficient parallel implementation
- Communication and synchronization overhead - Communication and synchronizations costs can become bottleneck at large scales (especially group communication)
- MPI standard quite complex - The basics are simple, but using MPI effectively requires substantial knowledge

What is MPI?



- MPI stands for **Message Passing Interface**
- MPI specifies a **library**, not a language
 - Specifies names, calling sequences, and results of functions / subroutines needed to communicate via message passing
 - Language bindings for C/C++ and Fortran
 - MPI programs are linked with the MPI library and compiled with ordinary compilers

What is MPI? (2)

- MPI is a **specification**, not a particular implementation
 - De-facto standard for message passing
 - Defined by the MPI Forum – open group with representatives from academia and industry
 - www mpi-forum.org
- Correct MPI program should be able to run on all MPI implementations without change
- Both proprietary and portable open-source implementations

History



- Version 1.0 (1994)
 - Fortran77 and C language bindings
 - 129 functions
- Version 1.1 (1995)
 - Corrections and clarifications, minor changes
 - No additional functions
- Version 1.2 (1997)
 - Further corrections and clarifications for MPI-1
 - 1 additional function

History (2)



- Version 2.0 (1997)
 - MPI-2 with new functionality (193 additional functions)
 - Parallel I/O
 - Remote memory access
 - Dynamic process management
 - Multithreaded MPI
 - C++ binding
- Version 2.1 (2008)
 - Corrections and clarifications
 - Unification of MPI 1.2 and 2.0 into a single document
- Version 2.2 (2009)
 - Further corrections and clarifications
 - New functionality with minor impact on implementations

History (3)



- Version 3.0 (2012)
 - New functionality with major impact on implementations
 - Non-blocking collectives
 - Neighborhood collectives
 - New one-sided communication operations
 - Fortran 2008 bindings
 - Tool information interface
- Version 3.1 (2015)
 - Mostly corrections and clarifications
 - Few new functions added

Minimal message interface



`send(address, length, destination, tag)`



Message buffer
(length in bytes)

Used for
matching

Actual message
can be shorter
than buffer



`receive(address, length, source, tag, actlen)`

Message buffer



(address, length) not really adequate

- Often message is non-contiguous
 - Example: row of matrix that is stored column-wise
 - In general, dispersed collection of structures of different sizes
 - Programmer wants to avoid “packing” messages
- Data types may have different sizes in heterogeneous systems
 - Length in bytes not an adequate specification of the semantic content of the message

Message buffer (2)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

MPI solution

- (address, count, datatype)
- Example: (a, 300, MPI_REAL) describes array (vector) a of 300 real numbers
- Data type can also be non-contiguous

Separating families of messages



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Matching of messages via tag argument

- Wildcard tags match any tag
- Entire program must use tags in a predefined and coherent way
- Problem – libraries should not by accident receive messages from the main program

MPI solution

- Message context
- Allocated at runtime by the system in response to the user and library requests
- No wild-card matching permitted

Naming processes



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Processes belong to groups
- Processes within a given group identified by ranks
 - Integers from 0 to $n-1$
- Initial group to which all processes belong
 - Ranks from 0 to 1 less than total number of processes

Communicators



- Combines the notions of context and group in a single object called communicator
- Becomes argument to most communication operations
- Destination or source specified in send or receive refers to rank of the process within group identified by communicator
- Two predefined communicators
 - `MPI_COMM_WORLD` contains all processes
 - `MPI_COMM_SELF` contains only the local process

Blocking send



```
MPI_Send(address, count, datatype, destination, tag, comm)
```

- Sends **count** occurrences of items of the form **datatype** starting at **address**
- **Destination** specified as rank in the group associated with communicator **comm**
- Argument **tag** is an integer used for message matching
- **comm** identifies a group of processes and a communication context

Blocking receive

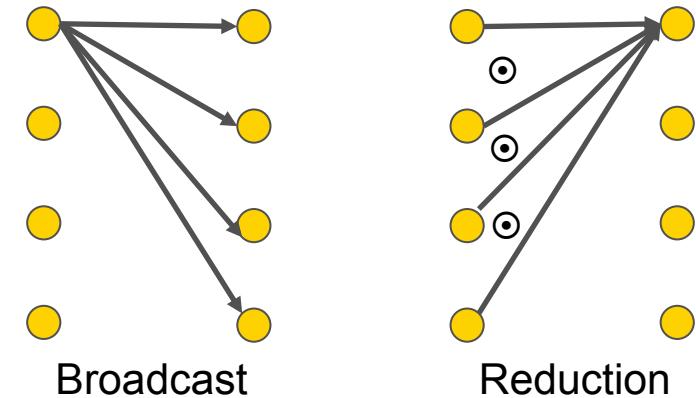


```
MPI_Recv(address, maxcount, datatype, source,  
tag, comm, status)
```

- Allows **maxcount** occurrences of items of the form **datatype** to be received in the buffer starting at **address**
- **Source** is rank in **comm** or wildcard **MPI_ANY_SOURCE**
- **tag** is an integer used for message matching or wildcard **MPI_ANY_TAG**
- **comm** identifies a group of processes and a communication context
- **status** holds information about the actual message size, source, and tag

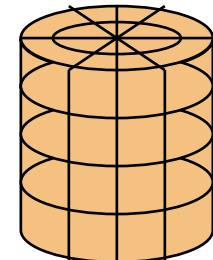
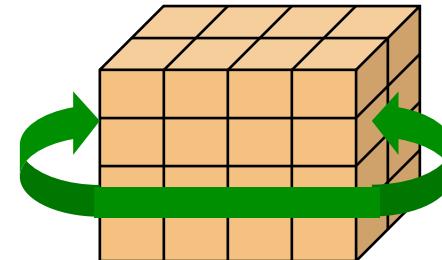
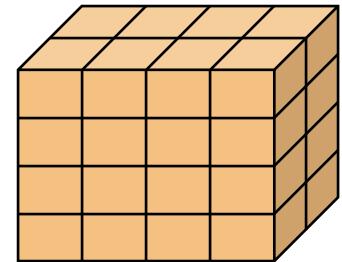
Collective communication & computation

- Recurring parallel group communication patterns (1->n, n->1, n->n)
 - Communication (e.g., broadcast)
 - Computation (e.g., sum reduction)
- Manual implementation via point-to-point messages cumbersome
 - Often suboptimal performance
- MPI offers a range of predefined **collective operations**
 - Use optimized algorithms
 - May take advantage of hardware-specific features (e.g., network)



Virtual topologies

- Applications often define logical adjacency relationships among processes
 - Example: domain decomposition using Cartesian grid
 - Communication occurs between neighbors
- Virtual topologies allow efficient mapping of these adjacency relationships onto the physical topology of the underlying machine
 - Optimization of communication between neighbors
 - Convenient process naming, e.g. using Cartesian coordinates



Debugging and profiling



- Users can intercept MPI calls via “hooks”
 - Allows the definition of custom profiling and debugging mechanisms
- MPI implementations can expose variables that provide insight into internal performance information

Communication modes



- There are also non-blocking versions of send and receive whose completion can be tested and waited for
 - Allows overlap of computation and communication
- Multiple communication modes
 - Standard mode – common practice
 - Synchronous mode – requires send to block until receive is posted
 - Ready mode – way for the programmer to notify the system that receive has already been posted
 - Buffered mode – provides user-controllable buffering

A six-function version of MPI



MPI_Init	Initialize MPI
MPI_Comm_size	Find out how many processes there are
MPI_Comm_rank	Find out which process I am
MPI_Send	Send a message
MPI_Recv	Receive a message
MPI_Finalize	Terminate MPI

Header file



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
#include <mpi.h>
```

Contains

- Definition of named constants
- Function prototypes
- Type definitions

Opaque objects



- Internal representations of various MPI objects such as groups, communicators, datatypes, etc. are stored in MPI-managed system memory
 - Not directly accessible to the user, and objects stored there are **opaque**
- Their size and shape is not visible to the user
- Accessed via handles, which exist in user space
- MPI procedures that operate on opaque objects are passed handle arguments to access these objects

Generic MPI function format



```
error = MPI_Function(parameter, ...);
```

- Error code is integer return value
- Successful return code will be MPI_SUCCESS
- Failure return codes are implementation dependent

MPI namespace:

The MPI_ and PMPI_ prefixes are reserved for MPI constants and functions (i.e., application variables and functions must not begin with MPI_ or PMPI_).

Initialization and finalization



```
int MPI_Init(int *argc, char ***argv)
```

- Must be called as the first MPI function
 - Only exception: MPI_Initialized
- MPI specifies no command-line arguments but does allow an MPI implementation to make use of them

```
int MPI_Finalize()
```

- Must be called as the last MPI function
 - Only exception: MPI_Finalized

Rank in a communicator



```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- Determines the rank of the calling process in the communicator
- The rank uniquely identifies each process in a communicator

```
int myrank;  
...  
MPI_Init(&argc, &argv);  
...  
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

Size of a communicator



```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- Determines the size of the group associated with a communicator

```
int size;  
...  
MPI_Init(&argc, &argv);  
...  
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Hello world



```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

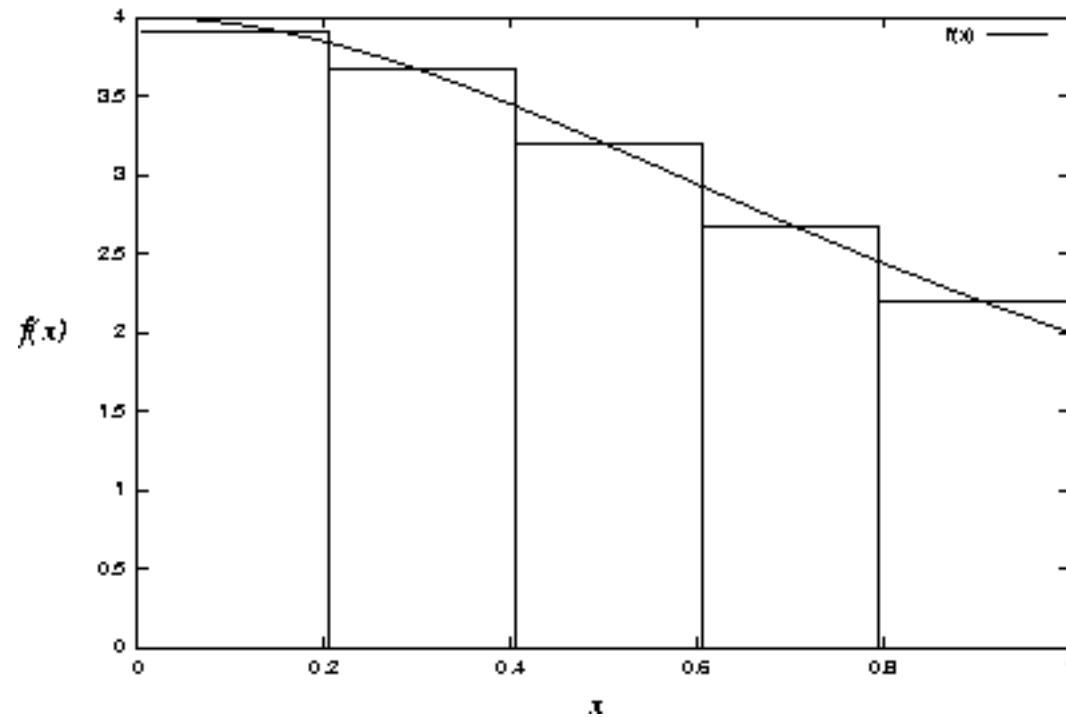
    printf("I am %d out of %d\n", myid, numprocs);
    MPI_Finalize();
    return 0;
}
```

```
> mpieexec -n 4 ./hello
I am 3 out of 4
I am 1 out of 4
I am 0 out of 4
I am 2 out of 4
```

Calculating π via numerical integration

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(x) \Big|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4}$$

$$\Rightarrow \int_0^1 \frac{4}{1+x^2} = \pi$$



Calculating π via numerical integration (2)



```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    if (myid == 0) {
        printf("Enter the number of intervals:");
        scanf("%d",&n);
    }
    [... next slide ...]
    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
    MPI_Finalize();
    return 0;
}
```

Calculating π via numerical integration (3)



```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

h = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += (4.0 / (1.0 + x*x));
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Broadcast



```
int MPI_Bcast(void *buf, int count,  
              MPI_Datatype datatype, int root,  
              MPI_Comm comm)
```

- Broadcasts a message from the process with rank root to all other processes of the group.
 - buf = starting address of buffer
 - count = number of entries in buffer
 - datatype = data type of buffer
 - root = rank of broadcast root
 - comm = communicator

Reduce



```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,  
               MPI_Datatype datatype, MPI_Op op,  
               int root, MPI_Comm comm)
```

- Combines the elements in the input buffer of each process using the operation op and returns the combined value in the output buffer of the process with rank root
 - sendbuf = address of send buffer
 - recvbuf = address of receive buffer
 - count = number of elements in send buffer
 - datatype = data type of elements of send buffer
 - op = reduce operation
 - root = rank of root process
 - comm = communicator



Basic datatypes in C

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MP_LONG_DOUBLE	long double
MPI_BYTE	(any C type)
MPI_PACKED	(any C type)
MPI_LONG_LONG_INT	longlong int (64 bit integer)

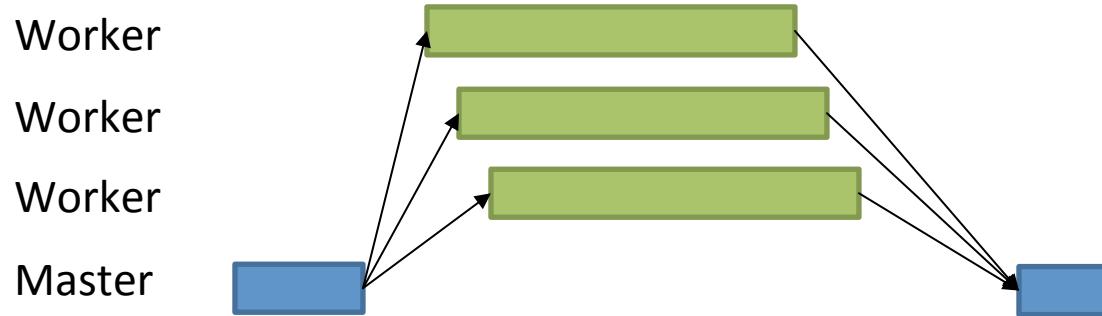
Further datatypes
defined in the
standard

Predefined reduction operations



Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or
MPI_BXOR	bit-wise exclusive or
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Master worker



- **Self-scheduling** algorithm - master coordinates processing of tasks by providing input data to workers and collecting results
- Suitable if
 - Workers need not communicate with one another
 - Amount of work each worker must perform is difficult to predict
- Example: matrix-vector multiplication

Matrix-vector multiplication



$$A * \vec{b} = \vec{c}$$

Unit of work = dot product of one row of matrix A with vector b

Master

- Broadcasts b to each worker
- Sends one row to each worker
- Loop
 - Receives dot product from whichever worker sends one
 - Sends next task to that worker
 - Termination if all tasks are handed out

Worker

- Receives broadcast value of b
- Loop
 - Receives row from A
 - Forms dot product
 - Returns answer back to master

Macros



```
#include "mpi.h"
#define MAX_ROWS 1000
#define MAX_COLS 1000
#define MIN(a, b) ((a) > (b) ? (b) : (a))
#define DONE MAX_ROWS+1
```

Matrix-vector multiplication: common part



```
int main(int argc, char **argv) {
    double A[MAX_ROWS][MAX_COLS], b[MAX_COLS], c[MAX_ROWS];
    double buffer[MAX_COLS], ans;
    int myid, master, numprocs;
    int i, j, numsent, sender, done;
    int anstype, row;
    int rows, cols;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    master = 0;
    rows   = 100;
    cols   = 100;

    if (myid == master) { /* master code */
    } else { /* worker code */
    }
    MPI_Finalize();
    return 0;
}
```

Matrix-vector multiplication: master



```
/* Initialize A and b (arbitrary) */
[...]
numsent = 0;

/* Send b to each worker process */
MPI_Bcast(b, cols, MPI_DOUBLE, master, MPI_COMM_WORLD);

/* Send a row to each worker process; tag with row number */
for (i = 0; i < MIN(numprocs - 1, rows); i++) {
    MPI_Send(&A[i][0], cols, MPI_DOUBLE, i+1, i, MPI_COMM_WORLD);
    numsent++;
}
```

Matrix-vector multiplication: master (2)



```
for (i = 0; i < rows; i++) {
    MPI_Recv(&ans, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);
    sender = status.MPI_SOURCE;

    /* row is tag value */
    anstype = status.MPI_TAG;
    c[anstype] = ans;

    /* send another row */
    if (numsent < rows) {
        MPI_Send(&A[numsent][0], cols, MPI_DOUBLE, sender,
                 numsent, MPI_COMM_WORLD);
        numsent++;
    } else {
        /* Tell sender that there is no more work */
        MPI_Send(MPI_BOTTOM, 0, MPI_DOUBLE, sender, DONE, MPI_COMM_WORLD);
    }
}
```

Matrix-vector multiplication: worker



```
MPI_Bcast(b, cols, MPI_DOUBLE, master, MPI_COMM_WORLD);

/* Skip if more processes than work */
done = myid > rows;

while (!done) {
    MPI_Recv(buffer, cols, MPI_DOUBLE, master, MPI_ANY_TAG, MPI_COMM_WORLD,
              &status);
    done = status.MPI_TAG == DONE;
    if (!done) {
        row = status.MPI_TAG;
        ans = 0.0;
        for (i = 0; i < cols; i++) {
            ans += buffer[i] * b[i];
        }
        MPI_Send(&ans, 1, MPI_DOUBLE, master, row, MPI_COMM_WORLD);
    }
}
```

C-binding of blocking send and receive



```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,  
            int dest, int tag,  
            MPI_Comm comm)
```

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



Return status

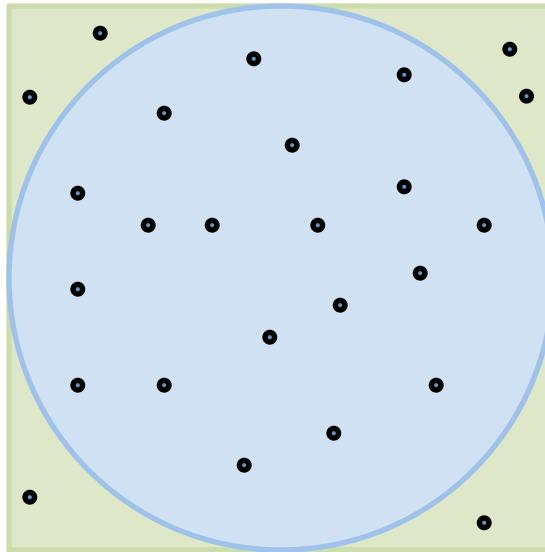
- In C, status is structure with three fields

MPI_SOURCE	rank of source process
MPI_TAG	tag of message
MPI_ERROR	error code

- The three fields provide information on the message actually received
- The number of entries received can be obtained using the function

```
int MPI_Get_count(MPI_Status *status,  
                  MPI_Datatype datatype, int *count)
```

Monte Carlo computation of π



Radius $r = 1$

Area of circle = π

Area of square = 4

Ratio of areas $q = \pi / 4$

$\Rightarrow \pi = 4q$

Compute ratio q

- Generate random points (x,y) in the square
- Count how many turn out to be in the circle

Monte Carlo computation of π (2)



Master

- Loop
 - Receives request from any worker
 - Generates pair of random numbers
 - Sends them to requesting worker

Worker

- Sends initial request to master
- Loop
 - Receives pair of random numbers and computes coordinates
 - Decides whether point sits inside circle
 - Synchronizes with all other workers to determine progress (i.e., accuracy of approximation)
 - Either terminates loop or requests new pair of random numbers

Monte Carlo computation of π (3)



- Every worker synchronizes with all other workers to determine progress
 - All workers calculate collectively current value of approximation
 - Then they compare it to the exact value of π

```
/* worker code */
[...]

MPI_Allreduce(&in, &totalin, 1 , MPI_INT, MPI_SUM,
               workers);
MPI_Allreduce(&out, &totalout, 1 , MPI_INT, MPI_SUM,
               workers);
Pi = (4.0*totalin)/(totalin + totalout);
error = fabs( Pi-3.141592653589793238462643);

[...]
```

Allreduce



```
int MPI_Allreduce(void *sendbuf, void *recvbuf,  
                  int count, MPI_Datatype datatype,  
                  MPI_Op op, MPI_Comm comm)
```

- Combines values from all processes and distributes the result back to all processes
 - sendbuf = starting address of send buffer
 - count = number of elements in send buffer
 - datatype = data type of elements of send buffer
 - op = reduce operation
 - comm = communicator

Using communicators



Two communicators

- World: all processes
- Workers: all processes except random number server

```
[...]
MPI_Comm_size(world, &numprocs);
MPI_Comm_rank(world, &myid);
server = numprocs-1;          /* last proc is server */
MPI_Comm_group( world, &world_group );
ranks[0] = server;
MPI_Group_excl( world_group, 1, ranks, &worker_group );
MPI_Comm_create( world, worker_group, &workers );
MPI_Group_free(&worker_group);
MPI_Group_free(&world_group);
[...]
MPI_Comm_free(&workers);
```

Frees only reference -
not necessarily the
entire object

Using communicators and groups



```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

Accesses the group associated with given communicator

```
int MPI_Group_excl(MPI_Group group, int n, int *ranks,  
                   MPI_Group *newgroup)
```

Produces a group by deleting those processes with ranks rank[0], ..., rank[n-1]

```
int MPI_Group_free(MPI_Group *group)
```

Marks a group object for deallocation (frees reference)

Using communicators and groups (2)



```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,  
                    MPI_Comm *newcomm)
```

Creates a new communicator from the specified group
(subset of parent communicator's group)

```
int MPI_Comm_free(MPI_Comm *comm)
```

Marks a communicator for deallocation (frees reference)

```
int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group,  
                         int tag, MPI_Comm *newcomm)
```

Similar to MPI_Comm_create except that MPI_Comm_create must be called by all processes in the group of comm, whereas MPI_Comm_create_group must be called by all processes in group, which is a subgroup of the group of comm (needed for fault tolerance)

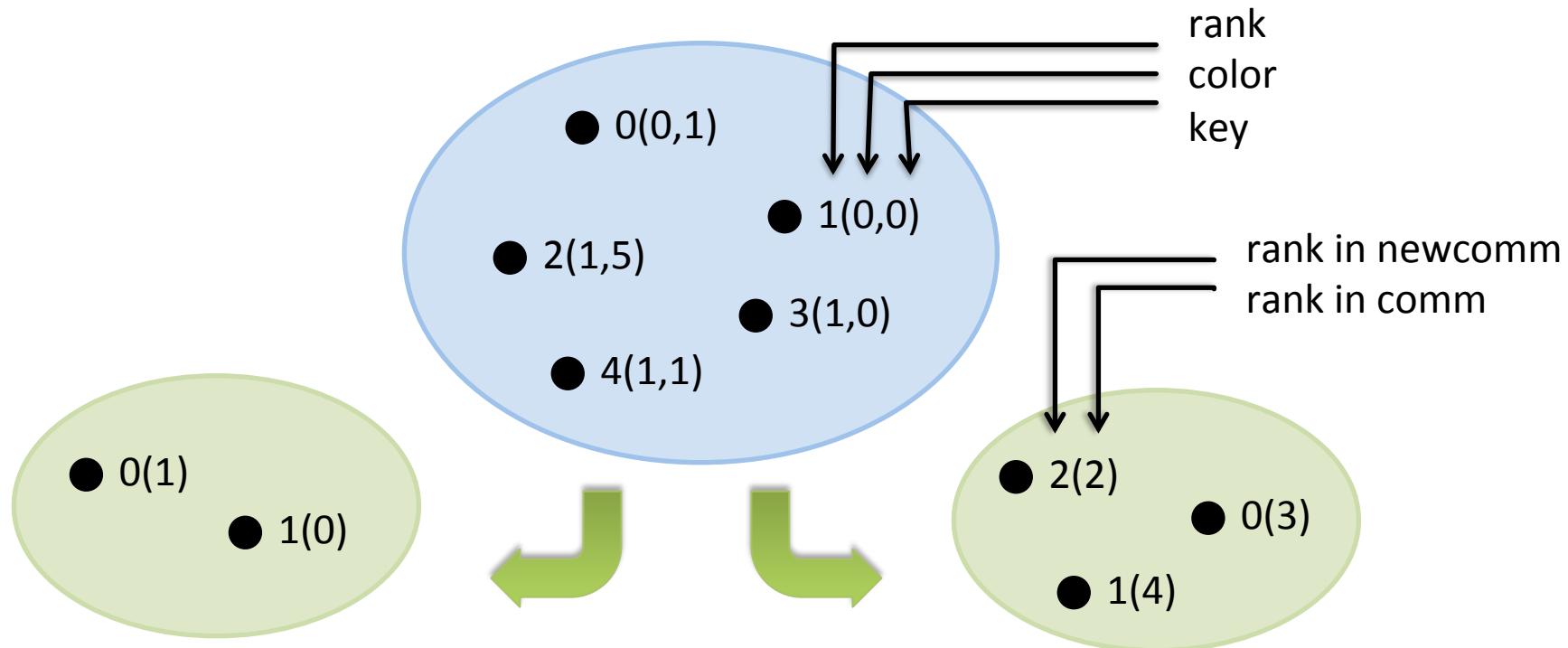
Splitting communicators



```
int MPI_Comm_split(MPI_Comm comm, int color, int key,  
                    MPI_Comm *newcomm)
```

- Partitions the group associated with `comm` into disjoint subgroups, one for each value of `color`. Within each subgroup, the processes are ranked in the order defined by the value of `key`

Splitting communicators (2)



Further group and communicator operations



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Groups

- Size and rank
- Translate ranks between groups
- Compare two groups
- Union, intersection, and difference
- New group from existing group via explicit inclusion
- Inclusion and exclusion of ranges with stride

Communicators

- Comparison
- Duplication

Summary



- Programming model – multiple processes with private address spaces communicate by exchanging messages
 - Between two processes via point-to-point communication
 - Between groups of processes via collective communication because more convenient & efficient
- Advantage – easy to understand
- Disadvantage – results in complex programs
- Central concept – **communicators**
 - Define group of processes
 - Define communication context (similar to wave length)



Large-Scale Parallel Computing

Prof. Dr. Felix Wolf

MESSAGE PASSING INTERFACE

PART 2

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Message-passing model
- Basic MPI concepts
- Essential MPI functions
- Simple MPI programs
- Virtual topologies
- Point-to-point communication
- Datatypes
- Collective communication

2D Poisson problem



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Simple PDE at the core of many applications

$$\nabla^2 u = f(x, y) \quad \text{in the interior} \quad (1)$$

$$u(x, y) = g(x, y) \quad \text{on the boundary} \quad (2)$$

Simplification

- Domain is unit square
- Discretization via square mesh
 - $n+2$ points along each edge

$$x_i = \frac{i}{n+1}, i = 0, \dots, n+1$$

$$y_j = \frac{j}{n+1}, j = 0, \dots, n+1$$

Jacobi iteration



We can approximate Equation (1) at each of these points using the formula:

$$\frac{u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - 4u_{i,j}}{h^2} = f_{i,j} \quad \text{with} \quad h = \frac{1}{n+1}$$

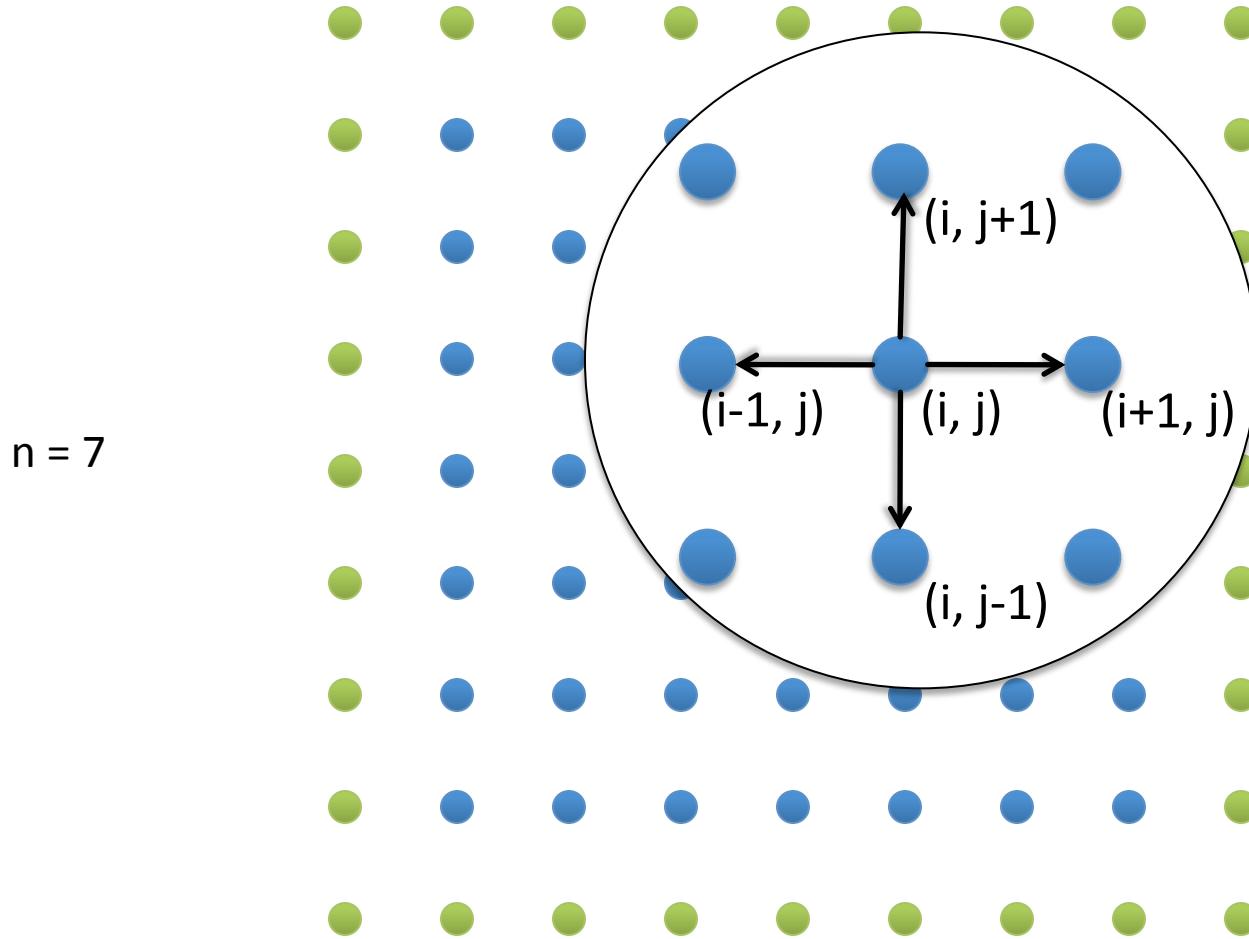
Can be rewritten as:

$$u_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - h^2 f_{i,j})$$

We iterate by choosing values for all $u_{i,j}$ and replace them using:

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k + u_{i+1,j}^k - h^2 f_{i,j})$$

Stencil approximation

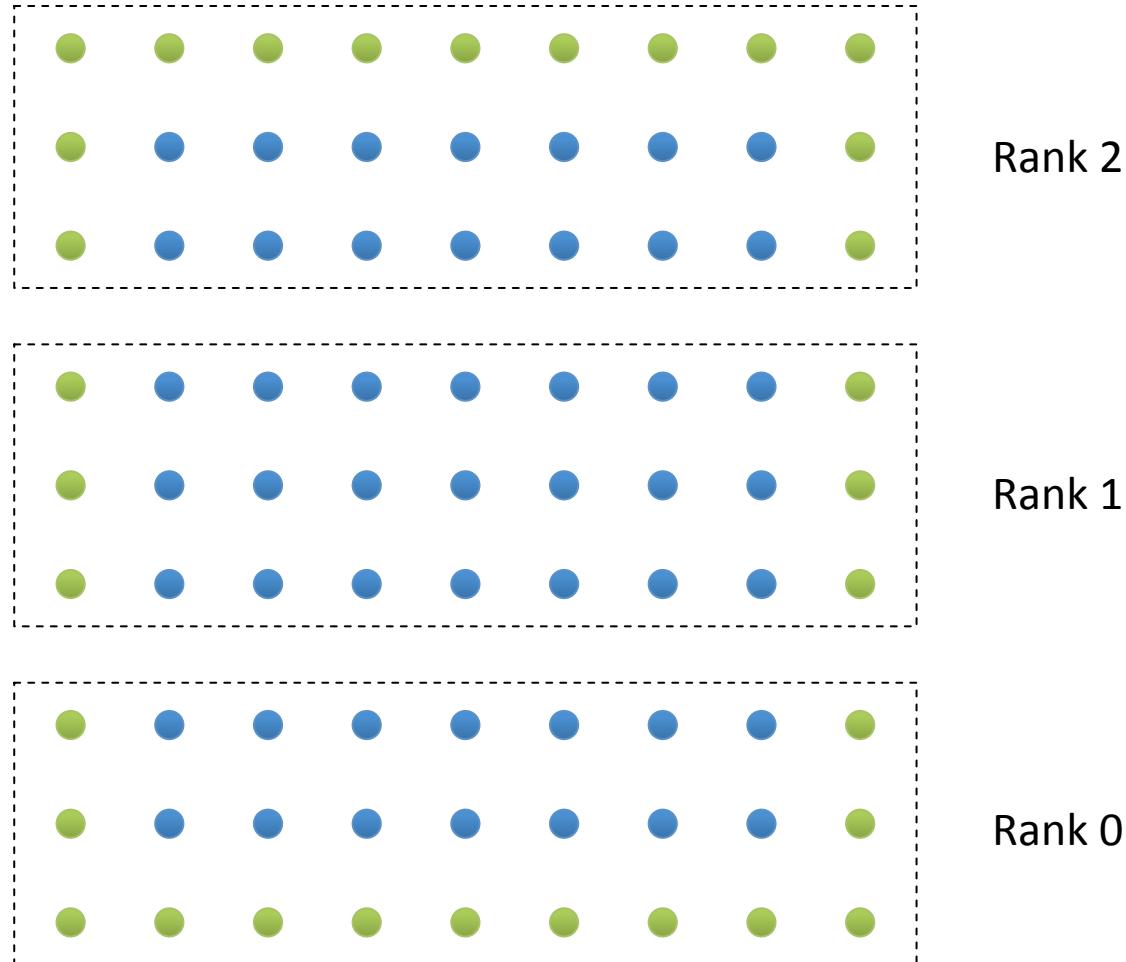


Jacobi iteration



```
integer i, j, n
double precision u(0:n+1,0:n+1), unew(0:n+1,0:n+1)
do 10 j=1, n
    do 10 i=1, n
        unew(i,j) = &
            0.25 * (u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - &
            h * h * f(i,j)
10: continue
```

1D decomposition



Jacobi iteration for a slice of the domain

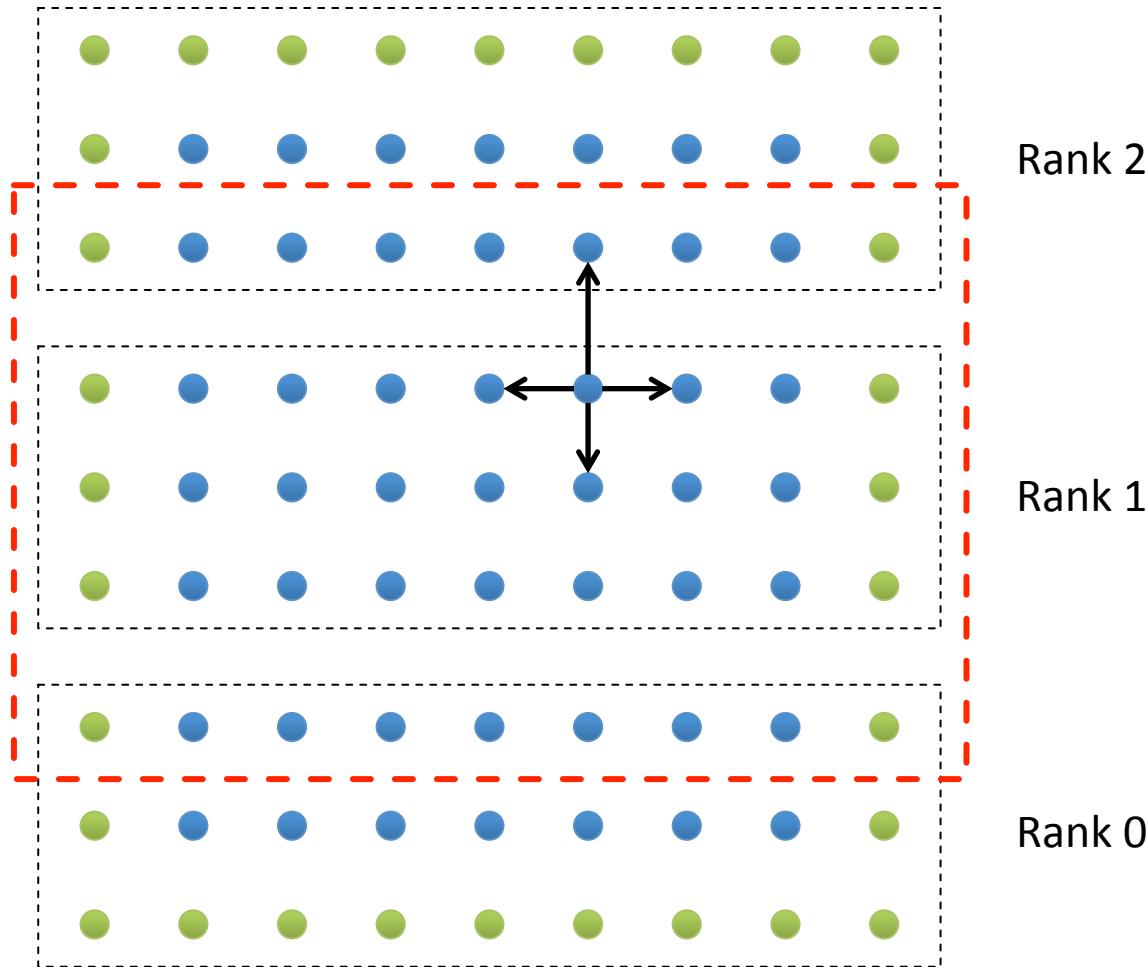


```
integer i, j, n
double precision u(0:n+1,s:e), unew(0:n+1,s:e)
do 10 j=s, e
    do 10 i=1, n
        unew(i,j) = &
            0.25 * (u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - &
            h * h * f(i,j)
10: continue
```

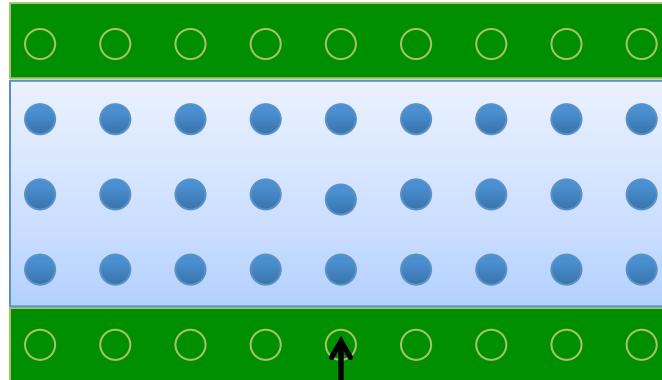
- Problem – the loop will require elements such as $u(i, s-1)$ that belong to another process
- Necessitates array expansion to hold **ghost points**

```
! center slice
double precision u(0:n+1,s-1:e+1), unew(0:n+1,s:e)
```

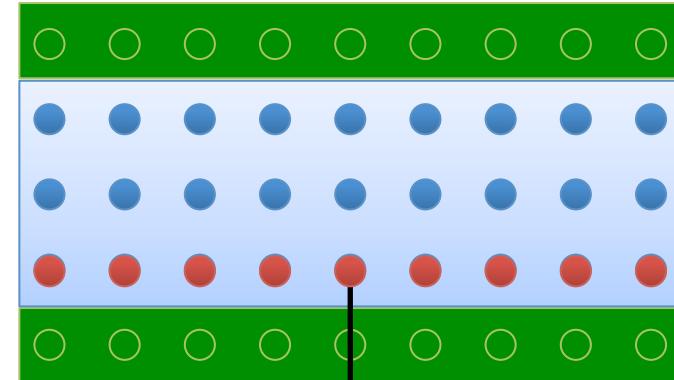
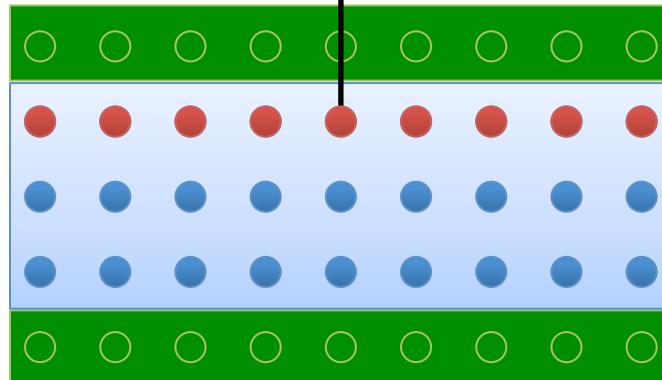
Computational domain with ghost points



Two-step data transfer



(1)



(2)

Topologies



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- The description of how processors in a parallel computer are connected to each other is called network topology or **physical topology**
- The description of which processes in a parallel program communicate with each other is called application topology or **virtual topology**

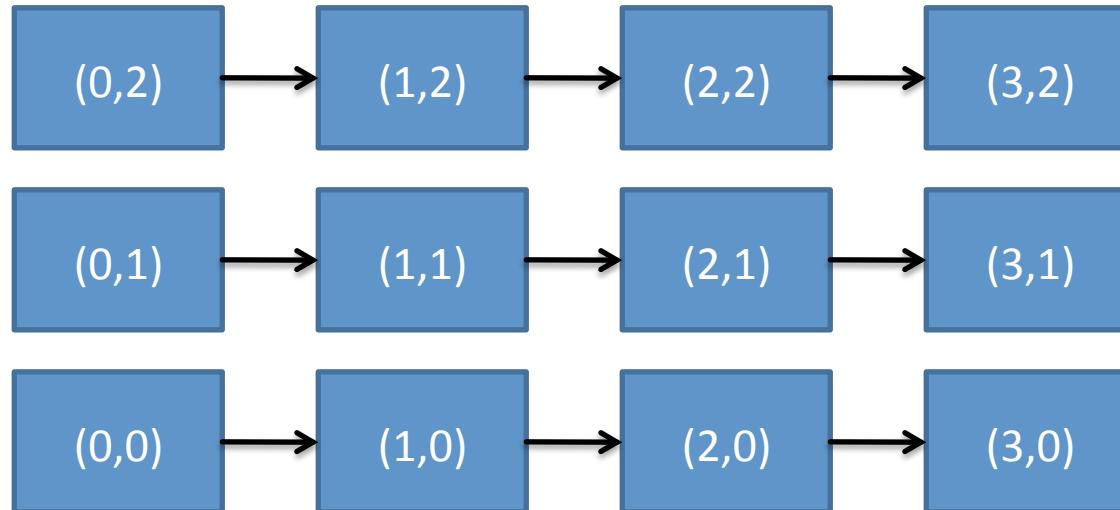
Topologies (2)



- Topology functions allow convenient process naming
- The way the virtual topology is mapped onto the physical topology can influence performance
 - Some mappings are better than others
- MPI allows the implementation to help optimize this aspect through topology functions

Cartesian topologies

- Decomposition in the natural coordinate directions
 - Arrows show shift in the first dimension



- MPI provides collection of routines for defining, examining, and manipulating Cartesian topologies

Creating a Cartesian topology



```
integer dims(2), ndim
logical isperiodic(2), reorder

dims(1)      = 4
dims(2)      = 3
isperiodic(1) = .false.
isperiodic(2) = .false.
reorder       = .true.
ndim          = 2
call MPI_CART_CREATE( MPI_COMM_WORLD, ndim, dims, isperiodic, &
                      reorder, comm2d, ierr )
```

Creates Cartesian topology from previous slide

- isperiodic indicates whether processes at the “end” are connected
- reorder allows function to reorder process ranking for better performance

How to access coordinates?



- In one dimension we can simply use the rank in the communicator unless processes have been reordered
- More complicated for more than one dimension

```
call MPI_CART_GET( comm2d, 2, dims, isperiodic, coords, ierr )
print *, '(', coords(1), ',', coords(2), ')'
```

- Returns coordinates of the calling process plus dimensions sizes and periodicity

```
call MPI_COMM_RANK( comm2d, myrank, ierr )
call MPI_CART_COORDS( comm2d, myrank, 2, coords, ierr )
```

- Returns coordinates of a given rank

How to find neighbors?



```
[...]  
call MPI_CART_CREATE( MPI_COMM_WORLD, ndim, dims, isperiodic, &  
    reorder, comm2d, ierr )  
call MPI_CART_SHIFT( comm2d, 0, 1, nbrleft, nbrright, ierr )  
call MPI_CART_SHIFT( comm2d, 1, 1, nbrbottom, nbrtop, ierr )
```

- 2nd argument indicates the direction
 - The coordinate dimension (0,..., ndim-1) to be traversed by the shift.
- 3rd argument indicates displacement (integer)
 - > 0: upwards shift
 - < 0: downwards shift
- Depending on the periodicity, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift
 - In the case of an end-off shift, the value MPI_PROC_NULL is returned

C-binding of topology functions



```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims,
                    int *dims, int *isperiodic,
                    int reorder, MPI_Comm *newcomm);

int MPI_Cart_shift(MPI_Comm comm, int direction,
                   int displacement, int *src, int *dest);

int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims,
                 int *isperiodic, int *coords);

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank);

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
                    int *coords);
```

Domain decomposition



- How to divide the domain among the processes?
- Trivial if number of processes evenly divides n

```
s      = 1 + myrank * (n / procs)
e      = s + (n / procs) - 1
```

- Otherwise

```
nlocal = n / numprocs
s      = myid * nlocal + 1
deficit = mod(n,numprocs)
s      = s + min(myid,deficit)
if (myid < deficit) nlocal = nlocal + 1
e      = s + nlocal - 1
if (e > n .or. myid == numprocs-1) e = n
```

Exchange of ghost points



```
subroutine exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
include 'mpif.h'
integer nx, s, e
double precision a(0:nx+1,s-1:e+1)
integer comm1d, nbrbottom, nbrtop
integer status(MPI_STATUS_SIZE), ierr
c
call MPI_SEND( a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, &
               comm1d, ierr )
call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, &
               comm1d, status, ierr )
call MPI_SEND( a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, &
               comm1d, ierr )
call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, &
               comm1d, status, ierr )
return
end
```

Sweep



```
subroutine sweep1d( a, f, nx, s, e, b )
integer nx, s, e
double precision a(0:nx+1,s-1:e+1), f(0:nx+1,s-1:e+1)
+                      b(0:nx+1,s-1:e+1)

c
integer i, j
double precision h
c
h = 1.0d0 / dble(nx+1)
do 10 j=s, e
    do 10 i=1, nx
        b(i,j) = 0.25 * (a(i-1,j)+a(i,j+1)+a(i,j-1)+a(i+1,j)) -
+                      h * h * f(i,j)
10   continue
      return
end
```

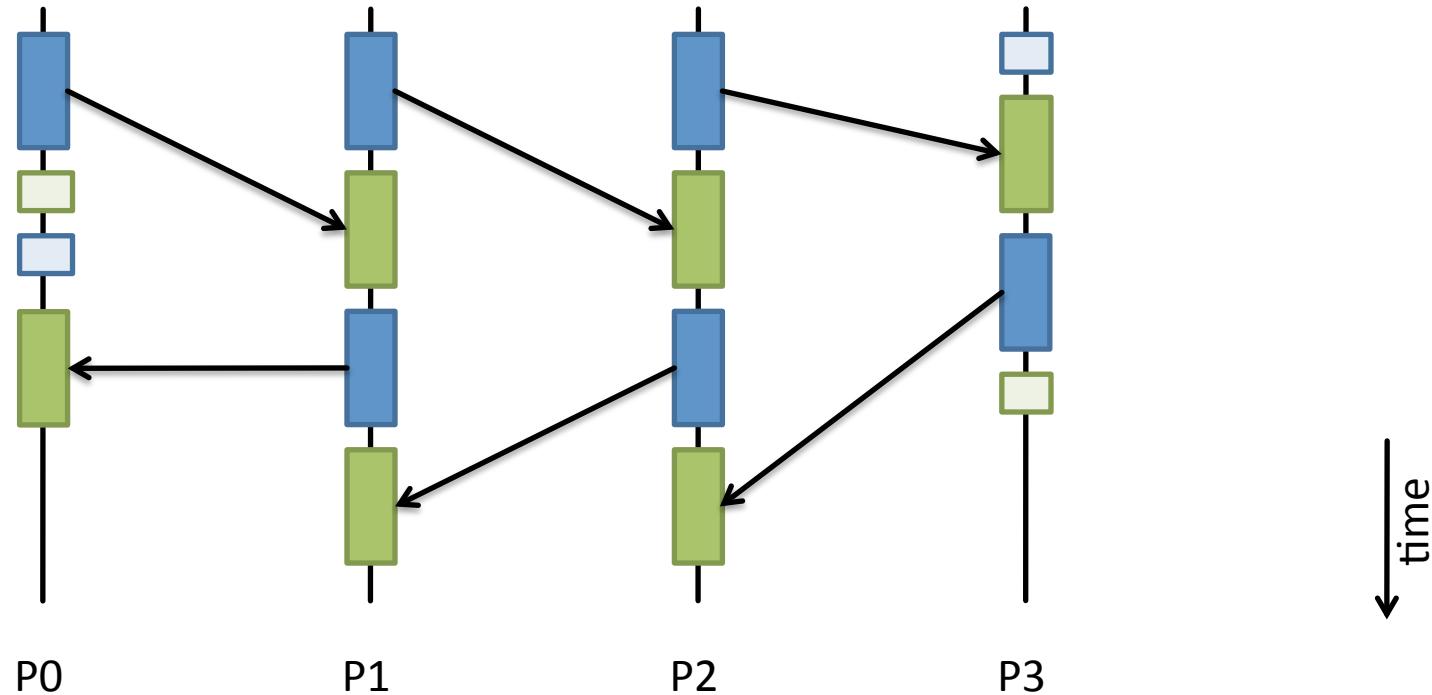
Jacobi iteration



```
call MPI_CART_CREATE( MPI_COMM_WORLD, 1, numprocs, .false., &
                      .true., comm1d, ierr )
call MPI_COMM_RANK( comm1d, myid, ierr )
call MPI_Cart_shift( comm1d, 0, 1, nbrbottom, nbrtop, ierr    )

c
c Compute the decomposition and initialize a, b, and f
[...]
c
do 10 it=1, maxit
    call exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
    call sweep1d( a, f, nx, s, e, b )
    call exchng1( b, nx, s, e, comm1d, nbrbottom, nbrtop )
    call sweep1d( b, f, nx, s, e, a )
    dwork = diff( a, b, nx, s, e )
    call MPI_Allreduce( dwork, diffnorm, 1, MPI_DOUBLE_PRECISION, &
                        MPI_SUM, comm1d, ierr )
    if (diffnorm .lt. 1.0e-5) goto 20
10  continue
    if (myid .eq. 0) print *, 'Failed to converge'
20  continue
    if (myid .eq. 0) print *, 'Converged after ', 2*it, ' Iterations'
```

Intended exchange pattern

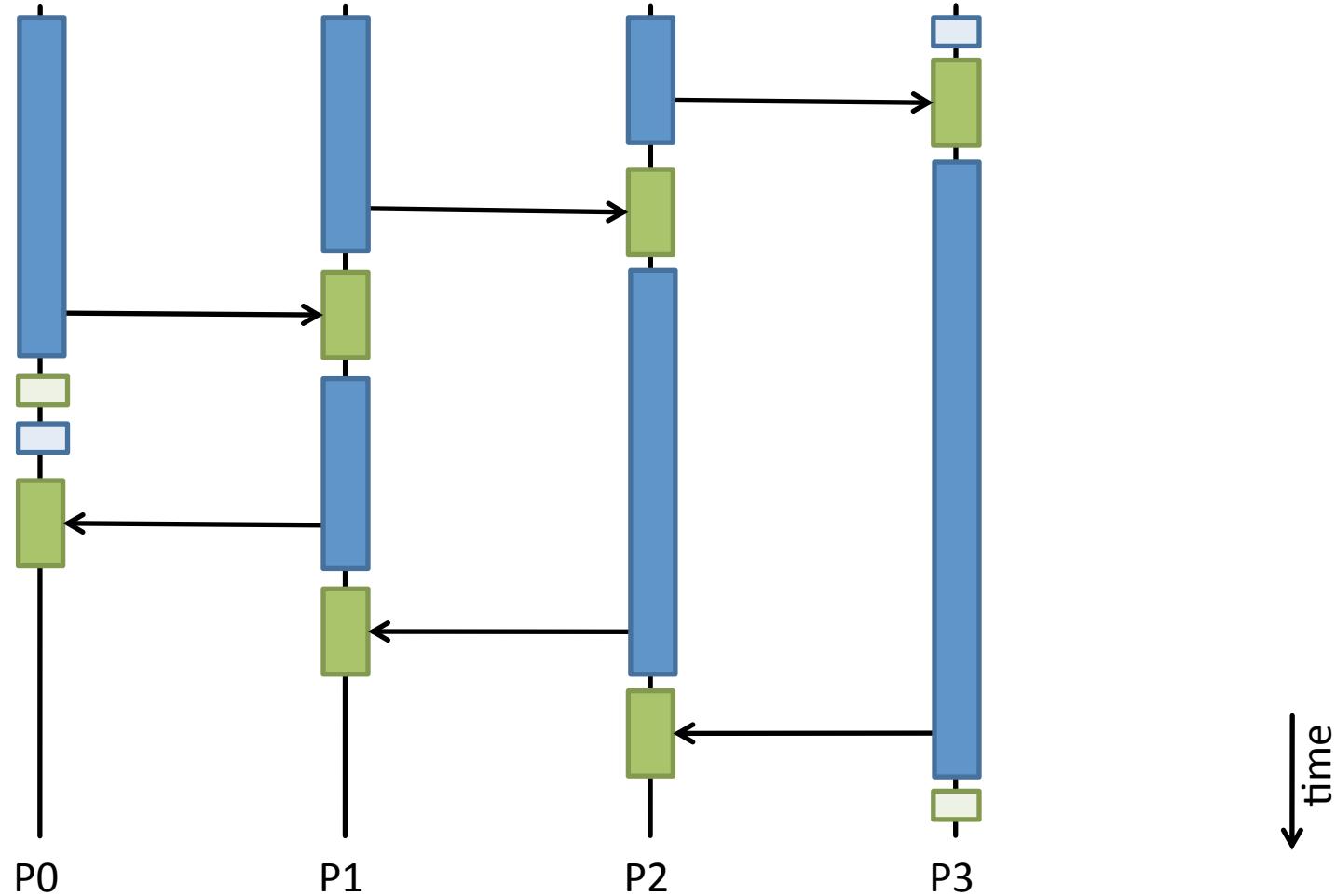


Assumes that messages can be buffered

Buffering semantics of MPI_Send

- An MPI implementation is permitted to copy the message to be sent into internal storage to allow the MPI_Send to return
 - This is called **buffering** the message
- But it is not required to do so because there simply might not be enough buffer space

Potential serialization in the case of large nx



Solution alternatives

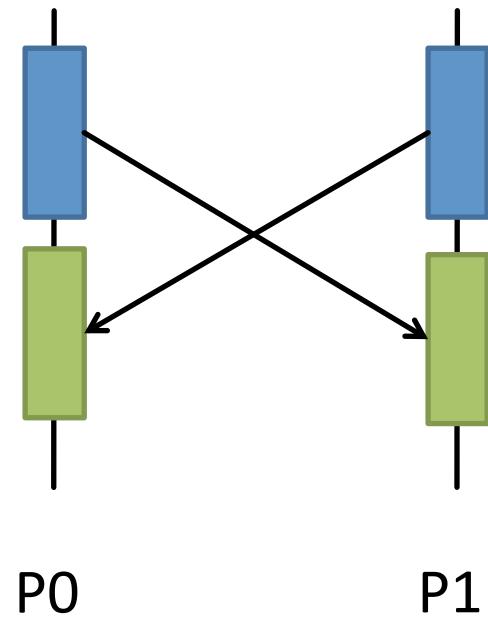


TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Pairing sends and receives
- Combined send and receive
- Buffered sends
- Non-blocking communication

Excursion: deadlock

- What happens if the two sends do not return until the receivers have posted their receive?
- Definition
 - Two processes are deadlocked if each process is waiting for an event that only the other process can cause
 - If more than two processes are involved in a deadlock then they are waiting in a circular chain
- Analogy: **chicken and egg** problem



Pairing sends and receives

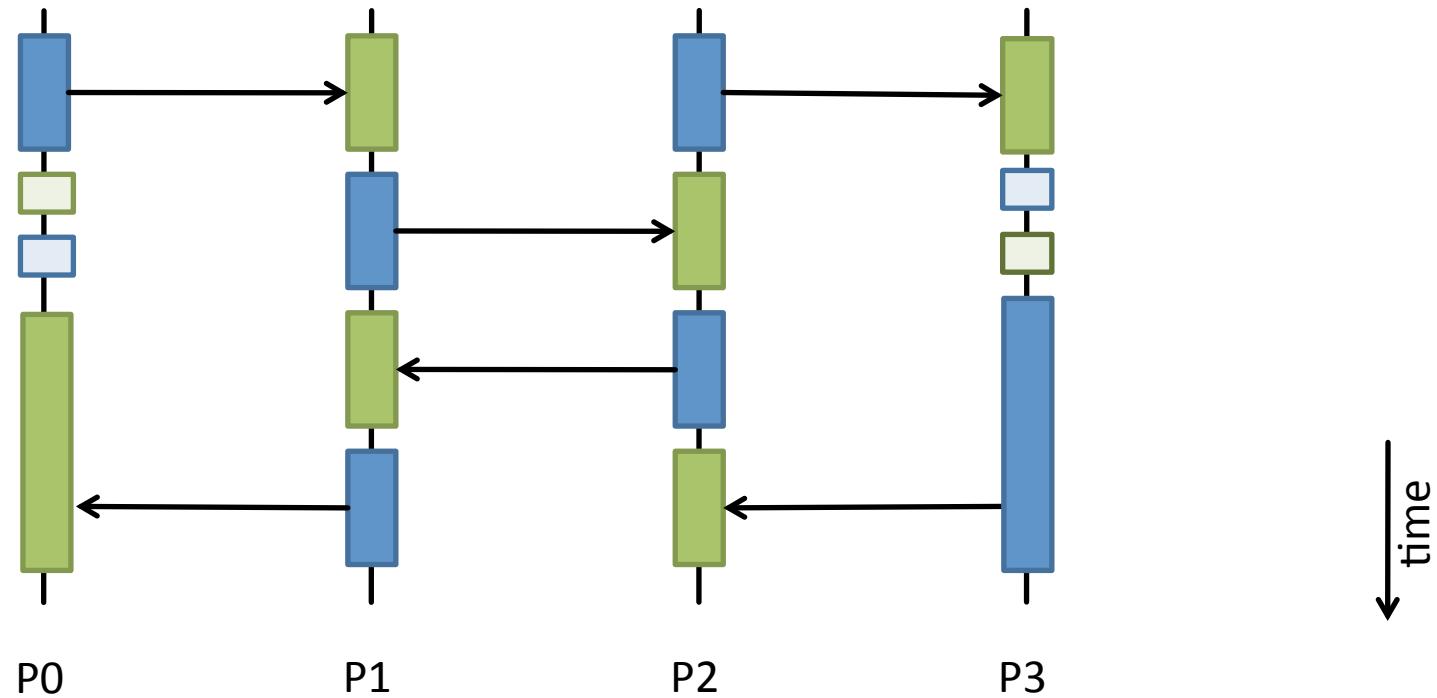


Order sends and receives so that they are paired

- If one process is sending to another, the destination will do a receive that matches that send before doing a send of its own

```
if (mod( coord, 2 ) .eq. 0) then
    call MPI_SEND( a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, ... )
    call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, ... )
    call MPI_SEND( a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, ... )
    call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, ... )
else
    call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, ... )
    call MPI_SEND( a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, ... )
    call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, ... )
    call MPI_SEND( a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, ... )
endif
```

Pairing sends and receives (2)



Combined send and receive



- Pairing sends and receives can be difficult when the arrangement of processes is complex
 - Example: irregular grids
- Alternative: [**MPI_Sendrecv**](#)
 - Allows process to send and receive without worrying about deadlock from lack of buffering

```
    call MPI_SENDRECV(
&          a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0,
&          a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0,
&          comm1d, status, ierr )
    call MPI_SENDRECV(
&          a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1,
&          a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1,
&          comm1d, status, ierr )
```

MPI_Sendrecv



```
int MPI_Sendrecv(void *sendbuf, int sendcount,
                  MPI_Datatype sendtype, int dest,
                  int sendtag,
                  void *recvbuf, int recvcount,
                  MPI_Datatype recvtype, int source,
                  int recvtag,
                  MPI_Comm comm, MPI_Status *status)
```

- Executes a blocking send and receive operation
- Both send and receive use the same communicator, but possibly different tags
- The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes
- The semantics of a send-receive operation is what would be obtained if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads

Buffered sends



- MPI allows the programmer to provide a buffer into which data can be placed until it is delivered (or at least until it leaves the buffer)
- Buffer must be large enough to hold all messages that must be sent before the matching receives are called

```
double precision buffer(2*MAXNX+2*MPI_BSEND_OVERHEAD)
integer size
[...]
size = 2*MAXNX*8 + 2*MPI_BSEND_OVERHEAD*8
call MPI_BUFFER_ATTACH( buffer, size, ierr )
```

- Once the buffer is no longer needed, it can be detached

```
call MPI_BUFFER_DETACH( buffer, size, ierr )
```



Buffered sends (2)

- To use buffer, replace MPI_Send with MPI_Bsend

```
call MPI_BSEND( a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, ...)  
call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, ...)  
call MPI_BSEND( a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, ...)  
call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, ...)
```

- Remember that buffering may incur a performance penalty

C-binding of buffered sends



```
int MPI_Buffer_attach(void *buffer, int size)
```

- Provides to MPI a buffer in the user's memory to be used for buffering outgoing messages. The size is given in bytes
- The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time

```
int MPI_Buffer_detach(void *buffer_addr, int *size)
```

- Detach the buffer currently associated with MPI
- Will block until all messages currently in the buffer have been transmitted
- Upon return, the user may reuse or deallocate the space taken by the buffer

```
int MPI_Bsend(void *buf, int count, MPI_Datatype  
datatype, int dest, int tag, MPI_Comm comm)
```

- Send in buffered mode

Further send modes



- Synchronous send (`MPI_Ssend`)
 - Will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send
 - Can be used to check a program's dependence on buffering
 - **Caveat:** `MPI_Send` may be implemented using synchronous mode
- Ready send (`MPI_Rsend`)
 - Same semantics as a standard send operation
 - The sender only provides additional information to the system –namely that a matching receive is already posted
 - Can save some overhead
- Same signatures as `MPI_Send`



Summary send modes

- Standard – Up to MPI to decide whether outgoing messages will be buffered
- Buffered – Send can be started whether or not a matching receive has been posted. May complete before a matching receive is posted
- Synchronous – Send can be started whether or not a matching receive was posted. Will complete successfully only if a matching receive is posted and the receive operation has started to receive the message
- Ready – Send may be started only if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined

There is only one receive mode, but it matches any of the send modes

Possible send protocols



- **Ready send** – The message is sent as soon as possible
- **Synchronous send** – The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message and the sender now sends the message
- **Standard send** – First protocol may be used for short messages (“eager” protocol). Second protocol for long messages
- **Buffered send** – The sender copies the message into a buffer and then sends it with a non-blocking send (using the same protocol as for standard send)

Non-blocking communication



- **Motivation 1** – improve performance by overlapping communication and computation
 - Especially useful on systems where communication can be executed autonomously by intelligent communication controller
- **Motivation 2** – avoid buffering by deferring completion of communication until receive operation specifies destination
 - Also allows the destination to be specified early in the program
- **Semantic advantage** – avoids deadlock problem

Sender

- Start send
- Do something else
- Complete send

Receiver

- Start receive
- Do something else
- Complete receive

Non-blocking send



- Request object used to determine whether an operation has completed
- Test returns immediately

```
call MPI_ISEND( buffer, count, datatype, dest, tag, &
                 comm, request , ierr )
    <do something else>
10 call MPI_TEST( request, flag, status, ierr )
    if (.not. flag) goto 10
```

- Wait blocks until completion

```
call MPI_ISEND( buffer, count, datatype, dest, tag, &
                 comm, request , ierr )
    <do something else>
call MPI_WAIT ( request, status, ierr )
```

- Receive works analogously



Multiple completions

- MPI provides a way to wait for a collection of non-blocking operations to complete

```
integer statuses(MPI_STATUS_SIZE, 2), requests(2)

call MPI_IRecv( . . . , requests(1) , ierr )
call MPI_IRecv( . . . , requests(2) , ierr )
[ . . . ]
call MPI_WaitAll ( 2, requests, statuses, ierr)
```

- Also supported
 - Waiting for any of a collection (MPI_Waitany)
 - Waiting for some of a collection (MPI_Waitsome)
 - Testing for all, any, or some of a collection (MPI_Testall, MPI_Testany, MPI_Testsome)

Non-blocking exchange of ghost points



```
    integer status_array(MPI_STATUS_SIZE,4), ierr, req(4)
c
call MPI_RECV ( &
    a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, &
    commId, req(1), ierr )
call MPI_RECV ( &
    a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, &
    commId, req(2), ierr )
call MPI_SEND ( &
    a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, &
    commId, req(3), ierr )
call MPI_SEND ( &
    a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, &
    commId, req(4), ierr )
c
call MPI_WAITALL ( 4, req, status_array, ierr )
```

C-binding of non-blocking operations



```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag,
              MPI_Comm comm, MPI_Request *request)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag,
              MPI_Comm comm, MPI_Request *request)

int MPI_Wait(MPI_Request *request, MPI_Status *status)

int MPI_Waitall(int count, MPI_Request *array_of_requests,
                MPI_Status *array_of_statuses)

int MPI_Waitany(int count, MPI_Request *array_of_requests,
                int *index, MPI_Status *status)

int MPI_Waitsome(int count, MPI_Request *array_of_requests,
                 int *numcompl, int *indices, MPI_Status *statuses)
```

C-binding of non-blocking operations (2)



```
int MPI_Test(MPI_Request *request, int *flag,  
             MPI_Status *status)  
  
int MPI_Testall(int count, MPI_Request *array_of_requests,  
                int *flag, MPI_Status *array_of_statuses)  
  
/* index identifies the one operation that completed */  
int MPI_Testany(int count, MPI_Request *array_of_requests,  
                int *index, int *flag,  
                MPI_Status *status)  
  
/* the first numcompl entries in indices are completed */  
int MPI_Testsome(int count, MPI_Request *array_of_requests,  
                  int *numcompl, int *indices,  
                  MPI_Status *statuses)
```

Persistent requests



- Enable reuse of all communication parameters
 - Sending or receiving rank
 - Buffer address of payload
 - Size of message
 - Datatype of message
- Persistent requests are either
 - **Inactive**: no transfer is ongoing
 - **Active**: transfer is ongoing
- They are created and started by special calls, but completed by usual non-blocking completion calls

Initialization of persistent requests



With the initialization, an implementation can optimize some of the things it would otherwise have to decide on ‘on-the-fly’

```
int MPI_Send_init(void *buf, int count,  
                  MPI_Datatype datatype, int dest, int tag,  
                  MPI_Comm comm, MPI_Request *request)  
  
int MPI_Recv_init(void *buf, int count,  
                  MPI_Datatype datatype, int src, int tag,  
                  MPI_Comm comm, MPI_Request *request)
```



Activation of persistent requests

- The user can activate single or multiple persistent requests at the same time
- Data is only transferred when requests are activated
- Buffers can be accessed while requests are inactive
 - After the init call, until the first start call
 - After a wait call, until the next start call

```
int MPI_Start(MPI_Request *request)
```

```
int MPI_Startall(int count, MPI_Request *request)
```

Summary



- Virtual topology
 - Description of which processes in a parallel program communicate with each other
 - Reasons to use MPI topology interface
 - Convenient process naming
 - Potentially more efficient
- Domain decomposition is common parallelization approach
- Different flavors of point-to-point communication
 - Blocking vs. non-blocking
 - 4 different send modes: standard, buffered, synchronous, ready
 - Persistent requests



Large-Scale Parallel Computing

Prof. Dr. Felix Wolf

MESSAGE PASSING INTERFACE

PART 3

Outline



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Message-passing model
- Basic MPI concepts
- Essential MPI functions
- Simple MPI programs
- Virtual topologies
- Point-to-point communication
- **Datatypes**
- **Collective communication**



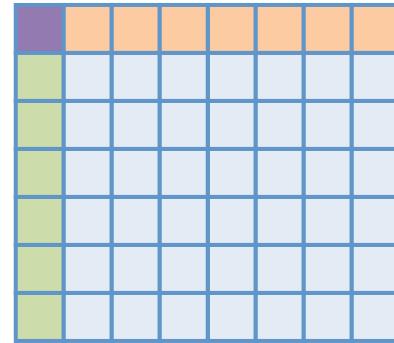
Non-contiguous data

Suppose we want to send column of a matrix stored row-wise

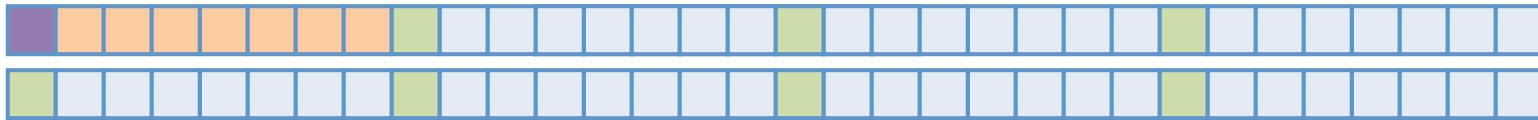
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Row-major and column-major order

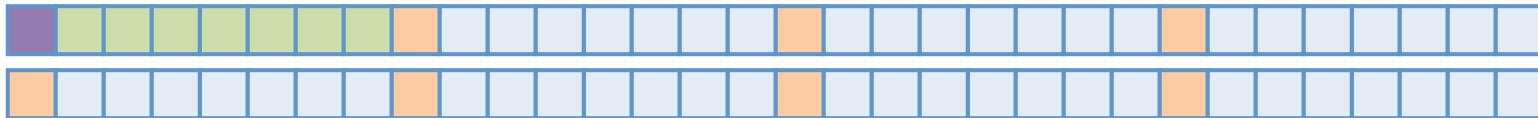
- Two-dimensional matrix



- Memory layout in C (row major)



- Memory layout in Fortran (column major)



Datatypes



- So far, communication involved only a contiguous sequence of identical datatypes
- Often, we need more flexibility
 - Mixing datatypes
 - Example: integer count followed by a sequence of real numbers
 - Non-contiguous data
 - Example: column of a matrix stored in row-major order or sub-block of a matrix
 - Possible solution – packing data into contiguous buffer
 - Disadvantage – local memory-to-memory copy operations at both sender and receiver
 - MPI allows the direct transfer of objects of various shapes and sizes

General datatypes



- A general datatype is an opaque object that specifies:
 1. A sequence of basic datatypes = **type signature**

$$Typesig = \{type_0, \dots, type_{n-1}\}$$

- 2. A sequence of integer (byte) displacements
 - Neither required to be positive, distinct, nor in increasing order
- Together also called **type map**

$$Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$$

- Together with base address specifies message buffer
 - i -th entry starts at $buf + disp_i$
 - Buffer will consist of n values of the types defined in type signature

Extent of a datatype



- The **extent** is the span from the first byte to the last byte occupied by its entries, rounded up to satisfy alignment requirements

$$Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\} \Rightarrow$$

$$lb(Typemap) = \min_j(displ_j)$$

$$ub(Typemap) = \max_j(displ_j + \text{sizeof}(type_j)) + \varepsilon$$

$$\text{extent}(Typemap) = ub(Typemap) - lb(Typemap)$$

Size of a datatype



TECHNISCHE
UNIVERSITÄT
DARMSTADT

The **size** of a datatype is the number of bytes the data takes up

$$\text{size}(\textit{Typemap}) = \sum_j \text{sizeof}(\textit{type}_j)$$

Example



$$\{(int, 0), (char, 4)\}$$

Assumption – integers to be aligned on 4-byte boundaries

$$lb = \min(0, 4) = 0$$

$$ub = \max(0 + 4, 4 + 1) + 3 = 8$$

$$extent = 8 - 0 = 8$$

$$size = 4 + 1 = 5$$

Query functions



- Size of a datatype

```
int MPI_Type_size(MPI_Datatype datatype,  
                  int *size)
```

Integer type that
can hold an
arbitrary address

- Lower bound and extent of a datatype

```
int MPI_Type_get_extent(MPI_Datatype datatype,  
                        MPI_Aint *lb,  
                        MPI_Aint *extent)
```

- The upper bound can be obtained by adding the extent to
the lower bound

Derived datatypes



- A type map is a general way of describing an arbitrary datatype
 - May be inconvenient if the resulting type map contains a large number of entries
- MPI provides a number of ways to create datatypes without explicitly constructing the type map
 - Ranging from count copies of an existing datatype to a fully general description

Contiguous



- Datatype constructor that makes count copies of an existing one
- If we assume that the old datatype has type map

$$\{(int, 0), (double, 8)\}$$

- Then

```
MPI_Type_contiguous( 2, oldtype, newtype)
```

- Produces a new datatype with type map

$$\{(int, 0), (double, 8), (int, 16), (double, 24)\}$$

Using derived datatypes in communication



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
MPI_Send( buffer, count, datatype, dest, tag, comm);
```

is exactly the same as

```
MPI_Type_contiguous( count, datatype, &newtype);
MPI_Type_commit( &newtype );
MPI_Send( buffer, 1, newtype, dest, tag, comm );
MPI_Type_free( &newtype );
```

Using derived datatypes in communication (2)



- A datatype object has to be committed before it can be used in a communication
- Uncommitted types can still be used in type constructors
- The system may „compile“ at commit time an internal representation for the datatype that facilitates communication and select the most convenient transfer mechanism

Commit and free



- Commit a datatype

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

- Free a datatype

```
int MPI_Type_free(MPI_Datatype *datatype)
```

- Created by replicating a datatype into locations that consist of equally spaced blocks
 - Each block is obtained by concatenating the same number of copies of the old datatype
 - The spacing between blocks (i.e., the stride) is a multiple of the extent of the old datatype

```
int MPI_Type_vector(
    int count,
    int blocklength,
    int stride,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

- count = number of blocks
- blocklength = number of elements in each block
- stride = number of elements between the starts of adjacent blocks



Vector - example

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

```
MPI_Type_vector( 8, 2, 8, MPI_DOUBLE, newtype);
```



Created by replicating a datatype into a sequence of blocks

- Like in a vector, each block is a concatenation of the old datatype
- However, each block can contain a different number of copies and can have a different displacement (multiple of the old type's extent)

```
int MPI_Type_indexed(int count,
                     int *array_of_blocklengths,
                     int *array_of_displacements,
                     MPI_Datatype oldtype,
                     MPI_Datatype *newtype)
```



Indexed - example

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

```
int blocklengths[] = {1,2,3,4,5,6,7,8};  
int displacements[] = {0,8,16,24,32,40,48,56};  
MPI_Type_indexed( 8, blocklengths, displacements,  
                  MPI_DOUBLE, newtype );
```

Struct



- Most general type constructor

```
int MPI_Type_create_struct(int count,
                           int      *array_of_blocklengths,
                           MPI_Aint *array_of_byte_displacements,
                           MPI_Datatype *array_of_types,
                           MPI_Datatype *newtype)
```

- Generalization of indexed constructor
 - Each block replicates different datatypes
 - Note that displacements are given in bytes (!)



Struct - example

- Let type1 have the type map

$$\{(double, 0), (char, 8)\}$$

- Let

- b = {2, 1, 3}
- d = {0, 16, 26}
- t = {MPI_FLOAT, type1, MPI_CHAR}

- Then `MPI_Type_struct(3, b, d, t, newtype)`
returns datatype with type map

$$\{(float, 0), (float, 4) , (double, 16) , (char, 24) ,
(char, 26) , (char, 27) , (char, 28)\}$$



Further constructors

- **Hvector**
 - Same as vector except that stride is given in bytes
- **Hindexed**
 - Same as indexed except that displacements are given in bytes
- **Indexed block**
 - Same as indexed except that the block length is the same for all blocks
- **Subarray**
 - n-dimensional subarray of an n-dimensional array
- **Darray**
 - Distributed array

Understanding extents



```
char *buffer;  
MPI_Send( buffer, n, datatype, ...);
```

sends the same data as

```
char *buffer;  
MPI_Type_get_extent( datatype, &lb, &extent);  
for ( i=0; i<n; i++)  
    MPI_Send( buffer + (i * extent), 1, datatype, ...);
```

- The extent of a datatype is not its size
- It is closest to being the stride of a datatype
 - From the start of one instance to the start of another instance in a contiguous type

Lower bound and upper bound markers



- Sometimes convenient to explicitly define the lower and upper bound of a type map
 - Allows a datatype to be defined with holes at its beginning or end
 - Allows alignment rules used to compute lower and upper bounds to be overridden. Some compilers allow changing default alignment rules for some structures

```
int MPI_Type_create_resized(MPI_Datatype oldtype,
                           MPI_Aint lb, MPI_Aint extent,
                           MPI_Datatype *newtype)
```

- Creates a modified data type with new lower and upper bounds
- Does not affect the size of the datatype but its extent

Lower and upper bounds - example



```
MPI_Type_create_resized( MPI_INT, -4, 12, type1 );
```

Creates a datatype with type map

$$\{(lb_marker, -4), (int, 0), (ub_marker, 8)\}$$

- The markers `lb_marker` and `ub_marker` are conceptual datatypes that occupy no space
- The extent of the datatype is 12, its size is `sizeof(int)`

```
MPI_Type_contiguous( 2, type1, type2 );
```

Creates a datatype with type map

$$\{(lb_marker, -4), (int, 0), (int, 12), (ub_marker, 20)\}$$

Summary datatypes



- Motivation
 - Non-contiguous data
 - Mixing of datatypes
- General datatype described by type map

$$Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$$

- Size is number of bytes data takes up
- Extent is distance between first and last byte
- Derived datatypes can be composed recursively from potentially non-contiguous blocks of existing datatypes

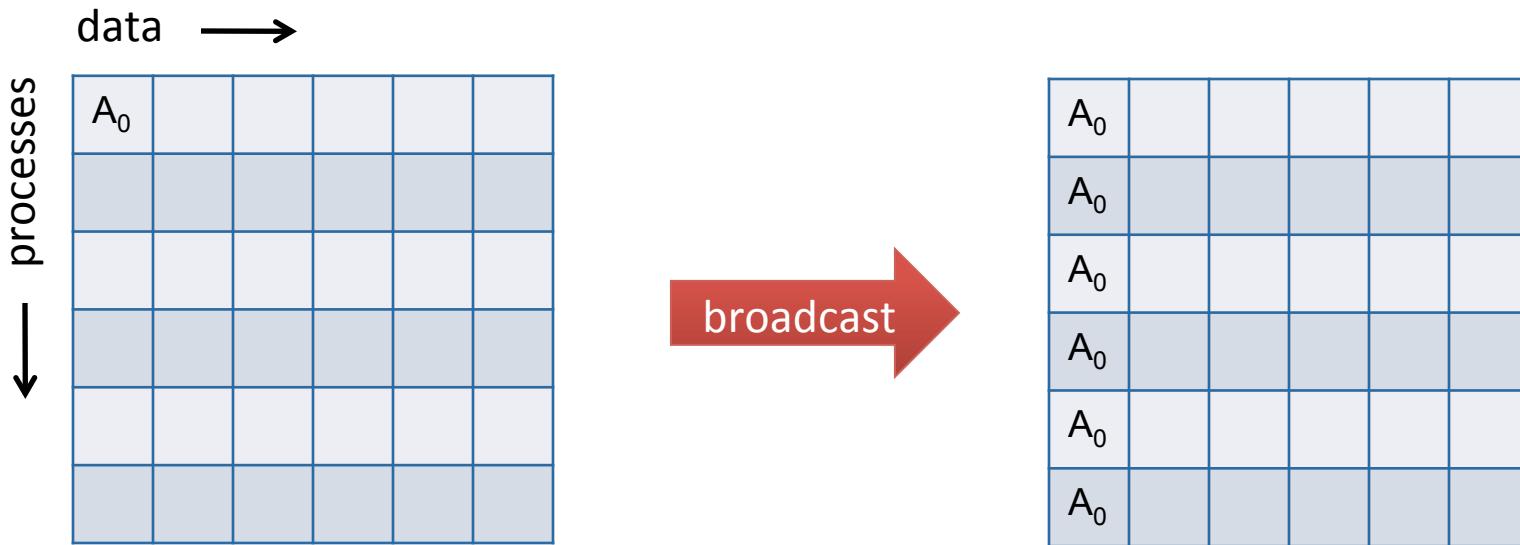
Collective operations



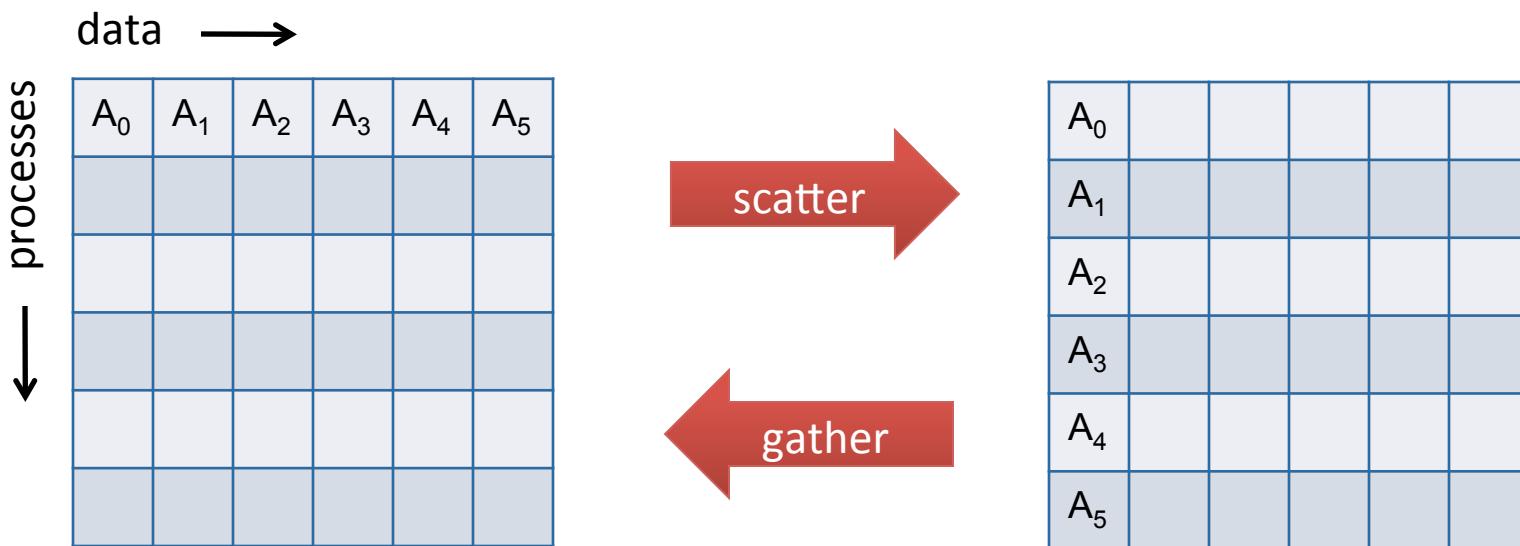
- Operation that involves a group of processes

Barrier synchronization across all members	MPI_Barrier
Broadcast from one member to all members	MPI_Bcast
Gather data from all members to one member / all members	MPI_Gather MPI_Gatherv MPI_Allgather MPI_Allgatherv
Scatter data from one member to all members	MPI_Scatter MPI_Scatterv
Complete exchange (scatter / gather) from all members to all members	MPI_Alltoall MPI_Alltoallv MPI_Alltoallw
Global reduction operations where the result is returned to one member / all members	MPI_Reduce MPI_Allreduce
Combined reduction and scatter	MPI_Reduce_scatter
Scan across all members of a group	MPI_Scan MPI_Exscan

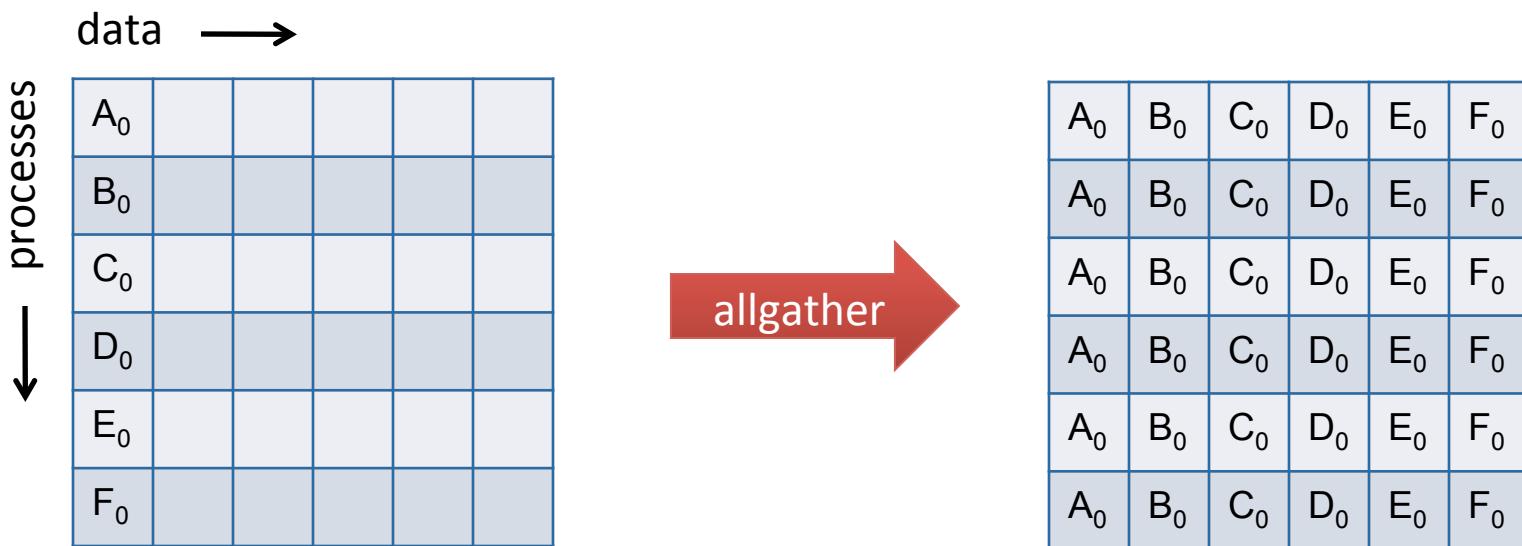
Broadcast



Scatter and gather



Allgather



Complete exchange



data →
↓ processes

A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
B ₀	B ₁	B ₂	B ₃	B ₄	B ₅
C ₀	C ₁	C ₂	C ₃	C ₄	C ₅
D ₀	D ₁	D ₂	D ₃	D ₄	D ₅
E ₀	E ₁	E ₂	E ₃	E ₄	E ₅
F ₀	F ₁	F ₂	F ₃	F ₄	F ₅

complete
exchange

A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₁	B ₁	C ₁	D ₁	E ₁	F ₁
A ₂	B ₂	C ₂	D ₂	E ₂	F ₂
A ₃	B ₃	C ₃	D ₃	E ₃	F ₃
A ₄	B ₄	C ₄	D ₄	E ₄	F ₄
A ₅	B ₅	C ₅	D ₅	E ₅	F ₅

Classification



All-to-all	
All processes contribute to the result and all processes receive the result	MPI_Allgather, MPI_Allgatherv MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw MPI_Allreduce, MPI_Reduce_scatter MPI_Barrier
All-to-one	
All processes contribute to the result and one process receives the result	MPI_Gather, MPI_Gatherv MPI_Reduce
One-to-all	
All processes receive the result	MPI_Bcast MPI_Scatter, MPI_Scatterv
Other	
Do not fit into one of the above categories	MPI_Scan, MPI_Exscan

Collective communication rules



- Several collective routines such as broadcast or gather have a single originating or receiving process
 - Such a process is called the **root**
 - Some arguments are significant only at root and are ignored by all others
- Type matching more strict than in the point-to-point case
 - The amount of data sent must exactly match the amount of data specified by the receiver
 - Different typemaps between sender and receiver still allowed
- Blocking
 - Collective calls may return as soon as their participation in the collective communication is complete
 - A collective call may or may not have the effect of synchronizing all calling processes

Collective communication rules (2)

- All processes in the communicator must call the collective routine
- Often, collective communication can occur “in place” with the output buffer being identical to the input buffer
 - Specified by providing MPI_IN_PLACE instead of send or receive buffer, depending on the operation performed

Barrier synchronization



```
int MPI_Barrier(MPI_Comm *comm)
```

- Blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call
- Applications
 - Correctness (e.g., ready send, I/O)
 - Performance (e.g., limiting the number of messages in transit)

Gather

```
int MPI_Gather(void *sendbuf, int sendcount,  
               MPI_Datatype sendtype,  
               void *recvbuf, int recvcount,  
               MPI_Datatype recvtype,  
               int root, MPI_Comm comm)
```

Number of items to
be received from
each process – not
the total number of
items

- Each process sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order
 - Outcome is as if each of the n processes in the group executed a call to

```
MPI_Send(sendbuf, sendcount, sendtype, root,...);
```

and the root had executed n calls to

```
MPI_Recv(recvbuf + i*recvcount*extent(recvtype),  
         recvcount, recvtype, i, ...);
```

Gather (2)

- Type signatures of sendcount, sendtype on each process must be identical to recvcount, recvtype at the root

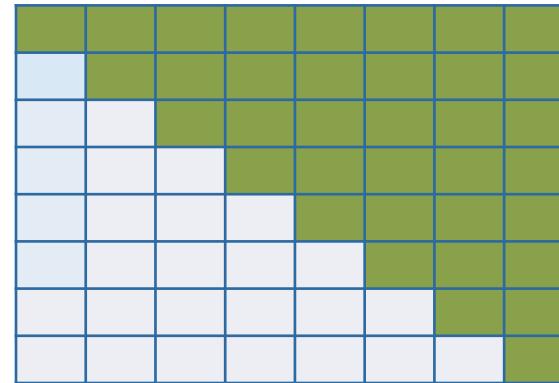
- Extends the functionality of gather by allowing
 - A varying count of data from each process
 - More flexibility as to where the data is placed on the root via displacements

```
int MPI_Gatherv(void *sendbuf, int sendcount,  
                 MPI_Datatype sendtype,  
                 void *recvbuf, int *recvcounts,  
                 int *displs, MPI_Datatype recvtype,  
                 int root, MPI_Comm comm)
```

- The data received from process j is placed into `recvbuf` of the root process beginning at `displs[j]` elements (in terms of the `recvtype`)

Example

- The root process gathers $8-i$ doubles from each rank i and places them into the rows of an 8×8 array, filling the upper triangular matrix



Example (2)



```
int myrank;
double recvcounts[] = {8,7,6,5,4,3,2,1};
double displs[] = {0,9,18,27,36,45,54,63};

MPI_Comm_rank(comm, &myrank);
MPI_Gatherv(sendbuf,
            8-myrank,                      /* send count */
            MPI_DOUBLE,                     /* send datatype */
            8x8_array,                     /* receive buffer */
            recvcounts,                    /* receive counts */
            displs,                        /* displacements */
            MPI_DOUBLE,                     /* receive datatype */
            root,
            comm);
```

Scatter and scatterv



- Scatter is the inverse operation to gather

```
int MPI_Scatter(void *sendbuf, int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf, int recvcount,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm)

int MPI_Scatterv(void *sendbuf, int *sendcounts,
                 int *displs, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

All-to-all exchange



- **MPI_Allgather / MPI_Allgatherv**
 - Can be thought of as a gather / gatherv, but where all processes receive the result, instead of just the root
- **MPI_Alltoall / MPI_Alltoallv**
 - An extension of allgather to the case where each process sends distinct data to each of the receivers
 - Alltoallv allows displacements to be specified for senders and receivers
- **MPI_Alltoallw**
 - Most general form of complete exchange
 - Allows separate specification of count, displacement, and datatype; displacement is specified in bytes

All-to-all exchange (2)



- **MPI_Allreduce**
 - Like a normal reduce, except that result appears in the receive buffer of all group members
- **MPI_Reduce_scatter_block** and **MPI_Reduce_scatter**
 - The result of the reduce operation is scattered to all group members

Reduction operations



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- MPI_Reduce
- MPI_Allreduce
- MPI_Reduce_scatter_block
- MPI_Reduce_scatter
- MPI_Scan
- MPI_Exscan

Predefined reduction operations



Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or
MPI_BXOR	bit-wise exclusive or
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

MINLOC and MAXLOC



- Operators that can be used to compute an extreme value (min/max) and the rank of the process containing this value

MPI_MAXLOC

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \max(u, v)$$

and

$$k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

MPI_MINLOC

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \min(u, v)$$

and

$$k = \begin{cases} i & \text{if } u < v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

MINLOC and MAXLOC (2)



- These reduction operations are defined to operate on arguments that consist of a pair
- The C binding of MPI provides suitable pair types

Name	Description
MPI_FLOAT_INT	float and integer
MPI_DOUBLE_INT	double and integer
MPI_LONG_INT	long and integer
MPI_2INT	pair of integers
MPI_SHORT_INT	short integer and integer
MPI_LONG_DOUBLE_INT	long double and integer



Example

- Each process has an array of 30 doubles. For each of the 30 entries, compute the value and rank of the process containing the largest value

```
/* each process has an array of 30 double: ain[30] */
double ain[30], aout[30];
int ind[30];
struct {
    double val;
    int rank;
} in[30], out[30];
int i, myrank, root = 0;
```

Example (2)



```
MPI_Comm_rank(comm, &myrank);
for (i=0; i<30; ++i) {
    in[i].val = ain[i];
    in[i].rank = myrank;
}
MPI_Reduce( in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm );
/* At this point, the answer resides on process root */
if (myrank == root) {
    /* read ranks out */
    for (i=0; i<30; ++i) {
        aout[i] = out[i].val;
        ind[i] = out[i].rank;
    }
}
```

- Inclusive scan
 - Performs a prefix reduction on data distributed across the group

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm)
```

- The operation returns, in the receive buffer of the process with rank i , the reduction of the values in the send buffers of the processes with ranks $0, \dots, i$ (inclusive)
- Exclusive scan (same arguments)
 - For processes with rank $i > 0$, the operation returns, in the receive buffer of the process with rank i , the reduction of the values in the send buffers of processes with ranks $0, \dots, i-1$ (inclusive)

User-defined reduction operations



```
int MPI_Op_create(MPI_User_function *function,  
                  int commute,  
                  MPI_Op *op)
```

- Binds a user-defined reduction operation to an operation handle that can subsequently be used in a collective reduction call
 - The user-defined operation is assumed to be associative, allowing the order of the evaluation to be changed
 - If commute = true, then it is also assumed to be commutative
 - If commute = false, then the order of arguments is fixed and defined to be in ascending rank order
- Can be deleted using

```
int MPI_Op_free(MPI_Op *op)
```

User-defined reduction operations (2)



- The user function must have the following prototype

```
void MPI_User_function(void *invec, void *inoutvec,  
                      int *len, MPI_Datatype *datatype)
```

- We can think of invec and inoutvec as arrays of len elements that the function is combining
- The results of the reduction overwrites values inoutvec
- Each invocation of the function results in the element-wise evaluation of the reduction operator
- $\text{inoutvec}[i] = \text{invec}[i] \odot \text{inoutvec}[i]$

Example: user-defined function



```
typedef struct {
    double real, imag;
} Complex;

void myProd( Complex *in, Complex *inout,
             int *len, MPI_Datatype *dptr ) {
    int i;
    Complex c;

    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real - inout->imag*in->imag;
        c.imag = inout->real*in->imag + inout->imag*in->real;
        *inout = c;
        in++; inout++;
    }
}
```

Example: how to use it



```
/* each process has an array of 100 Complexes */

Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

/* explain to MPI how type Complex is defined */
MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
MPI_Type_commit( &ctype );

/* create the complex-product user-op */
MPI_Op_create( myProd, 1, &myOp );
MPI_Reduce( a, answer, 100, ctype, myOp, root, comm );

/* At this point, the answer, which consists of 100
   Complexes, resides on process root */
```



Summary collective operations

- Motivation – convenience and efficiency
- Classification
 - All-to-all, all-to-one, one-to-all, other
- Rules
 - Some operations have distinct root process
 - All processes of a communicator must call the operation
- Synchronization
 - Explicit synchronization via barrier
 - Some operations may involve implicit synchronization
- NOTE: So far, we covered only blocking operations



This is what we learned

- Message-passing model
- Point-to-point communication
- Virtual topologies
- Datatypes
- Blocking collective communication



Features not covered so far

- Intercommunicators
- Profiling interface
- Error handling
- Dynamic process management
- One-sided communication
- Multithreaded MPI applications
- File I/O
- Non-blocking collectives
- Neighborhood collectives