

CS-E4690 – Programming parallel supercomputers D

1 - Course management

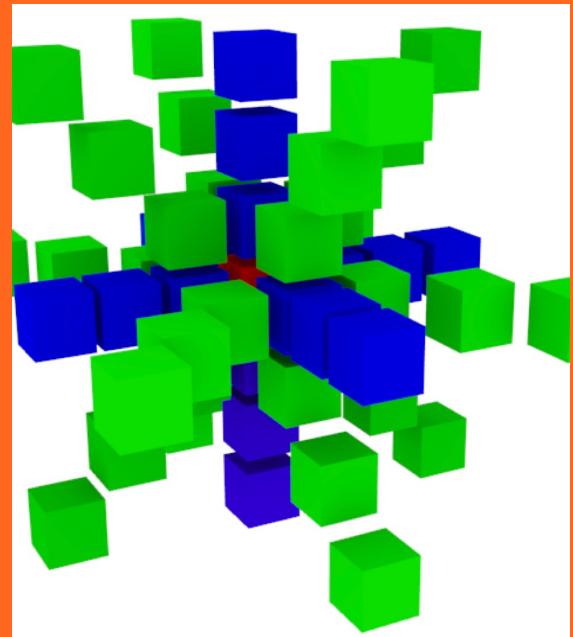
Maarit Korpi-Lagg

maarit.korpi-lagg@aalto.fi

24.10.2023



Aalto University
School of Science



Other teaching staff

Touko Puro (TA): touko.puro@aalto.fi (coding exercises)

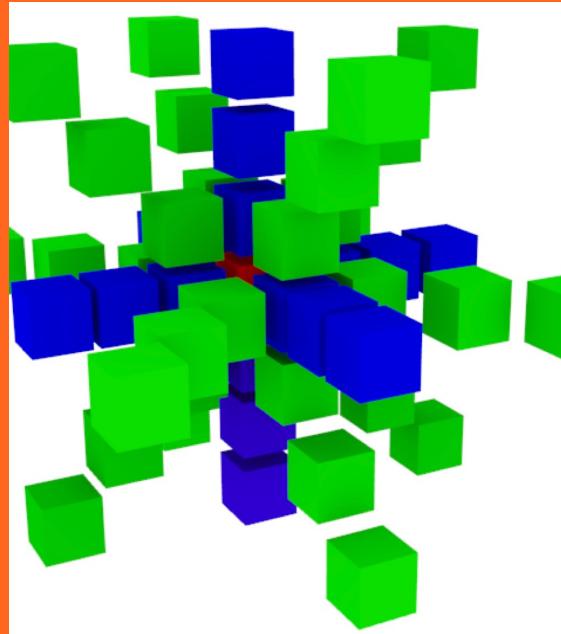
**Matthias Rheinhardt: matthias.rheinhardt@aalto.fi (back-up
lecturer & coding exercises)**

**Mira Salmensaari: mira.salmensaari@aalto.fi (Technical Triton
support)**

Learning outcomes



Aalto University
School of Science



Our learning objectives **are**

- **Get yourself familiarized with the current HPC landscape** to be able to choose the correct framework for your large-scale problem.
- **Learn basic concepts on how to build efficient applications** for clusters or supercomputers with thousand(s) to million(s) of cores
- **Master distributed memory and hybrid** (distributed + shared memory) programming models
- **Learn essentials of message-passing interface**
- **Learn essentials of HPC in hybrid architectures with graphics processing units (GPUs).**

Our learning objectives **are not**

- To become fluent in using Triton and CSC supercomputing environments; we get you started in Triton; SciComp and CSC trainings will support you further
- To solve any practical large-scale problem; the next course in the series, **CS-E4002 Large-scale computing and data analysis**, will deal with practical applications in the CSC environment.

The knowledge presented in this course, even quite theoretical, will be useful for your practical applications.

Break-down of learning objectives

Lecture1

Introduction to the current HPC landscape

Understanding how this course fits into that

Establishing understanding of the learning outcomes

Lecture2

Learning basic definitions and taxonomies

Understanding the importance of the “network”

Learning basic performance models

Lecture3

Becoming knowledgeable of the modern landscape of distributed memory programming

Understanding why in this course we will concentrate on low-level programming models

Getting acquainted with MPI: basics and synchronous and asynchronous point-to-point communication

Break-down of learning objectives

Lecture4

Learning more about MPI:

One-sided point-to-point communications

Collective communications

Lecture5

Programming MP hybrid architectures

Becoming knowledgeable of the spectrum of options

Understanding efficiency issues

Lecture6

Programming hybrid architectures with accelerators

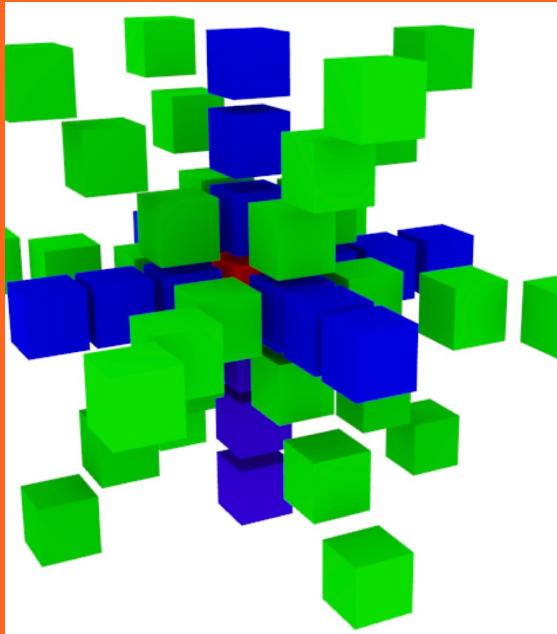
Acquiring knowledge of CUDA-MPI programming model

Some background from “Programming parallel computers” is essential

Practicalities



Aalto University
School of Science



Sessions

- Six 90 min "lectures" on Tuesdays 14:15, in T5.
- Pre-recorded **videos & slides** + extra reading
 - Compilation of **materials** in **MyCo** belonging to each lecture
 - Minimally watch/read through the **core material** (indicated) before the Tuesday session; **extra reading** (indicated) is for the curious/passionate.
 - The lecture times are intended for: **synthesis!**
 - Q&A about the materials; discussion points
 - going through the example codes
 - completing some exercises
 - You can also post questions or comments about the **lectures/exercises** in the **Zulip chat**, and these will be discussed/answered on **Tuesdays/Fridays**.

Exercises

- Course grading is solely based on the exercise points. **All sheets must be returned.** One exercise sheet per lecture. Available from the course GitLab during the entire course.
- **2 first ones** involve no coding, and will **preferably undertaken as group work during the first 2 lectures.**
- **4 last ones are coding exercises.**
- Exercise session on **Fridays 12:15-13:45 (Y342a)**: technical assistance on Triton environment and **help/hints on the coding exercises will be provided during this session. Be there!**
- DL for submissions is in the end of the evaluation week, **16:00 on Friday the 8th of Dec. The DL is strict.**

Basic exercises (2 first weeks)

- Reading, understanding + a little bit of maths
- You can get **full points** by participating in the lecture time group work.
- If you cannot attend the lecture, then you have to return the material in as a **learning diary** in MyCo, and **the contents will be graded**.
- **The completion of the first exercise is necessary to get a Triton account. DL is Tue 31st of Oct, 16:00.**

Triton

HPC environment to be used is Tier-2 semi-local **Triton cluster**; a temporary user account will be set up for those who do not already have one **and who have completed Ex. 1 in time**. Some relevant links for self-study:

<https://scicomp.aalto.fi/triton/#tutorials>

https://www.youtube.com/watch?v=13gikRotUVQ&list=PLZLVMs9rf3nPRb-QjrWsg_ftUfJ5Bbv07

Support available in Zulip and during exercises!

Coding exercises

- **HPC environment** to be used is Tier-2 semi-local **Triton cluster**; a temporary user account will be set up for those who do not already have one **and who have completed Ex. 1 in time**.
- Example codes, scripts, and the exercise sheets themselves are available through **GitLab**.
- **Everybody is encouraged to create repositories in Aalto GitLab**, as the user accounts are temporary (they will expire **31.12.2023**; for those who did not have one before)
- Submission of all materials to a dedicated submission directory set up per each participant in Triton. Codes will be **evaluated in Triton directly; readable only to course personnel and the participant**. **DL 8.12.2023 16:00**.
- **Model solutions** will be made available in GitLab to those who submitted **all sheets** (link available **11.12.2023**).

Course feedback

- Standard feedback query; please return!
- **Continuous anonymised feedback welcomed through MyCo at all times.**
- Dedicated (voluntary) feedback session on **Tue 12th of Dec, lecture time, online&physical, see MyCo calendar.**

Help to improve the relatively new course for future years!

Communications

- Pre-recorded videos, slides, and reading materials are posted in [MyCourses](#)
- Example codes, scripts, exercise sheets available from [GitLab repo](#).
- **Zulip chat** for the course

pps23.zulip.cs.aalto.fi

[Invite link](#)

CS-E4690 – Programming Parallel Supercomputers

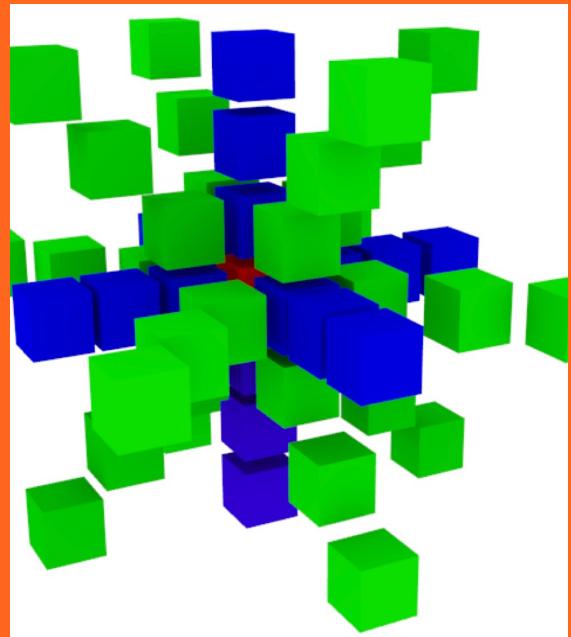
Current HPC landscape

Maarit Korpi-Lagg

maarit.korpi-lagg@aalto.fi



Aalto University
School of Science



Why HPC & What is it?

Task-oriented definition (currently)

"A computer system designed to execute applications that would take days/years/centuries on a desktop/mobile device, in seconds/minutes or require weeks or months to run, even at large scale."

Why HPC & What is it?

Hardware-oriented definition (currently)

Network of processing elements (PEs, currently of 10k or more) that enable them to process the task in parallel. In a common use case, the PEs need to exchange data, hence the HPC system needs to have fast memories and low-latency, high-bandwidth communication systems between the PEs and PEs and the associated memories (>100Gb/s).

Parallelism==computation + communication and making them concurrent

Example

- Let us assume that you want to solve a dense linear system of the form $Ax = b$, where A is a $n \times n$ matrix, and b is a n vector.
- You plan to use an unoptimized algorithm with computational complexity of $\mathcal{O}(n^3)$.
- You have a PC that is capable of computing 50 billion floating point multiplications a second.

How long will this take?

n	required ops	time
10	1000	20 ns
100	1M (10^6)	20 μ s
1,000	1G (10^9)	20 ms
10,000	1T (10^{12})	20s
100,000	1P (10^{15})	20,000s=5,5h
1,000,000	1E (10^{18})	20,000,000s=231d

If you have such processors working in parallel, the large-scale problem becomes computable

How long will this take?

If you have such processors working in parallel, the large-scale problem becomes computable *); ideally, doubling the number of processors will halve the computing time

n	nproc	time
1,000,000	1	231d
-"-	2	116d
-"-	4	58d
...
-"-	1024	5,4h

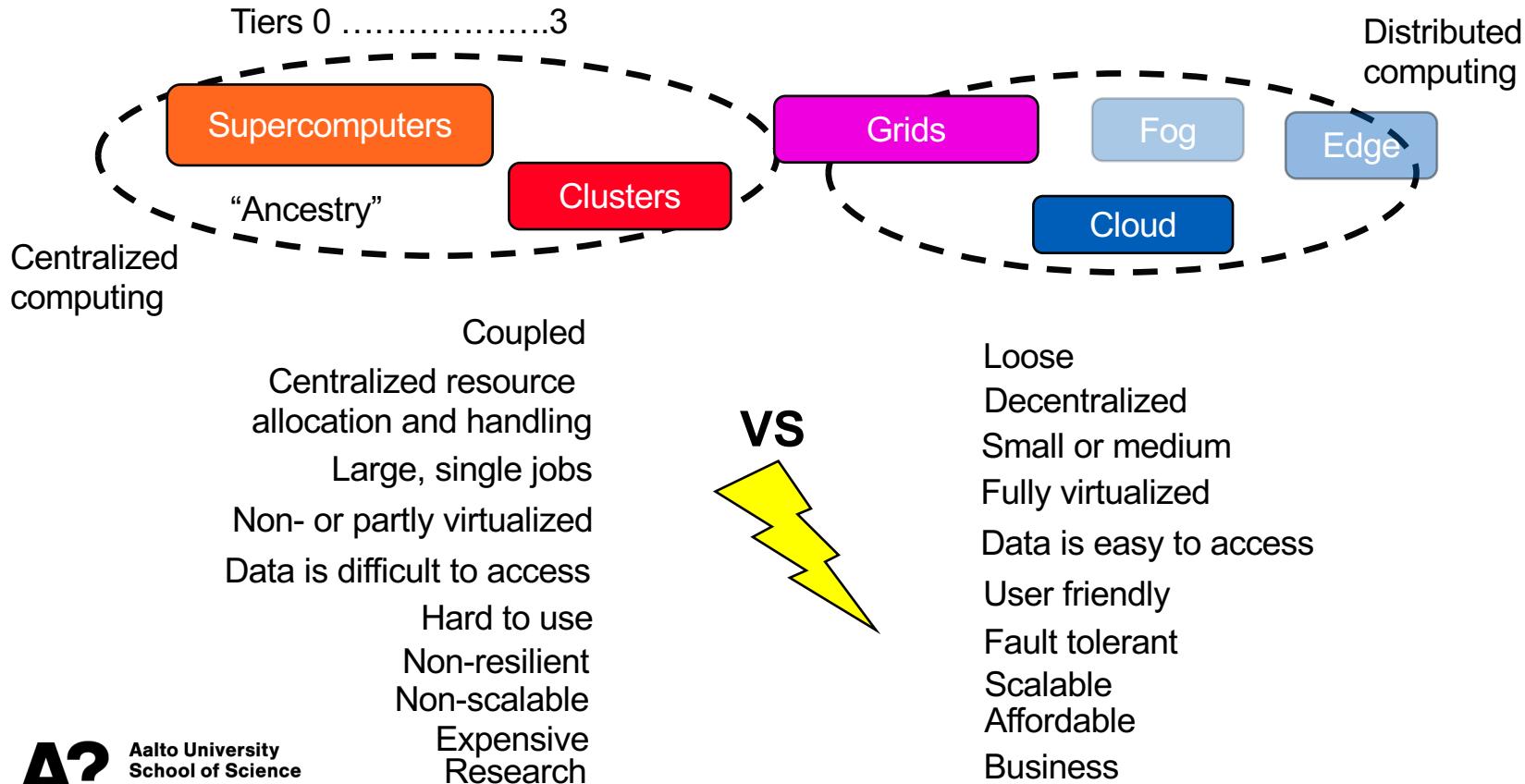
*) in this case, however, you should also optimize your algorithm, too.

Why HPC & What is it?

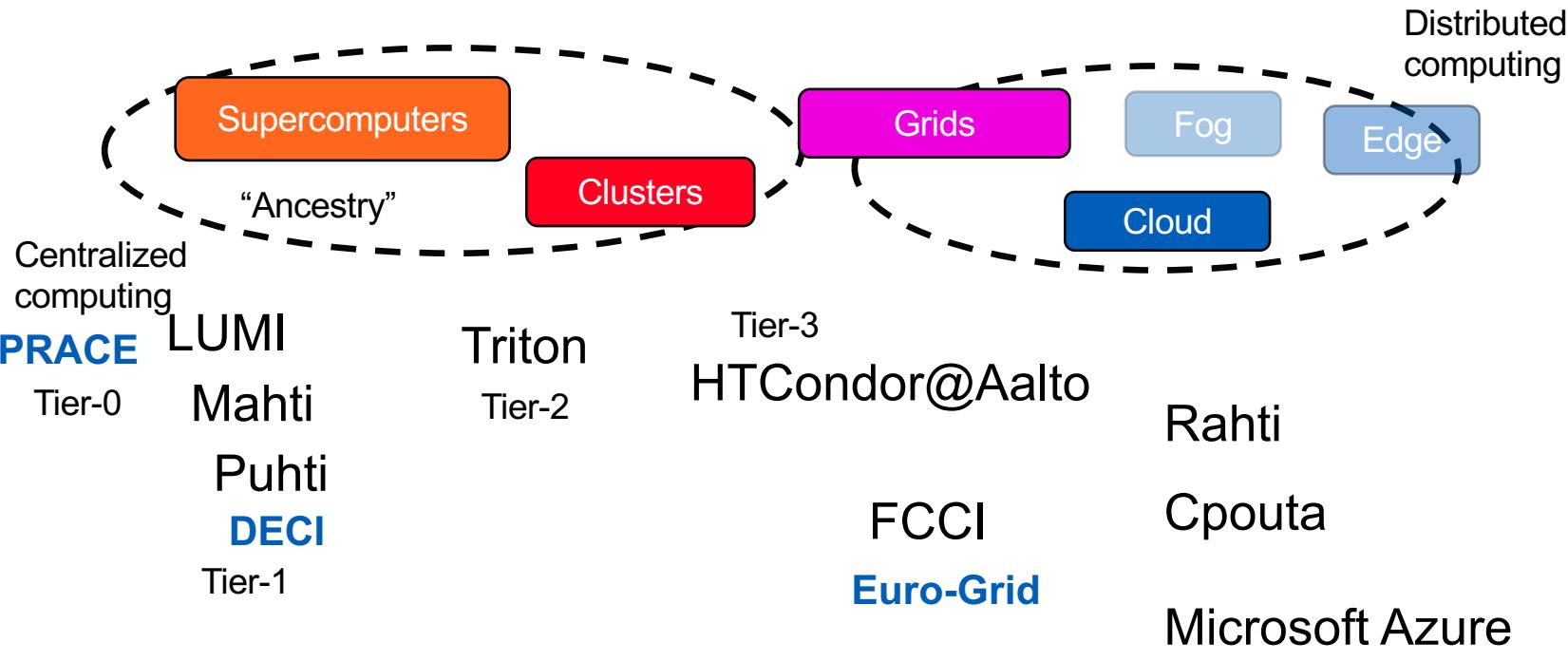
- - (IMO) HPC==any computation or data analysis task that you cannot perform in a standard desktop computer at hand
- This course: we learn to use supercomputing environments *)
 - **Minimalistic** computing environment
 - Constant, **rapid changes** to the environment – code becomes obsolete if you do not maintain it
 - Parallel programming is **hard** – producing scalable and optimized code is an effort
 - **HPC paradigms change fast** – need to be ready for complete turnaround (currently from homogeneous to heterogeneous systems)



A rough sketch of the current HPC landscape

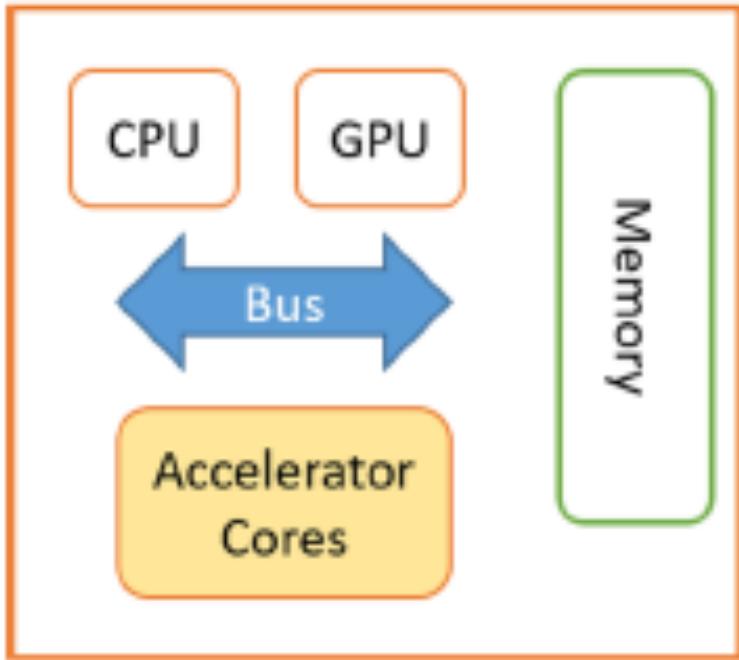


A rough sketch of the current HPC landscape



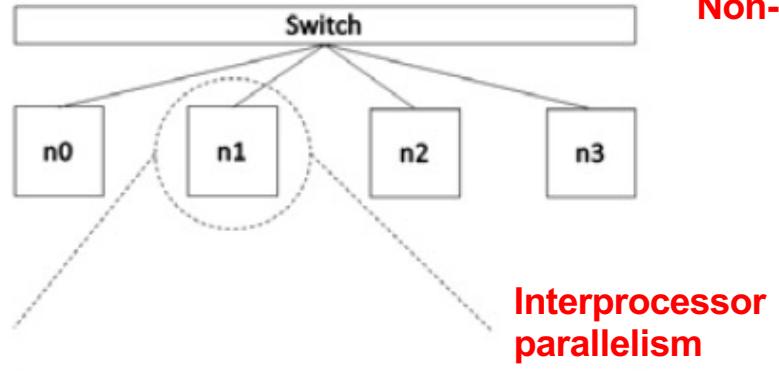
Read more from the websites linked to in the core material

Current paradigm: Heterogeneity

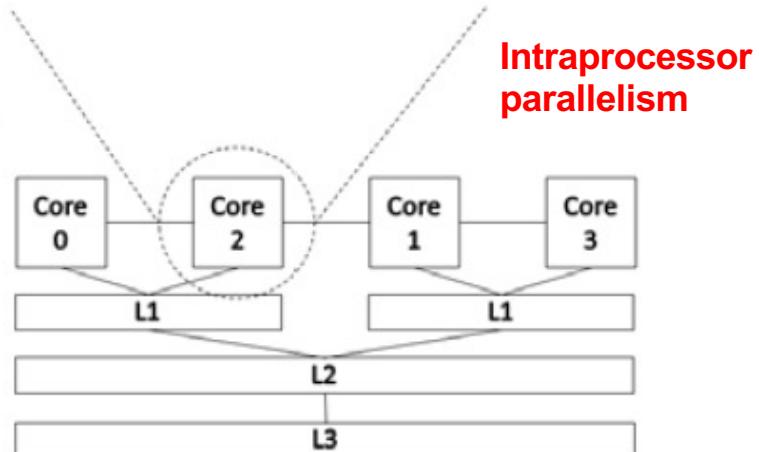
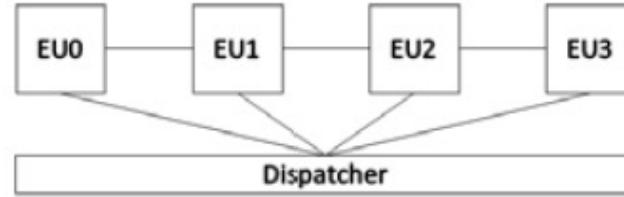
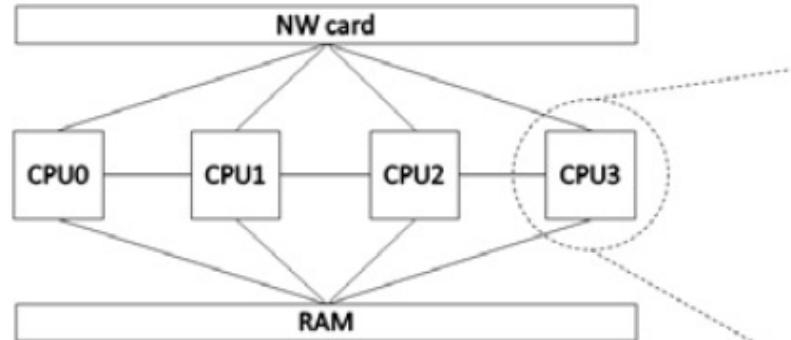


Heterogeneity can be predicted to increase with the inclusion of even more specialized hardware components with built-in logic for interfacing

Current paradigm: NUMA



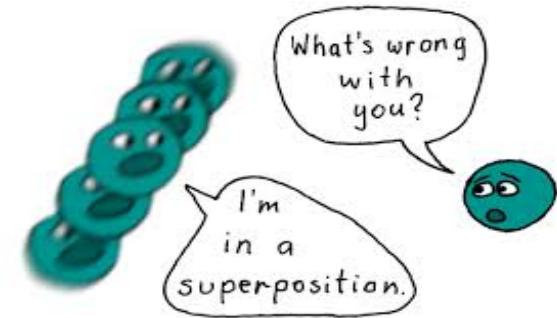
Non-Uniform Memory Access: Access to one's own memory is faster than somebody else's



Next paradigm: quantum computers

- From **bits** (transistor values of ‘1’ and ‘0’) to **qubits** (quantum states of, e.g., electron in superposition of ground (0) or excited states (1))
- Classical logic is replaced by the rules of quantum mechanics
- Quantum effects: Superposition – interference – entanglement
- Cubits are extremely sensitive – building and operating a stable quantum computer will not be trivial
- Currently < 100 cubits,

Quantum advantage – a few years
Quantum supremacy – decade

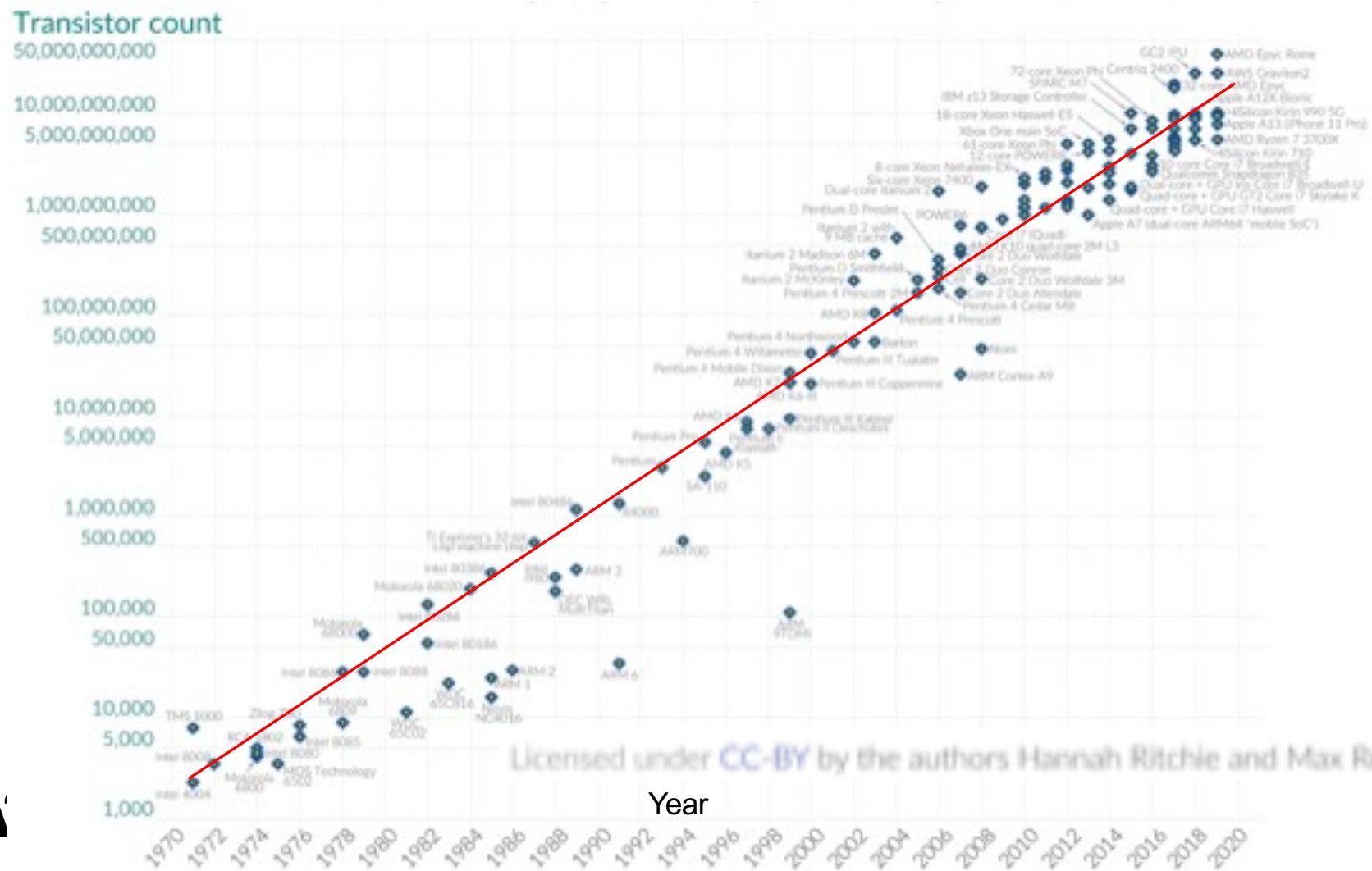


Credit: CC BY-SA 3.0

Moore's first law

“The number of transistors per chip doubles every two years”

Gordon Moore (1965)



A

Moore's second law

Although the cost for consumers has been constant....

The capital cost of a semiconductor fab has also increases exponentially over time

80 microns to few dozen nanometers
few million to 10 billion by 2018

Dennard scaling

Power density $\frac{P}{A}$ in CMOS transistors is constant to the first order when their size is reduced (Dennard, 1974)

- $P \propto aCfU^2$; a is the switching frequency, C the capacitance, f the clock frequency, and U is the voltage.
- Making the transistors smaller, their power use remains in proportion with the area.
- The clock frequency can be increased when shrinking the size, keeping the power consumption roughly the same.
- Hence, Dennard's recipe to make ever faster chips was to **make the transistor smaller**.

Power wall

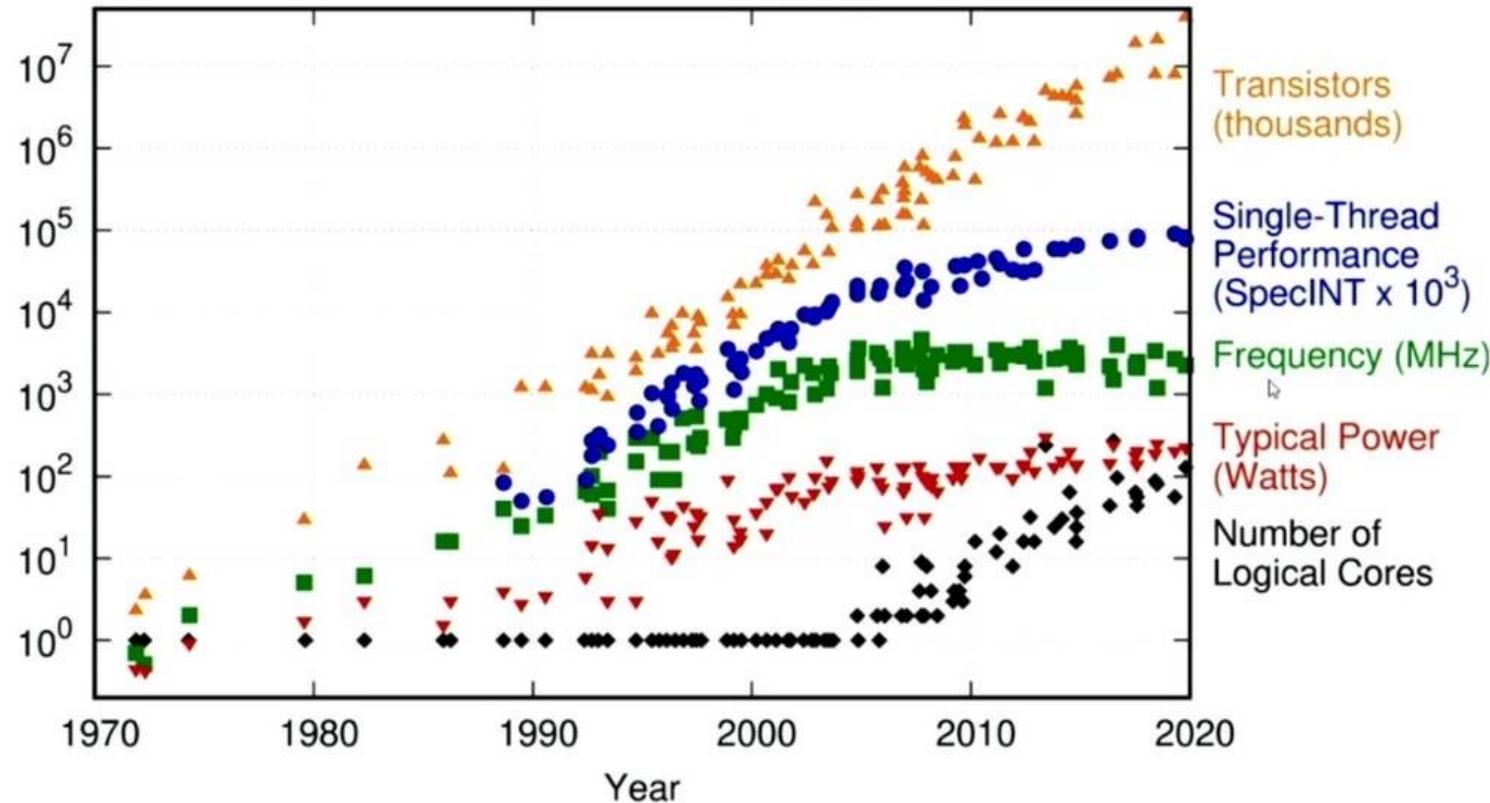
CMOS chip design evolution: Dennard scaling does no longer hold.

- $P \propto aCfU^2 + I_{leakage}U + I_{short\ circuit}U$
- When the voltage is reduced, higher-order effects, namely power losses become important, and finally dominant
- These heat up the chip, and lead to further power losses.
- The transistor no longer operates reliably, that can finally lead to thermal runaway.

More in Bose (2011)

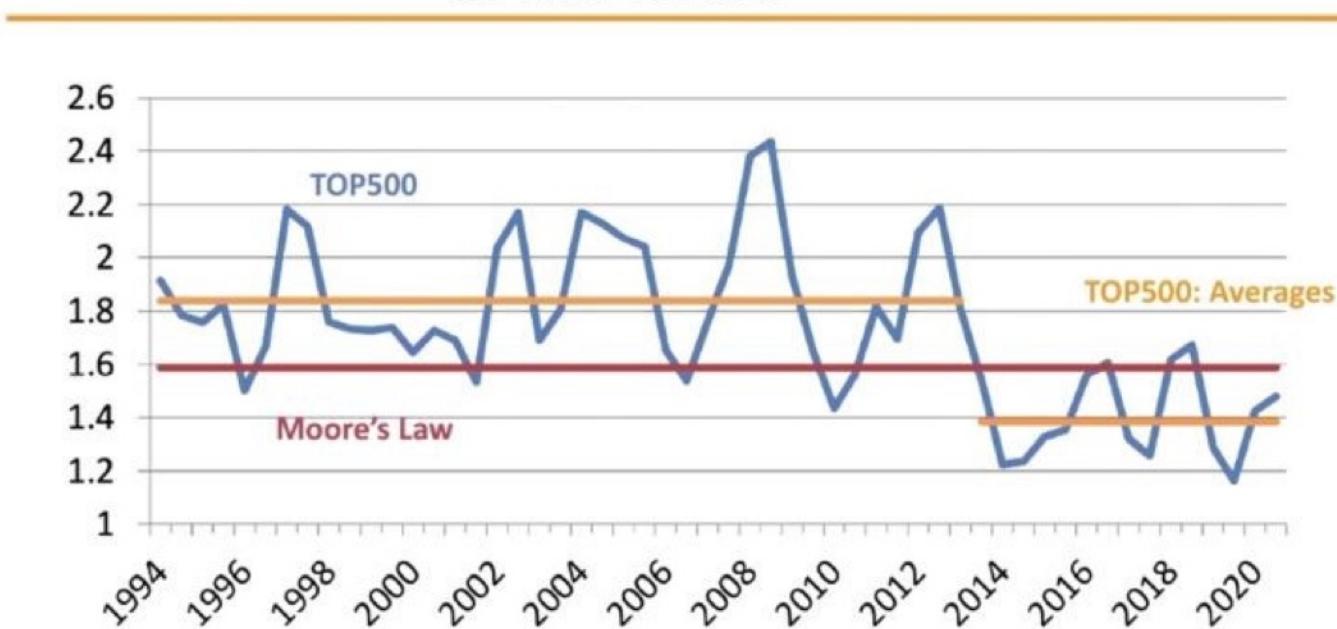
50 Years of Technology Scaling

48 Years of Microprocessor Trend Data



Trend seen in top500 list

ANNUAL PERFORMANCE INCREASE
OF THE TOP500



Materials (core)

N. Sadashiv and S. M. D. Kumar, "Cluster, grid and cloud computing: A detailed comparison," *2011 6th International Conference on Computer Science & Education (ICCSE)*, 2011, pp. 477-482, doi: 10.1109/ICCSE.2011.6028683, and references therein.

Aalto scientific computing: <https://scicomp.aalto.fi/about/>

CSC scientific computing: <https://research.csc.fi/csc-s-servers>

TOP500 supercomputer list <https://www.top500.org/>

PRACE resources: <https://prace-ri.eu>

FCCI infrastructure: <https://www2.helsinki.fi/en/infrastructures/fcci>

Materials (extra)

Asch, M. et al. (2018) 'Big data and extreme-scale computing: Pathways to Convergence-Toward a shaping strategy for a future software and data ecosystem for scientific inquiry', *The International Journal of High Performance Computing Applications*, 32(4), pp. 435– 479. doi: 10.1177/1094342018778123.

Guidi, G., Ellis, M., Buluç, A. Yelick, K., and Culler, D. 2021. 10 Years Later: Cloud Computing is Closing the Performance Gap. In Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE '21). Association for Computing Machinery, New York, NY, USA, 41–48. DOI:<https://doi.org/10.1145/3447545.3451183>

Bose P. (2011) Power Wall. In: Padua D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-09766-4_499

http://www.cs.virginia.edu/~robins/The_Limits_of_Quantum_Computers.pdf

<https://www.csc.fi/fi/web/training/-/quantum-computing>

CSC's Quantum Learning Machine: <https://research.csc.fi/-/kvasi>

Materials (extra)

Cardwell, S.G. *et al.* (2020). Truly Heterogeneous HPC: Co-design to Achieve What Science Needs from HPC. In: Nichols, J., Verastegui, B., Maccabe, A.'., Hernandez, O., Parete-Koon, S., Ahearn, T. (eds) Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI. SMC 2020. Communications in Computer and Information Science, vol 1315. Springer, Cham. https://doi.org/10.1007/978-3-030-63393-6_23; also the whole conference proceedings book is an interesting read.

Supalov, A., 201, Inside the Message Passing Interface: Creating fast communication libraries, Walter de Gruyter Inc., Boston/Berlin
<https://doi.org/10.1515/9781501506871>; recommendation for wannabe MPI geeks

CS-E4690 – Programming parallel supercomputers D

2nd Lecture

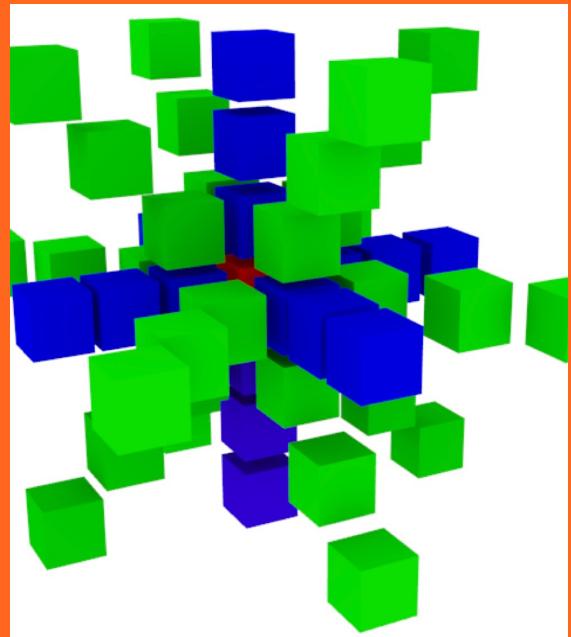
Maarit Korpi-Lagg

maarit.korpi-lagg@aalto.fi

31.10.2023



Aalto University
School of Science



Lecture 2

Basic definitions and the
importance of the network

- **Course practicalities repetition: 5 min**
- **Unbroken phone: key concepts recap (30 min)**
- **Exercise 2 as group work:**
 - 4 topics (from sheet2 in GitLab)
 - Row-wise group work (10 mins work, 2 mins for discussing answers)
 - Synthesis (5 mins max)
- **Signing up for points and feedback (2 mins)**

Break-down of learning objectives

Lecture1

Introduction to the current HPC landscape

Understanding how this course fits into that

Establishing understanding of the learning outcomes, specifically answering the question: “What are programming during this course?”

Lecture2

Learning basic definitions and taxonomies

Understanding the importance of the “network”

Learning basic performance models

Understanding the concept of a well-performing software in large-scale computing.

Lecture3

Becoming knowledgeable of the modern landscape of distributed memory programming

Understanding why in this course we will concentrate on low-level programming models

Getting acquainted with MPI: basics and synchronous and asynchronous point-to-point communication

Break-down of learning objectives

Lecture4

Learning more about MPI:

One-sided point-to-point communications

Collective communications

Lecture5

Programming MP hybrid architectures

Becoming knowledgeable of the spectrum of options

Understanding efficiency issues

Lecture6

Programming hybrid architectures with accelerators

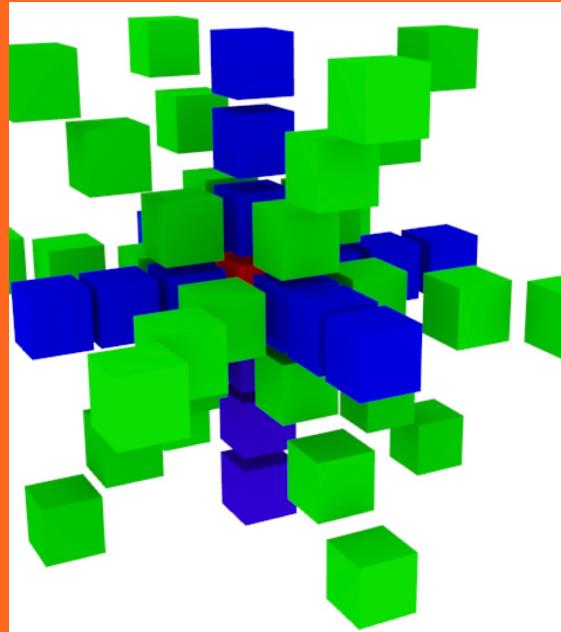
Acquiring knowledge of CUDA-MPI programming model

Lecture 2

Basic definitions and the
importance of the network

- Course practicalities repetition: 5 min
- Unbroken phone: key concepts recap (30 min)
- Exercise 2 as group work:
 - 4 topics (from sheet2 in GitLab)
 - Row-wise group work (10 mins work, 2 mins for discussing answers)
 - Synthesis (5 mins max)
- Signing up for points and feedback (2 mins)

Unbroken phone: Key concepts



A''

Aalto University
School of Science

Flynn's taxonomy

What is it in short?

Green(s) explain ~ 1 min to the group

Red/yellow explains it to the class ~ 1 min

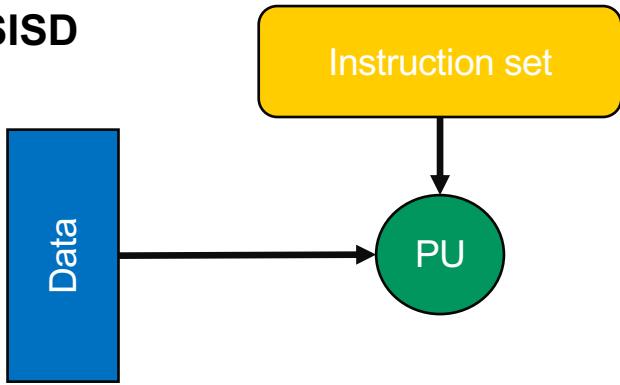
Model answer

Flynn's taxonomy

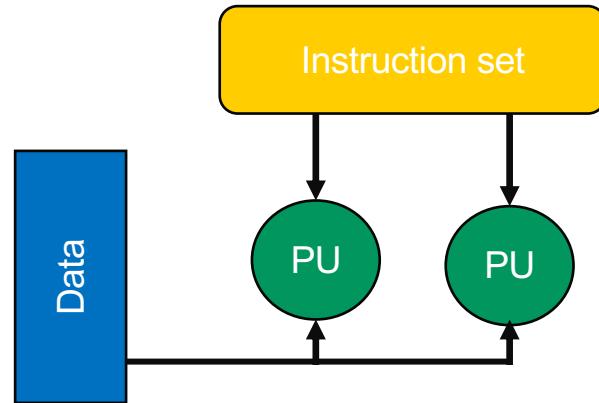
It is a coarse-grained classification of computer architectures using only three abstraction levels: processing unit, data streams and instruction streams. In its original form it contains four different classes.

Flynn's taxonomy

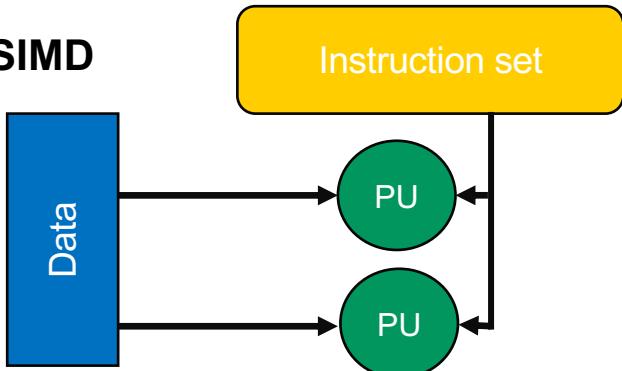
SISD



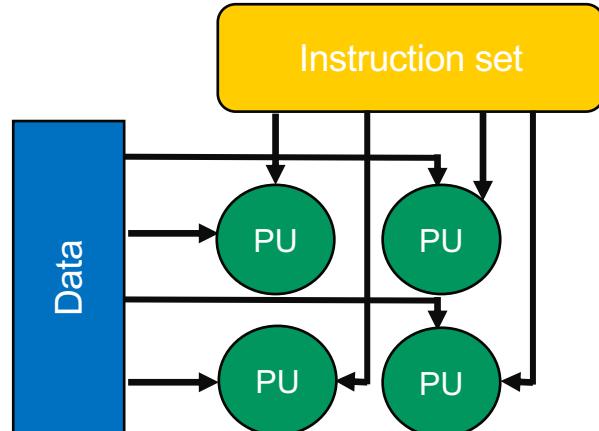
MISD



SIMD



MIMD



Iterative stencil loop

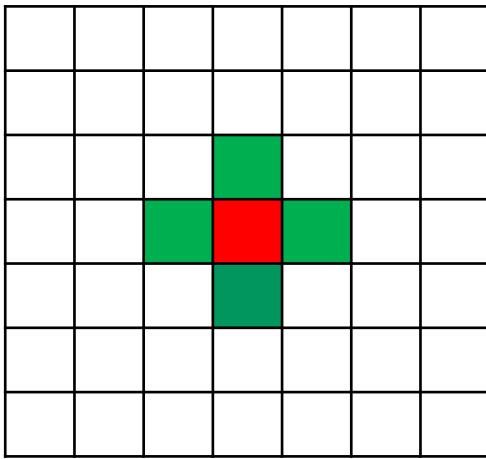
What is it in short?

Green(s) explain ~ 1 min to the group

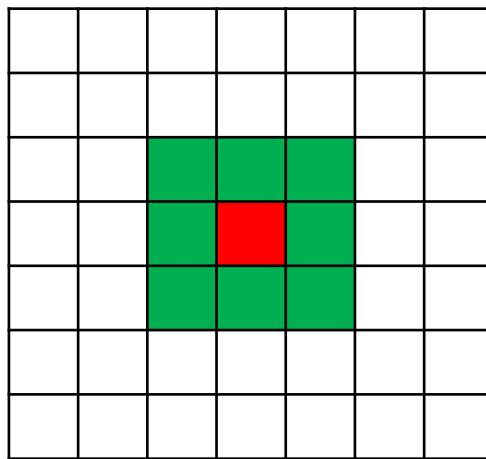
Red/yellow explains it to the class ~ 1 min

Model answer

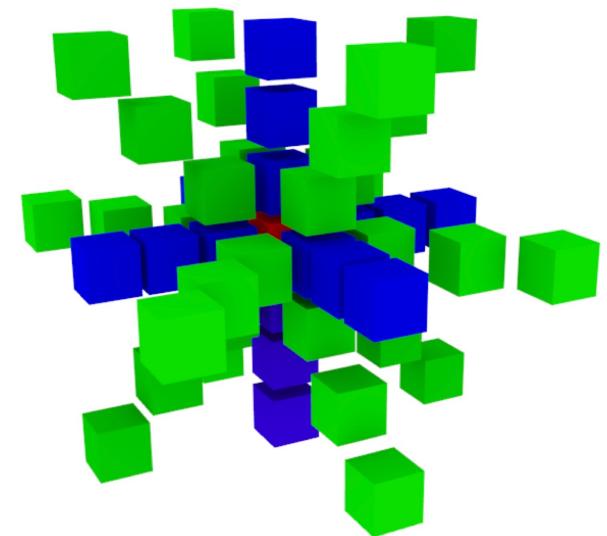
Iterative stencil loop



2D von Neumann



2D Moore



3D 55-point stencil

Recurring update pattern of array elements based on their neighbors.

Amdahl's law

What is it in short?

Green(s) explain ~ 1 min to the group

Red/yellow explains it to the class ~ 1 min

Model answer

Amdahl's law

A fixed computational problem is given to increasing number of processing elements. The serial parts of the code strongly dominate the scale-up.

Gustafsson's law

What is it in short?

Green(s) explain ~ 1 min to the group

Red/yellow explains it to the class ~ 1 min

Model answer

Gustafsson's law

When increasing the number of processing elements, the problem size is allowed to grow. The serial part of the code no longer constrain the speed up.

Interconnect topology: bisection or diameter

Pick one: what is it in short?

Green(s) explain ~ 1 min to the group

Red/yellow explains it to the class ~ 1 min

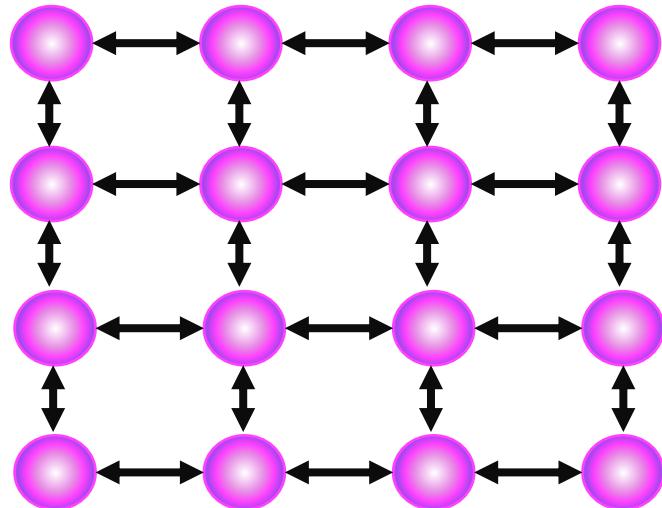
Model answer

Interconnect topology: bisection and diameter

Bisection: minimum number of links that divide the network into two equal halves (**estimates worst case bandwidth**)

Diameter: maximum number of links between nodes **over a path with minimal distance (worst case routing distance)**

2D mesh



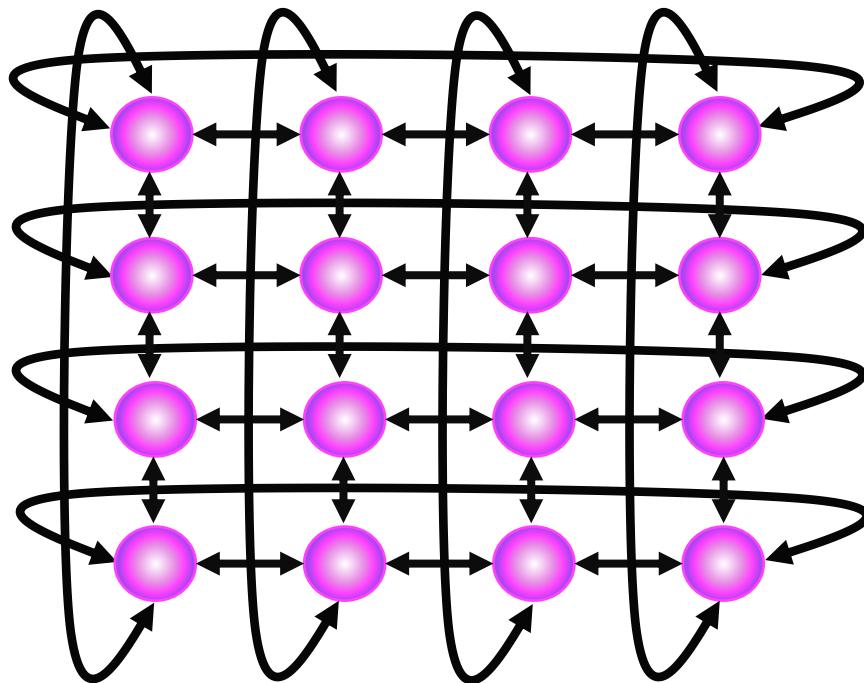
a. Two-dimensional 4x4 mesh.

Bisection: 4, Diameter: 6, generalization \sqrt{N} , $2(\sqrt{N} - 1)$

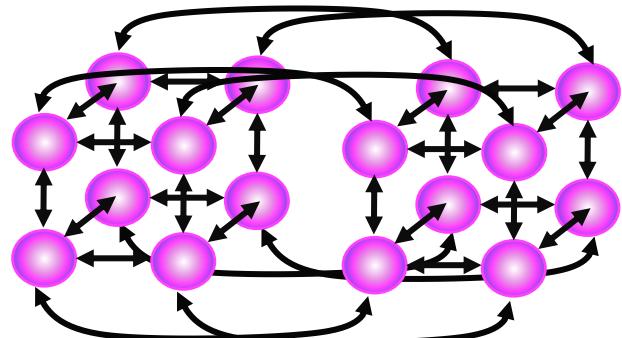
2D torus

b. Two-dimensional 4x4 torus.

Bisection: 8, Diameter: 4, generalization $2\sqrt{N}$, \sqrt{N}



“8+8” Hypercube



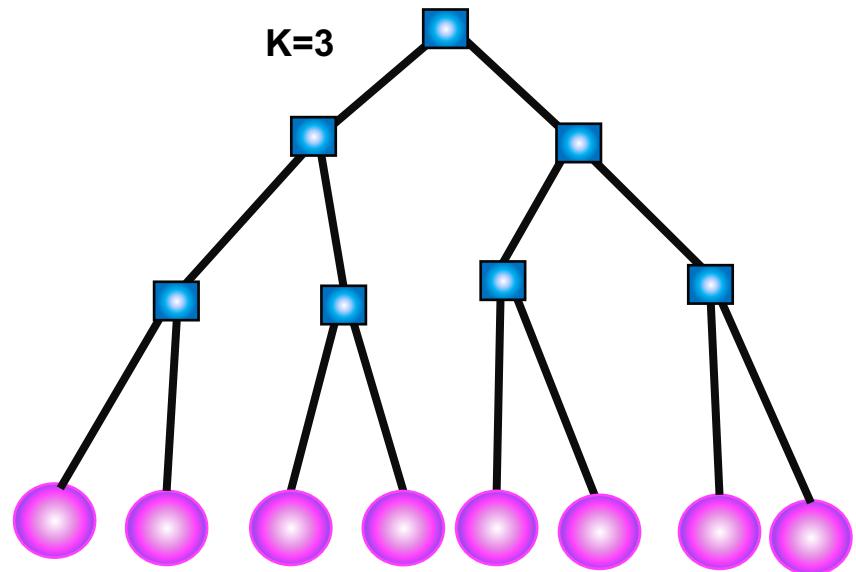
c. 8+8 hypercube.

Bisection: 8, Diameter: 4, generalization $N/2, \log_2 N$

K-Binary tree

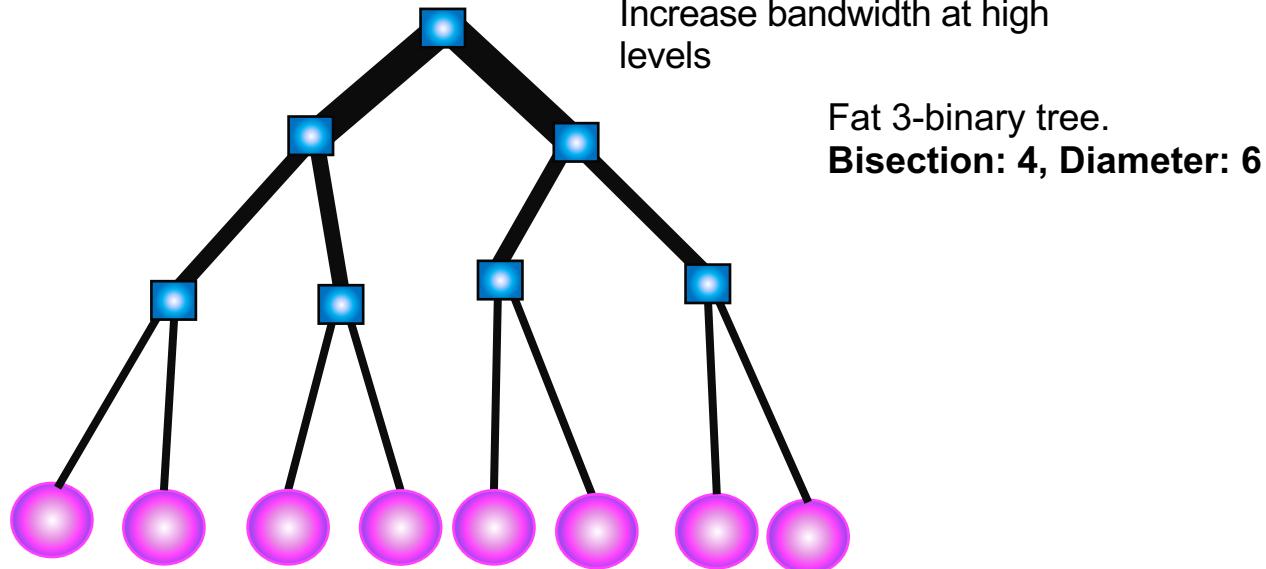
3-binary tree.

Bisection: 1 (constant) Diameter: 6, generalization 1 , $2\log_2 N$



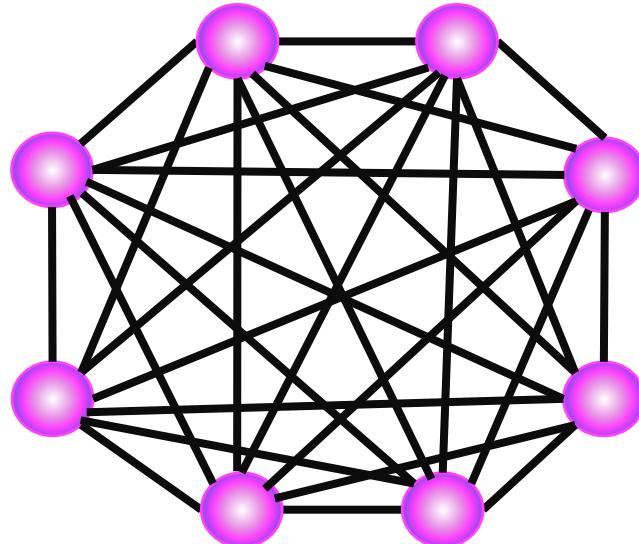
Topology: modern HPC

Fat tree



Topology: modern HPC

Dragonfly



Fully connected graph

Minimizes diameter and
maximises bisection

Diameter 1, Bisection 16

Lecture 2

Basic definitions and the
importance of the network

- Course practicalities repetition: 5 min
- Unbroken phone: key concepts recap (30 min)
- Exercise 2 as group work:
 - 4 topics (from sheet2 in GitLab)
 - Row-wise group work (10 mins work, 2 mins for discussing answers)
 - Synthesis (5 mins max)
- Signing up for points and feedback (2 mins)

Which one are you now?



I am still a bit lost



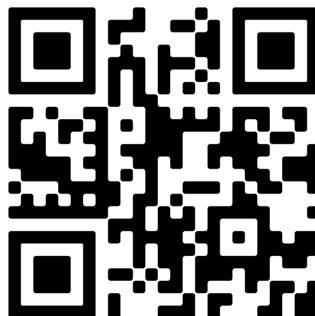
I understood
more, but still
would have some
questions



I understood
much more



Great
work!



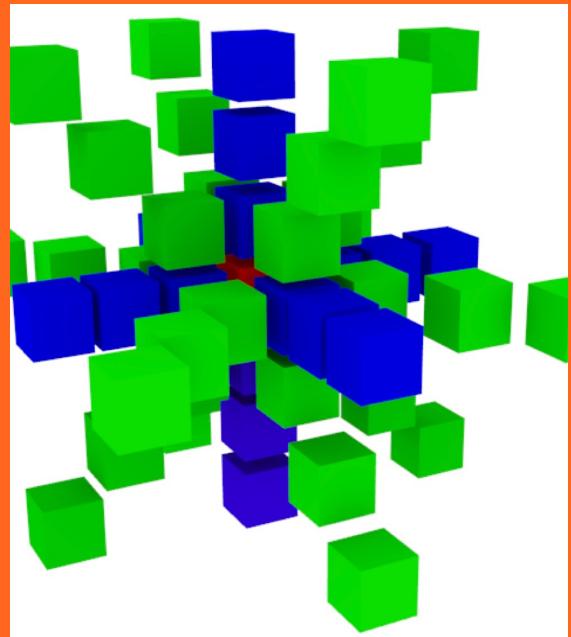
SCAN ME

CS-E4690 – Programming Parallel Supercomputers

Taxonomies and definitions

Maarit Korpi-Lagg

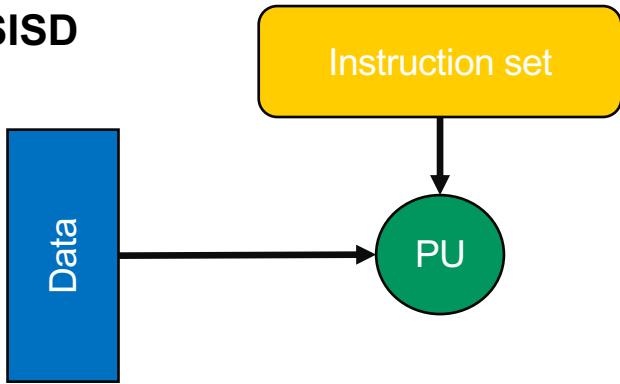
maarit.korpi-lagg@aalto.fi



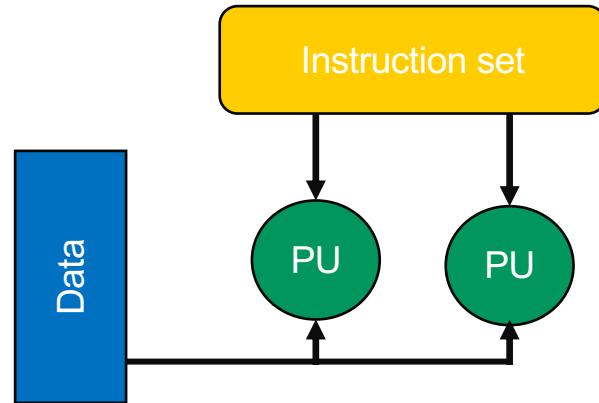
Aalto University
School of Science

Flynn's taxonomy

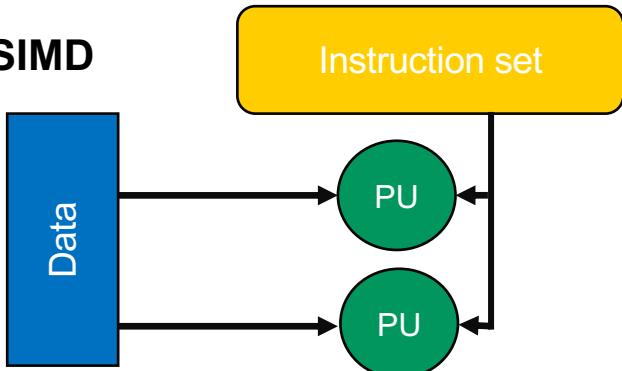
SISD



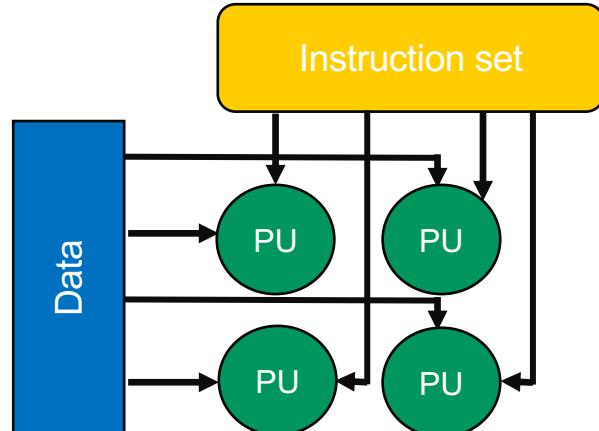
MISD



SIMD

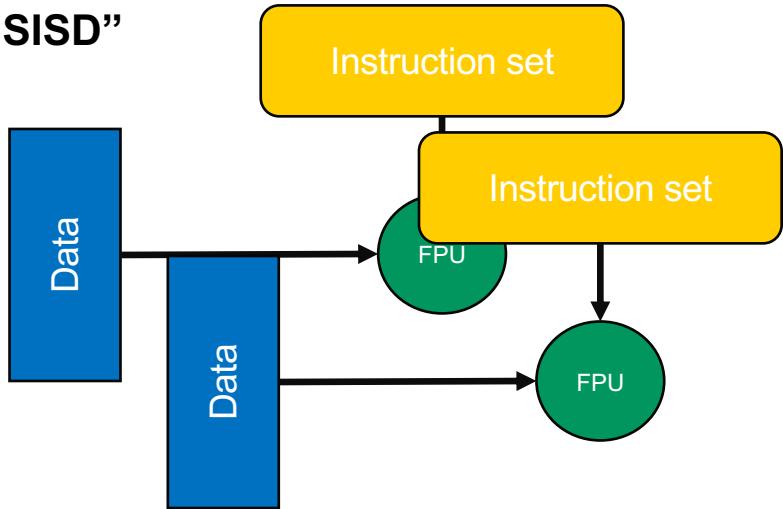


MIMD



Instruction level parallelism

“SISD”

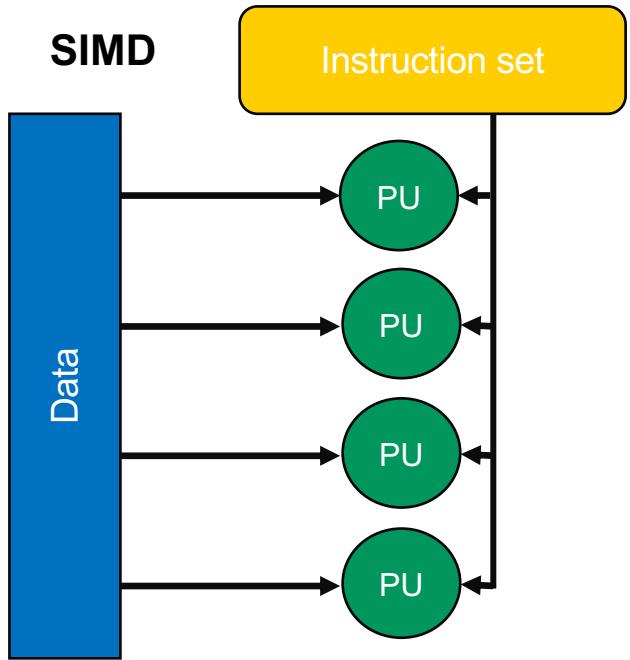


**Low level operational model
of a modern CPU**

**SISD: Scalar vs. superscalar
processor**

Pipelining: MISD-like setup

Data parallelism

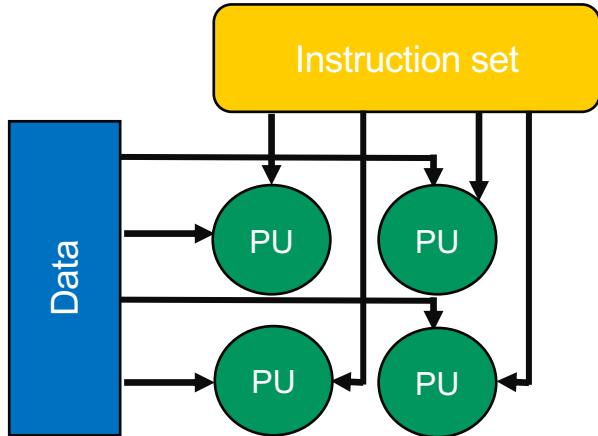


Vectorization

SIMT

Task parallelism

MIMD



**Top level operational
model of modern
supercomputer
application**

Adding concurrency

SPMD

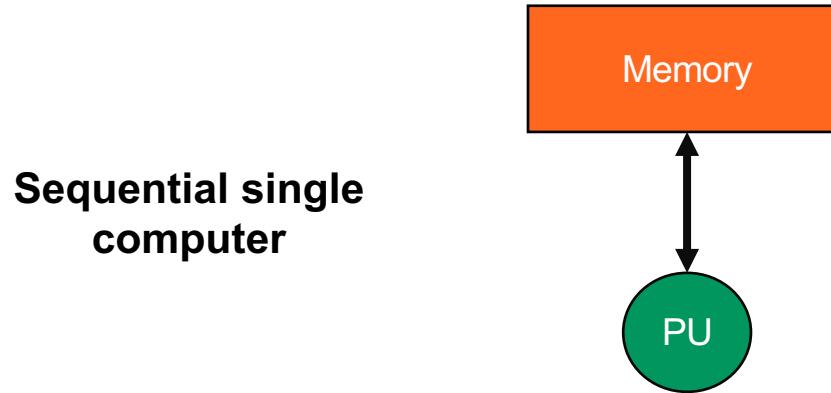
Modern hybrid architectures

Combining all Flynn's taxonomies in
a way or another

How to build
applications for
such systems?

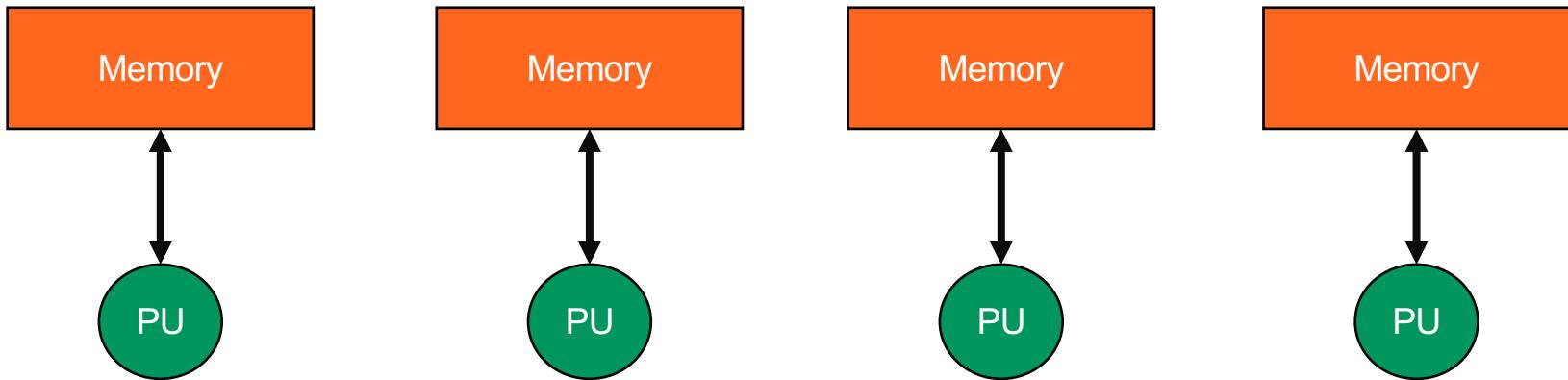
[1] A recent review of processor types used in HPC infras for those who are interested.

Memory access taxonomy



Memory access taxonomy

An array of (sequential) computers



Full task parallelism, no communication

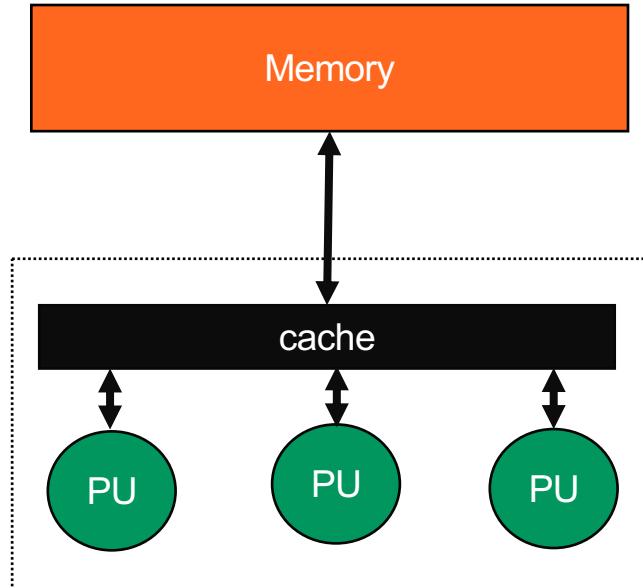
Coarse-grained

HTC

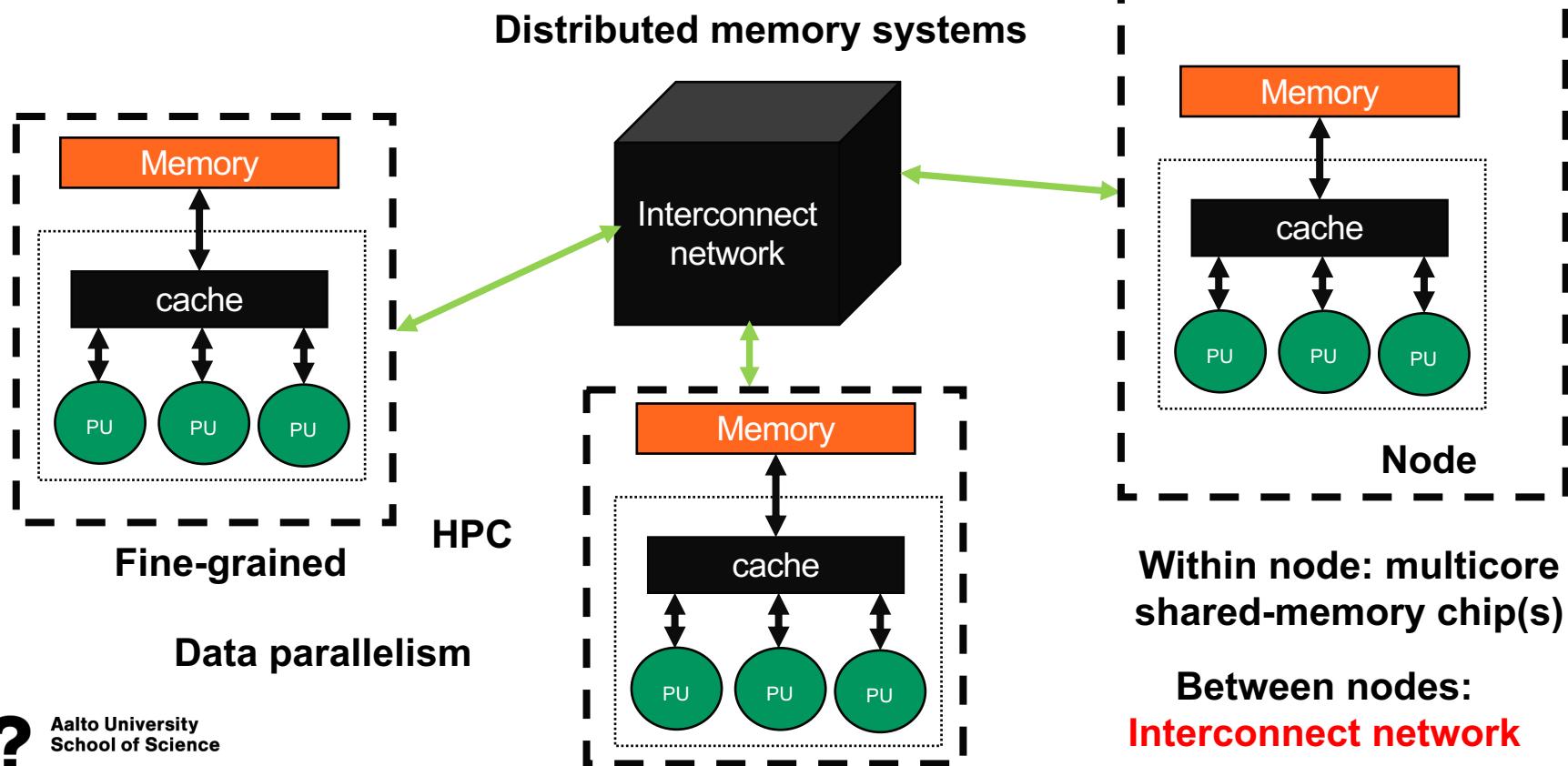
Embarassingly
parallel

Memory access taxonomy

Multicore shared-memory chips



Memory access taxonomy



Interconnect

Infiniband protocol and connector technology; Ethernet protocol.

Important properties affecting the performance (global bandwidth and latency)

- **Topology** (How are the links in between compute nodes organized; who can connect to who, through whom)
- **Connection type** (How is the processing unit connected to the network)

Topology

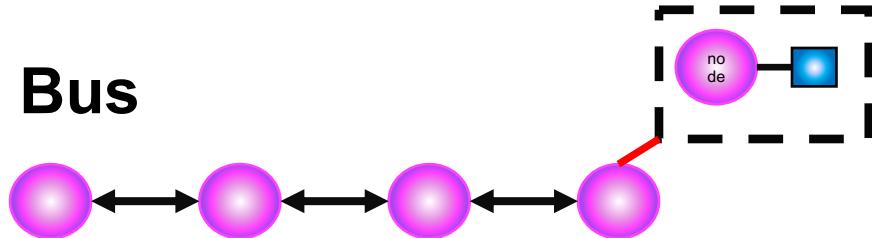
Physical networks + virtual mappings of the processes when designing a parallel program

- **Degree:** number of links from a node
- **Diameter:** maximum number of links between nodes **over a path with minimal distance** (**worst case routing distance**)
- **Average distance:** number of links to a random node
- **Bisection:** minimum number of links that divide the network into two equal halves (**can estimate worst case bandwidth**)
- **Bisection bandwidth:** **Bisection x link bandwidth**

Minimize diameter, maximize bisection

Topology: examples

Bus

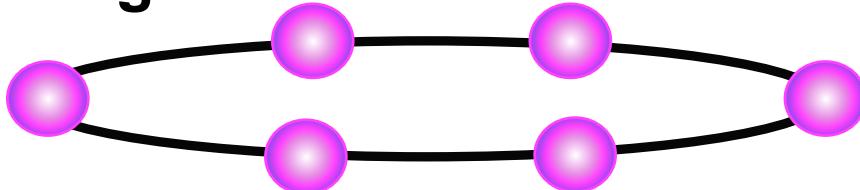


Diameter: $3 (n-1)$

Bisection: 1

Number of switches: n

Ring



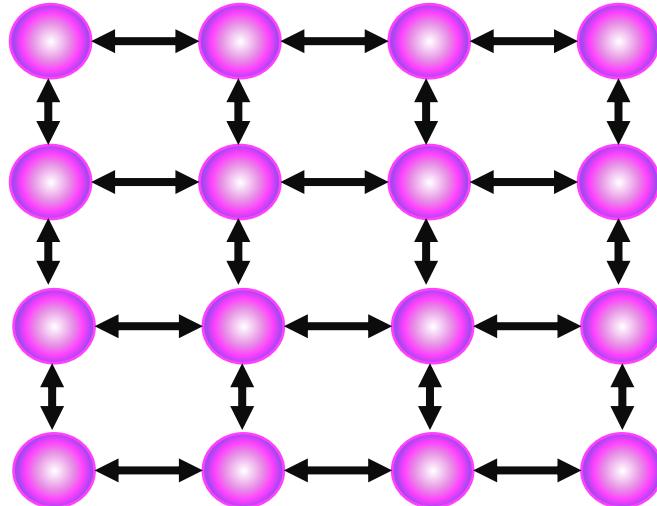
Diameter: $3 (n/2)$

Bisection: 2

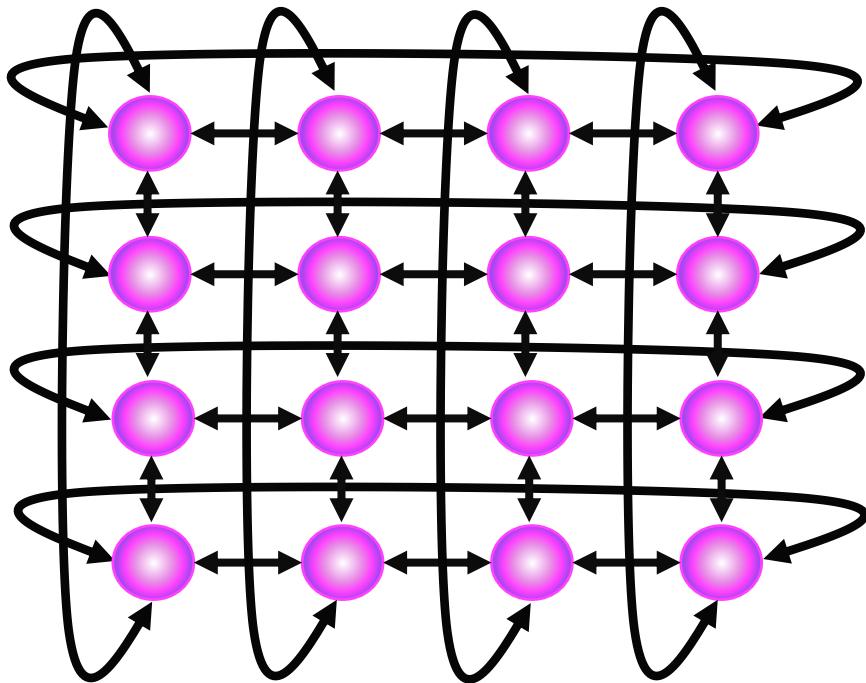
Number of switches: n

Topology: examples

2D mesh

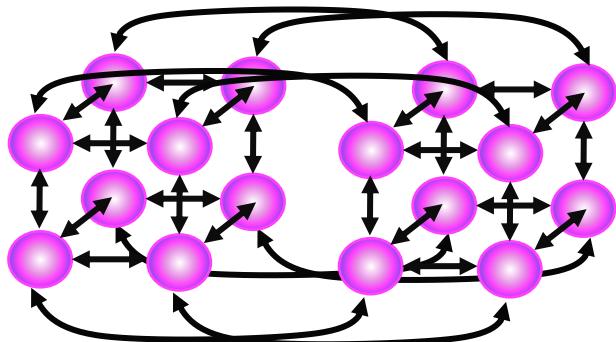


2D torus

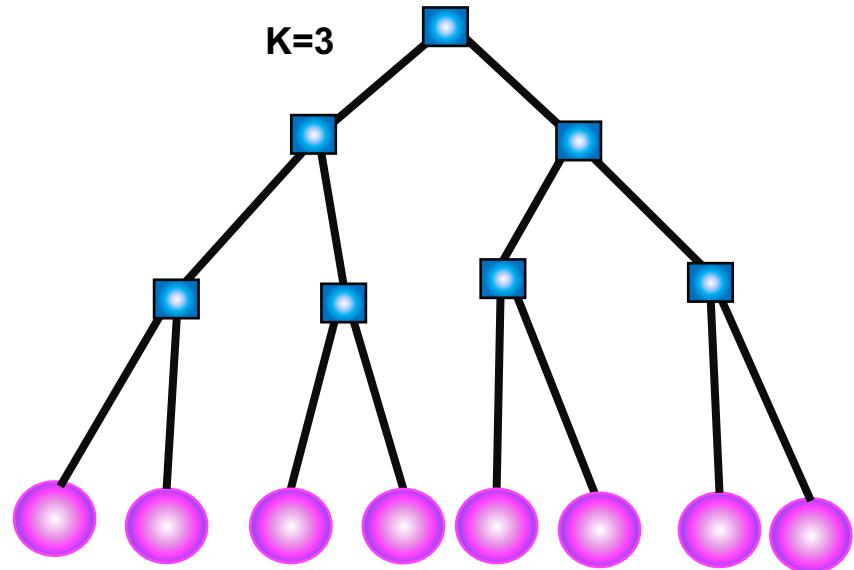


Topology: examples

“8+8” Hypercube



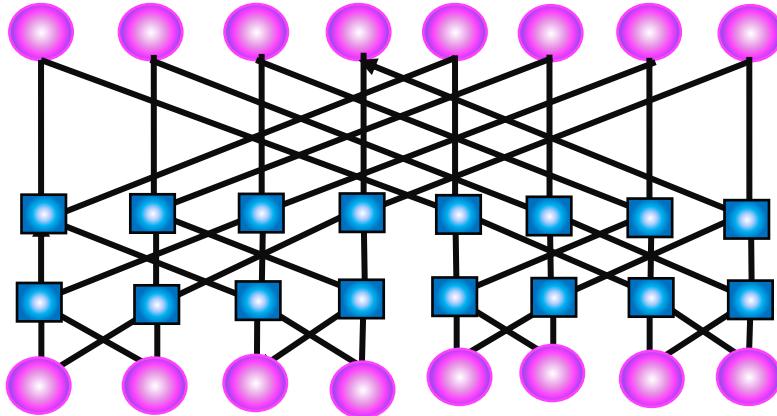
K-Binary tree



To improve further, multilevel networks

Topology: examples

Butterfly



Diameter: 3 ($\ln(n)-1$)

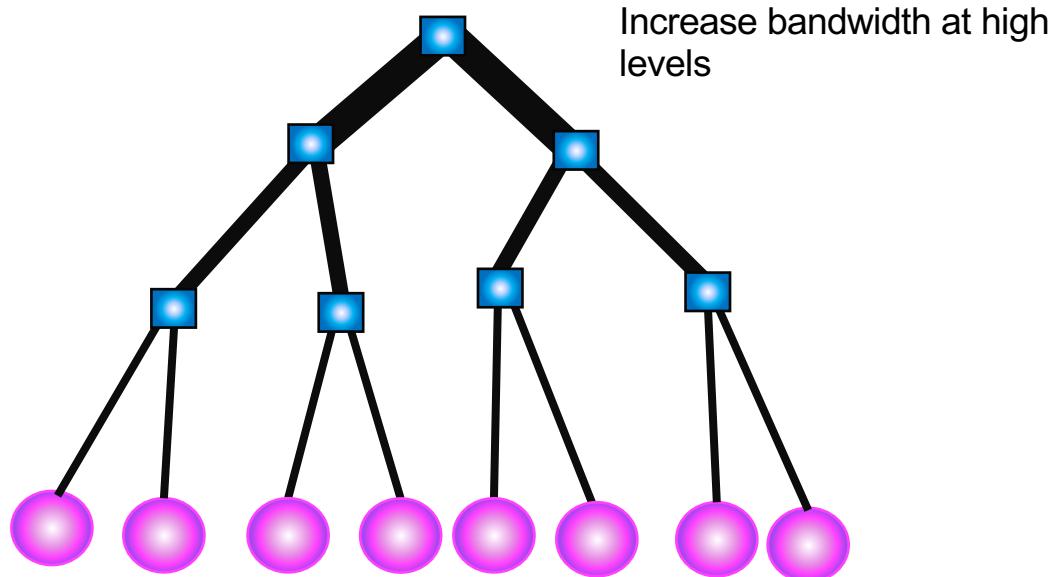
Bisection: 8 ($n/2$)

Number of switches: 32
($n*(\ln(n)+1)$)

Cost: the more links and switches (large hop count), the more resources it takes to build and operate

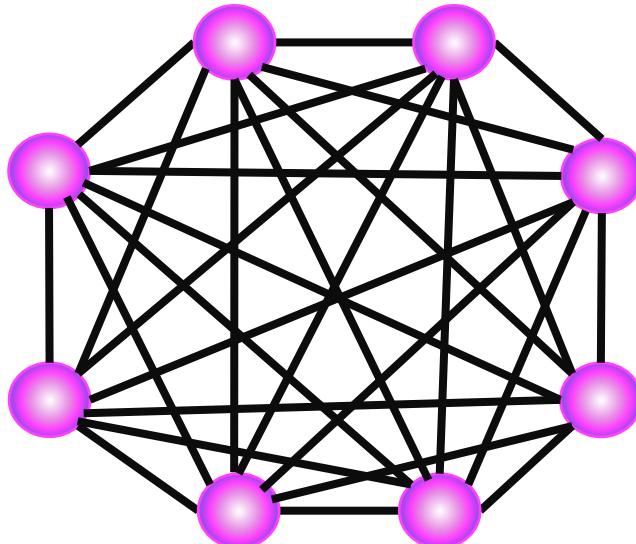
Topology: modern HPC

Fat tree



Topology: modern HPC

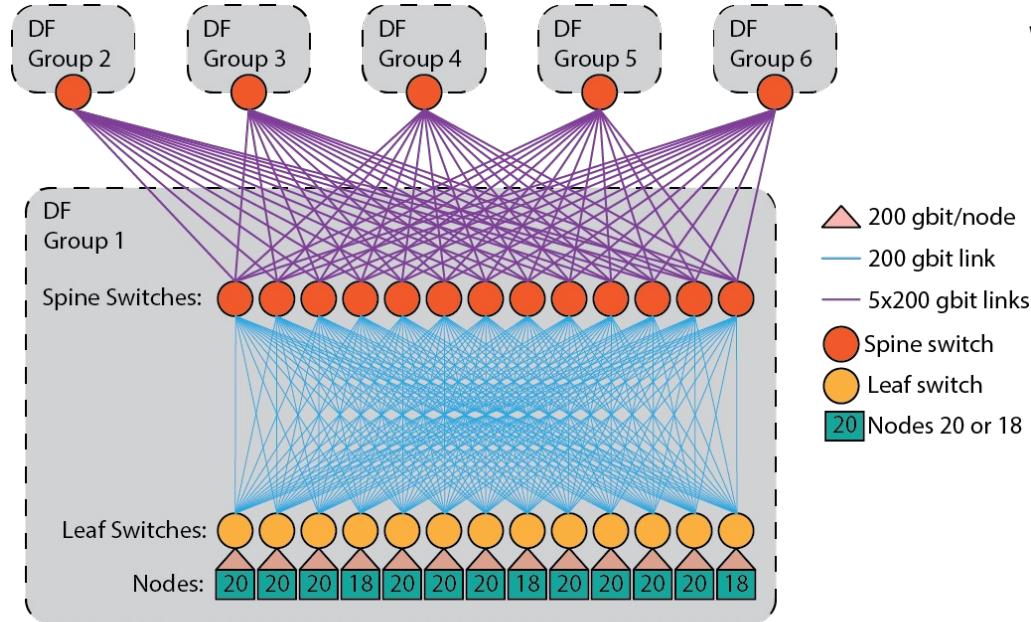
Dragonfly



Fully connected graph

Minimizes diameter and
maximises bisection

Topology: modern HPC



What would then this be?

Mahti@CSC, image credit CSC

Interconnect

Summary of current HPC interconnects

- **Topology** (Dragonfly, fat tree, torus, all sorts of combinations at multiple layers)
- **Connection type** (Currently most networks are multi-level, switches can handle an order of hundred of ports)
- **Latency** (1-2microsecs)
- **Bandwidth** (Nowadays around 100Gbit/s-200Gbit/s achieved through multiple lanes)

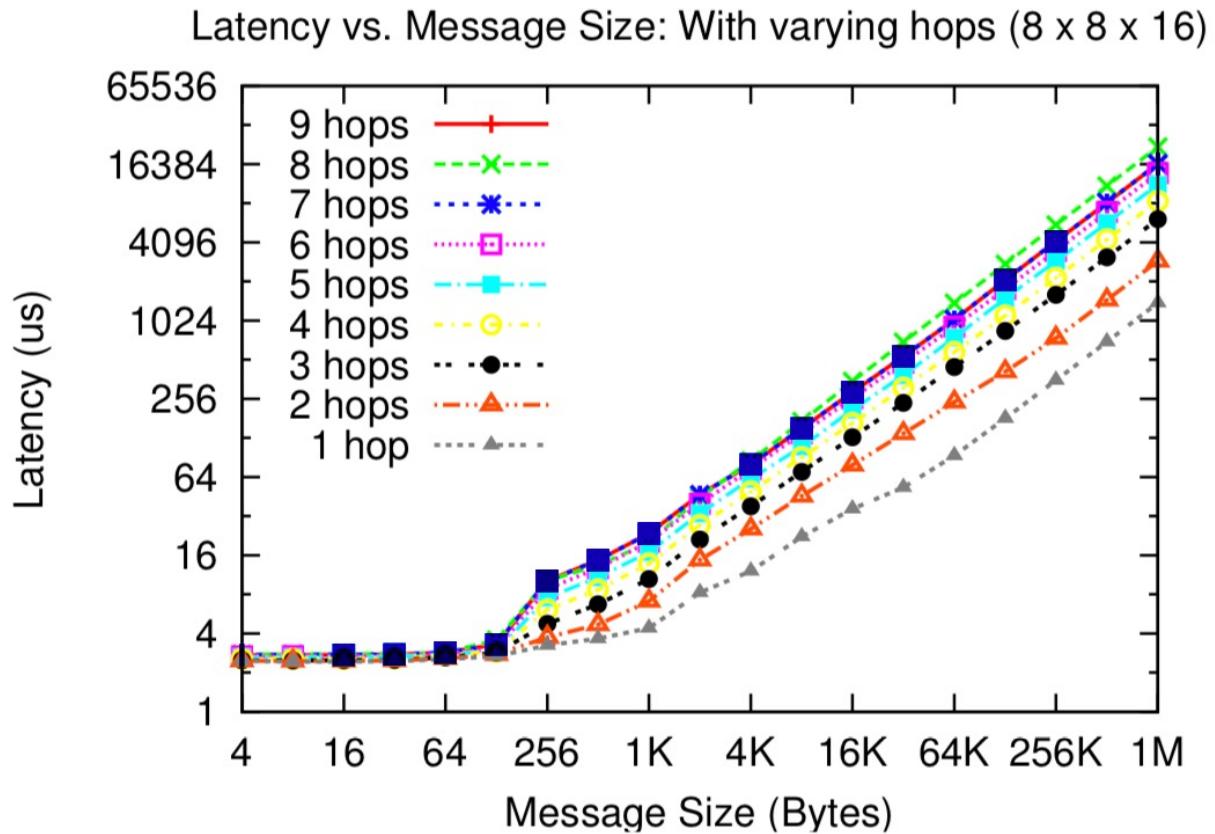
[1] A recent review of interconnect status for those who are interested.

Interconnect

How much (as a user) do you care about the interconnect topology?

- Thanks to libraries such as MPI, not much. Why? Should one?
- Jobs are usually small in comparison to the scale of the system (fit into a chunk or island).
- On this local scale, the job schedulers do a good job.
- Larger simulations challenge all this, and we are heading towards exascale computing.
- Hence, learning about topologies is not in vain!

**Bhatelé & Kalé
(2009)** IBM BG/P
using messages
between
equidistant pairs



Performance models without interconnect: RAM and PRAM

- If there was only shared memory...

- $T_{RAM} = N_C + N_M$

N_C the number of instructions completed

N_M the number of loads/stores from/to the memory

- Communication to other distant nodes is not an issue
- For sequential algorithms, still valid, but requires extensions to take into account multi-level caches.
- PRAM an extension to multiple processors

Latency-bandwidth performance model ($\alpha\beta$, used since the 1970's)

$$T_{LB} = \alpha + n(\beta + \gamma);$$

α = Latency (start up cost of communication)

β = Time cost per unit message length sent (bandwidth cost)

γ = Time consumed in actual computation

n = Message length

- Both receiver and sender block
- No multiple messages allowed
- Does not allow for overlap (concurrency) in communication and computation.

BSP model (Valiant et al. 1990)

- Bulk synchronous applications that perform **supersteps**

$$T_{BSP} = \max_{i=1}^p (\omega_i) + \max_{i=1}^p (gnh_i) + l$$

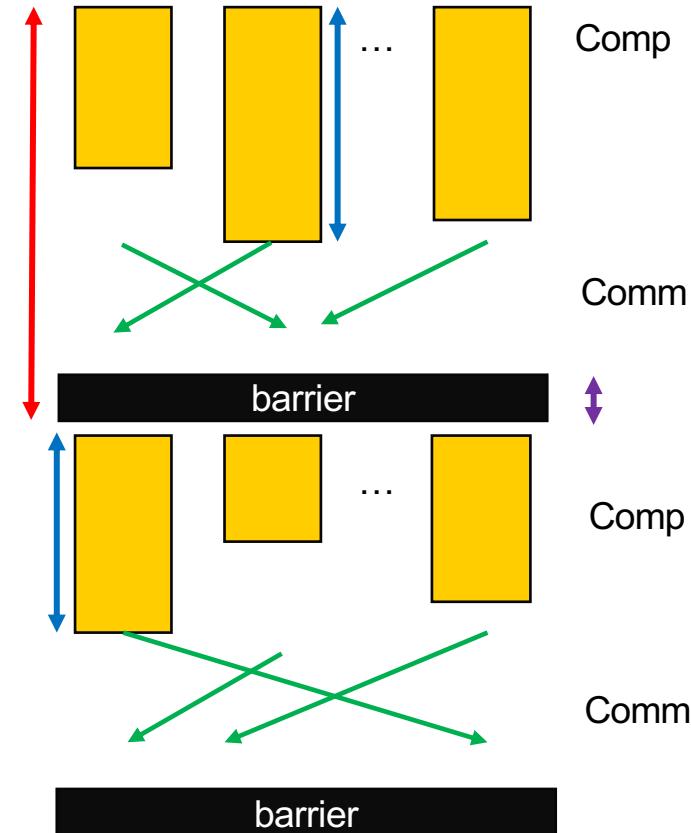
p number of processors

h number of messages, n their length

g bandwidth throughput

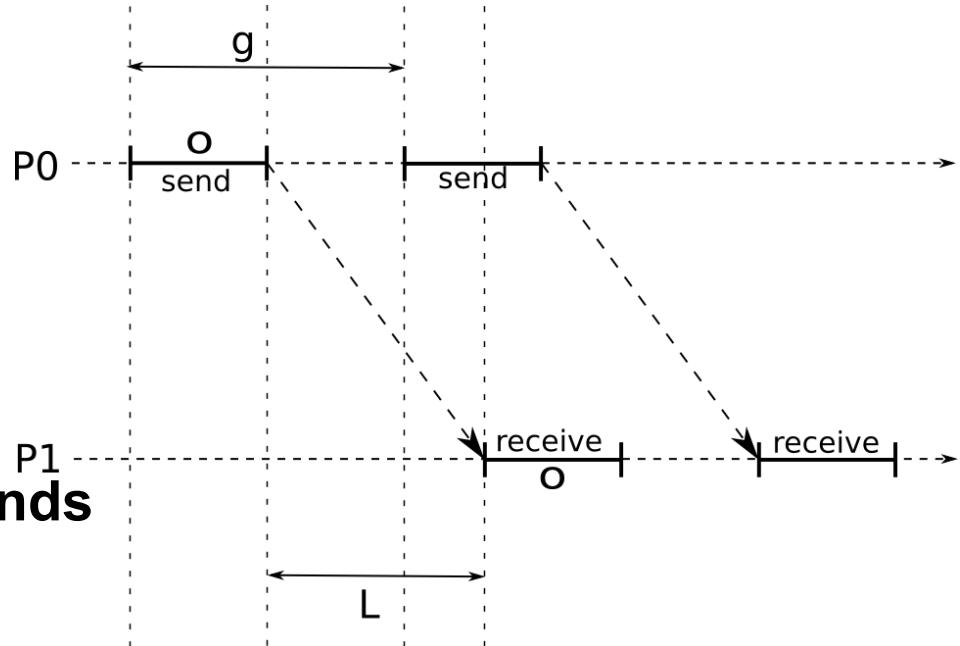
l barrier cost

- Topology is not accounted for
- Has many applications, including MapReduce.



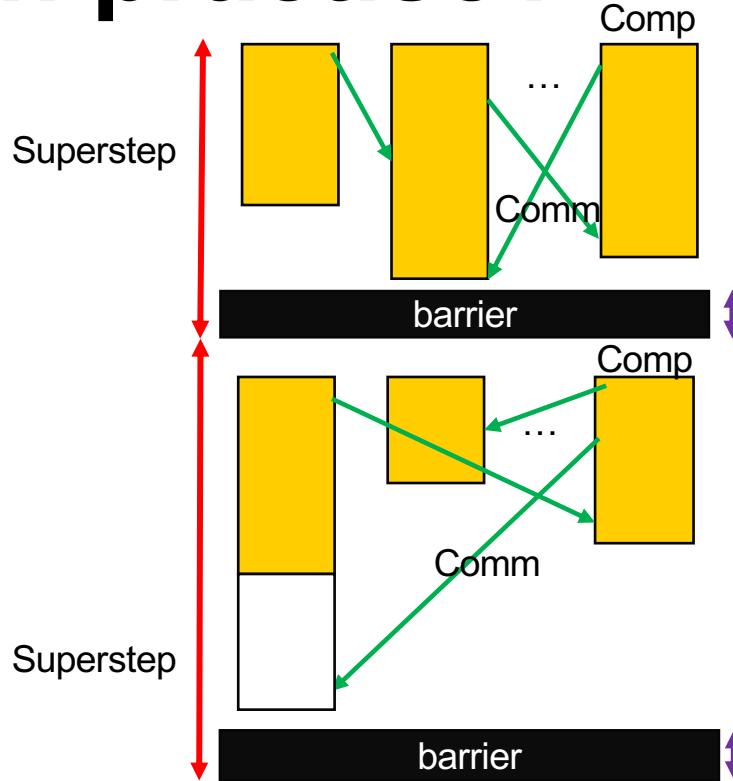
LogP model (Culler et al. 1993)

- **Asynchronous messages**
 - $T_{logP} = 2o + L + (n - 1)g$
- n is the data size
- o is the overhead (processor)
- L is the network latency
- $g \geq o$ is the gap between two sends or receives



What is required in practise?

Ideal situation: all communication is overlapped with computations.



Not-wanted situation: communication is not totally overlapped with computations.

Asynchronous communication-compute performance model for real applications (ACC) [3]

In reality, the “gap” does not only come from “preparations” to communicate, but from actual computations taking quite some time. These can be overlapped with comms.

Let us denote **latency of computations** as τ_W , when performing W operations on data items, and that of **communications** as τ_Q , when communicating Q data items. **π is the operational capability of the hardware as data item updates per second, and β the rate at which data items can be communicated.** The latencies of computation and communication are therefore, $\tau_W = W/\pi$ and $\tau_Q = Q/\beta$ respectively. Let us further assume that there is a portion of computations that cannot be overlapped (made concurrent) with the communication; let this fraction be τ_0 . The total latency is determined by the non-concurrent parts of the code

$$\tau = \max(\tau_W, \tau_Q) + \tau_0$$

ACC model cntd

Trivial limit: τ_0 dominates (rather a rare condition).

When τ_0 is small, there are two interesting limits:

- 1) When there is a lot to compute, and little to communicate, $\tau_W > \tau_Q$, **we are in the compute-bound limit.**
- 2) When there is a lot to communicate, and little to compute, $\tau_W < \tau_Q$, we are in the **communication-bound limit (where all the other performance issues of memory exchange/interconnect kick in)**.

Goal: always to stay in limit 1), never run an application in limit 2).

Amdahl's law

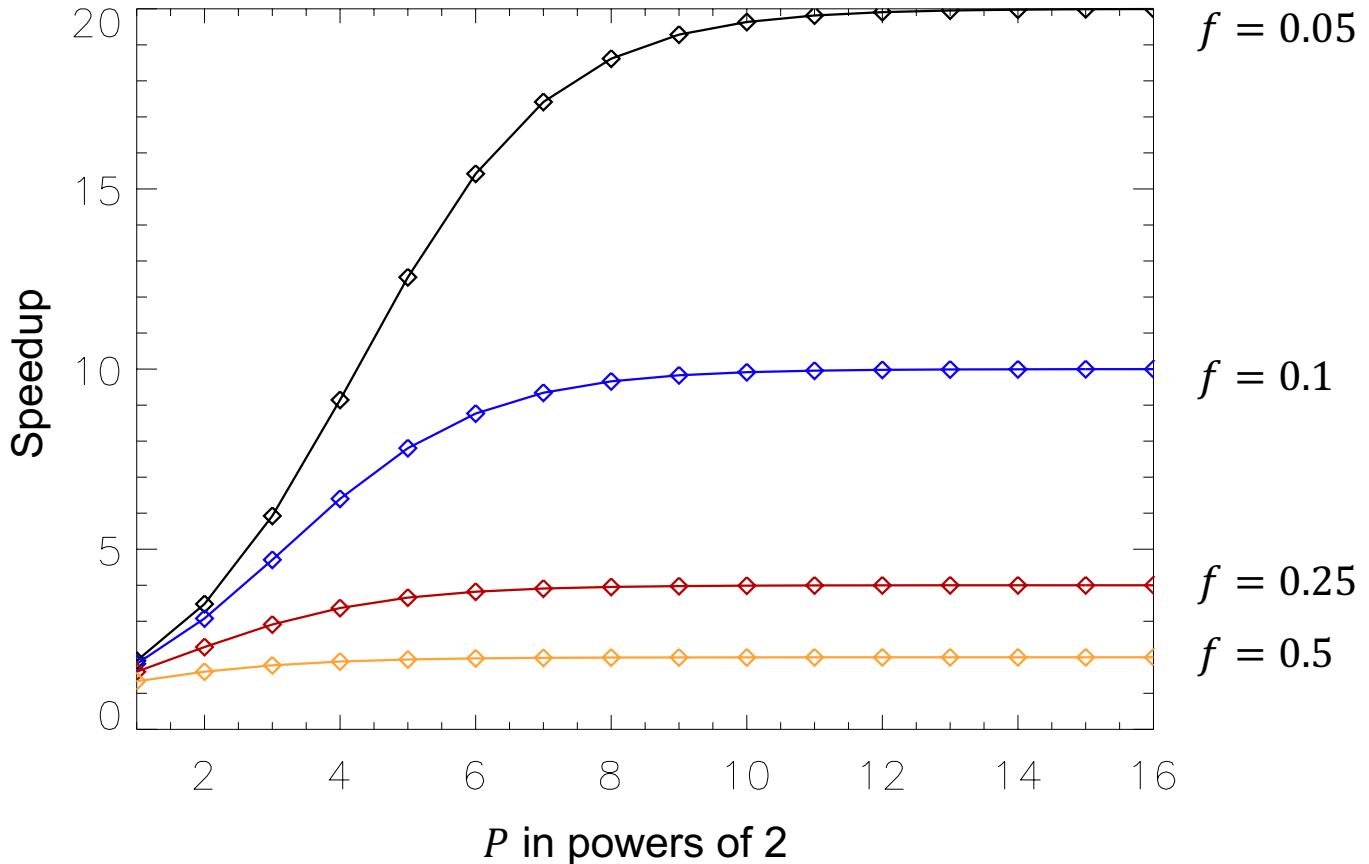
One way of determining speedup from parallelism:

Let f be the fraction of computations performed sequentially. The fraction that can be done in parallel is $(1 - f)$. The time that it takes, for a fixed problem size, to execute all the computations with P processing units is then

$$T_P = f + \frac{(1 - f)}{P}$$

The speedup is $\frac{T_1}{T_P} = \frac{1}{f + (1-f)/P} < \frac{1}{f}$; limited by the sequential part

Amdahl's law



Strong scaling

- The **problem size is kept fixed**, the number of PUs is increased, and the execution time is monitored.
- Due to the serial part of the code, the ideal linear scaling (when you double PUs, the computing time halves) saturates to a level $\frac{1}{f}$, after which increasing PUs does not make sense.
- Follows from Amdahl's law.
- Strong scaling to large number of PUs is very challenging to retain.

Gustafson's law

Another viewpoint on speedup from parallelism:

If the workload is growing, the serial fraction of the program does not limit speedup if the parallel part scales well. The speedup can be written as the ratio of the workloads

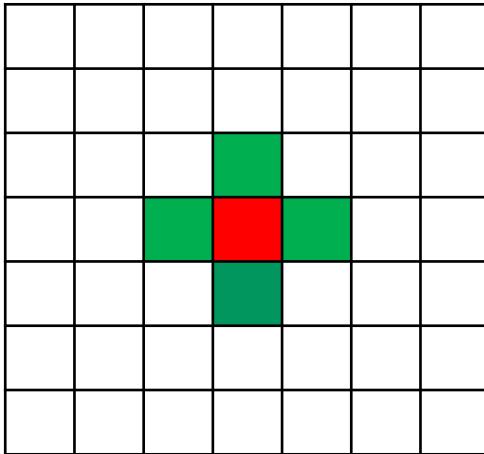
$$\frac{W_P}{w_1} = f + (1 - f)P.$$

Weak scaling

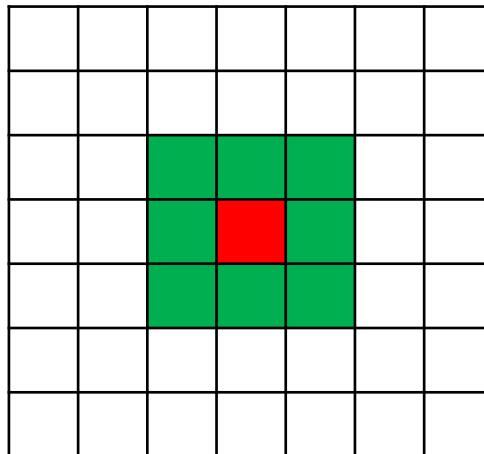
- Follows from Gustafson's law.
- The problem size is not kept constant, but increased so that each PU has a constant workload, and one monitors the execution time.
- Perfect scaling up to many more PUs is easier to maintain.
- Usually both strong and weak scalings are required to be shown when testing the parallel performance of a code.

Stencils

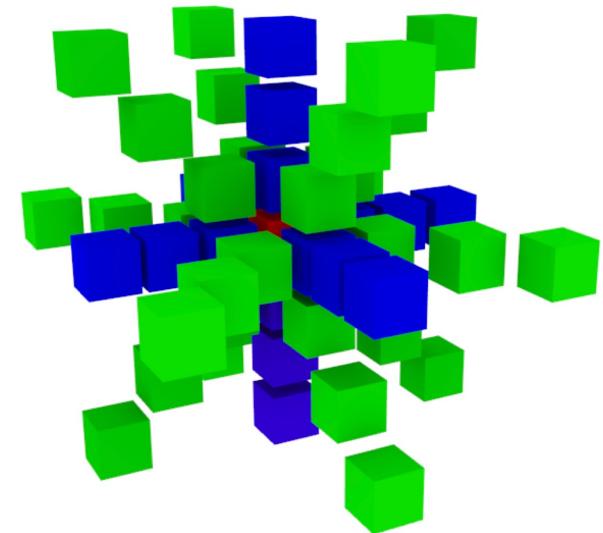
- Recurring update patterns (iterative stencil loops, ISL) of arrays
- Nearly everywhere, can be of any shape and complication



2D von Neumann



2D Moore

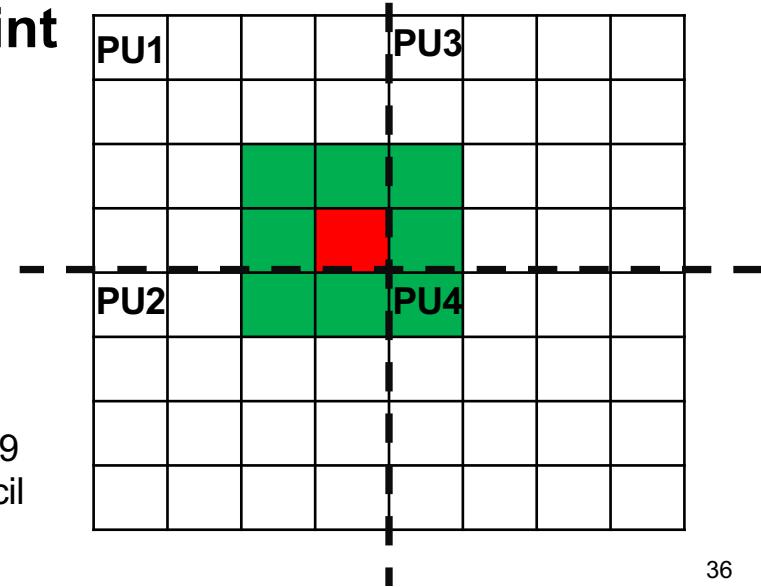


3D 55-point stencil

Stencils

- **Order** of stencil: how many neighbors in a certain cardinal direction it needs for updating the central point. (Not useful for asymmetric stencils)
- **N-point stencil:** N is the total number of points (including itself) it needs for updating the central point
- Challenge parallelism:
Tasks are not independent, as they need data from other PUs to complete the update

2nd order 9 point stencil

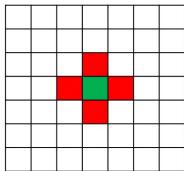


ISL: Halo exchange

The points that need to be exchanged form the so called **halo**.

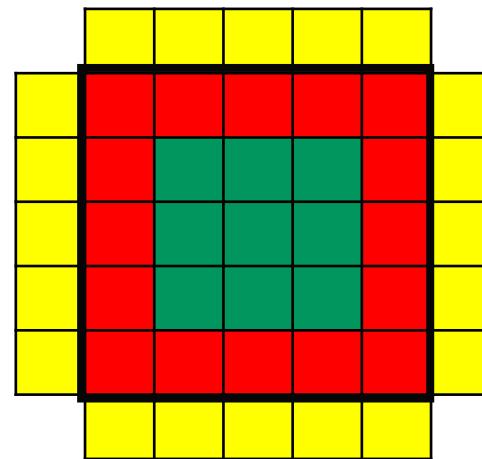
The size of the halo depends on the order and structure of the stencil.

Example: 2nd order, 2D von Neumann stencil, 5x5 computational subdomain



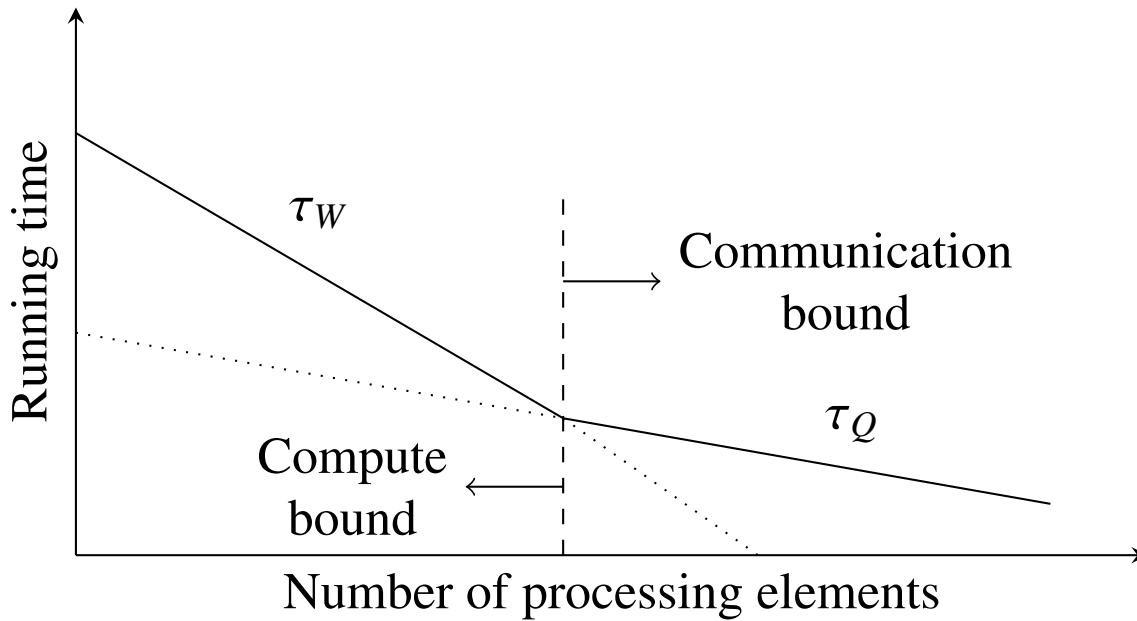
Red cells: **edge** points that can be updated when the halo points are received

Yellow cells: **halo** points required from neighbors

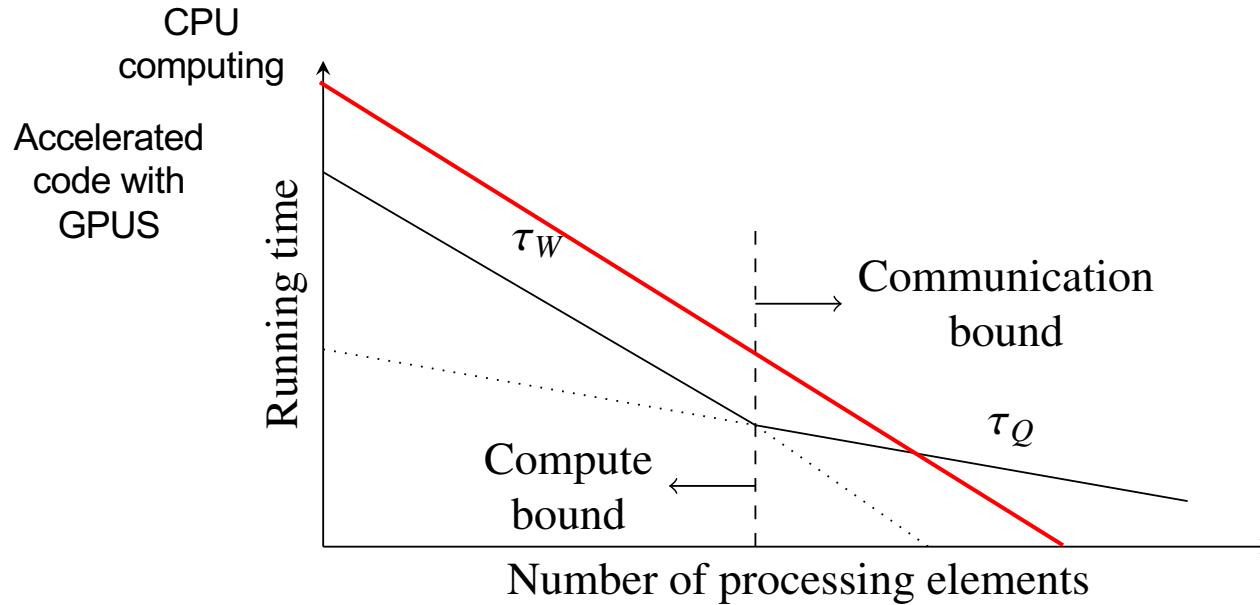


Green cells:
Interior points
that can be
updated without
data exchange

ACC model [3]: example curve for high-order ISLs.



ACC model [3]: example curve for high-order ISLs.



Core reading

- [1] Tekin et al., “State-of-the-Art and Trends for Computing and Interconnect Network Solutions for HPC and AI”, Prace publications.
- [2] Zhang, Y., Chen, G., Sun, G., and Miao, Q. (2007). Models of parallel computation: A survey and classification. *Frontiers of Computer Science in China*, 1:156–165.
- [3] Pekkilä, J. et al. "Scalable communication for high-order stencil computations using CUDA-aware MPI", *Parallel Computing*. 111, 102904.
<https://doi.org/10.1016/j.parco.2022.102904>

Extra reading

- Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.
- Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. (1993). LogP: towards a realistic model of parallel computation. Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, 28(7):1–12.
- Bhatelé and Kalé (2009), Quantifying network contention on large parallel machines, Parallel Processing Letters, 19(4),
<https://doi.org/10.1142/S0129626409000419>
- Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113.
- Pace, M. F., (2012), “BSP vs. MapReduce”, Procedia Computer Science, 9, 246-255, doi:10.1016/j.procs.2012.04.026

CS-E4690 – Programming Parallel Supercomputers

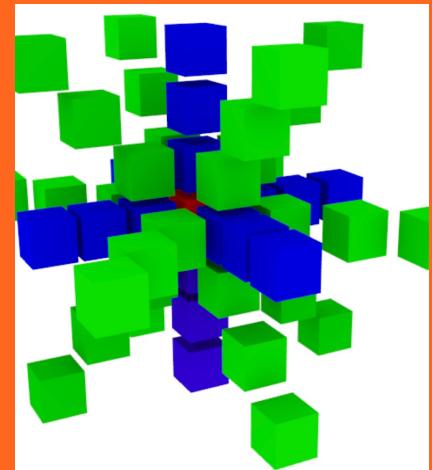
Basics of message passing interface (MPI)

Maarit Korpi-Lagg

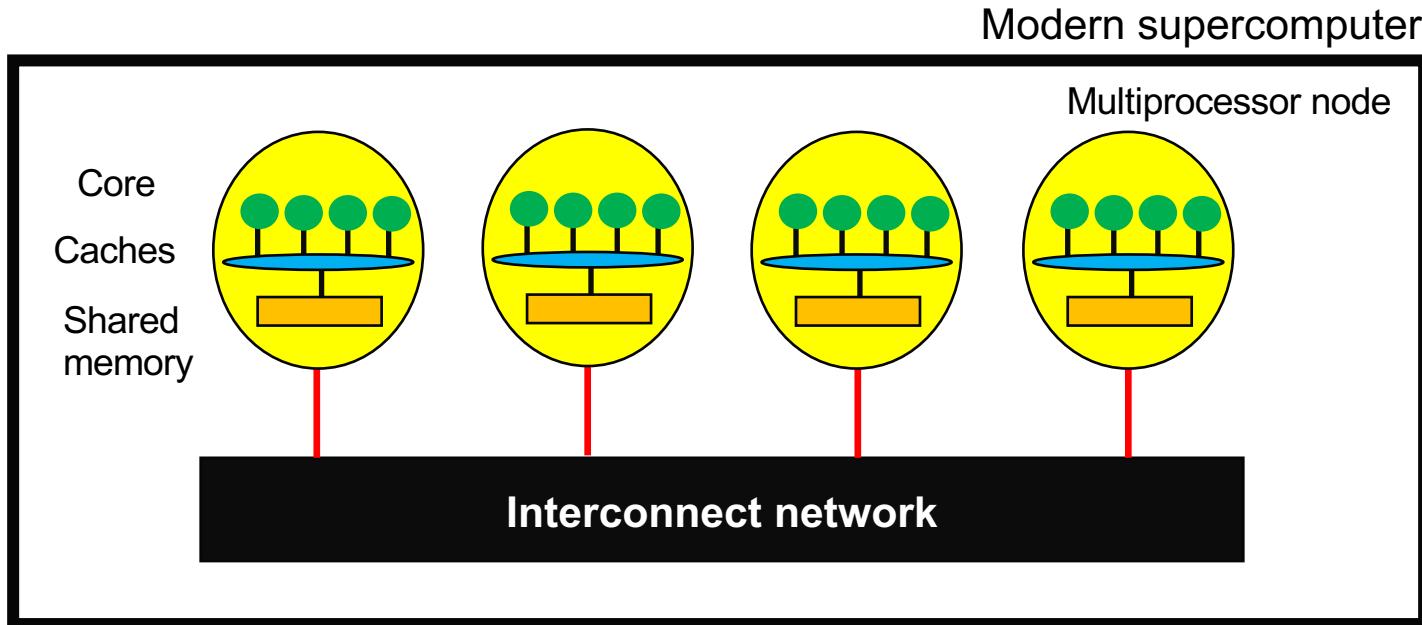
maarit.korpi-lagg@aalto.fi



Aalto University
School of Science



Recap of the situation



Current “software” landscape

- MPI (developed since 1991, standardized in 1994, now at MPI-3, MPI-4 soon coming): several implementations - OpenMPI, MPICH, MPICH...
 - Libraries that provide message passing functions
 - API to provide bindings to higher-level programming languages (Co-array Fortran, ..., Python, R, Matlab, Java/Scala, Julia, Chapel, ...)
- Big data programming models: MapReduce; Hadoop, Spark, ...
 - Instead of (only) passing messages, a distributed file system providing data locality is used

Low or high-level programming?

MPI:

- Low level, difficult to program
 - Fault tolerance is left to the user to take care about
 - Available and supported at every HPC center
 - Standardized
- During this course we use MPI

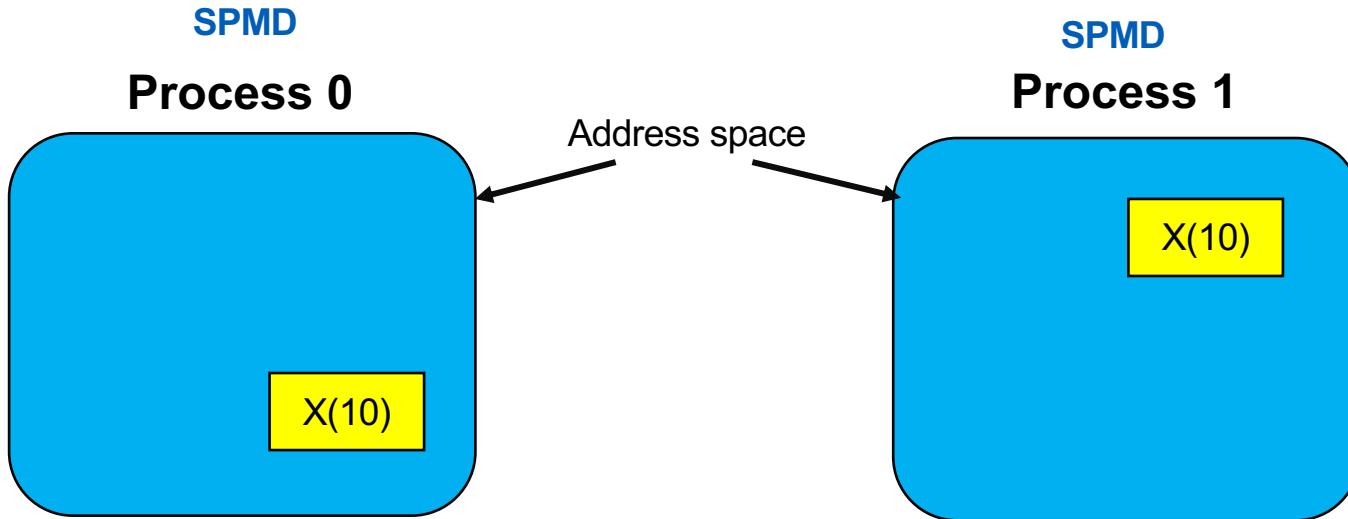
Higher-level languages:

- Easier to program
- Fault tolerance might be readily implemented
- Might not be provided everywhere
- You do not have to so much care,
but also do not learn, about the internal
workings of the distributed programming model

How to decide in practise?

1. I am lacking understanding of distributed memory programming, and will find the easiest way out with the high-level programming languages.
2. What is available in the system accessible for you now/near future?
3. I want to write portable code, and parallelize it only once, and keep on maintaining it with minimal effort

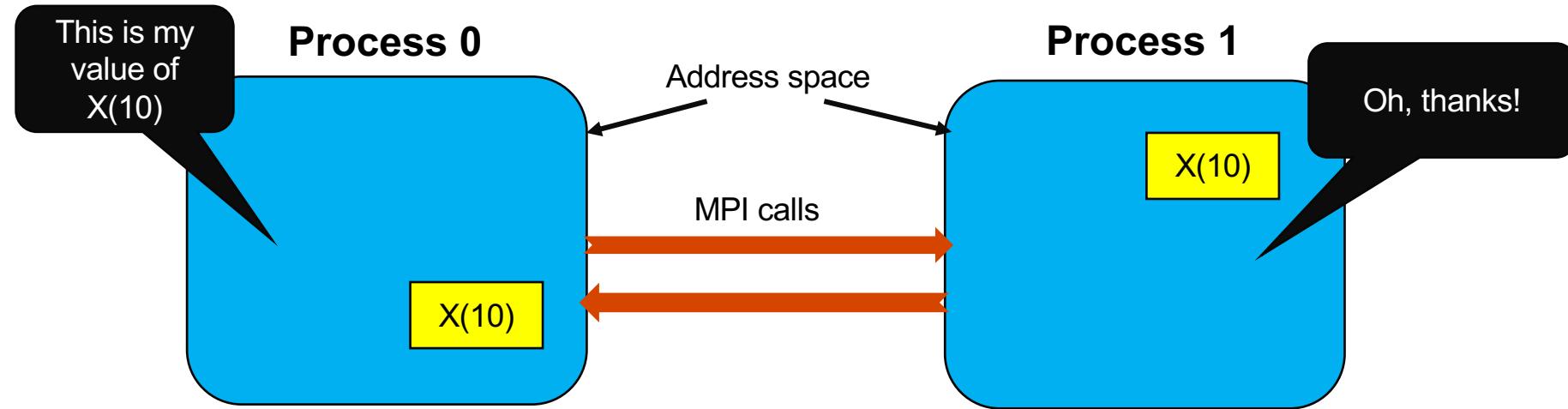
Distributed memory programming model



```
int X(100); ...
X(10)=my_rank*10;
printf("My value at index 10 %d",X(10));
```

```
int X(100); ...
X(10)=my_rank*10;
printf("My value at index 10 %d",X(10));
```

Distributed memory programming model

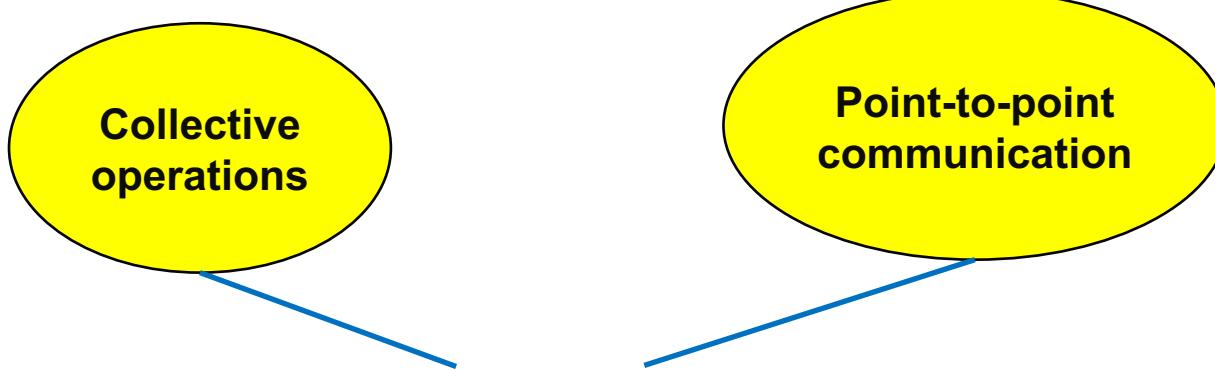


```
int X(100); ...
X(10)=my_rank*10;
printf("My value at index 10 %d",X(10));
```

```
int X(100); ...
X(10)=my_rank*10;
printf("My value at index 10 %d",X(10));
```

Fundamental idea

MPI libraries implement a message passing model, in which the **sending and receiving of messages combines both data movement and synchronization**. Processes have separate address spaces.

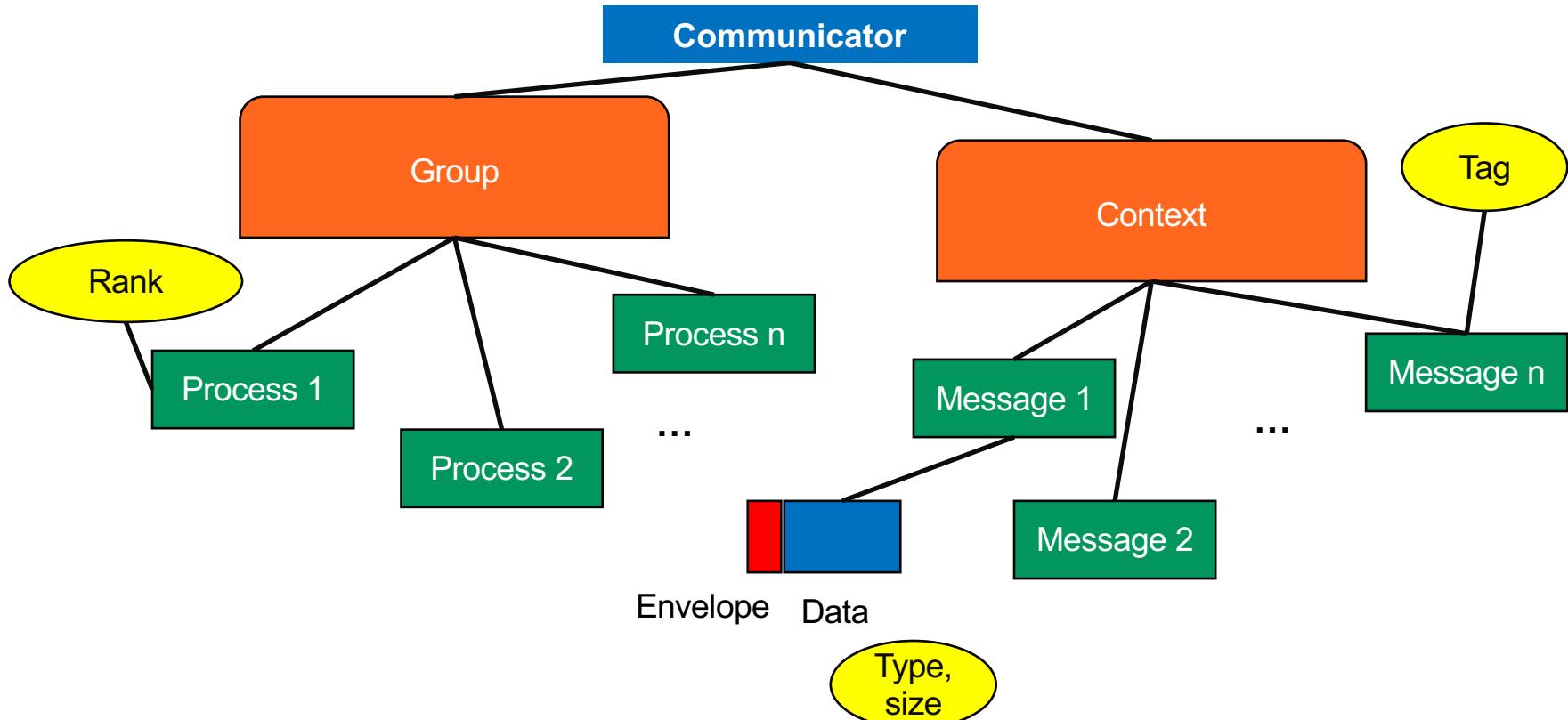


Two high-level modes of operation; during this lecture, we start with point-to-point

But, how to arrange

- How many others are there, and where amongst them am I?
- Identification of **sender** and **receiver**
- Communication about what is going to be sent and received
(prescription of **data**)
- Identification of the **message** (which data belongs where), if many are constantly sent?
- What is supposed to happen when the transmission is **complete?**

Communicator (def. MPI_COMM_WORLD)



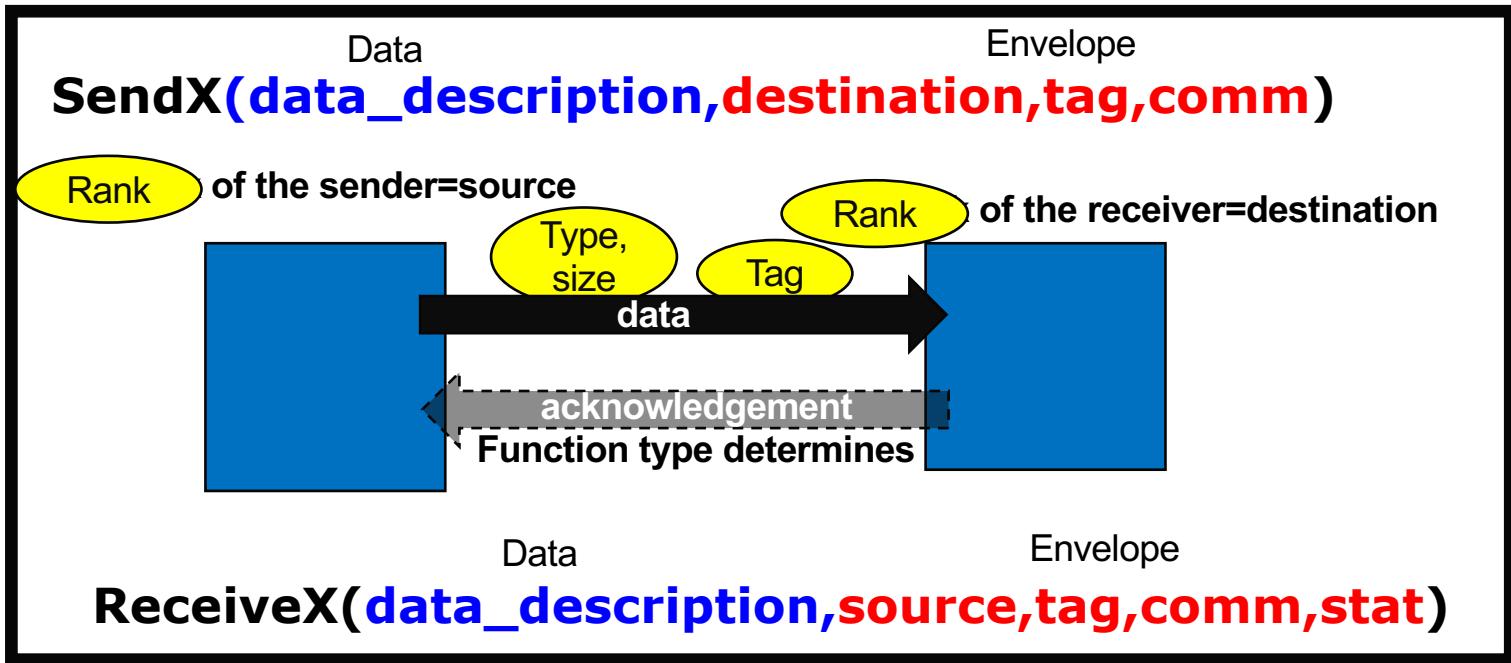
C code in practise

```
#include "mpi.h"
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init (&argc, &argv); /* Communicator set up */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("My rank %d of %d\n", rank, size);
    MPI_Finalize(); /* Communicator deallocated */
}
```

```
scripts/job_CPU_example.sh
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=2
```

More detailed functionality

Within 'comm' group of processes



Two operation modes

Point-to-point (P2P) communications

Collective communications

Co-operative communication

Lecture 3

Blocking

MPI_Send
MPI_Recv
MPI_SendRecv
MPI_Bsend
...

MPI_Isend
MPI_Irecv...

Non-blocking

MPI_BCast

Lecture 4

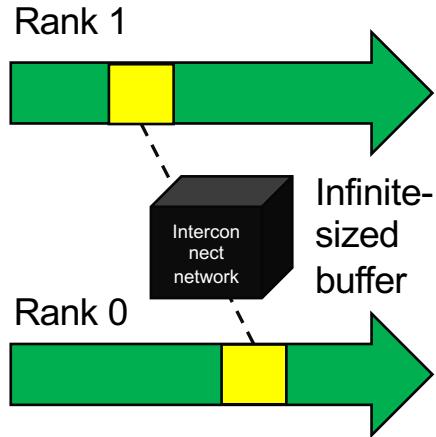
MPI_Scatter ...

One-sided communication (RMA ops)

MPI_Get
MPI_Put ...

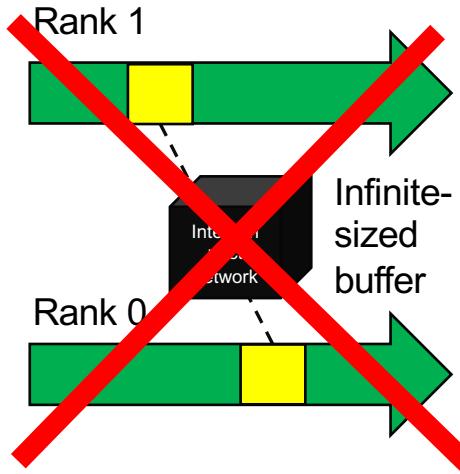
Lecture 4

Blocking communication



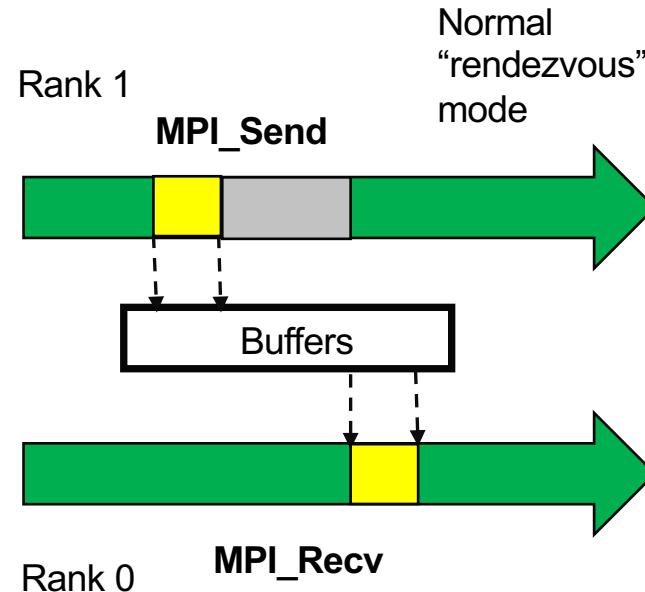
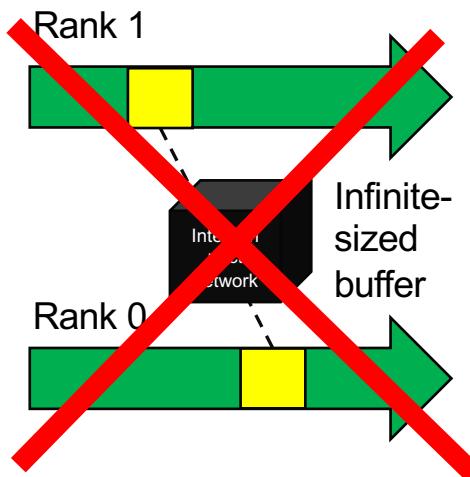
Yellow: communication
Green: computation
Grey: Idling

Blocking communication



Yellow: communication
Green: computation
Grey: Idling

Blocking communication



Sending call blocks until the receiving process has started.
Problem: If the receive cannot start for some reason, the system goes into a halt, called deadlock.

Blocking communication

- Exception: many MPI implementations optimize the non-blocking send with an **eager protocol** for short messages.
- The eager protocol keeps on sending the fully packed messages including the data and the envelope, assuming that the receiver can keep on receiving the full package.
- **Problem:** your code may work for with small system sizes, and deadlock with large system size.

Blocking communication

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

UNIQUE dest and tag

Push
communication
mechanism

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

Structure
containing source,
tag, error, and
length

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

Elementary data types

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char



Errors

- Virtually all function calls return an error. In C, the returned MPI function value is the error, 0 indicating success.
- Implementation specific; refer to the documentation of your MPI library
- If a MPI function call causes an error, it, as a thumb rule, aborts by itself (relatively safe not to handle errors).
- Programmer can also inspect the error and abort the code using the default error handle MPI_ERRORS_RETURN.

Questions: what would these lines of codes do?

1)

MPI/MPI_SR_1.c – MPI/MPI_SR_3.c code examples are related to these questions

...

your_id=1-my_id

```
MPI_Send(&sendbuf,1,MPI_INT,my_id,0,comm);  
MPI_Recv(&recvbuf,1,MPI_INT,my_id,0,comm,&status);
```

...

2)

What would happen if you used MPI_Rsend function?

...

your_id=1-my_id

```
MPI_Recv(&recvbuf,1,MPI_INT,my_id,0,comm,&status);  
MPI_Send(&sendbuf,1,MPI_INT,my_id,0,comm);
```

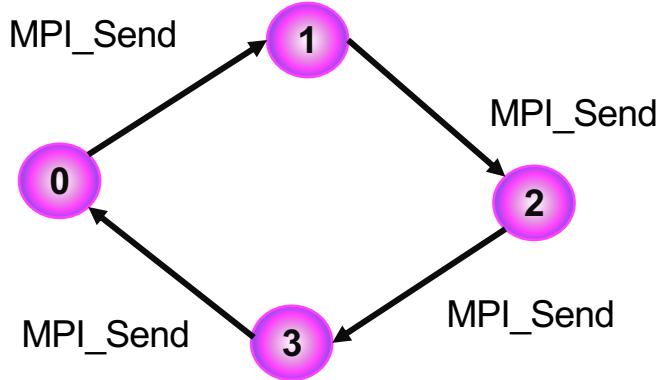
...

3)

Case 1) if you would send larger messages? What is happening here?

Deadlock

Processes wait for each other to do something, and the code hangs.



Cycles in waiting-for-graphs indicate deadlocks.

Question

Will the following pseudocode deadlock with MPI_Send and MPI_Recv?

MPI/MPI_SR_4.c code example is related to these questions

...

```
next_id = my_id+1; prev_id = my_id-1;  
if ( /* I am not the last processor */ ) send(target=next_id);  
if ( /* I am not the first processor */ ) receive(source=prev_id)
```

...

Would you call this efficient parallel execution? What actually happens?
Why are the results very difficult to interpret?

Pair-wise co-operative MPI_Sendrecv

- How to prevent deadlocks? 1. Avoid unsafe operations; one alternative is to use...
- Use MPI_Sendrecv(....from... ...to...); with the right choice of source and destination.
- For example:

```
MPI_Comm_rank(comm,&nproc); ....
```

```
MPI_Sendrecv( .... /* from: */ nproc-1 ... ... /* to: */ nproc+1 ... );
```

- Then you always need a “pair” to communicate with
- If not, then you need to use “MPI_PROC_NULL”

Question

Will the efficiency of this code be any better with MPI_Sendrecv?

...

```
next_id = my_id+1; prev_id = my_id-1;  
if ( /* I am not the last processor */ ) send(target=next_id);  
if ( /* I am not the first processor */ ) receive(source=prev_id)
```

...

MPI/MPI_SR_5.c code example is related to this question

Synchronous blocking send MPI_Ssend

- Another alternative is to use...
- **MPI_Ssend();**
- "S" for "Synchronous", meaning that the receiver is *always forced* to send an acknowledge.
- It will not avoid deadlocks.
- In this case, all unsafe operations should always deadlock, helping you out to debug and write "safer" code.

Buffered blocking communication

3. Force buffering

```
int bufsize; /* Size of data + MPI_BSEND_OVERHEAD */  
char *buf = malloc( bufsize );  
MPI_Buffer_attach( buf, bufsize );  
...  
MPI_Bsend( ... same as MPI_Send ... );  
...  
MPI_Buffer_detach( &buf, &bufsize );  
...
```

User is responsible for allocating large enough buffers.

Question: is this more efficient? You can try it out.



Blocking communication

Pros

Programmer has **full control** about where the data is: if the send call returns, the data has been successfully received, and the send buffer can be used for other purposes or de-allocated.

Buffering possible, so programmer can collect small messages into larger ones.

Cons

Unsafe operations cause deadlocks – one needs to be careful in ordering the calls.

Overlapping computation and communication is challenging.

Non-blocking communication

Immediate or Incomplete
MPI_Isend and
MPI_Irecv: they tell the runtime system
“Here is my data, please send it forward as I instruct”
or
“I am expecting certain type of data to come to this provided buffer space”.

“Posting”

Rank 1

MPI_Isend



Rank 0

MPI_Irecv

Non-blocking communication

```
int MPI_Isend(const 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)
```

Non-blocking routines yield an **MPI_Request** object. This request can then be used to query whether the operation has completed. **MPI_Irecv** routine does not yield an **MPI_Status** object. This is because the status object describes the actually received data, and at the completion of the **MPI_Irecv** call there is no received data yet.

Non-blocking communication

MPI_STATUS_IGNORE

```
Int MPI_Wait(MPI_Request *request, MPI_Status *status);  
int MPI_Waitall(int count, MPI_Request array_of_requests[],  
    MPI_Status array_of_statuses[])
```

MPI_STATUSES_IGNORE

One needs to **wait** for the completion of the non-blocking routines. There are various functions for that. They pass the **MPI_Request object** as a reference and return an **MPI_status**. If you are not interested in the status, then you can specify **MPI_STATUS(ES)_IGNORE** instead. These calls **deallocate** the handle after and set it to **MPI_REQUEST_NULL**. Waitall waits for **multiple** messages, and hence works with **arrays of requests and statuses**.

Non-blocking communication

```
int MPI_Waitany(int count, MPI_Request array_of_requests[], int
    *index, MPI_Status *status)      MPI_STATUS_IGNORE
int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
    int *outcount, int array_of_indices[], MPI_Status
    array_of_statuses[])           MPI_STATUSES_IGNORE
```

If one wishes to wait for **one or some** messages separately, then Waitany and Waitsome functions can be used. NB! Only after the corresponding wait call it is safe to use the buffer that has been sent, or has received its contents. To send multiple messages with non-blocking calls you therefore have to allocate multiple buffers (unlike in the blocking case).

MPI_Testx

- For every “Wait” there is a corresponding “Test”.
- While “Waits” are blocking, “Tests” are non-blocking, and can be used for **polling** if communication is completed.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

- Flag is set to true if the communication described by the specified handle has completed.

Useful reading:

MPI 4 standard: <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>

MPI 3 (version 3.1) standard: <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>

OpenMPI documentation: <https://www.open-mpi.org/doc/>

CS-E4690 – Programming parallel supercomputers

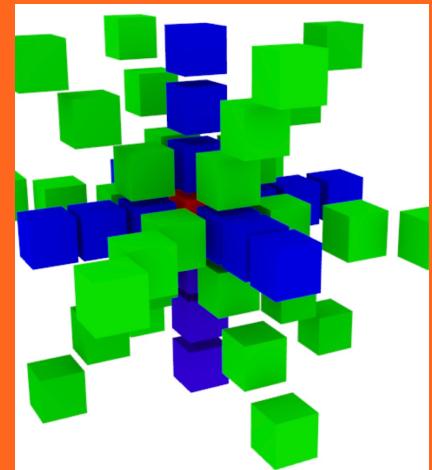
Designing parallel algorithms (EXTRA)

Maarit Korpi-Lagg

maarit.korpi-lagg@aalto.fi



Aalto University
School of Science

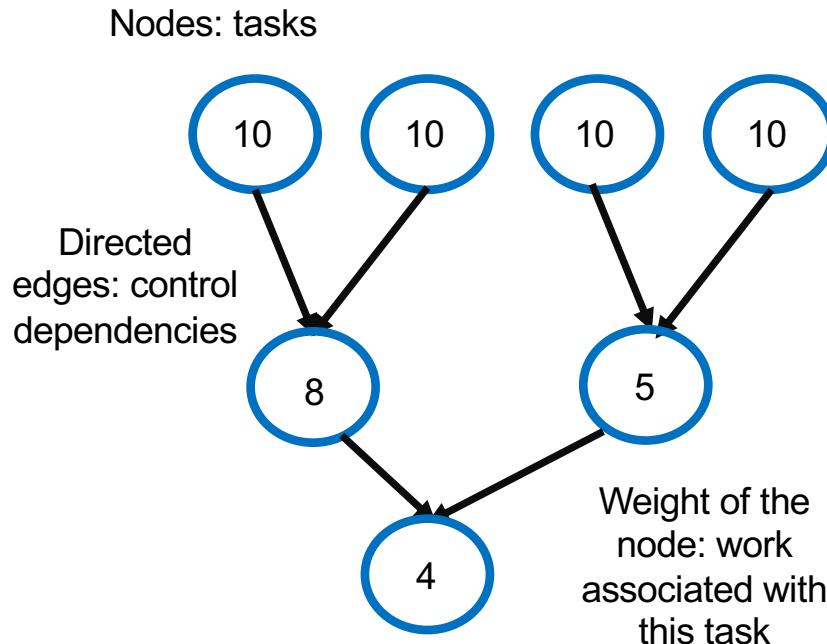


How to design a parallel algorithm?

- Determine **which parts** of your code can be computed **concurrently**
- **Decompose** these parts to **smaller pieces** that can be computed concurrently== **tasks**
- **Map** the obtained tasks to a “**virtual**” topology of **processes**, and **optimize** configuration
 - Maximize concurrency (Task dependency graphs) by mapping independent tasks onto different processes
 - Minimize interactions (Task interaction graphs) by mapping tasks with high degree of mutual interactions onto the same process
 - Make sure that there are processes to execute the next task when a previous task completes.

Task dependency graph (TDG)

- Optimum decomposition of the tasks for **concurrency**



Critical path:

The longest directed path between any pair of start (no incoming edge) and finish (no outgoing edge) node

Critical path length:

Sum of weights along critical path

Average degree of concurrency (to be maximized)

Total amount of work/critical path length

In the example: $57/22=2.59$

Examples

Data base query; imaginary phone sales catalogue

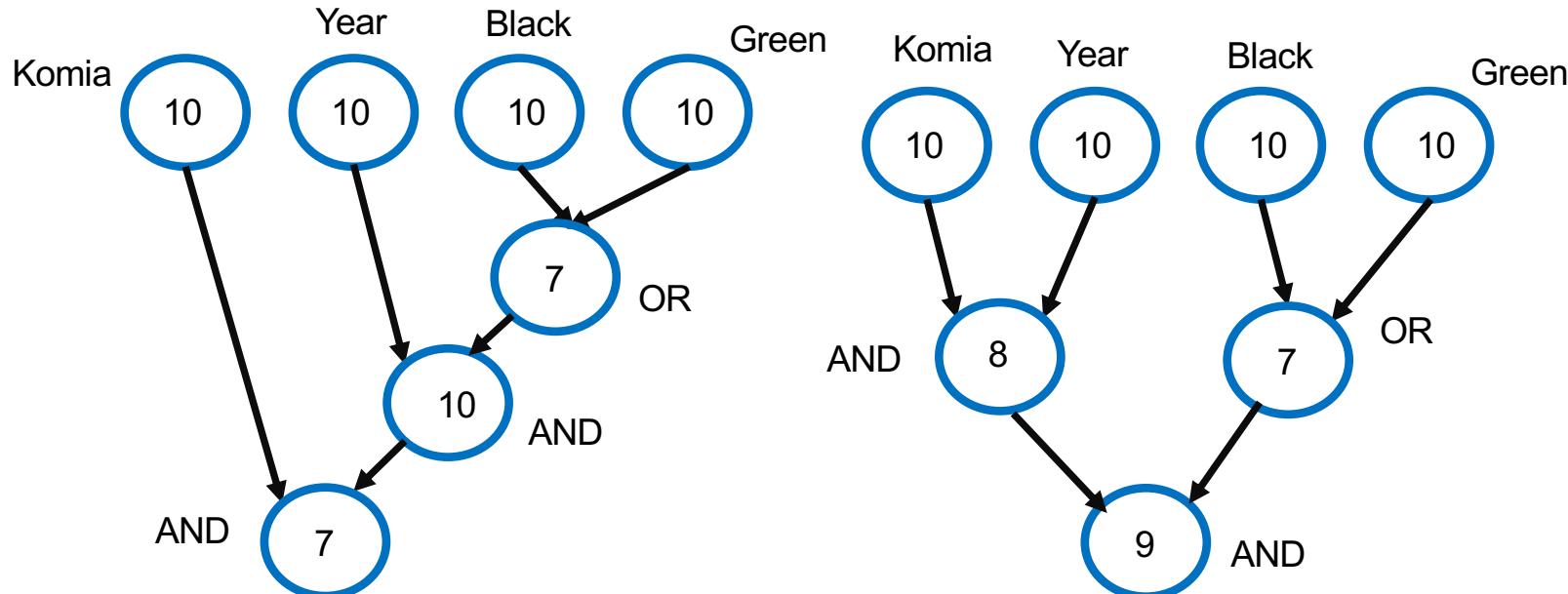
ID#	Year	Manufacturer	Model	Color	Retailer
23498	2018	Komia	Pulikka	Black	Kikantti
8734568	2019	OneMinus	6	Blue	Elise
265341	2017	Orange	10	Green	NDA
6743345	2019	Komia	Palikka	Black	Kikantti
3265	2016	OneMinus	6	Green	Elise
534876	2017	OneMinus	7	Red	NDA
762345	2019	Komia	Palikka	Green	Elise
34567	2020	Orange	11	Black	Kikantti
123867	2020	Komia	Pulikka	Blue	NDA
46556	2017	Komia	Palikka	Black	Elise

Query: Manufacturer="Komia" AND Year="2019" AND (Color="Black" OR Color="Green")

Examples

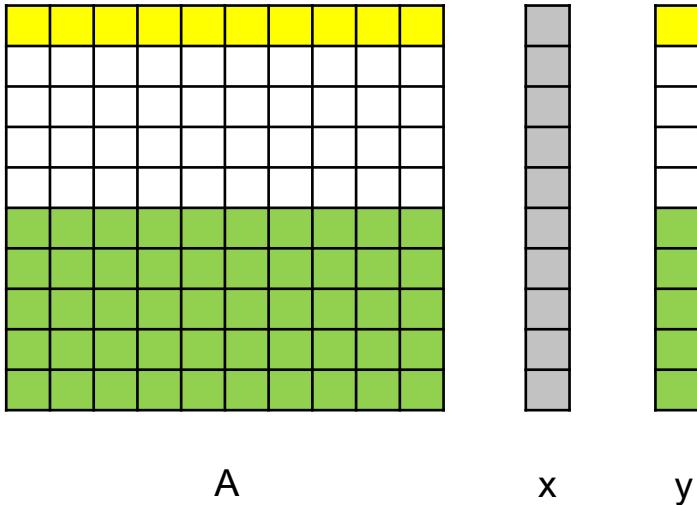
Data base query; imaginary phone sales catalogue

Query: Manufacturer="Komia" AND Year="2019" AND (Color="Black" OR Color="Green")



Examples

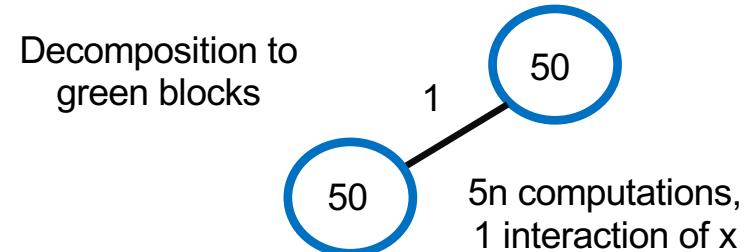
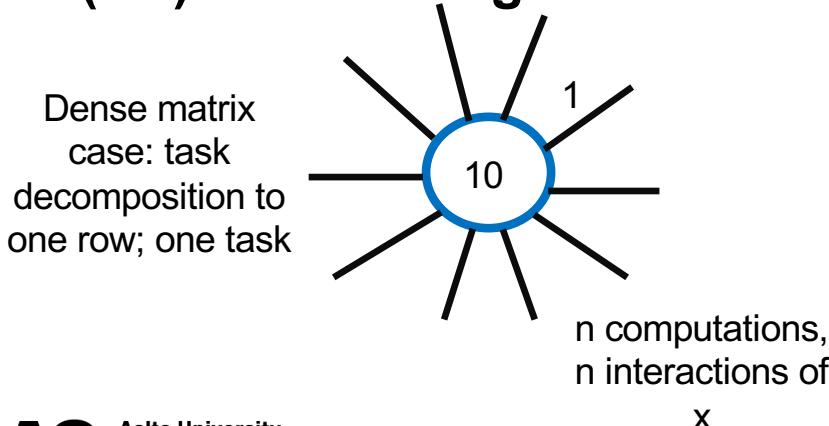
Matrix-vector multiplication; $y=Ax$



- All tasks are independent (no directed edges from nodes)
- Maximum concurrency according to TDG would be obtained by dividing to the smallest possible entity (one cell)
- Possibility to divide the work based on data in many different ways (for example to yellow or green blocks)
- No matter how you divide the work, **you will need totality of x for all tasks to update an element of y**

Task interaction graph (TIG)

- Optimize data dependencies (minimize interactions)
- To decide what is the optimum granulation level of the decomposition
- Nodes represent tasks and their computation times
- (Un)directed edges the interactions in between them

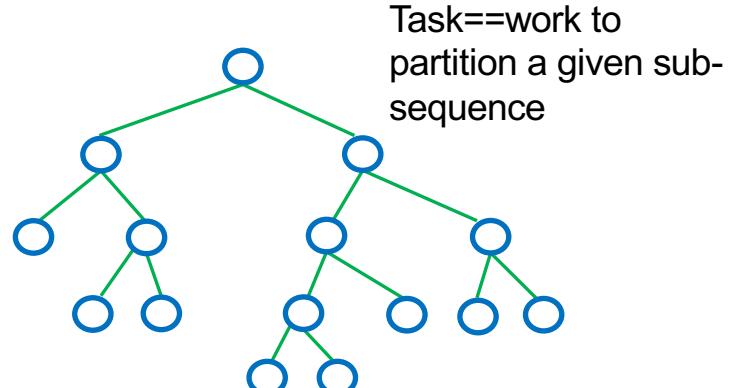
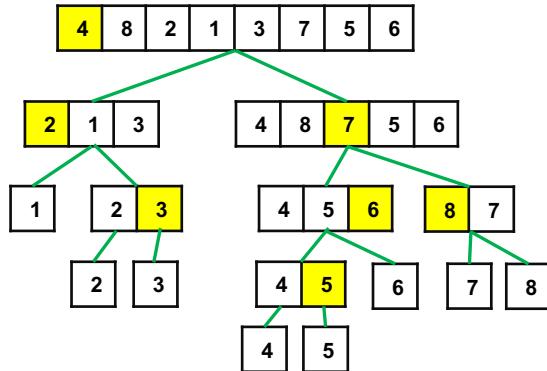


Decomposition

- **Task decomposition**
 - Recursive decomposition: “Divide and conquer”
- **Data decomposition (Input/Output/Intermediate/Hybrid)**
 - Input/Output: “Owner computes” model
- **(Exploratory)**
- **(Speculative)**

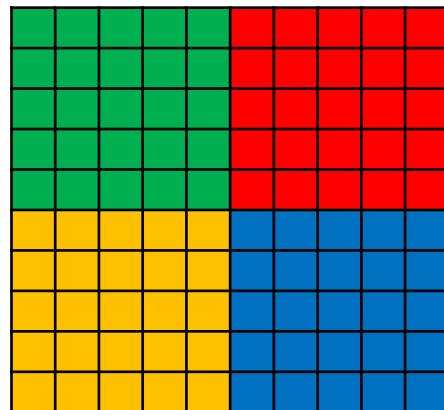
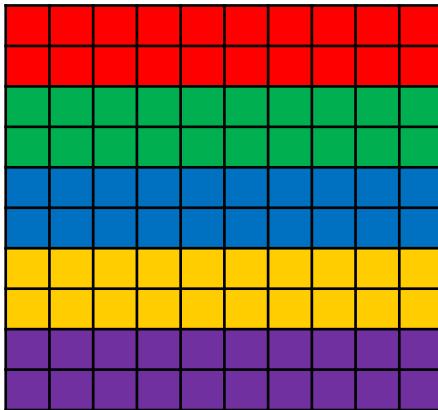
Recursive decomposition

- Decompose a problem into independent sub-problems
- Decompose sub-problems similarly using recursion
- Stop decomposing, when the granularity becomes sub-optimal or result is obtained
- Typical example: Quicksort



Data decomposition

- Manipulation of large data sets; matrix-vector multiplication was one good example
- Define tasks based on partitioning the data
- Output/Input/Intermediate/Hybrid



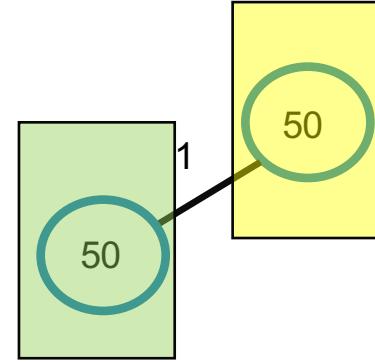
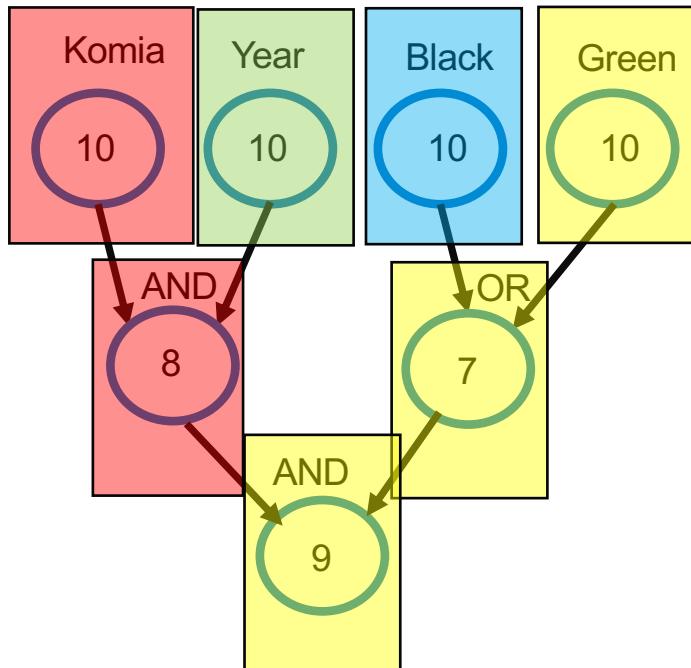
.....

How to map tasks to processes?

- Process is a logic entity performing the defined tasks
- Let us look at our example cases

How to map to processes?

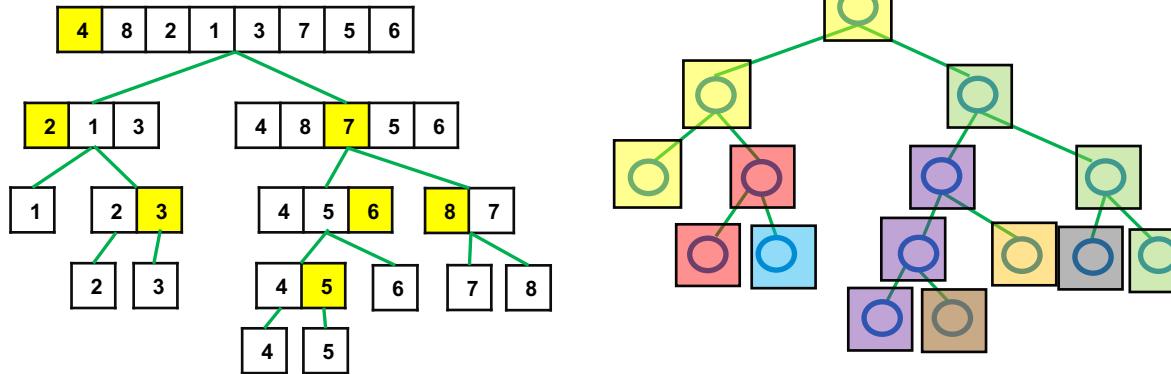
Data base query; best case concurrency-wise



Dense matrix-vector multiplication; If we decompose the data into two row-wise blocks, we can map them to two concurrent processes

How to map to processes?

Quicksort; tree-like mapping



Static versus dynamic tasks and mapping

- Dense matrix multiplication is suited for **static** task generation and mapping (no need to change them when repeating the operation for different data sets)
- Database query and sorting, for example, are suited for **dynamic** task generation and mapping (with a changing query, the optimal graphs will change)
- **Task dependency graph** is fixed for static, not known a priori for the dynamic case

Regular versus Irregular interactions

- Dense matrix multiplication has *regular* interactions (communication pattern between tasks repeats)
- If the matrix was sparse but did not possess any symmetry properties, then its communication pattern would become *irregular* (communication pattern would become dependent on where the zeros are in the matrix).
- Task interaction graph is fixed for regular, not known a priori for the irregular case
- Interactions can also be static and dynamic.

What to do in practice?

- **Static and regular mappings** are “trivial” cases for MPI.
- **Dynamic and irregular mappings** are the challenge
 - MPI can handle dynamicism with spawning more processes when needed (`MPI_Comm_spawn` and related functions); tedious
 - Also the way for implementing fault tolerance in MPI-4 standard; not ubiquitously available, hence we skip this year
 - MPI + openMP programming model; less tedious

CS-E4690 – Programming Parallel Supercomputers

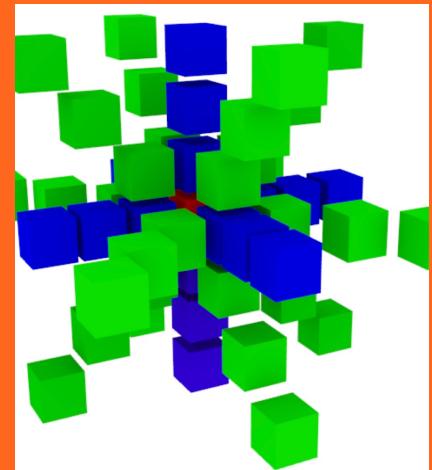
MPI: Collective communications and advanced topics

Maarit Korpi-Lagg

maarit.korpi-lagg@aalto.fi



Aalto University
School of Science



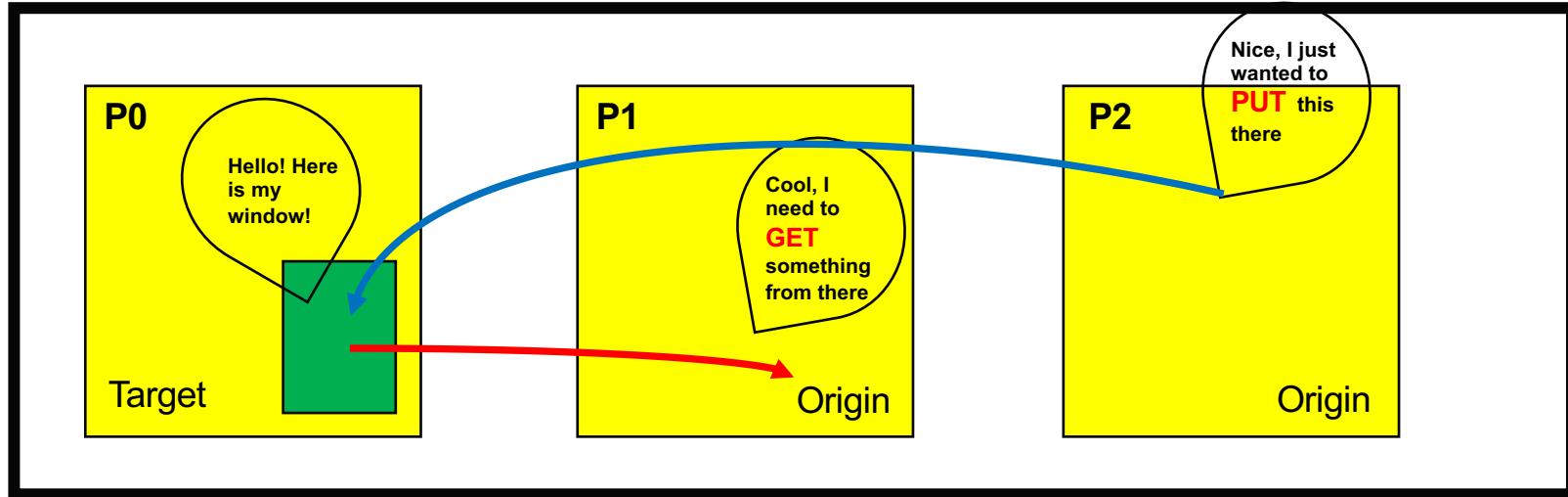
One-sided communication

Remote memory reads and writes (requires **RMA technology**, but nowadays standardly available on multiprocessors chips)

- Only **one process** needs to explicitly participate.
- An advantage is that communication and synchronization are **decoupled**
- Can, in principle, reduce synchronization overheads

Window

communicator



Origins could also wish to **ACCUMULATE** data to/from target process, which can include a reduction operation.

Typical workflow

```
MPI_Info info;
MPI_Win window;
MPI_Win_create( /* size info */, info, comm, &memory, &window );
// do put and get calls
MPI_Win_free( &window );
```

Window properties

- Create window call is **collective** (all in certain comm must call it)
- The window **size** can be set **individually** on each process (also to null)
- The window **location** can be set individually on each process
- The window is the **target** of data in a **put** operation, or the **source** of data in a **get** operation.
- There can be memory associated with a window, so it needs to be freed **explicitly** with `MPI_Win_free(win)`.

Creating a window (basic)

`MPI_Alloc_mem` to allocate the “base” buffer

```
int MPI_Win_create (void *base, MPI_Aint size, int disp_unit,  
                    MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

base (pointer to) local memory to expose for RMA

size of a local window in bytes

disp_unit local unit size for displacements in bytes

info info argument

comm communicator

win handle to window

MPI_Win_free(win)

Creating a window (with allocate)

```
int MPI_Win_allocate (MPI_Aint size, int disp_unit, MPI_Info info,  
MPI_Comm comm, void *baseptr, MPI_Win *win)
```

Allocates window segment

size: size of a local window in bytes

disp_unit local unit size for displacements, in bytes

info: info argument

comm: communicator

baseptr: address of local allocated window segment

win: window object returned by the call

MPI_Win_free(win)

Creating a window (dynamic)

```
int MPI_Win_create_dynamic (MPI_Info info, MPI_Comm comm,  
MPI_Win *win)
```

Similar to the others, but only a pointer to a window object, with its attached memory yet unspecified, is returned.

to an empty buffer is returned. To attach memory to the Window:

```
MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)  
MPI_Win_detach(MPI_Win win, void *base)
```

Advanced MPI, finding out details left for interested students

MPI_Win_free(win)

Synchronization?

Active synchronization:

- Both origin and target process perform synchronization calls

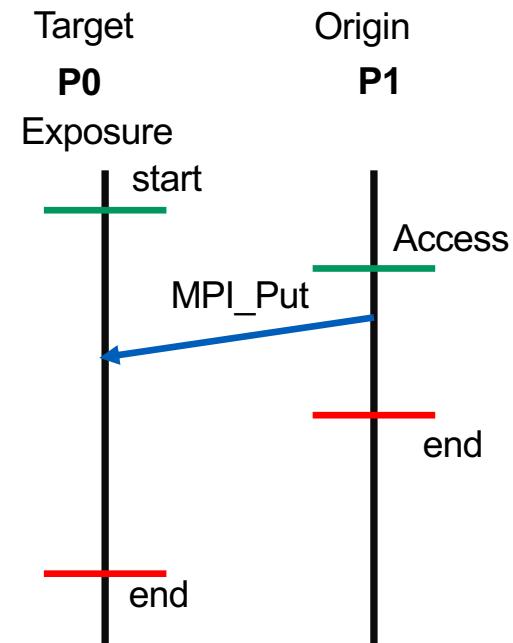
Passive synchronization:

- No synchronization calls at the target

Active mode:

Communication takes place within **epochs**

- Exposure epoch** on target, **access epoch** on origin
- Synchronization calls start and end an epoch
- An epoch is specific to a particular window



Epoch: time in
between the green
and red lines

Active (global) RMA synchronization: fences

MPI_Win_fence(assert, win)

assert optimize for specific usage. Valid values are "0",

MPI_MODE_NOSTORE, MPI_MODE_NOPUT,

MPI_MODE_NOPRECEDE, MPI_MODE_NOSUCCEED

win window handle

- Used both starting and ending an epoch
- Assertions with 0 will always work, but being more specific could help, see the next slide (advanced)

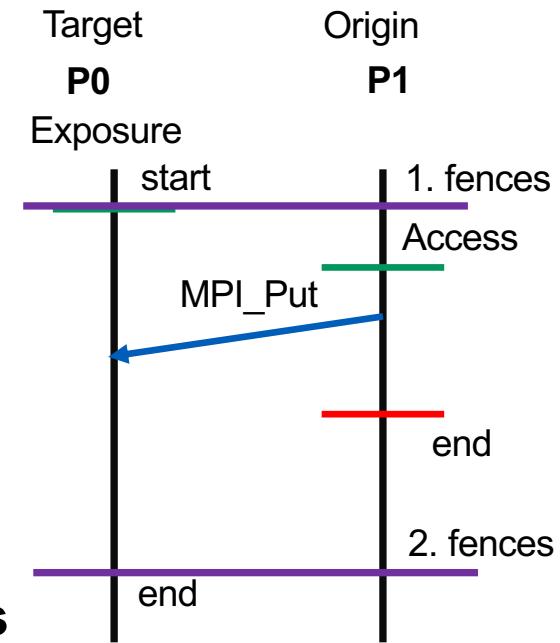
MPI/One_sided_1.c

Assertions for fence ops (advanced)

- **MPI_MODE_NOSTORE** the local window was not updated by local stores (or local get or receive calls) since last synchronization.
- **MPI_MODE_NOPUT** the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.
- **MPI_MODE_NOPRECEDE** the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.
- **MPI_MODE_NOSUCCEED** the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.
- Specifying these may help in optimizing the communication.

Fenced synchronization is restrictive

- **GLOBAL**: every process has to execute the calls
- Fences act as Barrier-like operations; recall the BSP model
- Can be inefficient
- The need for global synchronization can be avoided by **defining processor groups**; this requires additional knowledge about the communicators, and we will come back to this later on



Epoch: time in between
the purple lines

Passive synchronization

MPI/One_sided_3.c

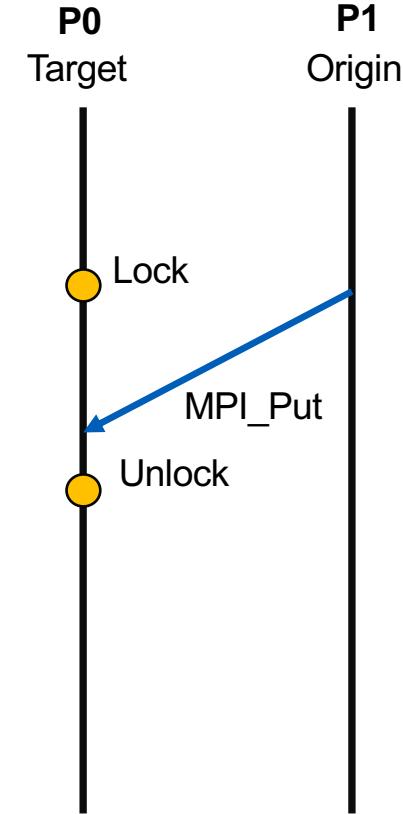
```
int MPI_Win_lock(int lock_type, int rank, int assert,  
MPI_Win win)
```

lock_type: Indicates whether other processes may access the target window at the same time (if MPI_LOCK_SHARED) or not (MPI_LOCK_EXCLUSIVE)

rank: rank of the process having the **locked** (target) window

assert: Used to optimize this call; **zero may be used as a default**.

win: window object



Moving data: put

```
int MPI_Put( const void *origin_addr, int origin_count,  
            MPI_Datatype origin_datatype, int target_rank, MPI_Aint  
            target_disp, int target_count, MPI_Datatype target_datatype,  
            MPI_Win win)
```

- Otherwise very normal-looking call, but the target data description is somewhat non-trivial
- When creating a window, you need to specify the displacement unit (from the window start)

MPI/One_sided_1.c

Moving data: get

```
int MPI_Get( const void *origin_addr, int origin_count,  
            MPI_Datatype origin_datatype, int target_rank, MPI_Aint  
            target_disp, int target_count, MPI_Datatype target_datatype,  
            MPI_Win win)
```

- Similar syntax to MPI_Put

Moving data: accumulate

```
int MPI_Accumulate (const void *origin_addr, int  
    origin_count,MPI_Datatype origin_datatype, int  
    target_rank,MPI_Aint target_disp, int  
    target_count,MPI_Datatype target_datatype, MPI_Op op,  
    MPI_Win win)
```

- Store data from the **origin** process to the memory window of the **target** process *and* combine it using one of the predefined MPI reduction operations
- **Predefined** operators are available (we talk about these in the connection of collectives), but **no user-defined ones**.
- There is one extra operator: **MPI_REPLACE**, this has the effect that only the last result to arrive is retained.

Moving data: get_accumulate

```
int MPI_Get_accumulate (const void *origin_addr, int  
                      origin_count, MPI_Datatype origin_datatype, void  
                      *result_addr, int result_count, MPI_Datatype result_datatype,  
                      int target_rank, MPI_Aint target_disp, int target_count,  
                      MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

- Store data from target window to the origin, and combine it with the predefined operation.
- **Predefined** operators are available (we talk about these in the connection of collectives), but **no user-defined ones**.
- There is one extra operator: **MPI_REPLACE**, this has the effect that only the last result to arrive is retained.

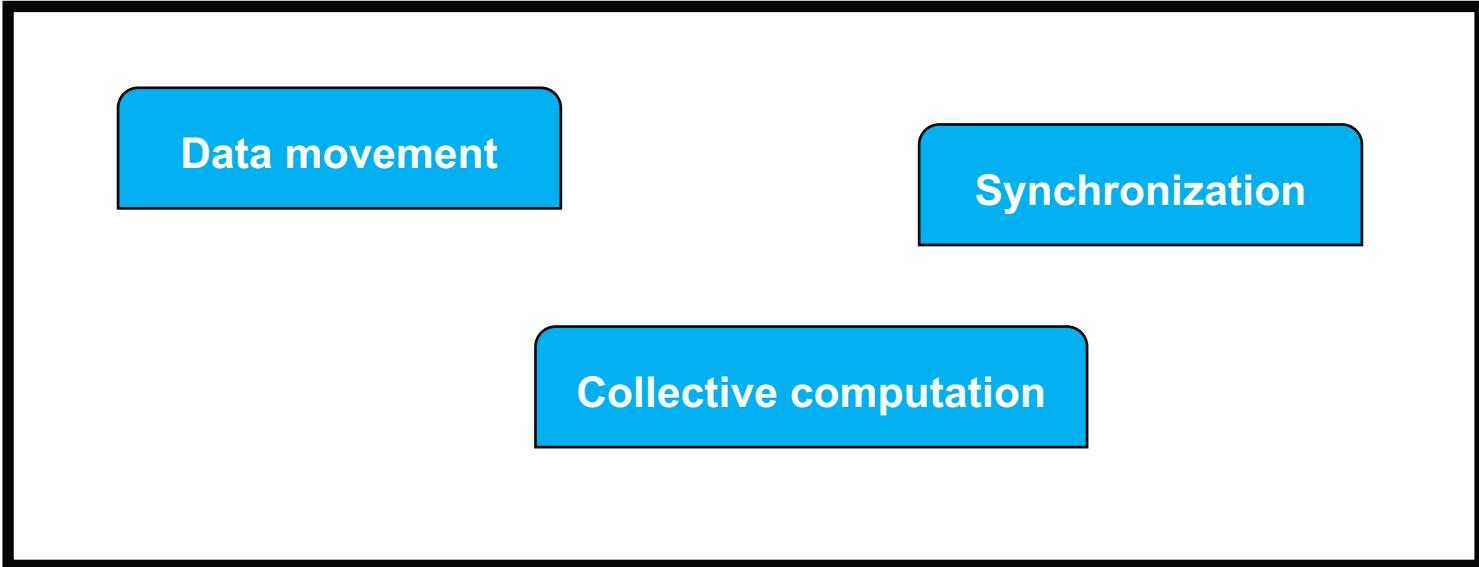
Note: many processes!

- **Within an epoch**, no guaranteed ordering of Get and Put operations: if you make many such calls in a mixture, there is no guarantee who gets to overlapping data first, and the results may turn out to be garbage (*race conditions*).
- Accumulates are ‘atomic’:
 - MPI_Accumulate with MPI_REPLACE implements an atomic put that has a well-defined order
 - MPI_Get_accumulate with MPI_NO_OP implements an atomic get that has a well-defined order.
- **Multiple atomic operations are safe within an epoch.**

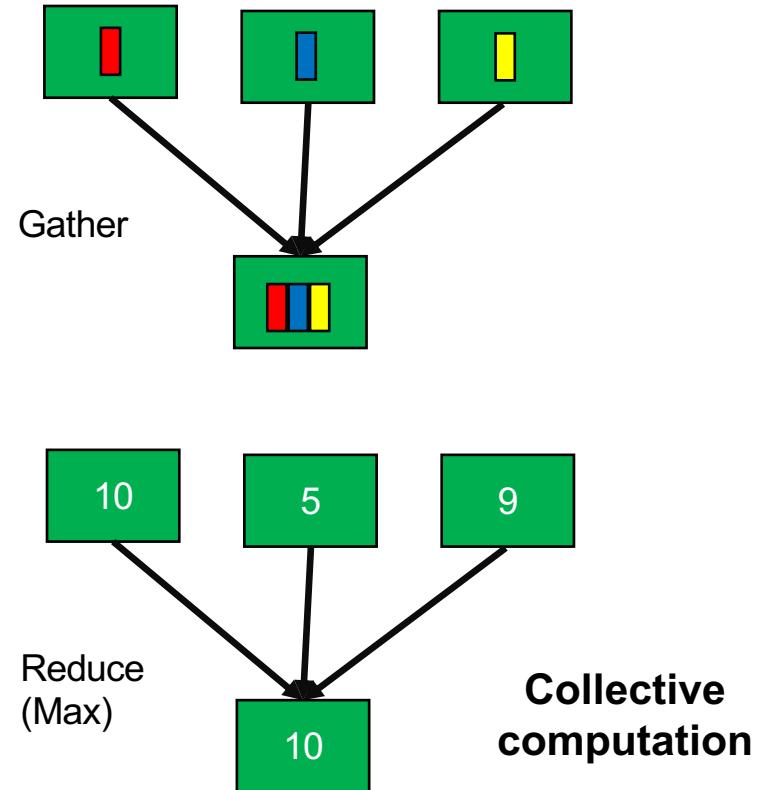
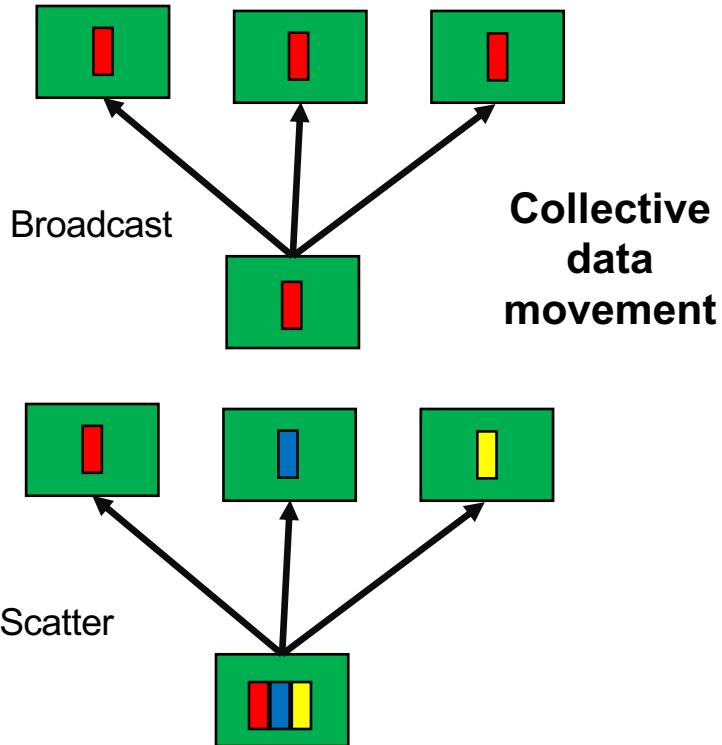
MPI/One_sided_1.c

Collective communications

Communicator



Typical collectives



Collective data movement; Broadcast

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

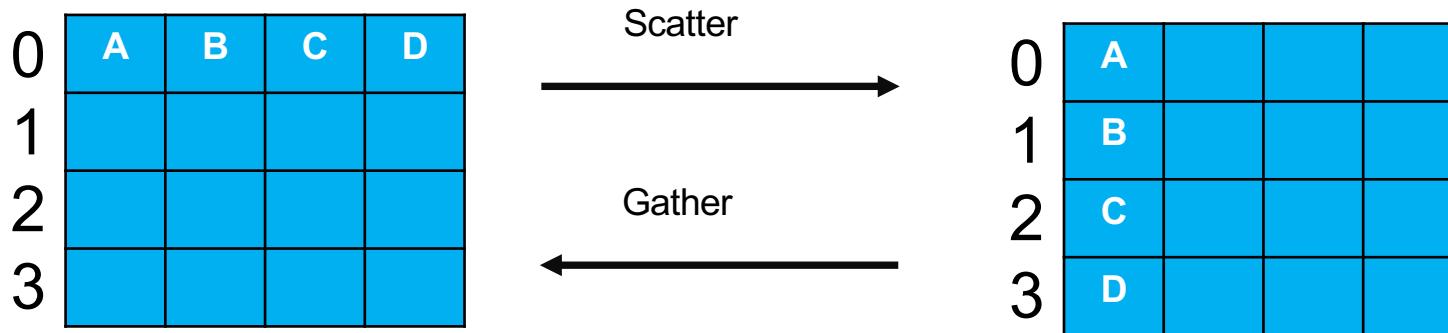
Example of “Rooted Collectives”

0	A			
1				
2				
3				



0	A			
1	A			
2	A			
3	A			

Collective data movement; Scatter, Gather



Collective data movement; Gather & Scatter

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

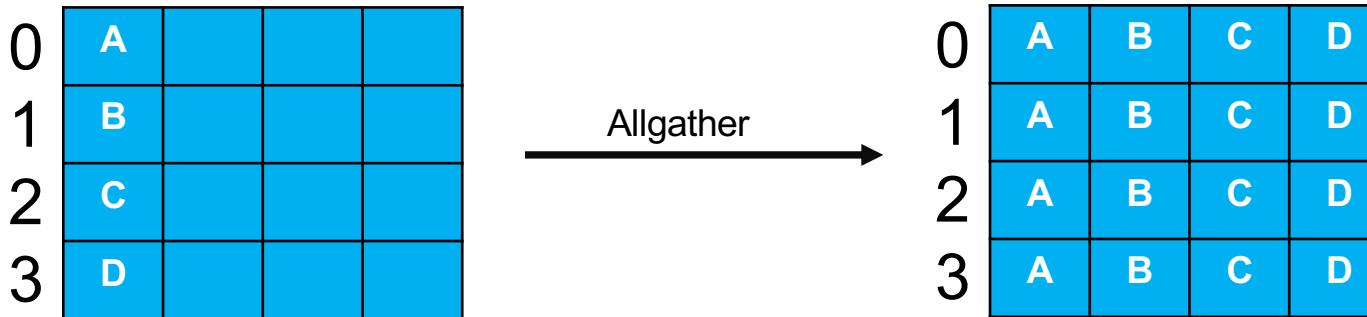
```
int MPI_Scatter (void* sendbuf, int sendcount, MPI_Datatype sendtype,  
    void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,  
    MPI_Comm comm)
```

- Send and receive buffers are **no longer of the same size**, hence need to specify two buffers.
- Root receives/sends np sized buffer of data, others send/receive data of the size n .
- Counterintuitively, root's **recvcount/sendcount** is **NOT** np , but n .
- SPMD code; everybody will have to allocate the large buffer; is that not awkward? Yes, other than 'root' processes,
 - use a null pointer in place of the larger buffer
 - Or use the option "**MPI_IN_PLACE**" for the unnecessary buffers.

Collective data movement; Allgather

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

```
int MPI_Iallgather (const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
                    void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm,  
                    MPI_Request *request)
```

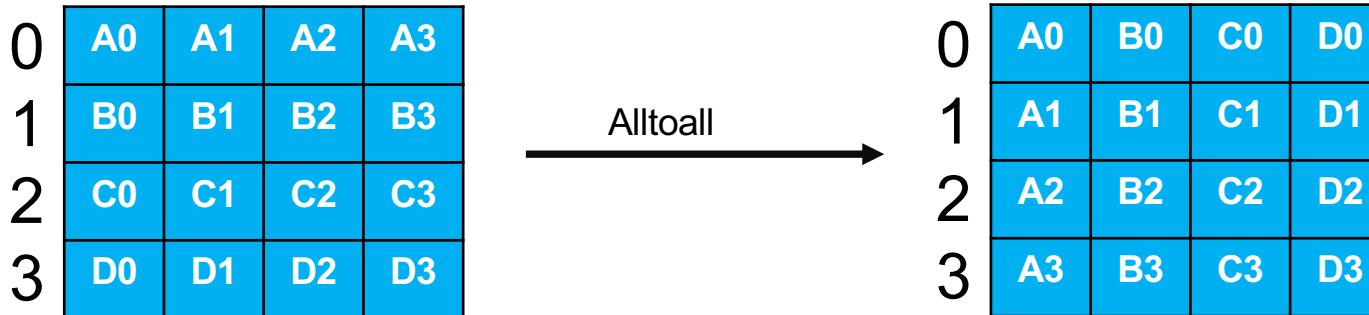


Questions: What is this equivalent of (using simpler functions)?
Can this be useful in matrix ops? Which implementation is faster?

Collective data movement; Alltoall

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

```
int MPI_Ialltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
                  void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm,  
                  MPI_Request *request)
```



Questions: What is this operation actually doing? Can it be useful in matrix manipulations?

Special variants

- The basic routines send/receive the same amount of data from each process
- “v” for vector routines to allow the programmer to specify a message of different length for each destination (one-to-all) or source (all-to-one) or destination and source (all-to-all)

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

```
int MPI_Alltoallv (void *sendbuf, int *sendcnts, int *sdispls, MPI_Datatype sendtype,  
void *recvbuf, int *recvcnts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

- May need to use some other collectives to compute the required displacements

MPI/Coll_1.c

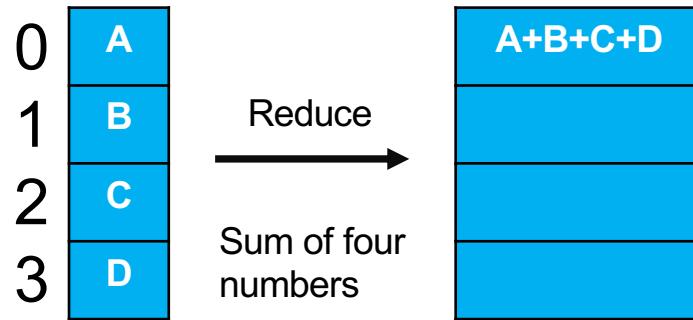
Collective computation

- **Combines communication with computation**
- **Combination operations either**
 - Predefined
 - User defined

Pre-defined

MPI type	meaning	applies to\
MPI_MAX	maximum	integer, floating point
MPI_MIN	minimum	
MPI_SUM	sum	integer, floating point, complex, multilanguage types
MPI_REPLACE	overwrite	
MPI_NO_OP	no change	
MPI_PROD	product	
MPI_LAND	logical and	C integer, logical
MPI_LOR	logical or	
MPI_LXOR	logical xor	
MPI_BAND	bitwise and	integer, byte, multilanguage types
MPI_BOR	bitwise or	
MPI_BXOR	bitwise xor	
MPI_MAXLOC	max value and location	MPI_DOUBLE_INT and such
MPI_MINLOC	min value and location	

Collective computation; Reduce



Collective computation; Reduce

All-to-one, “Rooted”

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

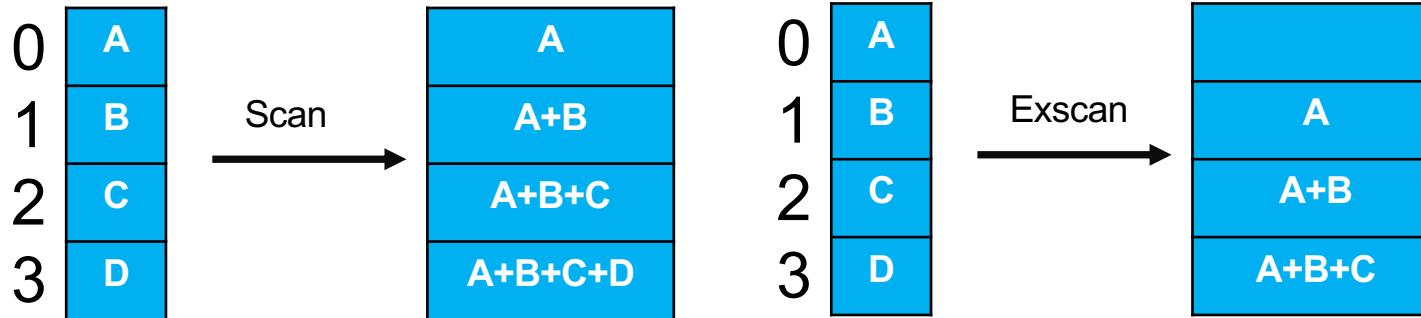
All-to-all

```
Int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,  
                MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,  
                MPI_Request *request)                                The reduction result will be returned only  
                                                          to root process receive buffer
```

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

```
int MPI_Iallreduce(const void *sendbuf, void *recvbuf, int count,  
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,  
                   MPI_Request *request)                                The reduction result will be returned to  
                                                               every rank's receive buffer.
```

More collective computation functions



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

```
int MPI_Iscan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,  
               MPI_Comm comm, MPI_Request *request)
```

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

```
int MPI_Iexscan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,  
                  MPI_Comm comm, MPI_Request *request)
```

More collective computation functions

```
int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf, const int recvcounts[],MPI_Datatype  
datatype, MPI_Op op, MPI_Comm comm)
```

```
int MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf, const int recvcounts[],MPI_Datatype  
datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)
```

User defined operations

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

Prototype

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

`inoutvec[i] = invec[i] op inoutvec[i]` from `i=0;len-1`

- The operation is assumed to be associative
- You can use flag “commute” to indicate whether the function is in addition commutative or not.
- Void return as no errors are expected

Synchronization

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

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
```

- Waits until all processes have called it
- Forces time synchronization
- Not needed very often, as collectives impose synchronization on their own

About communicators

So far we have used the default communicator only

```
MPI_Comm comm = MPI_COMM_WORLD;
```

But you can do much more with them, and here we just give a short introduction to those possibilities

Duplicate

Split

Define new communicators by groups of processes

Spawn new communicators (highly advanced MPI)

Intercommunicate (highly advanced MPI)

Duplicating

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)  
int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm,  
                  MPI_Request *request)
```

Sounds strange, but is handy. Consider the following scenario

```
MPI_Isend(...);  
// library call that also issues sends and receives  
MPI_Irecv(...);  
MPI_Waitall(...);
```

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

Splitting

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

comm: communicator (handle)

color: control of subset assignment (integer)

key: control of rank assignment (integer)

newcomm: new communicator (handle)

Splitting: problem

How to split a 2D grid of processes into a column communicator?

MPI/Split_1.c

Constructing new by groups

- Get group of communicator

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

comm : Communicator (handle)

group : Group in communicator (handle)

- Manipulate the groups with functions like **MPI_Group_incl**, **MPI_Group_excl**, ...
- Create the communicator(s) by

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

newcomm : new communicator (handle).

Constructing new by groups

```
int MPI_Group_incl(MPI_Group group, int n, const int ranks[],  
MPI_Group *newgroup)
```

group Group (handle).

n Number of elements in array **ranks** (and size of *newgroup*)(integer).

ranks Ranks of processes in group to appear in *newgroup* (array of integers).

Constructing new by groups

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

group Group (handle).

n Number of elements in array ranks (integer).

ranks Array of integer ranks in group not to appear in newgroup.

Constructing new by groups: question

How to set up groups based on even or odd rank of processes?

MPI/Split_2.c

Using groups to improve one-sided comms

- Define exposure epoch, on target, and access epoch, on origin, epochs using process groups
- Target runs exposure epoch by issuing

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)  
int MPI_Win_wait(MPI_Win win)
```

- Origin runs access epoch by issuing

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)  
int MPI_Win_complete(MPI_Win win)
```

Using groups for one-sided comms: example

```
if (my_id==origin) {  
  
    MPI_Group_incl(all,1,&target,&tgroup);  
  
    // access  
  
    MPI_Win_start(tgroup,0,the_window);  
  
    MPI_Put( /* data on origin: */ &my_number, 1,MPI_INT, /* data on target: */ target,0, 1,MPI_INT,  
    the_window);  
  
    MPI_Win_complete(the_window); ...}  
  
if (my_id==target) {  
  
    MPI_Group_incl(all,1,&origin,&ogroup);  
  
    // exposure  
  
    MPI_Win_post(ogroup,0,the_window);  
  
    MPI_Win_wait(the_window); ...}
```

Intercommunicators

What if your subcommunicators would need to communicate?

Can be achieved with **intercommunicators** (highly advanced MPI).

Look up the function **`MPI_Intercomm_Create`** from openMPI manual

Both **p2p** and **collective** comms then possible between the subcommunicators through this intercommunicator.

User defined data types

Brief introduction

- You would like to send data of different types in one and the same message
- Your data is not contiguous

Defining and decommissioning a new datatype

```
int MPI_Type_XXX(...MPI_Datatype oldtype,  
                  MPI_Datatype *newtype)  
int MPI_Type_commit(MPI_Datatype *newtype)...  
int MPI_Type_free(MPI_Datatype *newtype)
```

newtype the new datatype to commit, use, and decommission

oldtype the datatype to use for constructing newtype
XXX stands for one of the constructors

Datatype constructors

<code>MPI_Type_contiguous</code>	contiguous datatypes
<code>MPI_Type_vector</code>	regularly spaced datatype
<code>MPI_Type_indexed</code>	variably spaced datatype
<code>MPI_Type_create_subarray</code>	subarray within a multi-dimensional array
<code>MPI_Type_create_hvector</code>	like vector, but uses bytes for spacings
<code>MPI_Type_create_hindexed</code>	like index, but uses bytes for spacings
<code>MPI_Type_create_struct</code>	fully general datatype

Contiguous data

```
int MPI_Type_contiguous(int count,MPI_Datatype  
                      oldtype, MPI_Datatype *newtype)  
int MPI_Type_commit(MPI_Datatype *newtype)...  
int MPI_Type_free(MPI_Datatype *newtype)
```

newtype the new datatype to commit, use, and decommission

oldtype the datatype to use for constructing newtype
count number of replicas

Vector data

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

count number of blocks

blocklength number of replicated oldtype elements in each block

stride total number of elements in each block

MPI/Datatypes_1.c

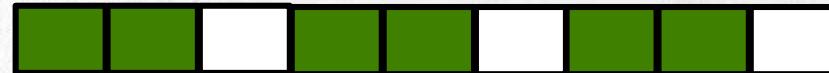
MPI_Type_vector(3, 2, 3, oldtype, newtype)

oldtype



BLOCKLEN=2

newtype



STRIDE=3

Subarrays of data

```
int MPI_Type_create_subarray(int ndims, const int sizes[], const  
int subsizes[], const int offsets[], int order, MPI_Datatype oldtype,  
MPI_Datatype *newtype)
```

ndims number of array dimensions

sizes number of array elements in each dimension

subsizes number of subarray elements in each dimension

offsets starting point of subarray in each dimension

order storage order of the array. Either MPI_ORDER_C or
MPI_ORDER_FORTRAN

Subarrays of data; simple example

```
int array_size[2] = {5,5};  
int subarray_size[2] = {2,2};  
int subarray_start[2] = {1,1};  
MPI_Datatype subtype;  
double **array  
// Put in some data to the subarray of rank 1  
MPI_Type_create_subarray(2,array_size,  
    subarray_size, subarray_start,  
    MPI_ORDER_C, MPI_DOUBLE, &subtype);  
MPI_Type_commit(&subtype);  
if (rank==0)  
MPI_Recv(array[0], 1, subtype, 1, 123,  
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
if (rank==1)  
MPI_Send(array[0], 1, subtype, 0, 123,  
    MPI_COMM_WORLD);
```

Rank 0: original array					
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
Rank 0: array after receive					
0.0	0.0	0.0	0.0	0.0	0.0
0.0	1.0	1.0	0.0	0.0	0.0
0.0	1.0	1.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

Topologies

- How to tell to MPI **your wish to map created ranks onto the physical topology?**
- MPI provides routines to create **new communicators** that order the process ranks in a way that *may* be a better match for the physical topology
- **Virtual topologies supported**
 - Cartesian grid
 - Graph
- Topology routines all create a *new communicator with properties of the specified virtual topology*

Cartesian grid topology

- Useful if two neighbours in each dimension; think of a von Neumann stencil; contrast it to Moore's stencils
- Even though multi-dimensional topologies are usually the most performant, allowing MPI to use this mapping may still not give great performance gains.
- May simplify your code, though.

Cartesian grid topology

```
int MPI_Cart_create( MPI_Comm comm_old, int ndims, const int  
dims, const int periods, int reorder, MPI_Comm *comm_cart);  
  
int MPI_Cart_coords( MPI_Comm comm, int rank, int maxdims, int  
coords);  
  
int MPI_Cart_rank( MPI_Comm comm, init coords, int *rank);
```

ndims f.ex. 2 for 2-dim, 3 for 3-dim

dims of grid in each ndim (size of ndim)

periods which of the boundaries are periodic?

reorder can MPI re-order ranks as to what it sees optimal?

coords Coordinate of the process in the Cartesian topology

rank the rank of the process in the Cartesian topology

MPI/Comm_1.c

Cartesian grid topology

Determine the neighbors for communication

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int  
*source, int *dest)
```

direction Shifting direction in the defined dim

displ displacement in ranks >0 for up <0 down in the direction

source Neighbor rank in decreasing index

dest Neighbor rank towards increasing index

“Names” of the neighbor ranks come from **MPI_Sendrecv**, in the context of which this routine is often used

MPI/Comm_1.c

Graph topologies

- More elegant way of defining complex recurring communication patterns
- Graph vertices represent processes
- Edges denote interactions with neighbours
- Weights can be assigned to describe additional information on the edges

Graph topologies

```
Int MPI_Dist_graph_create_adjacent(MPI_Comm oldcomm, int  
    indegree, int sources[], int sourceweights[],  
    int outdegree, int dests[], int destweights[], MPI_Info info, int  
    reorder, MPI_Comm *newcomm)
```

indegree : number of **source** nodes; **sources** : array containing the ranks of the source nodes; **sourceweights** : weights for source to destination edges or **MPI_UNWEIGHTED**; **outdegree** : array specifying the number of **destinations**, **dests** : ranks of the destination nodes, **destweights** : weights for destination to source edges or **MPI_UNWEIGHTED**; **info** : hints on optimization and interpretation of weights, **reorder** : the process may be reordered?

Graph topologies

```
int MPI_Dist_graph_create (MPI_Comm comm_old, int n, const int  
    sources[], const int degrees[], const int destinations[], const  
    int weights[], MPI_Info info, int reorder, MPI_Comm  
    *comm_dist_graph)
```

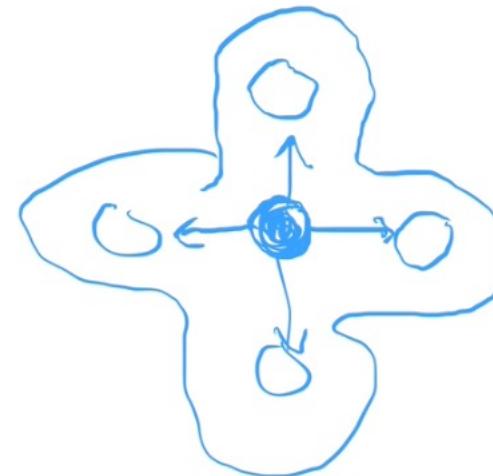
n: number of source nodes; **sources** : array containing the ranks of the source nodes; **degrees** : array specifying the number of **destinations** for each source node, **destinations** : ranks of the destination nodes, **weights** : weights for destination to source edges or **MPI_UNWEIGHTED**; **info** : hints on optimization and interpretation of weights, **reorder** : the process may be reordered?

MPI_Dist_graph_neighbors &
MPI_Dist_graph_neighbors_count

Graph topologies: example

2nd order von Neumann stencil halo communication

$n=1$
degrees = 4



Neighbor collectives

- Once the graph topology has been successfully defined, then neighbor collectives, such as **MPI_Neighbor_gather**, **MPI_Neighbor_allgather** and derivatives can be used to collect data only applying to this neighborhood.
- This can have certain benefits over p2p communication
 - More optimized topology
 - Collectives may use more efficient ways of communication (pipelining, trees)

CS-E4690 – Programming parallel supercomputers D

4th lecture

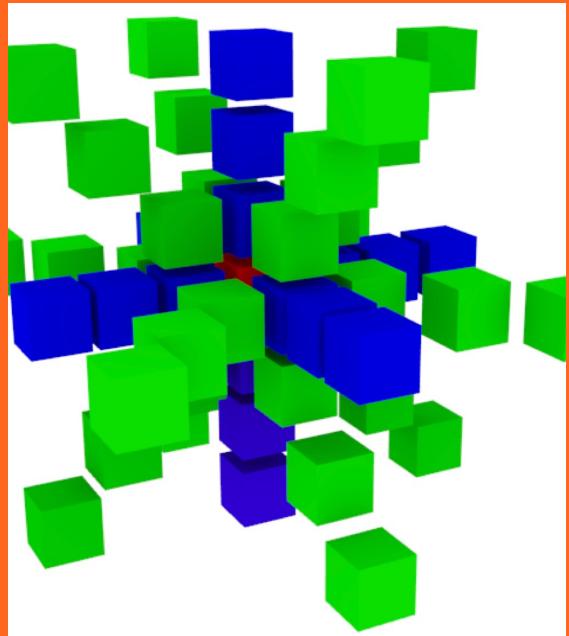
Maarit Korpi-Lagg

maarit.korpi-lagg@aalto.fi

14.11.2023



Aalto University
School of Science



Lecture 4

Collectives (finalizing basic MPI)
One-sided communication
(entering the advanced domain)

- Course practicalities: 5 min
- Key concepts recap (40 min) – old and new
- Break (max 15 mins)
- Example codes cntd. (20 mins)
- Exercise sheet tasks tips (20 mins)
- Wrap up (poll & feedback; 5 mins)

Break-down of learning objectives

Lecture1

Introduction to the current HPC landscape

Understanding how this course fits into that

Establishing understanding of the learning outcomes, specifically answering the question: “What are programming during this course?”

Lecture2

Learning basic definitions and taxonomies

Understanding the importance of the “network”

Learning basic performance models

Understanding the concept of a well-performing software in large-scale computing.

Lecture3

Becoming knowledgeable of the modern landscape of distributed memory programming

Understanding why in this course we will concentrate on low-level programming models

Getting acquainted with MPI: basics and synchronous and asynchronous point-to-point communication

Break-down of learning objectives

Lecture4

Learning more about MPI:

One-sided point-to-point communications

Collective communications

Lecture5

Programming MP hybrid architectures

Becoming knowledgeable of the spectrum of options

Understanding efficiency issues

Lecture6

Programming hybrid architectures with accelerators

Acquiring knowledge of CUDA-MPI programming model

Repetition of key concepts

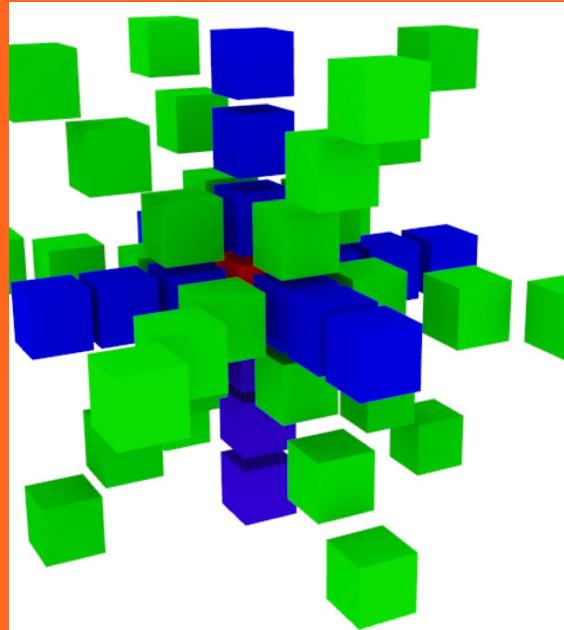
- old and new

Aim: to explain the key concept in short

Discuss 1-5 mins in row-wise groups

Randomly selected group(s) present(s) and
another one comment(s)

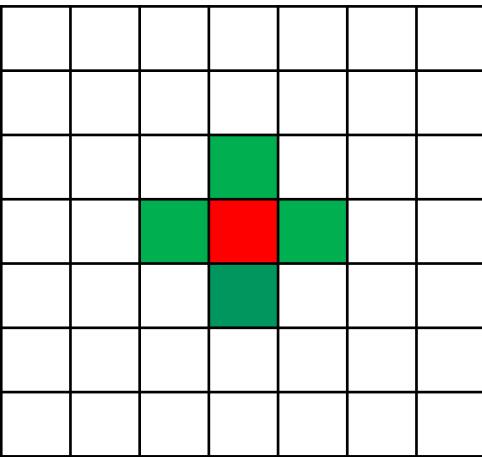
Model answer



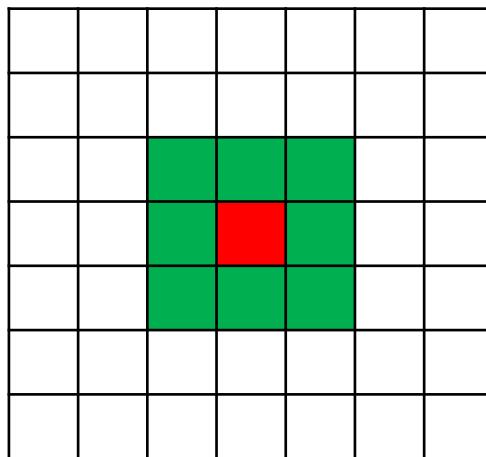
Iterative stencil loop

**Recurring update pattern of array elements
based on their neighbors.**

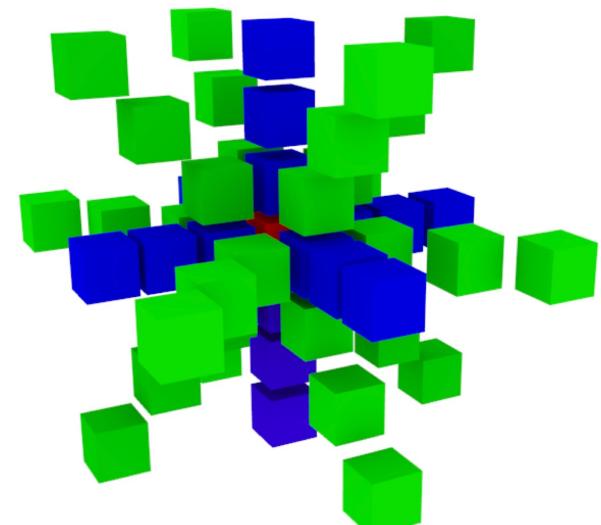
Iterative stencil loop



?



?



?

Draw the stencil of Sheet 3 ex. 2

$$\frac{\partial c}{\partial x}(x_i, y_j, t_n) \approx \frac{+3c_{i,j}^n - 4c_{i-1,j}^n + c_{i-2,j}^n}{2\Delta x} \quad \text{for } v_x > 0$$

$$\frac{\partial c}{\partial x}(x_i, y_j, t_n) \approx \frac{-c_{i+2,j}^n + 4c_{i+1,j}^n - 3c_{i,j}^n}{2\Delta x} \quad \text{for } v_x < 0,$$

... and the same for y velocity in j direction.

**Two-sided p2p communication;
Examples belonging/not belonging
(at least one per group)?**

Two parties – corresponding sender and receiver



What does "MPI messages are non-overtaking" mean?

If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending.

Does this apply to one-sided communication?

It depends. Many MPI_Put and MPI_Get mixed within an epoch do not guarantee the order and can lead to race conditions.

Are there ways to avoid race conditions in one-sided communication?

Yes!

Remember to synchronize

OR

**Use MPI_Accumulate with MPI_REPLACE
and MPI_Get_accumulate with MPI_NO_OP
which implement atomic operations for the
same target – origin pairs.**

What is the difference with rooted and all-to-all collectives?

**Results of reductions are collected to
ROOT's receive buffer versus to all ranks'
receive buffers**

MPI_Reduce vs. MPI_Allreduce

Are user defined ops allowed in RMA reductions?

Not yet.

MPI type	meaning	applies to\
MPI_MAX	maximum	integer, floating point
MPI_MIN	minimum	
MPI_SUM	sum	integer, floating point, complex, multilanguage types
MPI_REPLACE	overwrite	
MPI_NO_OP	no change	
MPI_PROD	product	
MPI_LAND	logical and	C integer, logical
MPI_LOR	logical or	
MPI_LXOR	logical xor	
MPI_BAND	bitwise and	integer, byte, multilanguage types
MPI_BOR	bitwise or	
MPI_BXOR	bitwise xor	
MPI_MAXLOC	max value and location	MPI_DOUBLE_INT and such
MPI_MINLOC	min value and location	

**Is there a way to make blocking two-sided
ops safe?**

**Not fully. Using MPI_Sendrecv, MPI_Ssend
and MPI_Bsend may help to write safe
code.**

Will blocking two-sided ops give you the optimum performance?

No, as these functions do not allow for concurrency in computation and communication.

Will non-blocking two-sided ops give you the optimum performance?

It depends: these functions allow for concurrency in computation and communication, but still require some implicit synchronization, buffers, and activity from both sender and receiver.

How could you replace Allgather with simpler MPI collective functions?

0	A			
1	B			
2	C			
3	D			

Allgather

0	A	B	C	D
1	A	B	C	D
2	A	B	C	D
3	A	B	C	D

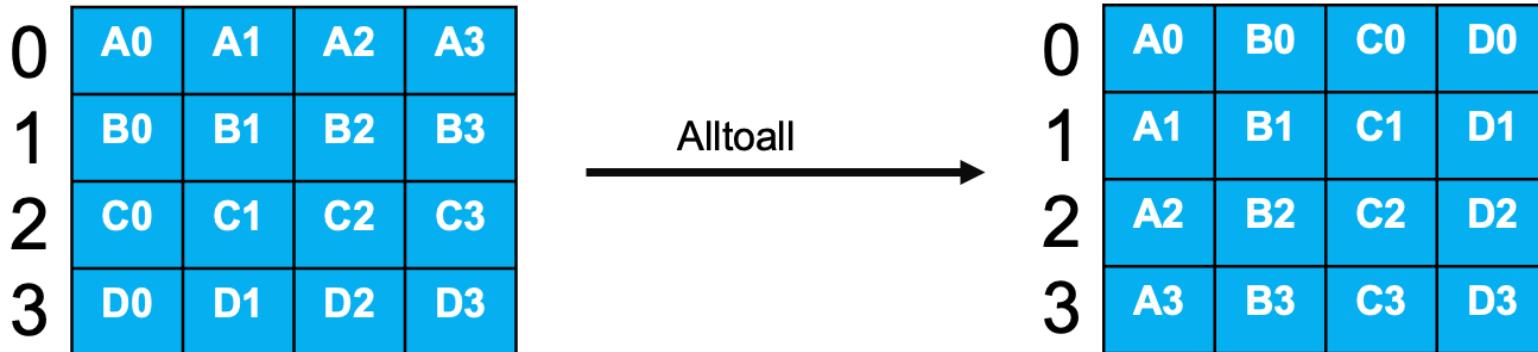
First Gather, and then Broadcast

0	A			
1	B			
2	C			
3	D			

Allgather →

0	A	B	C	D
1	A	B	C	D
2	A	B	C	D
3	A	B	C	D

What, in terms of matrix operations, is done here?



Transpose

0	A0	A1	A2	A3
1	B0	B1	B2	B3
2	C0	C1	C2	C3
3	D0	D1	D2	D3

Alltoall



0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3



Phew!

Great
work!

CS-E4690 – Programming parallel supercomputers

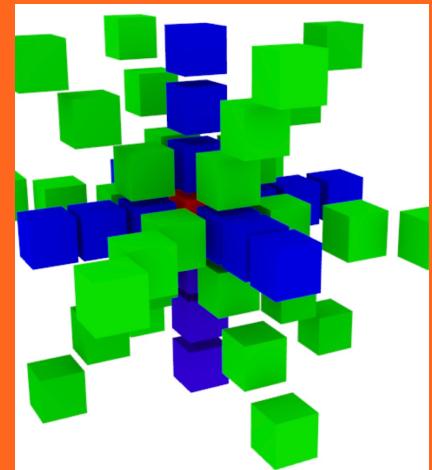
Hybrid computing in the CPU paradigm

Maarit Korpi-Lagg

maarit.korpi-lagg@aalto.fi



Aalto University
School of Science



Recap

The two trajectories resulting from the power wall

Multicore processors (core==CPU)

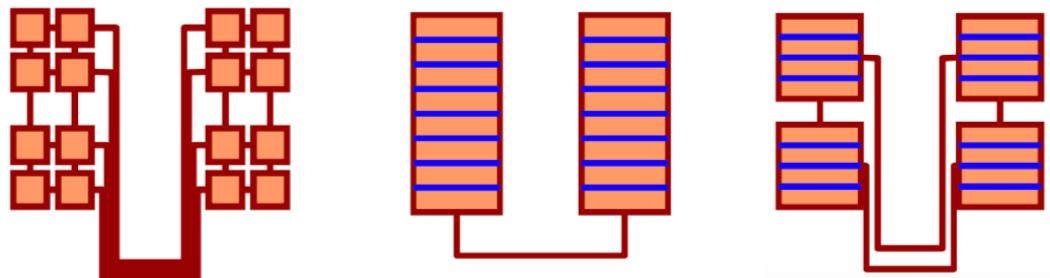
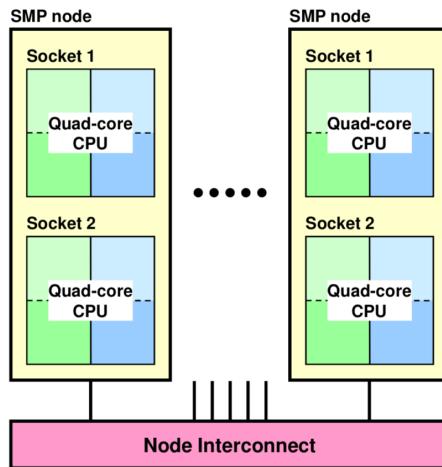
Lecture 5 (this material)

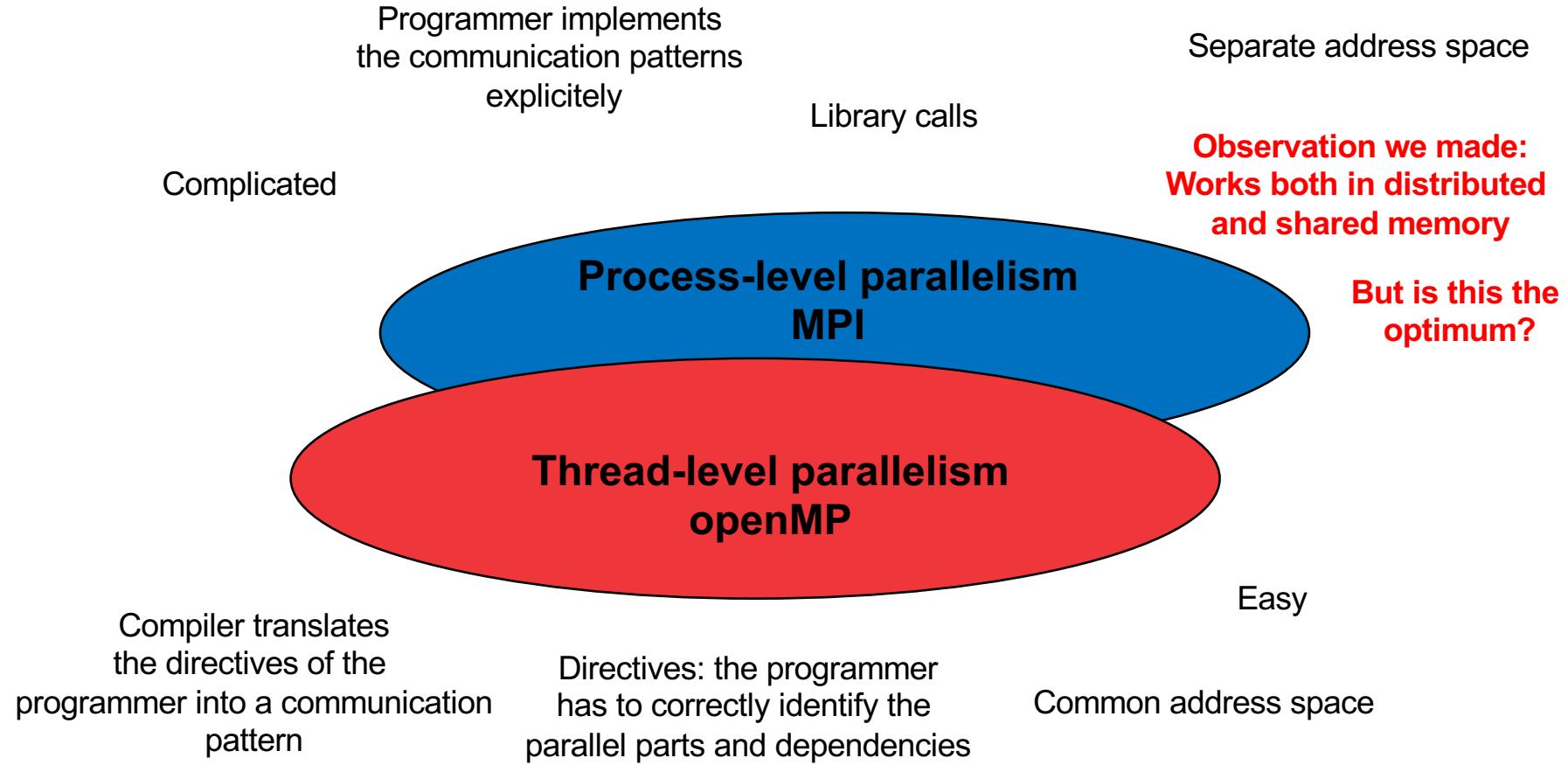
Multi-thread processors (e.g. processors with GPUs)

Lecture 6 (next week's material)

How to combine MPI distributed memory programming models with shared-memory ones?

One of the ultimate questions to answer to create efficient programs for the hybrid HPC platforms



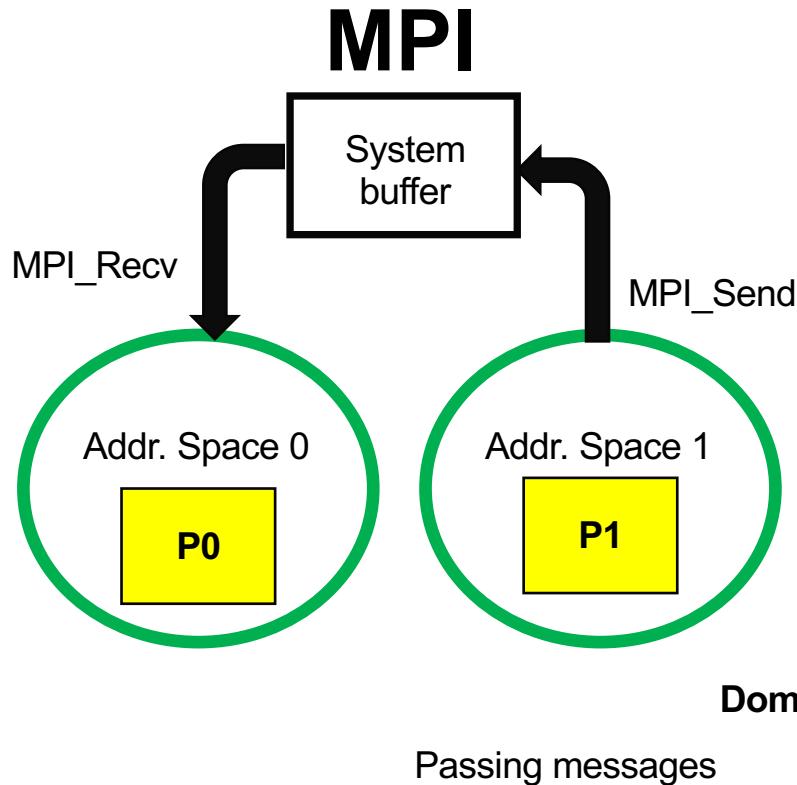


**To get yourself started
with/reminded about openMP,
recommended reading includes**

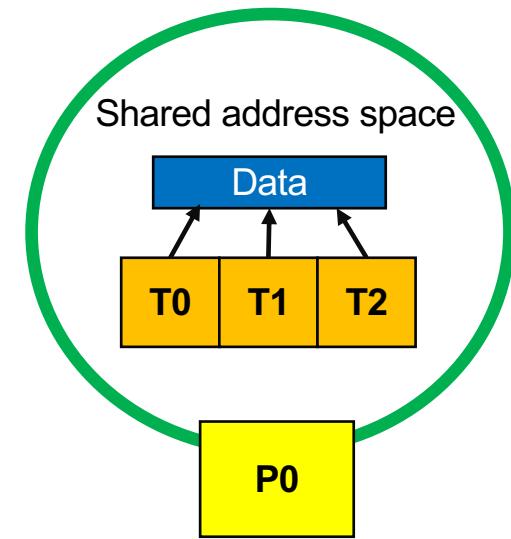
<https://ppc.cs.aalto.fi/ch3/>

More and docs
<https://www.openmp.org>

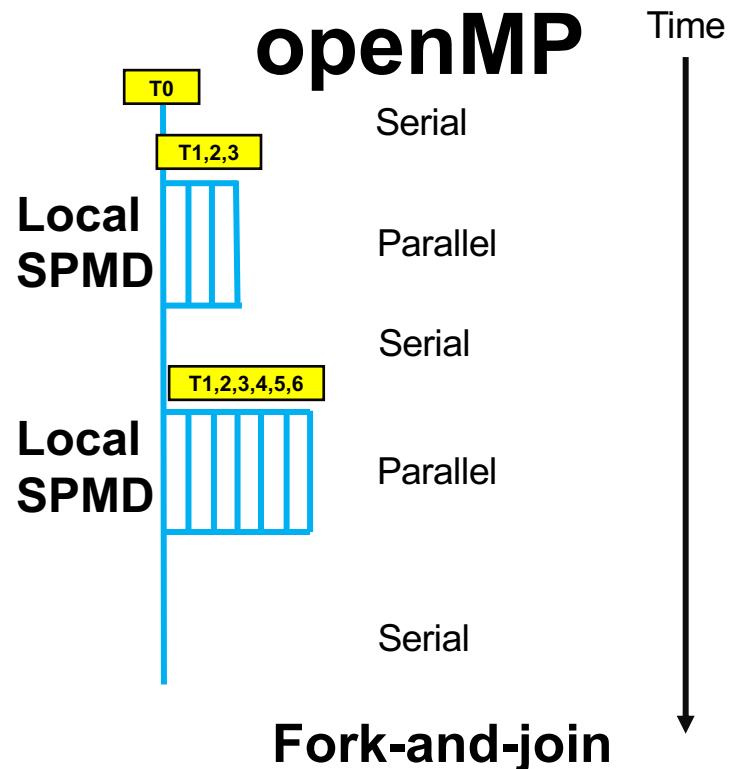
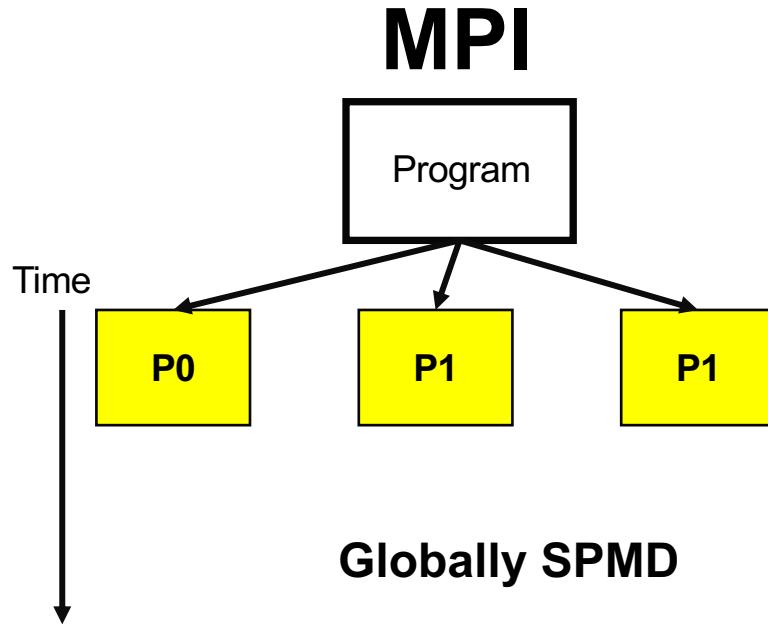
Memory models



openMP



Execution models



What are the (lower level) hybrid comp. options?

Pure MPI

MPI+ Shared
mem MPI

MPI+openMP

openMP

The mode that
has been used
so far....

Now provides
reference
cases

**Modes that are discussed today,
and tried out in Sheet 5**

Use shared mem.
MPI within a node
and MPI across
nodes

Use OpenMP
within a node and
MPI across nodes

Lecture 4: one-
sided p2p comms
material

No capacity to
investigate here,
but **please read**
[1], if you are
interested in trying
out.

Current consensus:
not the way to go
for distributed
memory comp.

What benefits are we expecting?

Two types of improvements can be envisaged

1. Reducing memory usage, both in the application and by the MPI library (e.g. decreased usage of communication buffers)
2. Improved performance and extended scale-up to higher number of CPU cores.

Memory consumption issues with MPI

Strong scaling scenario: if only shared memory, total consumption remains constant; with MPI there can be an increase due the replication (application) and buffering (system) of data.

Why is this a problem? Core issue: some applications are limited by the amount of memory per core (1-2GB nowadays); this is not going to increase dramatically in the future; better to try to optimize the memory consumption.

Halo sizes in
strong scaling case
with 2nd order
Moore stencil in 3D
periodic case



Aalto University
School of Science

Local domain size	Size of halos	Fraction of halos/domain size
$64^3 = 262,144$	$66^3 - 64^3 = 25,352$	10%
$32^3 = 32,768$	$34^3 - 32^3 = 6,536$	20%
$16^3 = 4,096$	$18^3 - 16^3 = 1,736$	42%

Goals?

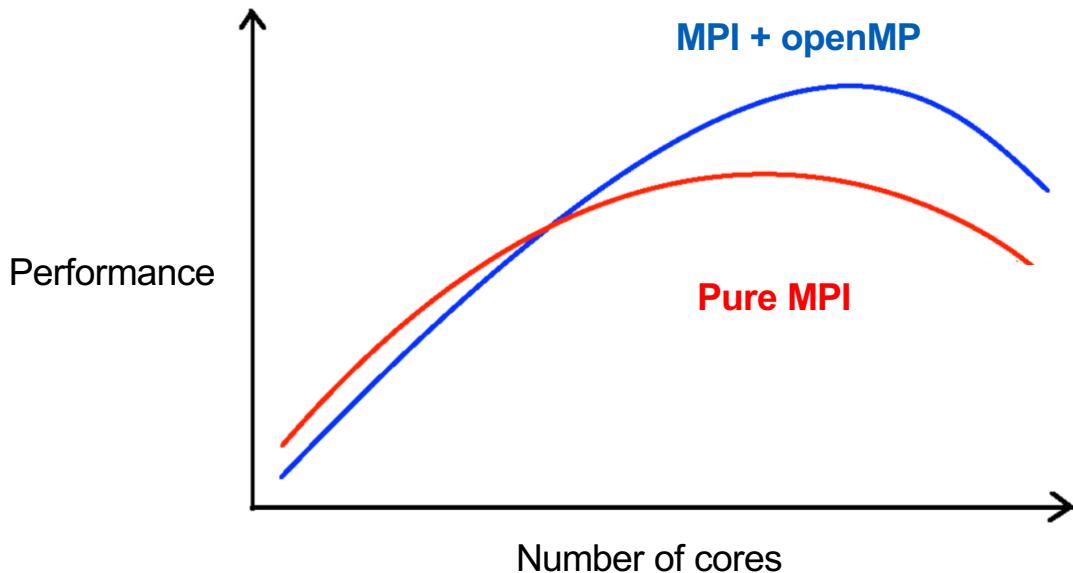
- **To reduce the total memory requirement;** larger problem sizes can then be computed with the same amount of cores
- **Reduced memory footprint per core may also give performance benefit, as data locality is improved:** Data can fit into cache, reducing the demand on memory bandwidth.

How could this be done?

Investigate whether the following strategy is possible:

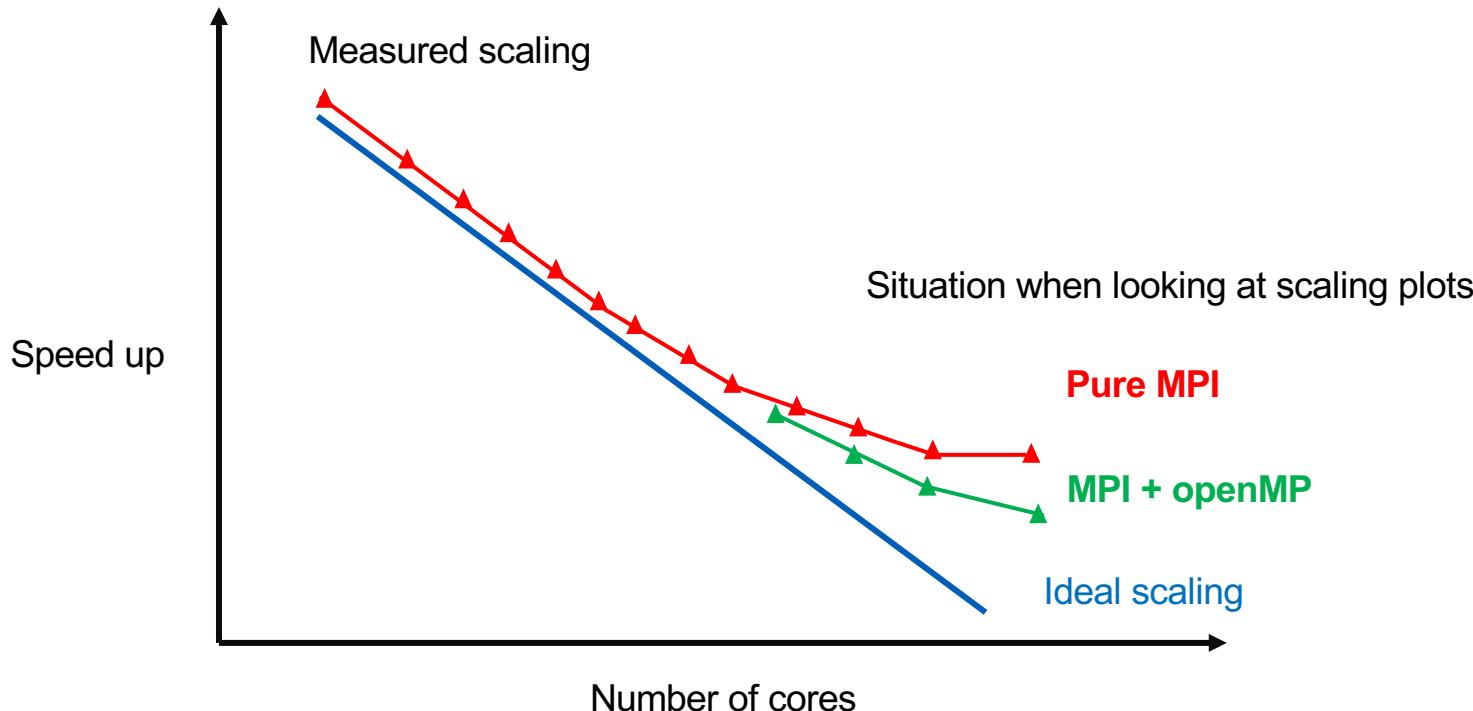
- Request less MPI processes on each node than there are cores.
- This results in some cores being idle
- Use openMP threads to make the idle cores work (**non-trivial, but possible**)

Performance



At low and intermediate core counts the performance of pure MPI is typically better than hybrid. At high core counts, parallelization overheads with pure MPI kill performance, but hybrid performance **can** overtake and the code **may continue** to perform to higher number of cores.

Scale-up



How can these benefits be achieved?

Investigate if there is a possibility to add lower-level parallelism into the application

Typical example: ISLs using MPI

Loop-level parallelism to be added

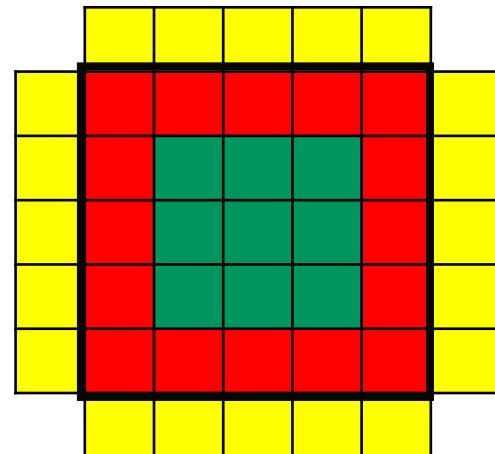
Repeat:

Initiate communication of yellow halos;

Do update of the green zones;

Wait for communications to finalize;

Update the red zones;



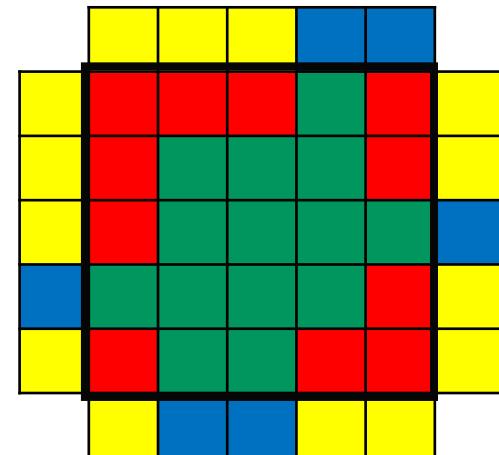
How can these benefits be achieved?

Investigate whether you can reduce communication overhead by removing redundant comm. operations

Typical example: dynamical ISLs;
conditional communication of part of the
halos required, but often implemented as
full

Do conditional communication in shared
memory programming model

Decrease number of MPI processes, give
that work to openMPI threads



OK, makes sense to implement

What to do in practise?

**Use shared
mem. MPI
within a
node and
MPI across
nodes**

**Use OpenMP
within a
node and
MPI across
nodes**

First, check if your MPI library supports threading

```
ompi_info | grep "Thread support"
```

Triton:

Thread support: posix (**MPI_THREAD_MULTIPLE:** yes, OPAL support: yes, OMPI progress: no, ORTE progress: yes, Event lib: yes)

**Hybrid/hello_class.c
scripts/job_hybrid_example.sh**

How to make MPI to co-operate with threads?

Instead of `MPI_Init()` one should call

```
int MPI_Init_thread(int *argc, char ***argv, int required, int  
*provided)
```

MPI_THREAD_SINGLE (0) Only one thread will execute (Equiv. of `MPI_Init()`). No openMP parallel regions in the code expected.

MPI_THREAD_FUNNELED (1) If the process is multithreaded, only the thread that called `MPI_Init_thread` will make MPI calls.

MPI_THREAD_SERIALIZED (2) If the process is multithreaded, only one thread will make MPI library calls at one time.

MPI_THREAD_MULTIPLE (3) If the process is multithreaded, multiple threads may call MPI at once with no restrictions.

Case MPI_THREAD_FUNNELED

All MPI calls are made by the openMP master thread OUTSIDE parallel regions, or inside openMP master regions.

```
int main(int argc, char ** argv) {  
    int data[100], provided;  
MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
#pragma omp parallel for  
for (i = 0; i < 100; i++)  
compute(data[i]);  
/* Do MPI stuff */  
MPI_Finalize();  
return 0; }
```

Master-only style, if calls
only outside parallel
regions

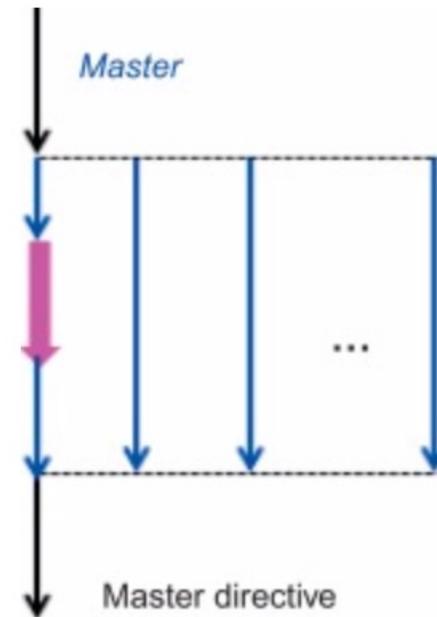
Master-only type programming

- All MPI calls **outside** openMP parallel regions
- Straightforward **fork-and-join** parallelism typical for openMP
- **Easy and safe:** Each parallel region imposes a synchronization, hence programmer does not have to worry about it. **High overhead.**
- During the MPI calls by master, all **other threads are idling**; using **derived data types** can be especially devastating, as the packing/unpacking of data is serialized
- **Poor data locality;** all data passes through the cache of the master thread

Funneled type programming

- MPI calls are made by OpenMP master thread, but take place inside OpenMP parallel “master” regions.

```
#pragma omp parallel {  
    ... work  
    #pragma omp barrier  
    #pragma omp master {  
        MPI_Send(...);  
    }  
    #pragma omp barrier ...  
    work  
}
```



Funneled type programming

- Two restrictions are relaxed in comparison to master-only programming:
 - there are now cheaper ways available to synchronise threads than opening and closing parallel regions
 - It possible for other threads to do useful computation while the master thread is executing MPI calls.

Serialized mode of programming

- Any thread inside an OpenMP parallel region may make calls to the MPI library, but the threads must be synchronised in such a way that only one thread at a time may be in an MPI call.

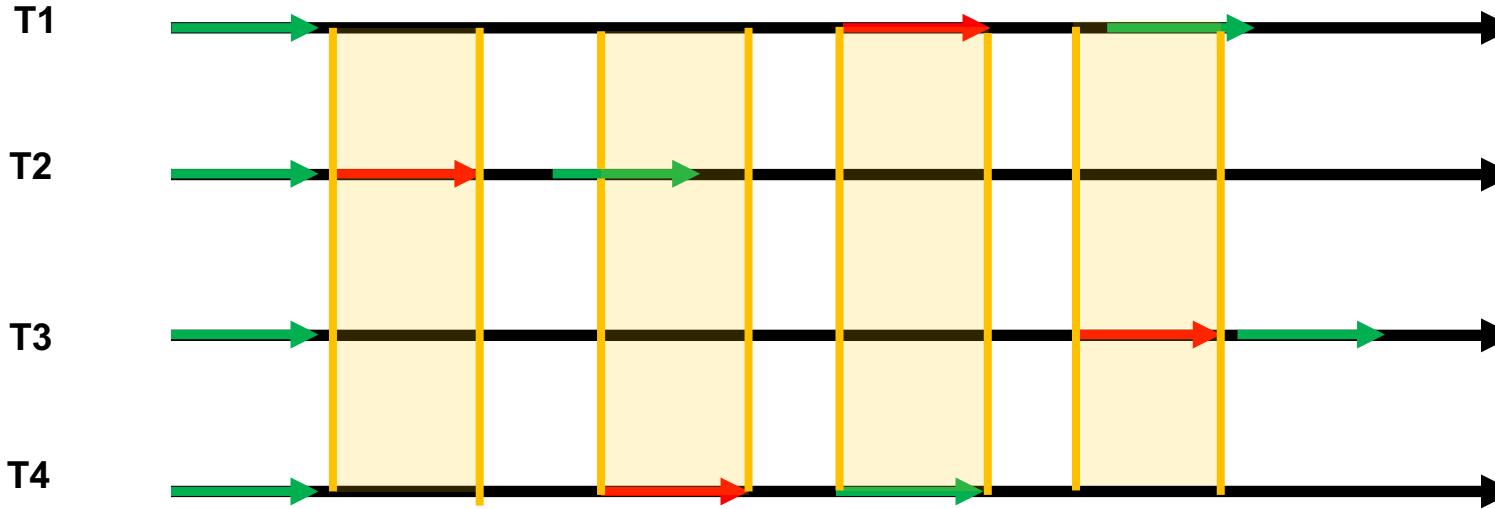
```
#pragma omp parallel {  
    ... work  
    #pragma omp critical {  
        MPI_Send(...); }  
    ... work }
```

Serialized mode of programming

- Threads can communicate their own data to other threads in other processes. This **improves locality**, since the message data is not all being cycled through one cache.
- It is now often necessary to **use tags or communicators** to distinguish between messages from (or to) different threads in the same MPI process. This is because the ordering of the sends and receives posted by different threads is non-deterministic.
- Ensuring threads do not enter MPI calls at the same time, by enclosing the MPI calls into **openMP critical regions**, may result in **idle threads**.

Serialized mode of programming

On a certain MPI rank of processes:



MPI calls are embedded in omp critical sections

Black: time
Green: computation
Red: communication
Orange: critical

Multiple style programming

- Any thread inside (or outside) an openMP parallel region may call MPI, and there are no restrictions on how many threads may be executing MPI calls at the same time.

```
#pragma omp parallel {  
    ... work  
    MPI_Send(...);  
    ... work  
}
```

- MPI assumes that it should take care of thread safety internally.
- Application code can become very inefficient. Efficient usage of this model requires advanced knowledge on openMP; skip but if interested, read [2].

OK, makes sense to implement

What to do in practise?

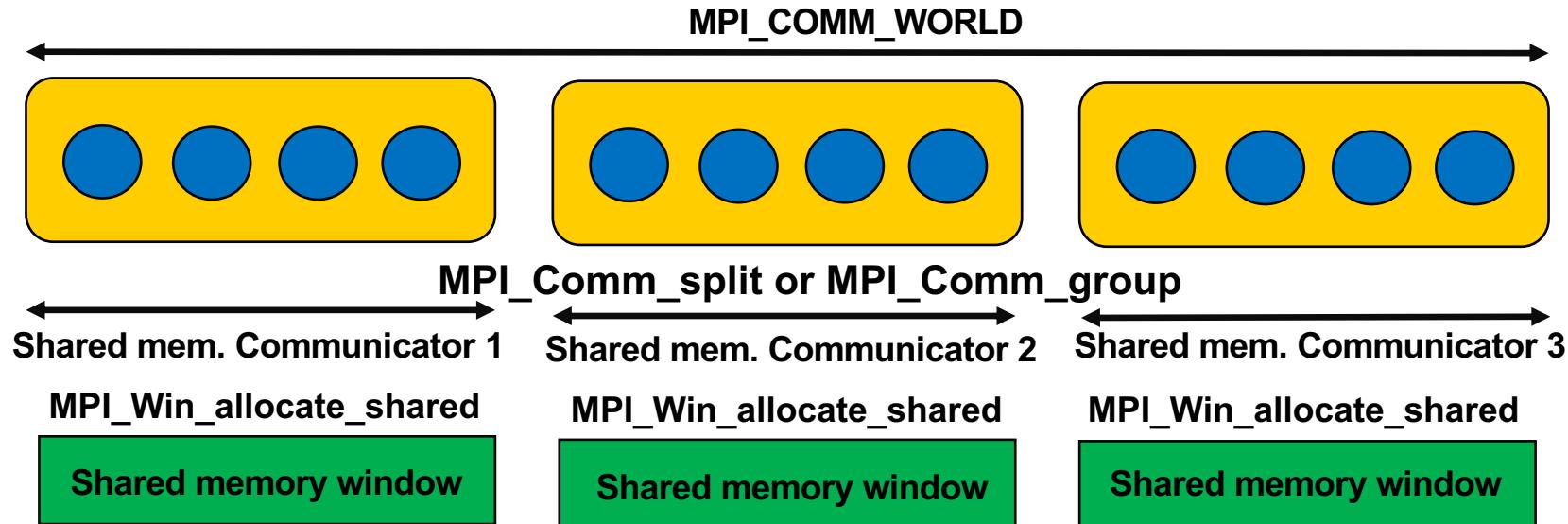
**Use shared
mem. MPI
within a
node and
MPI across
nodes**

**Use OpenMP
within a
node and
MPI across
nodes**

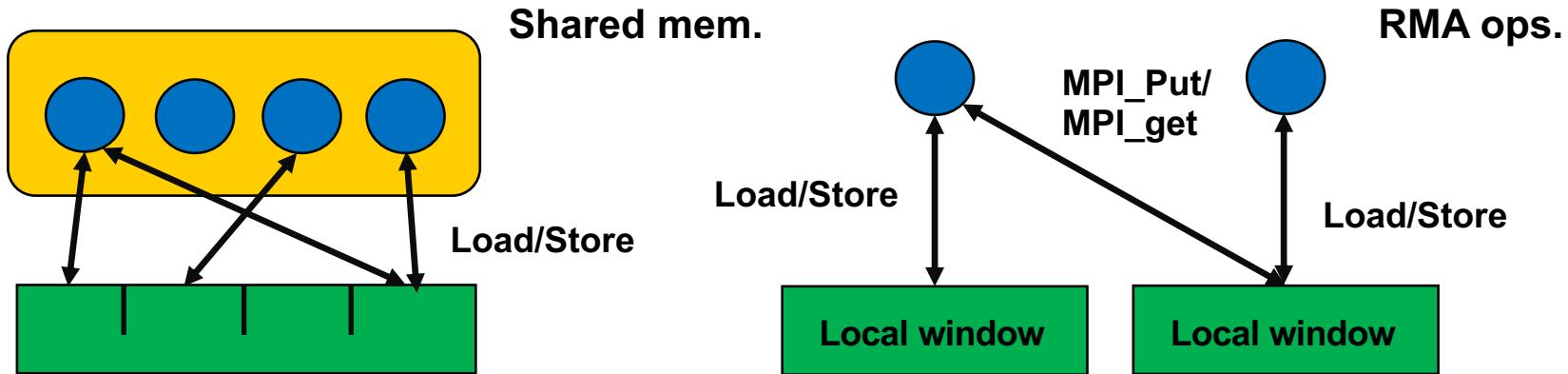
Example programs:
`Hybrid/OMP_MPI_X.c`

What is shared memory computing using MPI?

- “Standard” MPI mode for internode comms, shared memory mode for the intranode comms; altogether only one programming standard



Similarities and differences between RMA ops.



- No `MPI_Put/MPI_get` used in shared memory MPI mode; only loads/stores to the correct address of each core
- All RMA ops. are available, e.g. the atomic `MPI_Accumulate` and `MPI_Get_accumulate`
- Synchronization as in the RMA ops, e.g., fencing

OK, makes sense to implement

What to do in practise?

**Use shared
mem. MPI
within a
node and
MPI across
nodes**

Example programs:
[Hybrid/MPIs_MPI_X.c](#)

**Use OpenMP
within a
node and
MPI across
nodes**

Example programs:
[Hybrid/OMP_MPI_X.c](#)

Useful reading

[1] A. Basumallik, S. Min and R. Eigenmann, "Programming Distributed Memory Systems Using OpenMP," 2007 IEEE International Parallel and Distributed Processing Symposium, 2007, pp. 1-8, doi: 10.1109/IPDPS.2007.370397.

[2] http://www.intertwine-project.eu/sites/default/files/images/INTERTWinE_Best_Practice_Guide_MPI%2BOmpSs_1.0.pdf

Basics of openMP: <https://ppc.cs.aalto.fi/ch3/>

More and docs: <https://www.openmp.org>

CS-E4690 – Programming parallel supercomputers D

5th lecture

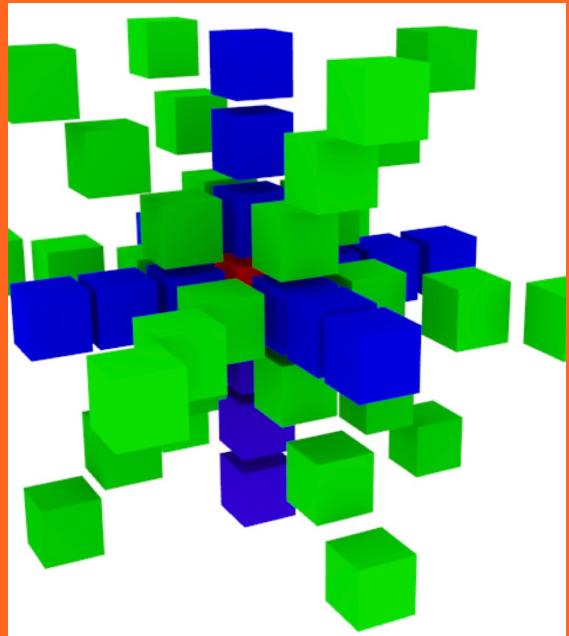
Maarit Korpi-Lagg

maarit.korpi-lagg@aalto.fi

21.11.2023



Aalto University
School of Science



Lecture 5

*Combining distributed memory
and shared memory programming
concepts*

- **Course updates:** 5 min
- **Quicksort stability card game:** 10 min
- **Key concepts recap - old and new (30 min)**
- **Break (max 15 mins)**
- **Example codes cntd. (10 mins)**
- **Exercise Sheet 5 tasks and tips (30 mins)**
- **Wrap up (Feedback; 5 mins)**

Break-down of learning objectives

Lecture1

Introduction to the current HPC landscape

Understanding how this course fits into that

Establishing understanding of the learning outcomes, specifically answering the question: “What are programming during this course?”

Lecture2

Learning basic definitions and taxonomies

Understanding the importance of the “network”

Learning basic performance models

Understanding the concept of a well-performing software in large-scale computing.

Lecture3

Becoming knowledgeable of the modern landscape of distributed memory programming

Understanding why in this course we will concentrate on low-level programming models

Getting acquainted with MPI: basics and synchronous and asynchronous point-to-point communication

Break-down of learning objectives

Lecture4

Learning more about MPI:

**One-sided point-to-point
communications**

**Collective
communications**

Lecture5

**Programming MP hybrid
architectures**

**Becoming
knowledgeable of the
spectrum of options**

**Understanding
efficiency issues**

Lecture6

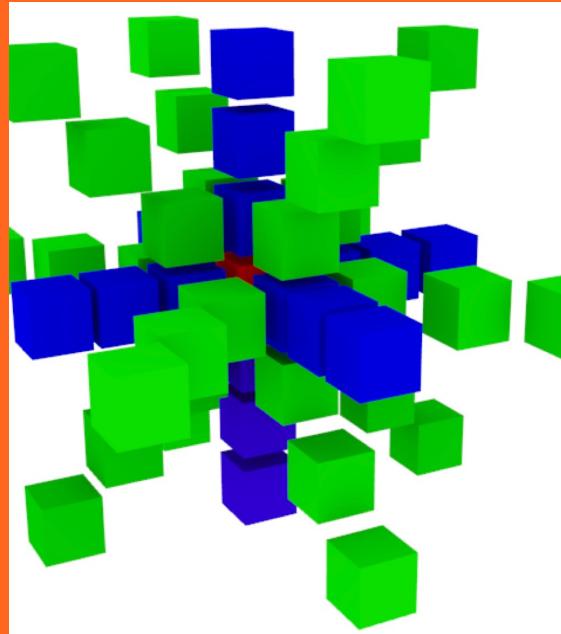
**Programming hybrid
architectures with
accelerators**

**Acquiring knowledge
of CUDA-MPI
programming model**

Course updates

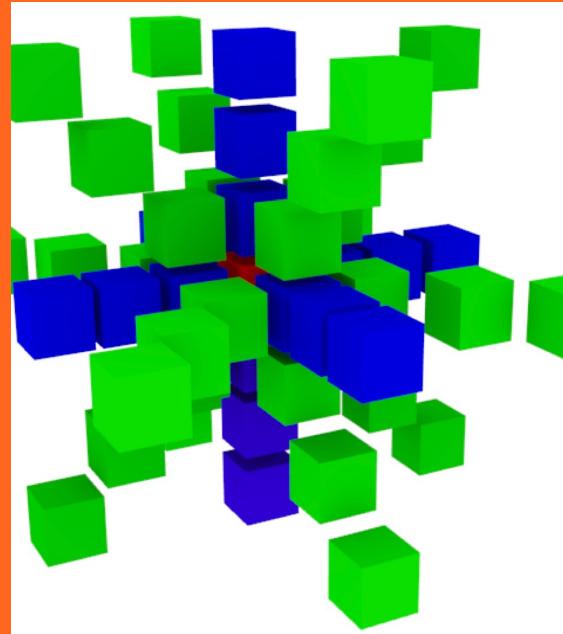


Aalto University
School of Science



Card game

Task: Demonstrate that Quicksort
is unstable



Quicksort stability card game;

Recurse (until all subarrays are sorted)

- Choose pivot (**P**) to be the leftmost index of (sub)array
- When pivot is on the left, compare it to the rightmost element; if **P>R**, swap and increment **L** index; otherwise decrement right index
- When pivot is on the right, compare it to the leftmost element; if **P<L**, swap and decrement **R** index; otherwise increment left index
- When **P, L, R** are pointing to the same index, divide into subarrays based on **P**, which value is now considered sorted (left out).

Try out at home! Make small pack of cards with at least two same numbers and record your findings about the position of the two

Repetition of key concepts - old and new

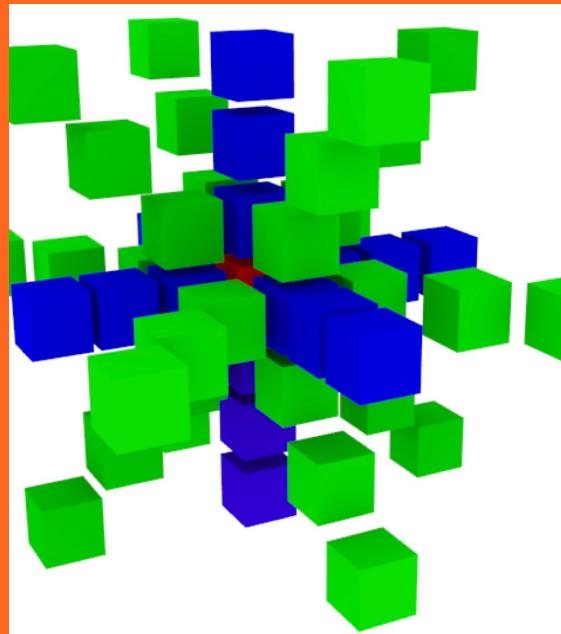
Aim: to explain the key concept in short

Discuss 1-5 mins in row-wise groups

Lecturer selects groups to answer and comment.

Model answer

Feedback with RED POST-ITS ONLY



A''

What is this?

Many correct answers.

Try to find at least two!

$$\begin{aligned} & I(x_i, y_j, t^n) \\ & \approx (I(x_{i-1}, y_j, t^n) + I(x_i, y_{j-1}, t^n) \\ & + I(x_{i-1}, y_{i-1}, t^n) + I(x_i, y_j, t^n) + I(x_{i+1}, y_j, t^n) \\ & + I(x_{i+1}, y_{j-1}, t^n) + I(x_{i+1}, y_{j+1}, t^n) \\ & + I(x_{i-1}, y_{j+1}, t^n) + I(x_i, y_{j+1}, t^n))/9 \end{aligned}$$

Iterative stencil loop (ISL)

Very basic image blurring operation

Nine-point stencil

Second order Moore stencil

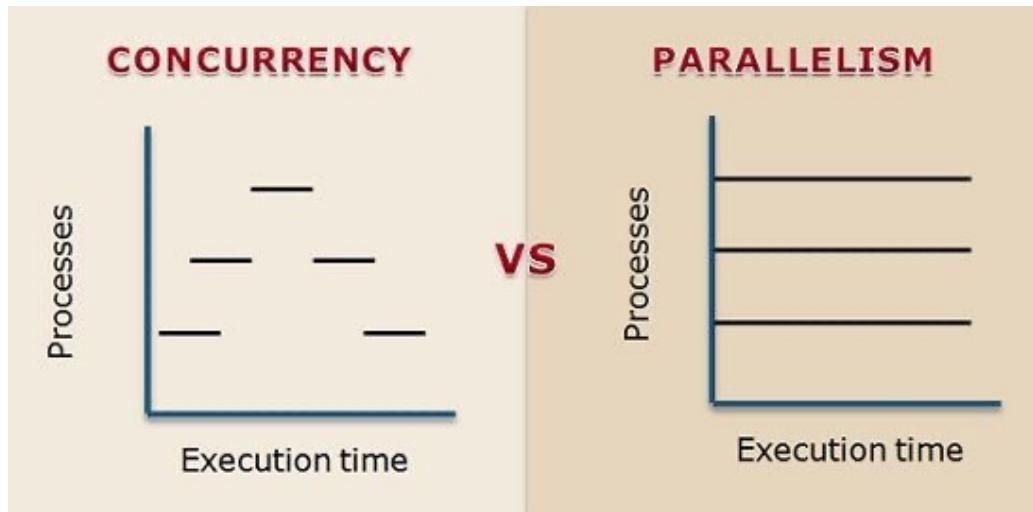
Symmetric stencil

Static stencil

**Discuss concurrency versus parallelism
with MPI; when and what?**

Parallelism: executing (nearly) similar computing tasks simultaneously.

Concurrency: executing different kinds of tasks overlapping with each other.



**Parallelism: Share (nearly) identical tasks
for many workers (SPMD model)**

**Concurrency: Overlap communication and
memory transactions with computation
(non-blocking communication, RMA
operations)**

What else than MPI could and should be used to optimize for parallelism and concurrency?

Multithreading, using accelerators, various types of co-processors, instruction-level parallelism, MPI RMA programming model within the nodes

Blue means topics that are dicussed and rehearsed in practise during the remaining of the course

Look at Hybrid/hello_class.c

From which line of the code does the parallel execution start/end with MPI?

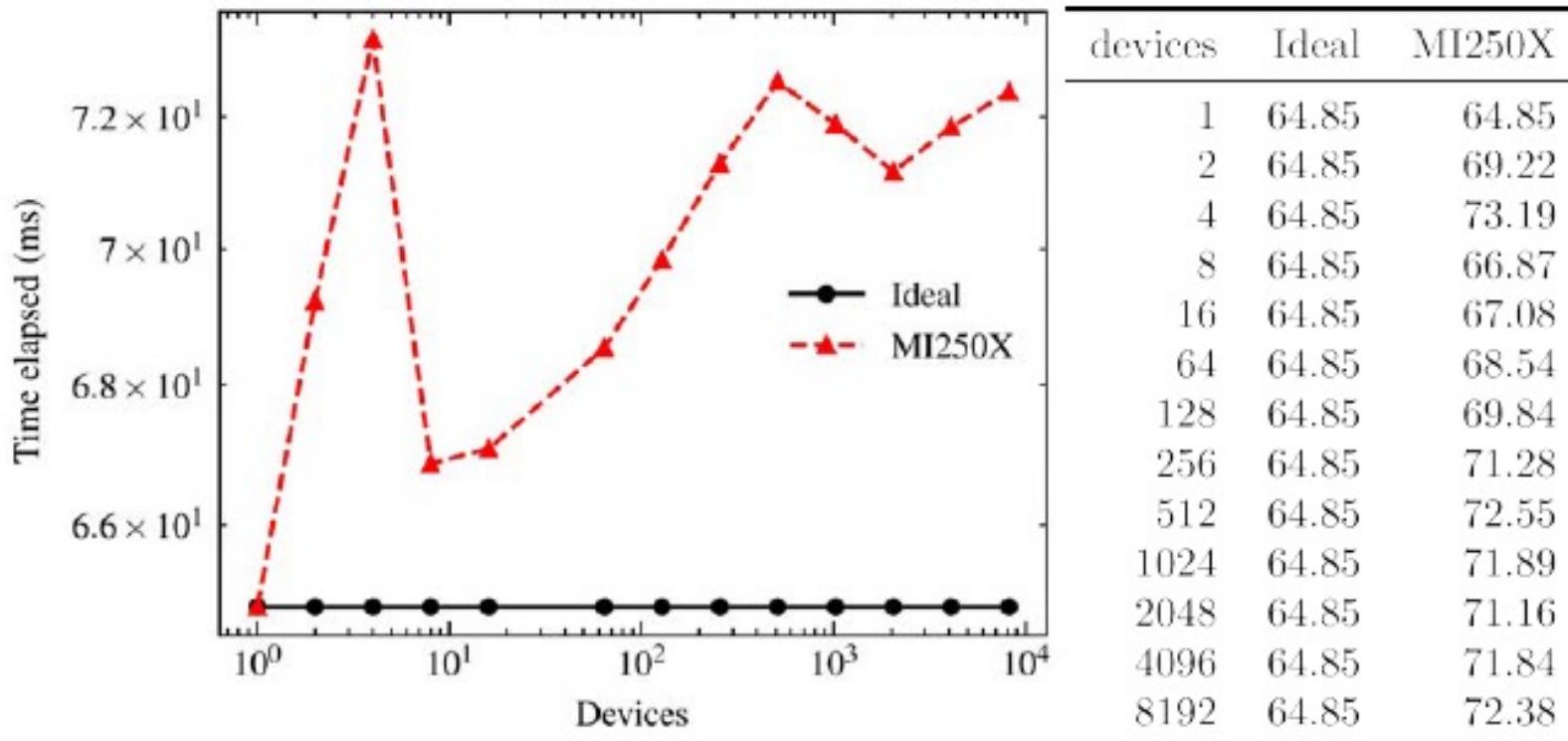
From which line of the code does the parallel execution start/end with openMP?

Look at Hybrid/hello_class.c

First/Last line.

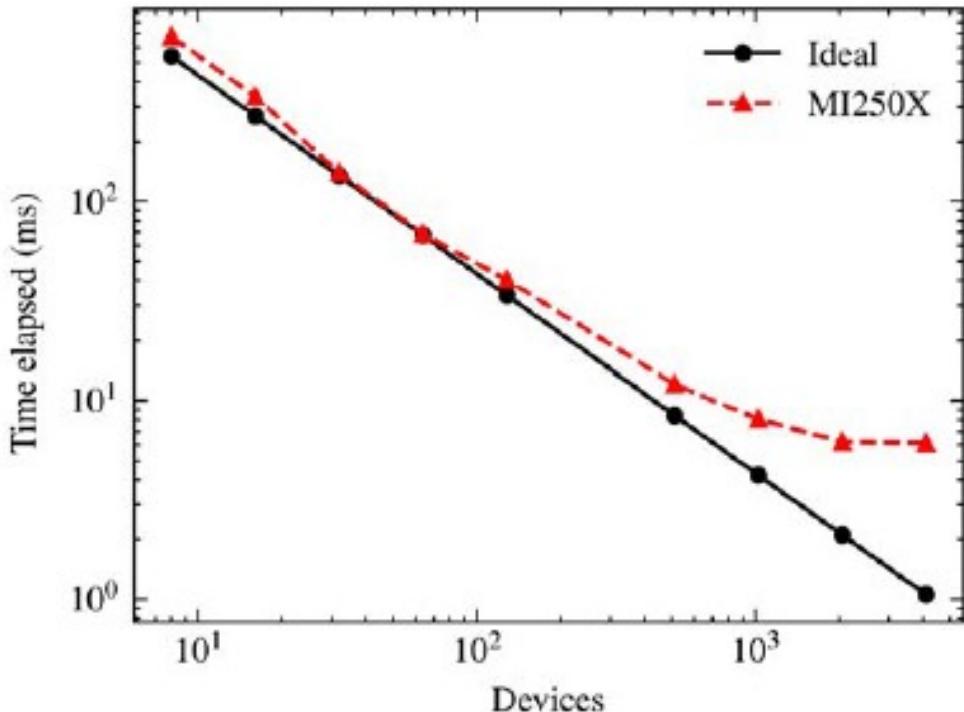
Line 38/40.

What does the data and graph illustrate?



Weak scaling

What does the data and graph illustrate?



devices	Ideal	MI250X
8	539.50	677.23
16	269.75	336.76
32	134.87	139.31
64	67.44	68.54
128	33.72	40.20
512	8.43	12.05
1024	4.21	8.07
2048	2.11	6.20
4096	1.05	6.12

Strong scaling

Which machine could this be?

Triton

LUMI

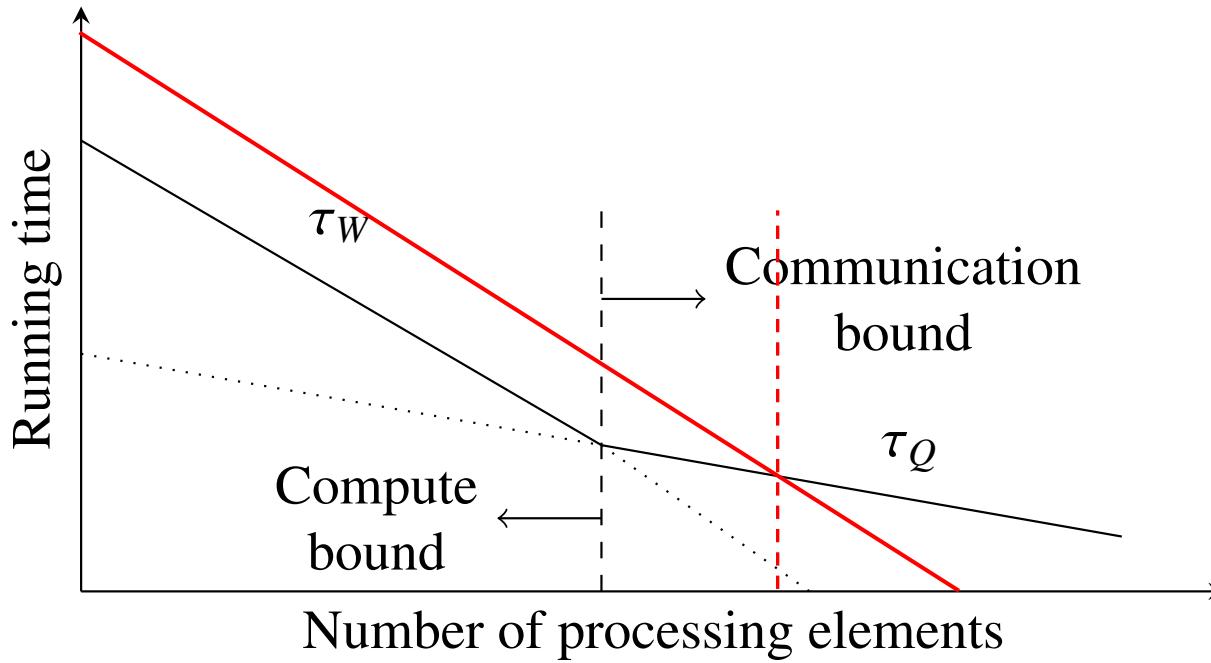
Mahti

Frontier

Puhti

LUMI or Frontier

What does the graph demonstrate?



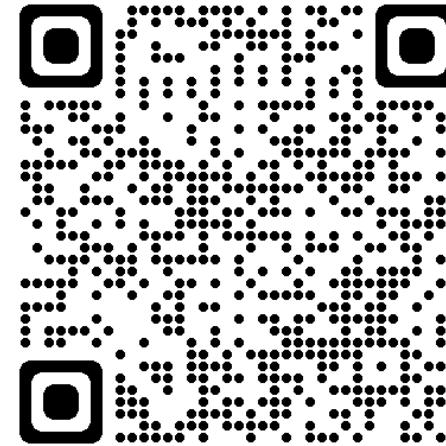
If speed of computations is increased, then concurrency will be limiting the scale-up even for smaller number of processes.

Remember Sheet 2 trends!

Wohoo!

Next week
SCITEC
teaching
evaluation
committee will
visit us.

Great
work!



You can give feedback
already now!

CS-E4690 – Programming parallel supercomputers

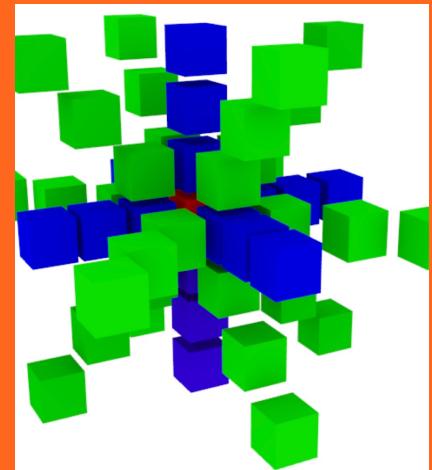
Hybrid computing using GPUs

Maarit Korpi-Lagg

maarit.korpi-lagg@aalto.fi



Aalto University
School of Science



Recap

The two trajectories resulting from the power wall

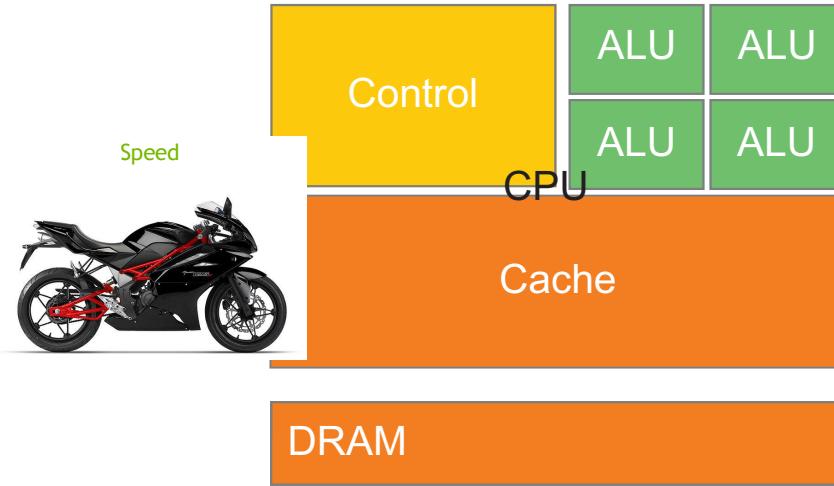
Multicore processors (core==CPU)

Lecture 5

Multi-thread processors (e.g. processors with GPUs)

This material

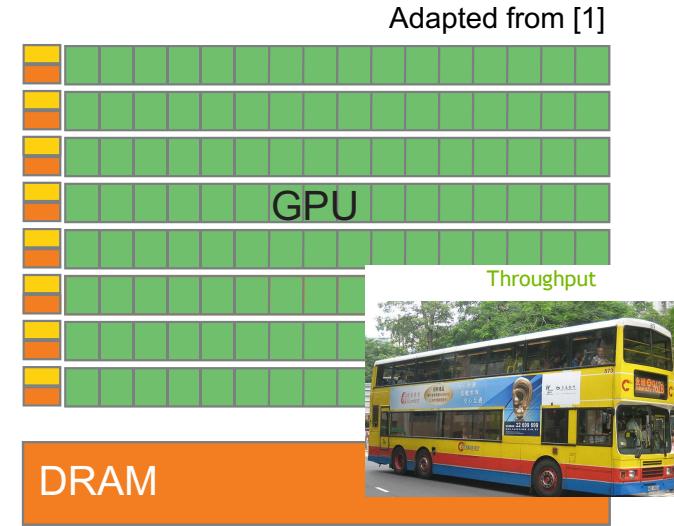
Schematic comparison



Optimising **sequential** execution

Peak throughput of GPUs about 10x higher than multi-core CPUs

DRAM access speed 10x higher for GPUs (made
for gaming)

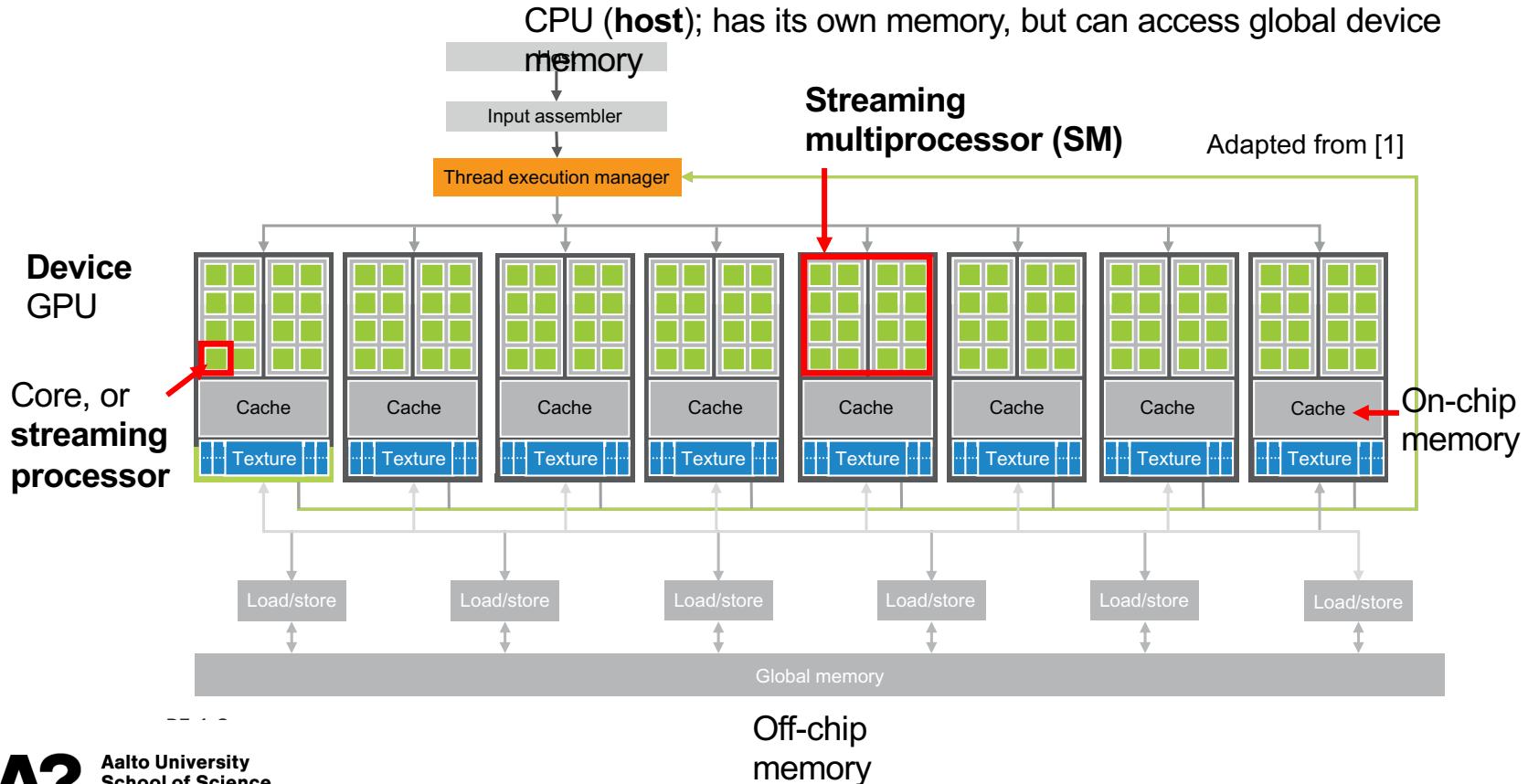


DRAM

Leading idea of **large-scale** computation

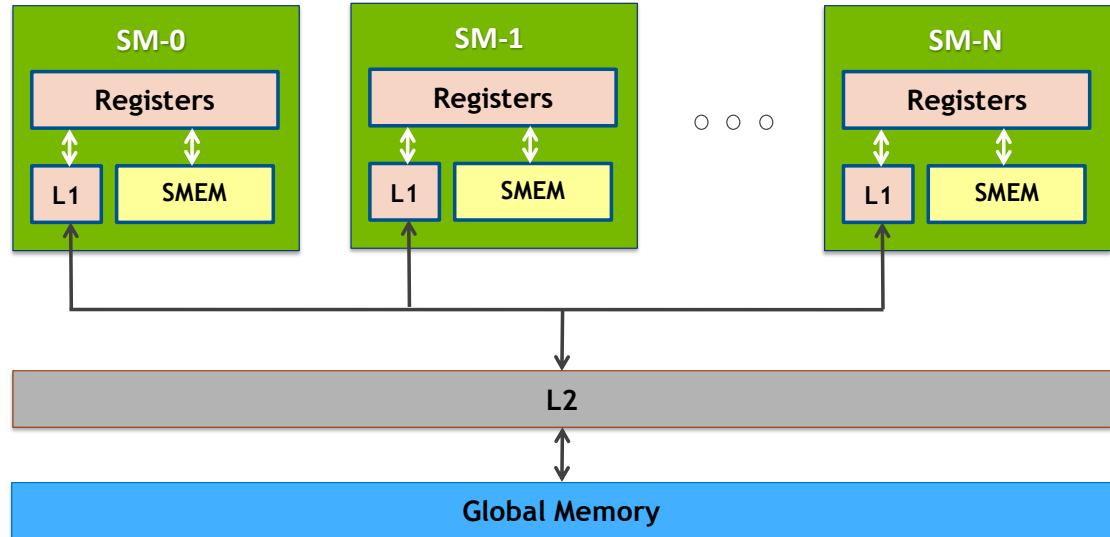
Execute sequential parts on CPUs
and parallel parts faster on GPUs;
Communications between GPUs
using MPI

Schematic model of a GPU



Memory hierarchy

- Memory transfers between host and **device global memory** have the highest latency (as bad as 100x the smem); to be minimized
- Access to **shared memory** and **registers** have much lower latency
 - Registers are seen by single threads
 - Shared memory is for fast communication between threads in a block
- The sizes of the shared memory and registers are very limited.



Chapter 4 of Programming parallel computers teaches you how to make efficient code by optimizing the memory usage; please read through

The GPUs in Triton

Card	total amount	nodes	architecture	compute threads per GPU	memory per card	CUDA compute capability	Slurm feature name	Slurm gres name
Tesla K80*	12	gpu[20-22]	Kepler	2x2496	2x12GB	3.7	kepler	teslak80
Tesla P100	20	gpu[23-27]	Pascal	3854	16GB	6.0	pascal	teslap100
Tesla V100	40	gpu[1-10]	Volta	5120	32GB	7.0	volta	v100
Tesla V100	40	gpu[28-37]	Volta	5120	32GB	7.0	volta	v100
Tesla V100	16	dgx[1-7]	Volta	5120	16GB	7.0	volta	v100
Tesla A100	28	gpu[11-17]	Ampere	7936	80GB	8.0		a100
gpuamd1	1	Dell PowerEdge R7525	2021	rome avx avx2 mi100	2x8 core AMD EPYC 7262 @3.2GHz	250GB DDR4-3200	EDR	3x MI100 32GB

A?
Aa
Sc

The GPUs in Triton

Card	total amount	nodes	architecture	compute threads per GPU	memory per card	CUDA (*) compute capability	Slurm feature name	Slurm gres name
Tesla K80*	12	gpu[20-22]	Kepler	2x2496	2x12GB	3.7	kepler	teslak80
Tesla P100	20	gpu[23-27]	Pascal	3854	16GB	6.0	pascal	teslap100
Tesla V100	40	gpu[1-10]	Volta	5120	32GB	7.0	volta	v100
Tesla V100	40	gpu[28-37]	Volta	5120	32GB	7.0	volta	v100
Tesla V100	16	dgx[1-7]	Volta	5120	16GB	7.0	volta	v100
Tesla A100	28	gpu[11-17]	Ampere	7936	80GB	8.0		a100
gpuamd1	1	Dell PowerEdge R7525	2021	rome avx avx2 mi100	2x8 core AMD EPYC 7262 @3.2GHz	250GB DDR4-3200	EDR	3x MI100 32GB

A?
As
Sc

(*) https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications

Kepler architecture

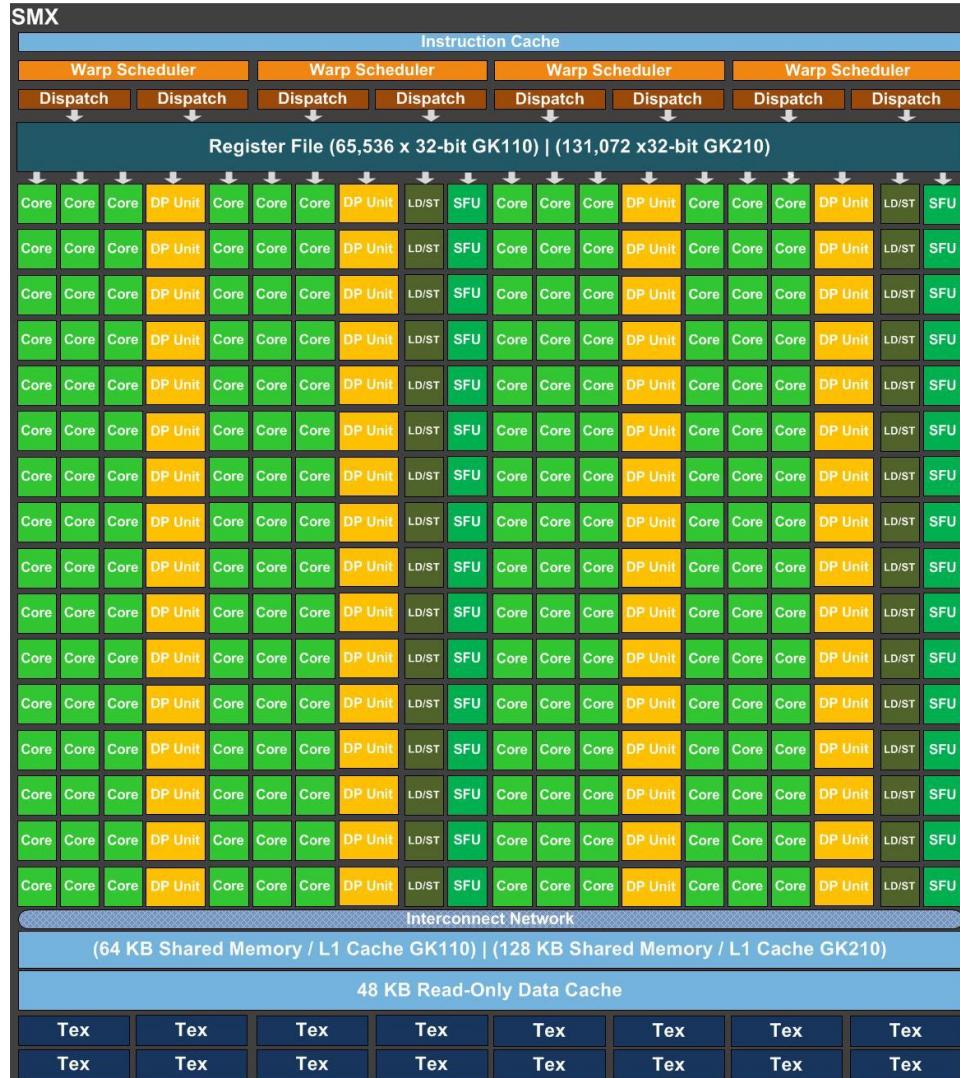


A?

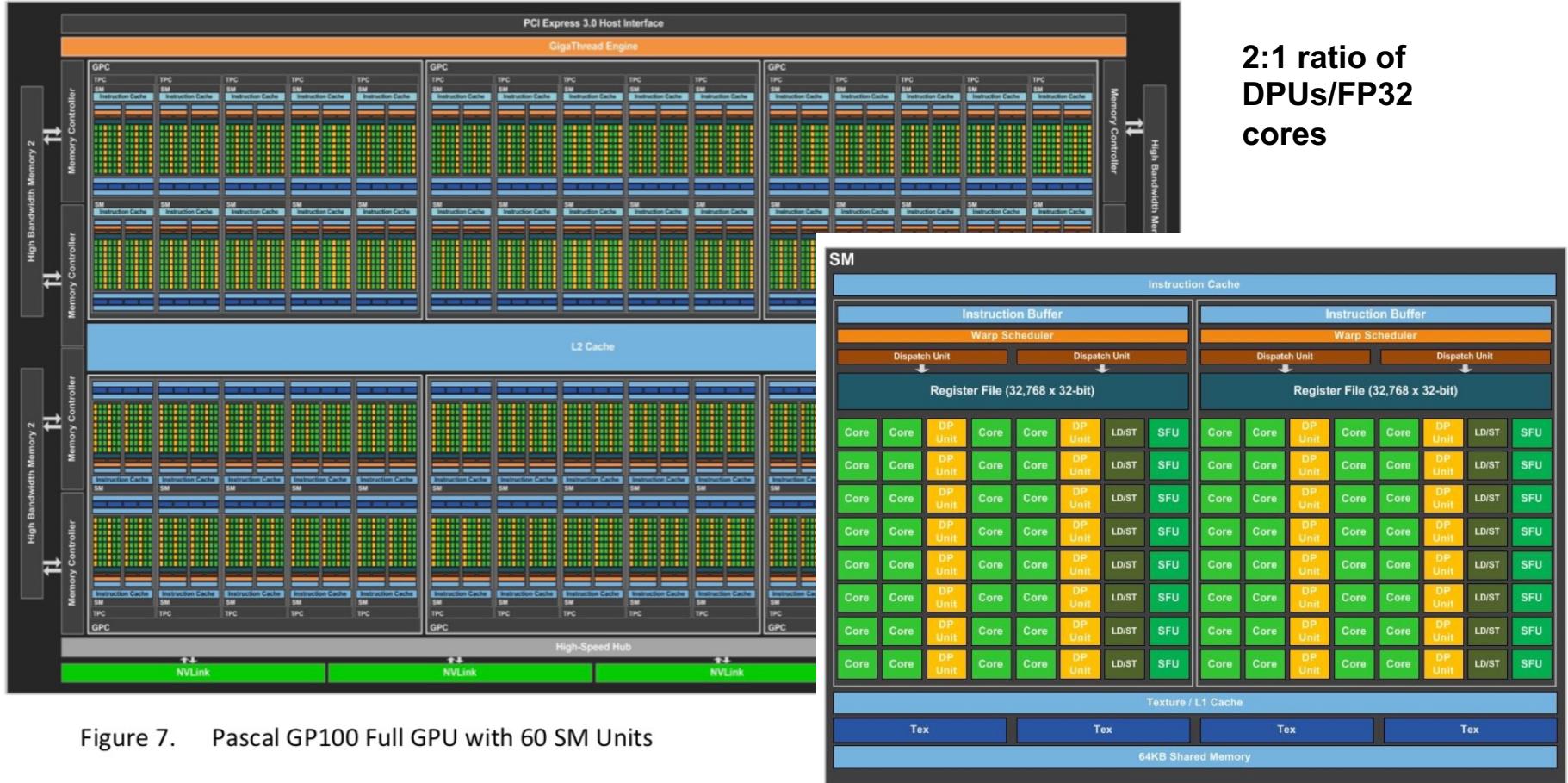
Aalto Uni
School of

Kepler SM

- Each SM has its own control units, registers, execution pipelines, caches
- Many cores per SM; how many is architecture dependent
- Special-function units (cos/sin/tan, etc.)
- Shared memory/L1 cache
- Thousands of 32-bit registers
- Double precision units with architecture variable ratio; in Kepler 3:1, nowadays more DPUs.



Pascal architecture



Programming models

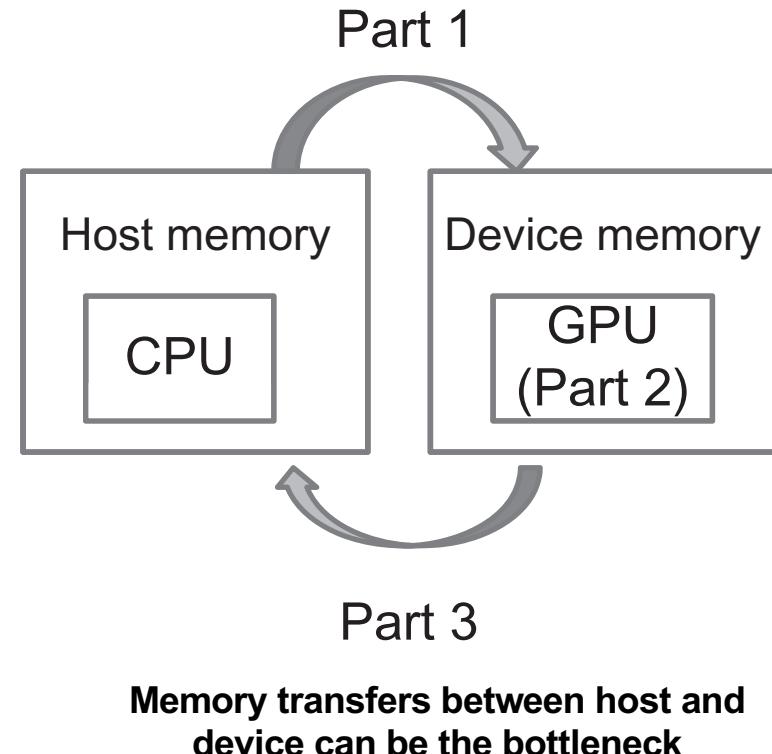
- CUDA (NVIDIA)
- Radeon Open Compute (ROCm) (AMD)
- HIP
- OpenCL
- OpenACC OpenMP
- ...

For openCL examples, please refer to

<https://ppc.cs.aalto.fi/ch4/v0opencl/>

CUDA Execution model

- Main program is executed by the CPU
- CPU needs to communicate with the GPU (Part I)
 - Upload the data to the GPU memory
 - Upload program to the GPU
- Wait (or do something useful) for the GPU to finish computations (Part 2)
- Fetch the results back from the GPU memory (Part 3)



CUDA programming model

Block

- **threads** that run on the same streaming multiprocessor (SM) form **blocks**;
- they communicate with each other through **shared memory** located on the SM;

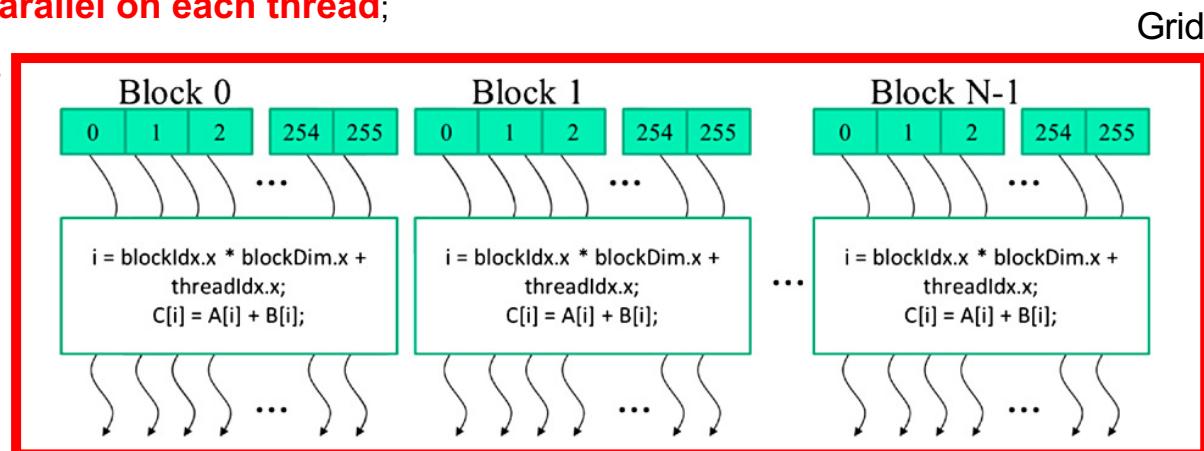
Grid

- Blocks are grouped into a **grid**; both threads and blocks have a **unique identification number**

Kernel

- Is a function that gets **executed in parallel on each thread**;
- Are executed as a grid of thread blocks

How to Identify who is who and operating on which part of the data?



CUDA programming model

Block

Phone number

- **threads** that run on the same streaming multiprocessor (SM) form **blocks**;
- they communicate with each other through **shared memory** located on the SM;

Grid

Phonebook

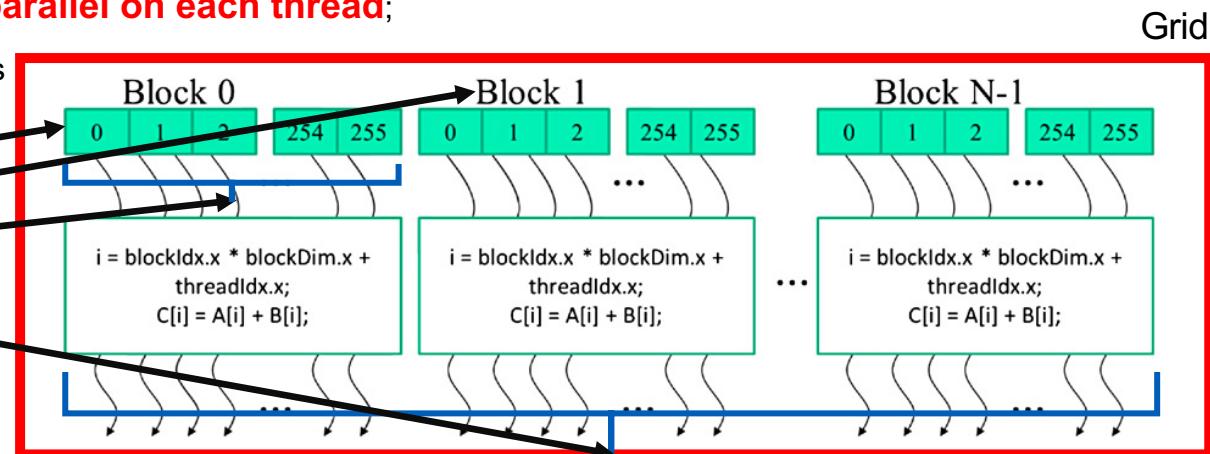
- Blocks are grouped into a **grid**; both threads and blocks have a **unique identification number**

Kernel

Call the number

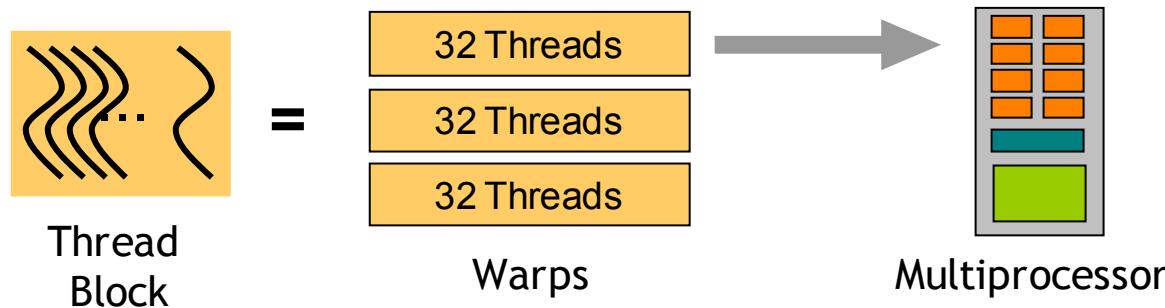
- Is a function that gets **executed in parallel on each thread**;
- Are executed as a grid of thread blocks

threadIdx.x
blockIdx.x
blockDim.x
gridDim.x



CUDA concept of warps

- Thread blocks are divided into warps; can be implementation dependent. In NVIDIA GPUs warps have 32 threads, in AMD's 64.
- Warps are physically executed in parallel on the SMs in “SIMD”-like manner.



Programming model in practise

Let us illustrate the difference of a normal C program and a CUDA one by adding together two numbers

C program (full)

```
// Compute vector sum h_C =  
// h_A+h_B  
void vecAdd(float* h_A, float*  
h_B, float* h_C, int n)  
{  
    for (int i = 0; i < n; i++)  
        h_C[i] = h_A[i] + h_B[i];  
  
    int main() {  
        // Memory allocation for h_A,  
        // h_B, and h_C // I/O to read h_A  
        // and h_B, N elements each ...  
        vecAdd(h_A, h_B, h_C, N);  
    }
```

Cuda (host code)

```
// Compute vector sum h_C = h_A+h_B  
void vecAdd(float* h_A, float* h_B, float* h_C, int  
n) // "h_" refers to host  
{  
    int size = n * sizeof(float);  
    float *d_A, *d_B, *d_C; // Pointers to device mem, hence start  
    // with "d_"  
    cudaMalloc((void **) &d_A, size); // Allocating device mem  
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);  
    // Copying data over to device mem  
    cudaMalloc((void **) &d_B, size); // Same stuff for B  
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);  
    cudaMalloc((void **) &d_C, size); // Allocation C that'll hold the  
    // result  
    vecAddKernel<<<256,256>>>(d_A, d_B, d_C, n);  
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);  
    // Copying result to host  
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);  
}
```

Prgramming model in practise

CUDA (device code)

Kernel function

```
// Compute vector sum C = A+B  
// Each thread performs one pair-wise addition
```

global

```
void vecAddKernel(float* A, float* B, float* C, int n) {  
    int i = blockDim.x*blockIdx.x + threadIdx.x;  
    if(i<n) C[i] = A[i] + B[i];  
}
```

blockDim.x=dimension of the blocks requested

blockIdx.x=Block ID amongst all blocks reserved

threadIdx.x=Unique identified of the thread in a block

Step by step autopsy of the CUDA code

Allocation of global device memory

cudaMalloc((void) &DevPtr, size_t size)**

- ° Address of a pointer to the allocated object in device memory
- ° Size of allocated object in terms of bytes

cudaFree(DevPtr)

Frees object from device global memory

- ° Pointer to freed object

Step by step autopsy of the CUDA code

Transferring data to/from device global memory

cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)

- o Pointer to destination
- o Pointer to source
- o Number of bytes copied
- o Type/Direction of transfer:
 - cudaMemcpyHostToDevice**
 - cudaMemcpyDeviceToHost**

Step by step autopsy of the CUDA code

Calling the kernel function

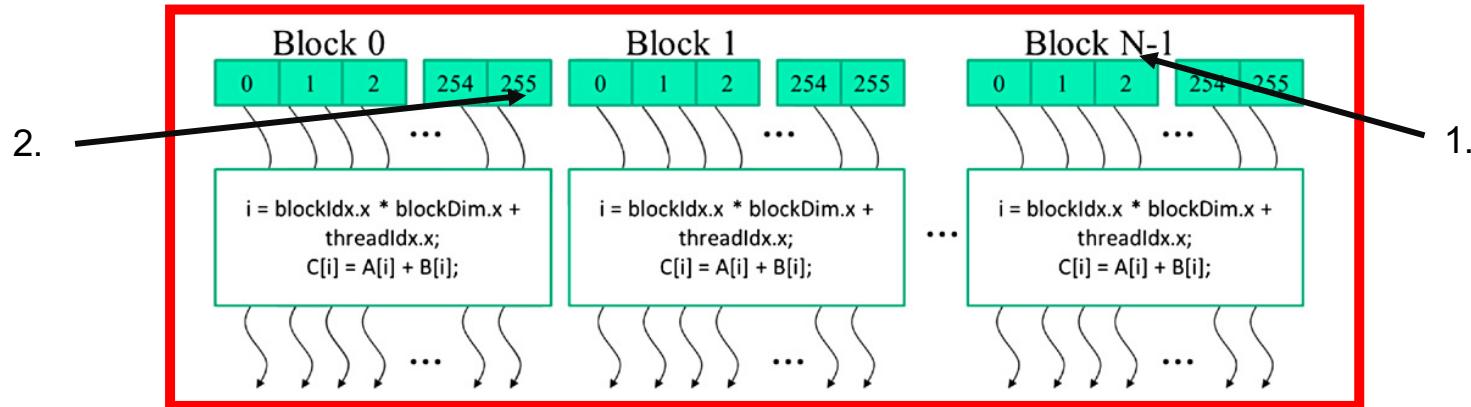
```
vecAddKernel<<<256,256>>>(d_A,d_B,d_C, n);
```

execution configuration parameters

Traditional C function arguments

Number of BLOCKS. Now hardcoded, but in reality should depend on n, e.g. using `ceil(n/256.0)`

Number of THREADS in a Block. Max number of threads is 1024
Grid



1. exec. config param.: Number of blocks
2. exec. config param.: Number of threads

Again works like a phonenumbers: areacode-number

Step by step autopsy of the CUDA code

Construction of the kernel function

Cuda (device code)

Kernel function

The order of execution is random

// Compute vector sum C = A+B
// Each thread performs one pair-wise addition

global

```
void vecAddKernel(float* A, float* B, float* C, int n) {  
    int i = blockDim.x*blockIdx.x + threadIdx.x;  
    if(i<n) C[i] = A[i] + B[i];  
}
```

blockDim.x=dimension of the blocks requested

blockIdx.x=Block ID amongst all blocks reserved

threadIdx.x=Unique identified of the thread in a block

Two *built-in variables* that enable threads to identify themselves amongst others and know their own data area.

With the ceil function we might have reserved extra threads, hence now we need to prevent their execution with this if

CUDA C keywords for function declaration.

		Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>		device	device
<code>__global__ void KernelFunc()</code>		device	host
<code>__host__ float HostFunc()</code>		host	host

vecAdd is a bad candidate for a CUDA code

- You do **a lot of data transfers** between the host and device
- **Very little computations**
- Your CUDA code will be performing worse than a sequential code; there should always be more to compute than communicate to make a reasonable application on GPUs.
Remember the ACC model!
- You are **REALLY** encouraged try this out.

[GPU/vecAdd_CPU.c](#)
[GPU/vecAdd_GPU.cu](#)

Generalization to multidimensional grids

The autopsied example case was dealing with **one-dimensional** thread blocks. Generally, however, the exec. config params

```
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

dimGrid and **dimBlock** are **dim3** type, which is a C struct with three unsigned integer fields: **x**, **y**, and **z** specifying the sizes of the three dimensions. Less than three dimensions are chosen by setting the size of the unused dimensions to 1.

```
dim3 dimGrid(2, 2, 1);
```

```
dim3 dimBlock(4, 2, 2);
```

Generalisation to multidimensional grids

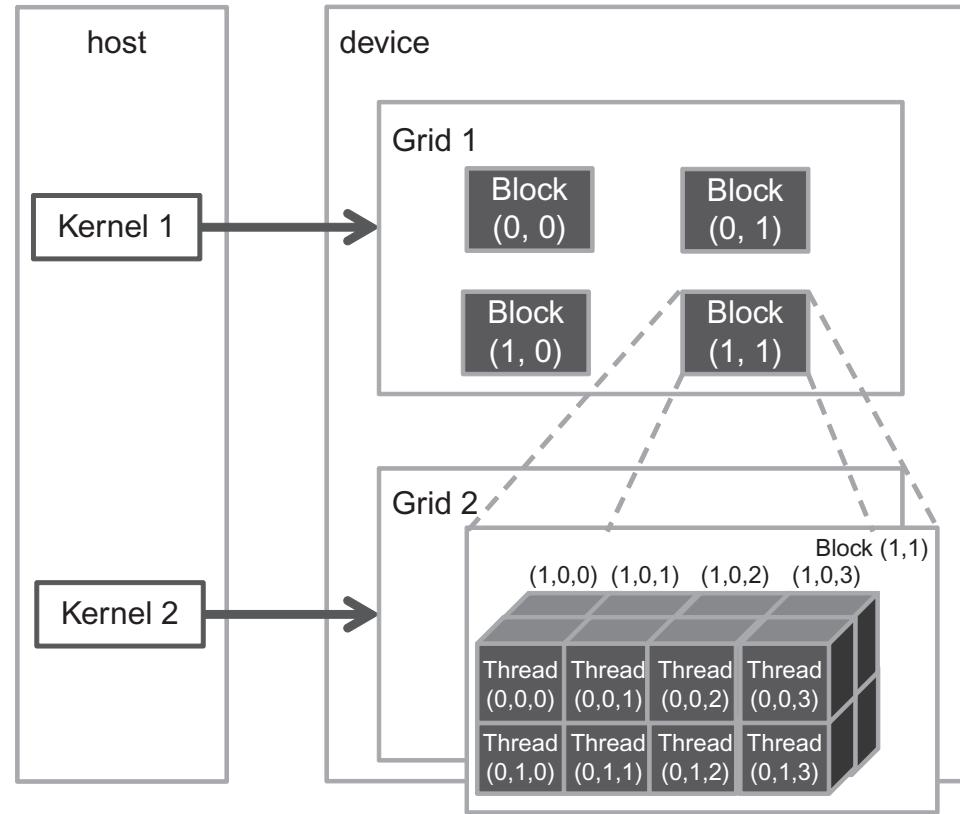
```
dim3 dimGrid(2, 2, 1);  
dim3 dimBlock(4, 2, 2);
```

KernelFunction<<<dimGrid, dimBlock>>>(...);

Launch of a kernel makes the following structures available

blockDim.x, blockDim.y, blockDim.z
threadIdx.x, threadIdx.y, threadIdx.z
blockIdx.x, blockIdx.y, blockIdx.z

which tell the placement of the thread in the hierarchy.



Adapted from [1]

Brief intro to shared mem programming model

Static shared memory device code

```
__global__
void staticReverse(int *d, int n) {
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

Dynamic shared memory device code

```
__global__
void dynamicReverse(int *d, int n) {
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

Calling this kernel from the host:

```
dynamicReverse<<<1, n, n*sizeof(int)>>>(d_d, n);
```

*Third execution configuration parameter
allocating the shared memory*

CUDA streams

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a);  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- **Kernel calls are asynchronous**; after the kernel is launched, the code returns to the host
- **CUDA calls are blocking or synchronous**, such as `cudaMemcpy`
- All device operations run in a **stream**; if no stream is specified, the default (or “null”) stream is used.

CUDA streams

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a);  
DoSmtghOnHost();  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- Overlapping host and device tasks is trivial due to the asynchronous nature of the kernel calls.
- How to make CUDA calls concurrently, f. ex. the computation and data transfers in the above example, requires further techniques with the concept of streams.

CUDA streams

```
cudaStream_t stream1;  
cudaError_t result;  
result = cudaStreamCreate(&stream1);  
result = cudaStreamDestroy(stream1);
```

Non-default streams in CUDA are

- Declared (1st line),
- Created (3rd line), and
- Destroyed (4th line)

in host code as above.

CUDA streams

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice,  
                      streamN)
```

- Async data transfers can be accomplished by CUDA functions such as `cudaMemcpyAsync`, `cudaMemcpy2DAsync()`, and `cudaMemcpy3DAsync()`, where the **5th argument** is the **stream identifier**.

```
increment<<<1,N,0,streamX>>>(d_a)
```

- **Kernel calls** to be executed on non-default stream will have to specify the stream identifier as the **4th argument**. The third argument is to declare the allocation of shared memory, here none is requested, hence 0.

CUDA streams

1. `cudaDeviceSynchronize();`
 2. `cudaStreamSynchronize(stream);`
 3. `cudaEventSynchronize(event)` (ADVANCED)
- Since all operations in **non-default streams** are **non-blocking with respect to the host code**, you need to **synchronize** the host code with stream operations.
 - Ways relevant to us:
 1. the host code is blocked until **all previously issued operations on the device** have completed
 2. The host thread is blocked **until all previously issued operations in the specified stream** have completed

How to run on multiple GPUs?

- Nowadays commonly possible, also in Triton.
- You ask for multiple GPUs using **--gpus-per-node**, where **N** stands for number of requested GPUs. **Use N>1** to reserve more than one GPU. See example programs and scripts in GitLab repo GPU/X.

How to setup a code for multiple GPUs and MPI?
GPU/sheet6/src/main.cu, reduce-multi.cu and reduce-mpi.cu

How to run on multiple GPUs?

Two general cases:

- **GPUs within a single network node: data transfers through peer-to-peer or shared host memory**
 - **peer-to-peer**: `cudaDeviceEnablePeerAccess(...)`,
`cudaDeviceCanAccessPeer(...)`, `cudaMemcpyPeerAsync(...)`
[advanced, not needed to solve Sheet 6].
 - Host launches **streams on different devices** and **collects** the results.
- **GPUs across network nodes**
 - Communication through **CUDA-aware MPI**

How to run on multiple GPUs?

cudaError_t `cudaGetDeviceCount`(int* count)

Returns the number of devices

cudaError_t `cudaSetDevice`(int device)

Device on which the active host thread should execute the device code.

cudaError_t `cudaGetDevice`(int* device)

Returns the device on which the active host thread executes the device code.

CUDA-aware MPI

The most likely case, as MPI tends to be SOOO complicated

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,
           cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,
         100,MPI_COMM_WORLD);
//MPI rank 1
MPI_Recv(r_buf_h,size,MPI_CHAR,0,
          100,MPI_COMM_WORLD,
          &status);
cudaMemcpy(r_buf_d,r_buf_h,size,
           cudaMemcpyHostToDevice);
```

Or can we perhaps do this, and life becomes wonderful?

```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,1,100,
          MPI_COMM_WORLD);
//MPI rank 1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,
          MPI_COMM_WORLD, &status);
```

Yes, this is how it works!!!!!!

Thanks to Unified Virtual Addressing (UVA) feature in CUDA; read more from [5]

Useful reading

- [1] David Kirk & Wen-Mei Wuu: “Programming massively parallel processors”, third edition, 2017, Morgan Kaufmann, Cambridge, USA
- [2] <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf>
- [3] <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>
- [4] <https://ppc.cs.aalto.fi/ch4/v1/, .../v2 and .../v3>
- [5] <https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/>