Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205
Resource-efficient programming

5 Production

5.1 Performance metrics
5.2 Greedy vs. backtracking
5.3 Load balancing

5.4 CMake
5.5 HPC deployment
5.6 MPI input/output

Weekly glossary concepts

Norwegian University of Life Sciences

What are essential concepts from the previous lecture?

Let us include them in the **INF205 glossary**.[1]

???

trace

distributed system

???

Petri net

???

[1] https://home.bawue.de/~horsch/teaching/inf205/glossary-en.html

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

5 Production

5.1 Performance metrics

INF205

22. april 2024

Requirements: The traditional view

In most cases, discussion of computational resources limits itself to "**space**" and "**time**." This is also motivated by tradition in theoretical computer science. In practice, then, *time usually becomes the main performance metric*, whereas *space becomes the main bottleneck* (memory access, communication, file I/O).

Strong scaling (**Amdahl**, *constant* problem size) on parallel architectures:

- Runtime reduction as number of processes increases (ideally, linear).
- Total CPU time increase as there are more processes (ideally, none).
- Rate of CPU operations (*e.g.*, FLOP/s) as fraction of peak performance.
- *Amdahl's law: Deterioration of performance at some point is inevitable.*

Weak scaling (**Gustafson**, *proportional* problem size) on parallel architectures:

- CPU time per problem size as problem and core usage are scaled up.
- Runtime increase during the scale-up.
- Rate of CPU operations (*e.g.*, FLOP/s) as fraction of peak performance.
- *Some algorithms and codes don't show a major decay in these metrics.*

Requirements: The traditional view

Time, in theory:

- Number of steps executed by a *Turing machine* (or similar formalisms, such as *random-access machines*)
- Number of statements to be executed when going through the code

Time, in practice:

- CPU time, *i.e.*, number of cores × measured runtime of the program

Space, in theory:

- Legth of tape used by a *Turing machine* (or number of registers used by a *random-access machine*)
- Number of elementary variables, or their total size in bytes, in the code

Space, in practice:

- Actual memory use measured during program execution

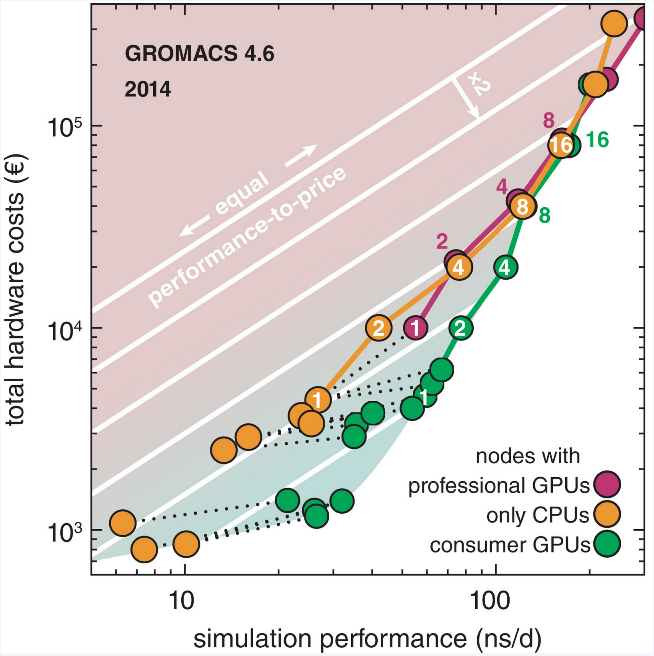
In **complexity theory**, the theoretical metrics are used to define **computational complexity classes**, such as DTIME($f(n)$) and DSPACE($f(n)$) for deterministic $O(f(n))$ time and space, respectively, as function of the problem size n.

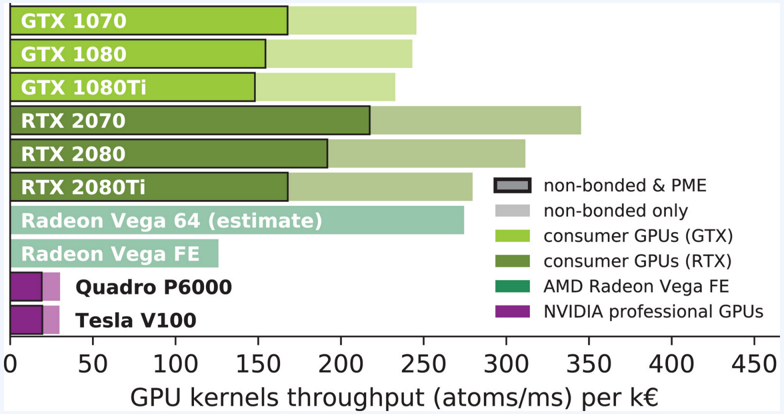
Economic metrics

Investment and operational costs can be considered. For an analysis of investment costs[1] to be reasonably actionable, it must include multiple representative use cases.

Operational costs usually also have a major ecological aspect (computational and cooling electricity cost). They might be taken into account for scheduling/workflow management.

Example on the right: C. Kutzner *et al.*, "More bang for your buck: Improved use of GPU nodes for GROMACS 2018," *J. Comp. Chem.* **40**(27): 2418–2431, doi:10.1002/jcc.26011, **2019**.





Ecological and socioeconomic metrics

How about bitcoins and similar blockchain technologies?

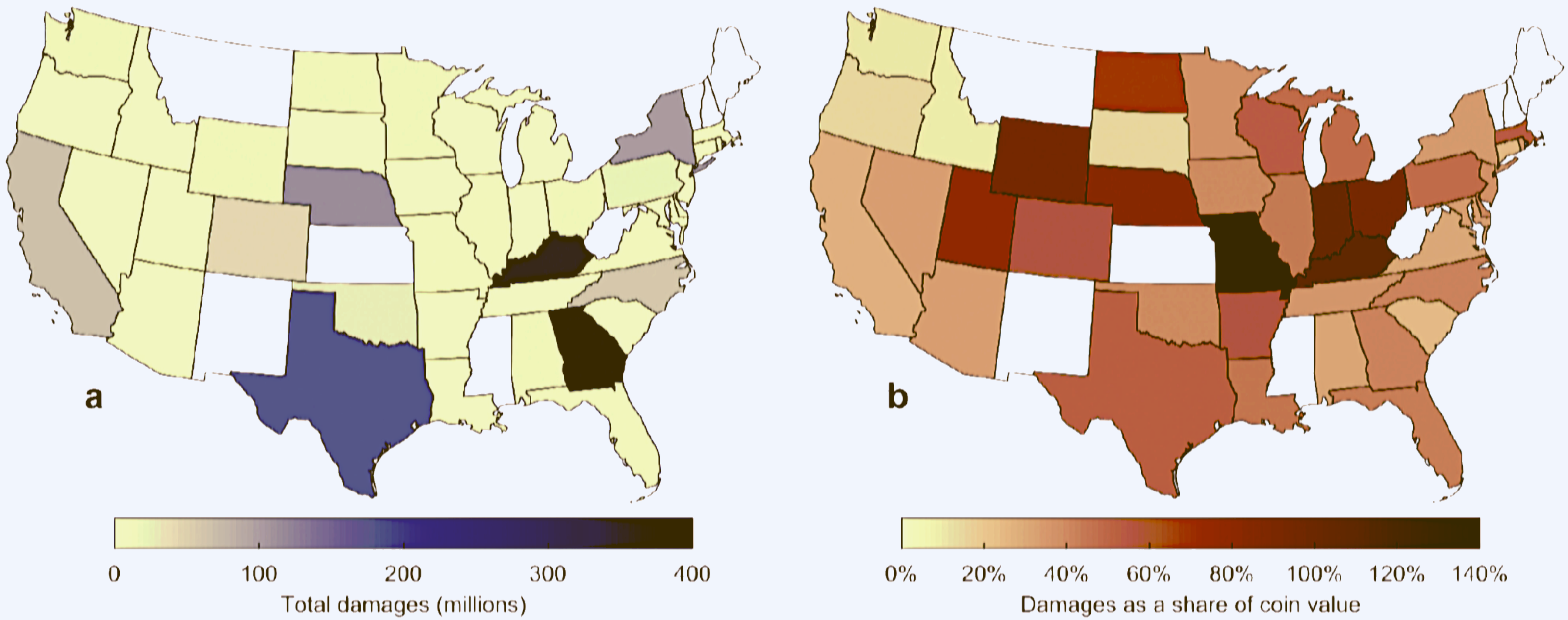


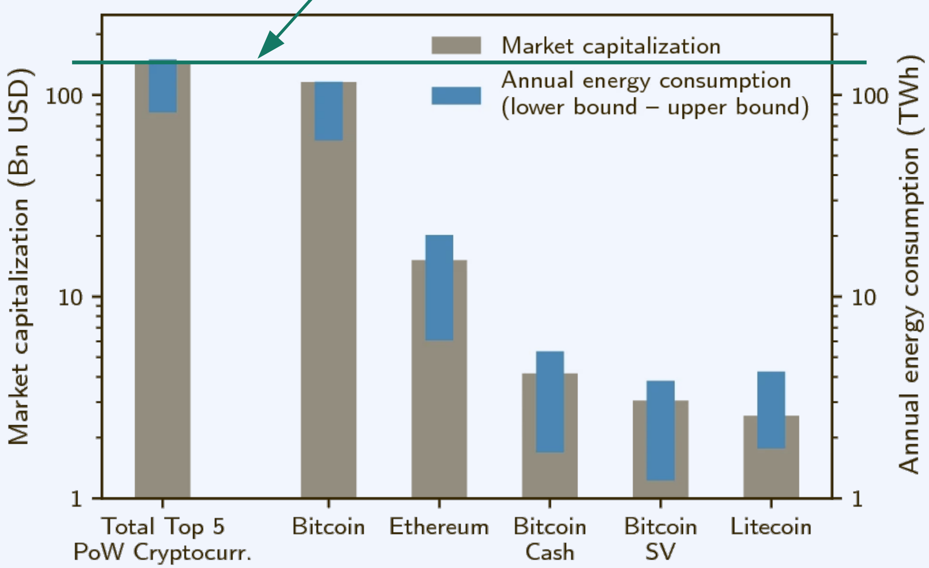
Figure 2. BTC mining damages (climate plus human health) across US states. Panel **a** (left): total damages of BTC mining between 1 September 2019 to 31 December 2021. Panel **b** (right): average damages per BTC mined as a share of the market price of the coin. States in white did not mine BTC over this period according to the CBECI dataset.

From A. L. Goodkind *et al.*, *Appl. Econ. Lett.*, doi:10.1080/13504851.2022.2140107, **2022**.

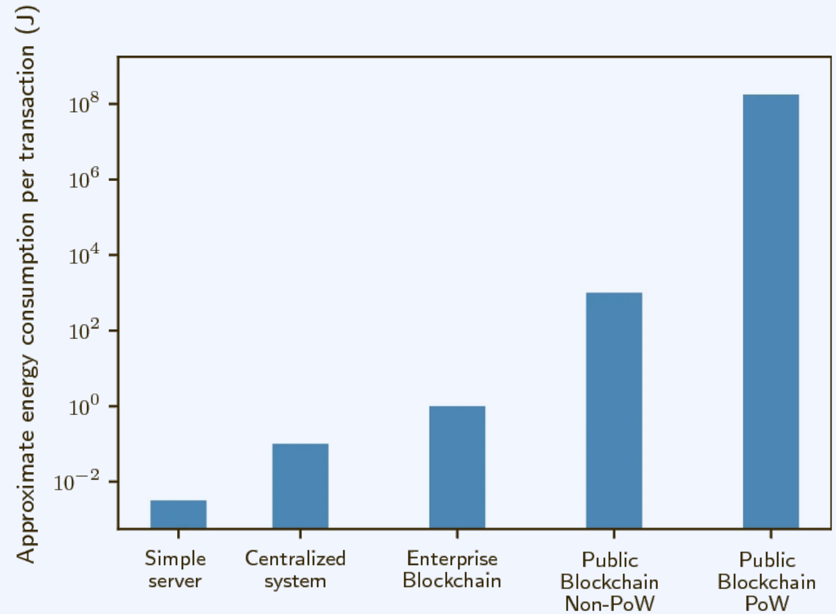
Ecological and socioeconomic metrics

How about bitcoins and similar blockchain technologies?

electric power production of Norway



Market capitalization and the computed bounds on energy consumption for the 5 highest valued Proof-of-Work cryptocurrencies. Note the logarithmic scale on the y-axis



A rough comparison of the order of magnitude of energy consumption per transaction for different architectures. A simple server can operate transactions with very low energy consumption. A typical non-blockchain, centralized system in applications will use a more complex database and backups, thus mildly increasing the energy consumption. A small-scale permissioned blockchain as used in cross-enterprise use-cases has a similar degree of redundancy, but some additional yet limited overhead due to, e.g., PoA consensus and more complex cryptographic operations. A non-PoW permissionless blockchain with a large number of nodes can already exhibit a significantly increased energy consumption due to the high degree of redundancy. However, compared to a major Proof-of-Work blockchain, energy consumption is still negligible

From J. Sedlmeir *et al.*, *Bus. Inf. Syst. Eng.* **62**: 599, doi:10.1007/s12599-020-00656-x, **2020**.

Energidepartementet (energifaktanorge.no), “I 2023 ble det produsert 154 TWh kraft i Norge.”

Ecological and socioeconomic metrics

Key performance indicators (KPIs) for ecological performance[1] can include:

- Abiotic-resource depletion potential (ADP)
- Cumulative energy consumption
- Greenhouse warming potential, including from use of refrigerants
- Water consumption
- *DCiRE: Contribution of building infrastructure to these categories*

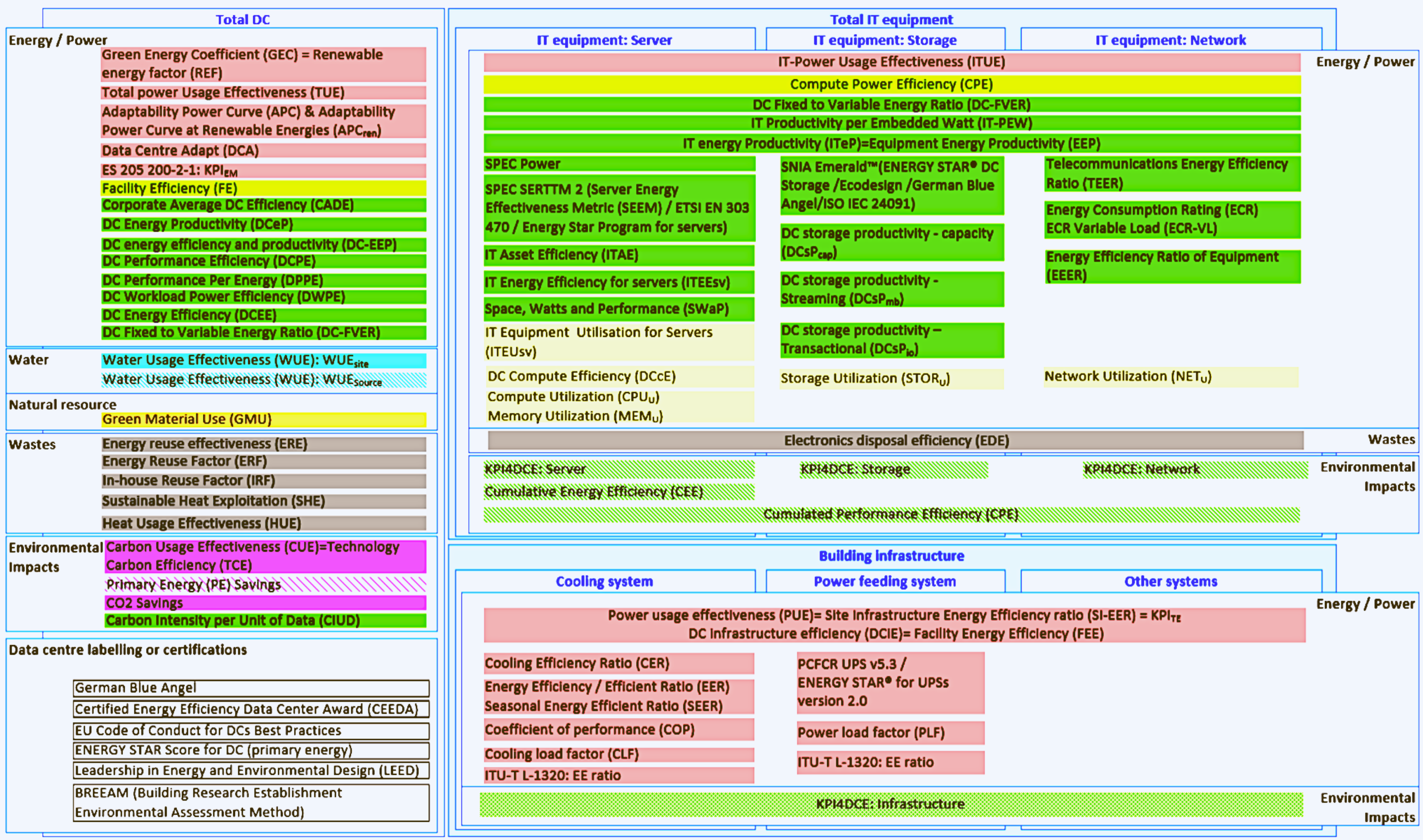
Blue Angel: "Type I environmental label" (ISO 14024) based on full life-cycle-analysis, targeting customers.[2]

EC explores introduction of additional Type II and III labels.[3]

[1] B. Schödwell, R. Zarnikow, KPI4DCE report, UBA no. 19/2018, *pp.* 25, 154, **2018**.
[2] See *e.g.* https://www.hlrs.de/about/certifications.
[3] EC report "Study on Greening Cloud Computing […]," doi:10.2759/116715, *pp.* 128, 289, **2022**.

Ecological and socioeconomic metrics

Table 16: Overview of 71 selected metrics and 6 DC-relevant labelling or certification scheme



p. 98, "Study on Greening Cloud Computing [...]," EC report, doi:10.2759/116715, **2022**.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

5 Production

5.1 Performance metrics

5.2 Greedy ./. backtracking

Algorithm design strategies

Brute force	Easy to implement and to verify	Scales with size of solution space, often forbiddingly expensive
Greedy algorithms	Easy to implement, often very efficient	Not all problems are accessible to this kind of approach
Decomposition techniques	Powerful by reduction to subproblems	Requires a thorough analysis of the problem and its subproblems

Design strategies concern algorithm and code development at a more abstract level than that of its implementation. They are established approaches for designing algorithms; they all have their own strengths and weaknesses.

- **Brute force:** Check all possible solutions, determine the right/best one.
- **Greedy algorithms:** Build the solution step by step until it is complete.
- Decomposition by **divide-and-conquer** or by **dynamic programming**.

Algorithm design strategy: Greedy algorithms

Greedy algorithms are based on the idea of making the **best local improvement** (*i.e.*, the best immediately visible small change) to a partial solution. They consider **one candidate solution** only and build it up gradually.

Image source: City College Norwich

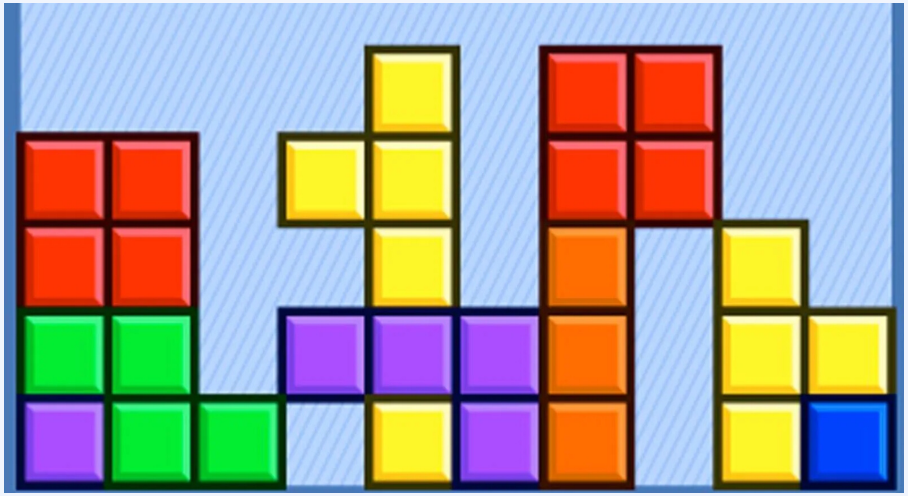


Image source: BBC

Strength: Systematic and easy to implement.

Weakness: It does not solve all problems correctly; but even then, it might return an acceptable suboptimal result or an approximation to the solution.

Problems for which greedy algorithms work

Example: **Selection sort**

- Initially, we cannot assume any part of the list to be sorted.
- Search for the smallest element of the list and move it to position 0; now, the list is sorted* up to index 0. It is unsorted** from 1 onward.
- In step *k*, go through the whole unsorted part, find the smallest of its elements, and move it to the end of the sorted part.

**It contains the smallest element.*

***And it contains the n–1 greatest elements, for problem size n.*

Image source: Wikipedia

Problems for which greedy algorithms don't work

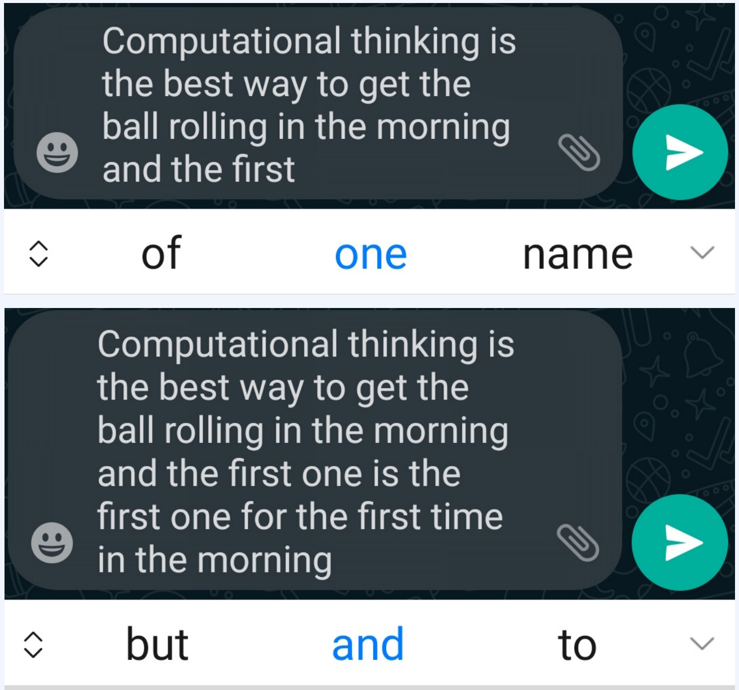
Constructing a solution incrementally, step by step, is not always correct.

Greedy algorithms will produce an incorrect or suboptimal result whenever the overall ("global") solution does not consist of parts that are best "locally."

Example: Autocomplete problem

"Computational thinking is the best way to get the ball rolling in the morning and the first one is the first one for the first time in the morning …"

To be fair, these are not all "locally optimal" incremental improvements. But even if they were, the word-by-word method would not lead to the globally optimal sentence as defined through some well-defined metric.



Greedy algorithm + backtracking

Greedy (priorization) + DFS
When exploring the state space using DFS (*i.e.*, with backtracking), *follow the most promising path first*: The greedy solution. Then, follow all other options *in order of their expected outcome*.

Problem from the example code
Cost function[1] $f(\mathbf{x})$ with argument:

- int **x**[4], values 0 to 999
- (parameter space size 10^{12})

Return value between 0.0 and 6.0.

- minimize (aim: close to 0.0)
- accept solutions below a threshold (default 0.00666)

- If the problem can be solved by a greedy algorithm, no backtracking will be needed. The program produces the greedy solution first.
- Different paths are prioritized according to some metric such as how "promising" they look. This needs to be defined and implemented.
- If necessary, the whole state space (parameter space) will be explored. Then this becomes equivalent to the brute force method.

[1]Other names (used instead of *cost function*) include *loss function*, *potential*, and *minimization objective*.

Example file: backtracking-vs-metropolis.zip

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

Remark on the Metropolis algorithm

INF205

22. april 2024

Metropolis Monte Carlo algorithm

Parameters of the method:

- Temperature T
- Maximum test-move distance s
  (go up to ± s in each dimension)

Position before step is $\mathbf{x} = (x_0, \ldots, x_{k-1})$.

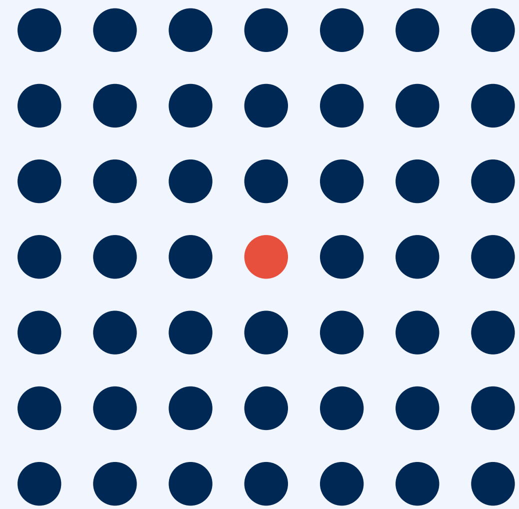
Note that in the example problem, $k = 4$.

For each $0 \leq d < k$, determine a random value Δx_d between $-s$ and $+s$. The new *test* coordinate is $\bar{x}_d = x_d + \Delta x_\mathrm{d}$.

Determine the change of the cost function: $\Delta f = f(\bar{\mathbf{x}}) - f(\mathbf{x})$. If $\Delta f \leq 0$, accept the test move. If $\Delta f > 0$, accept with probability $\exp(-\Delta f\ / T)$.

If and only if the test move was accepted, take over $\bar{\mathbf{x}}$ as the new value for $\mathbf{x}$.

present position; max. distance 3:
possible new positions after test move



The code, visualization, *etc.*, here are discrete (test moves by integer distances) *because the problem is discrete*.

For a problem defined over a continuous space, you must use test moves suitable for sampling that space – a continuum.

Example file: backtracking-vs-metropolis.zip

Simulated annealing

When applying the MC method outside of physics, the temperature T does not have any physical meaning, it is just a tuning parameter.

For minimization problems,

- high T means that the configuration space is sampled more evenly (we don't penalize a large increase in cost Δf very severly);
- low T means that we mostly restrict ourselves to local optimization (where there is a barrier between basins corresponding to multiple local optima, that is harder to overcome, since the increase Δf is normalized by T, which now is small).

Simulated annealing is a heuristic that uses this to our favour when sampling a space where we expect multiple local minima:

- Begin with a *high temperature* to *explore the whole space*,
- close in on the *local minimum* by continuing at a *low temperature*.

(The example code goes through repeated cycles of this, using three T levels.)

Example file: backtracking-vs-metropolis.zip

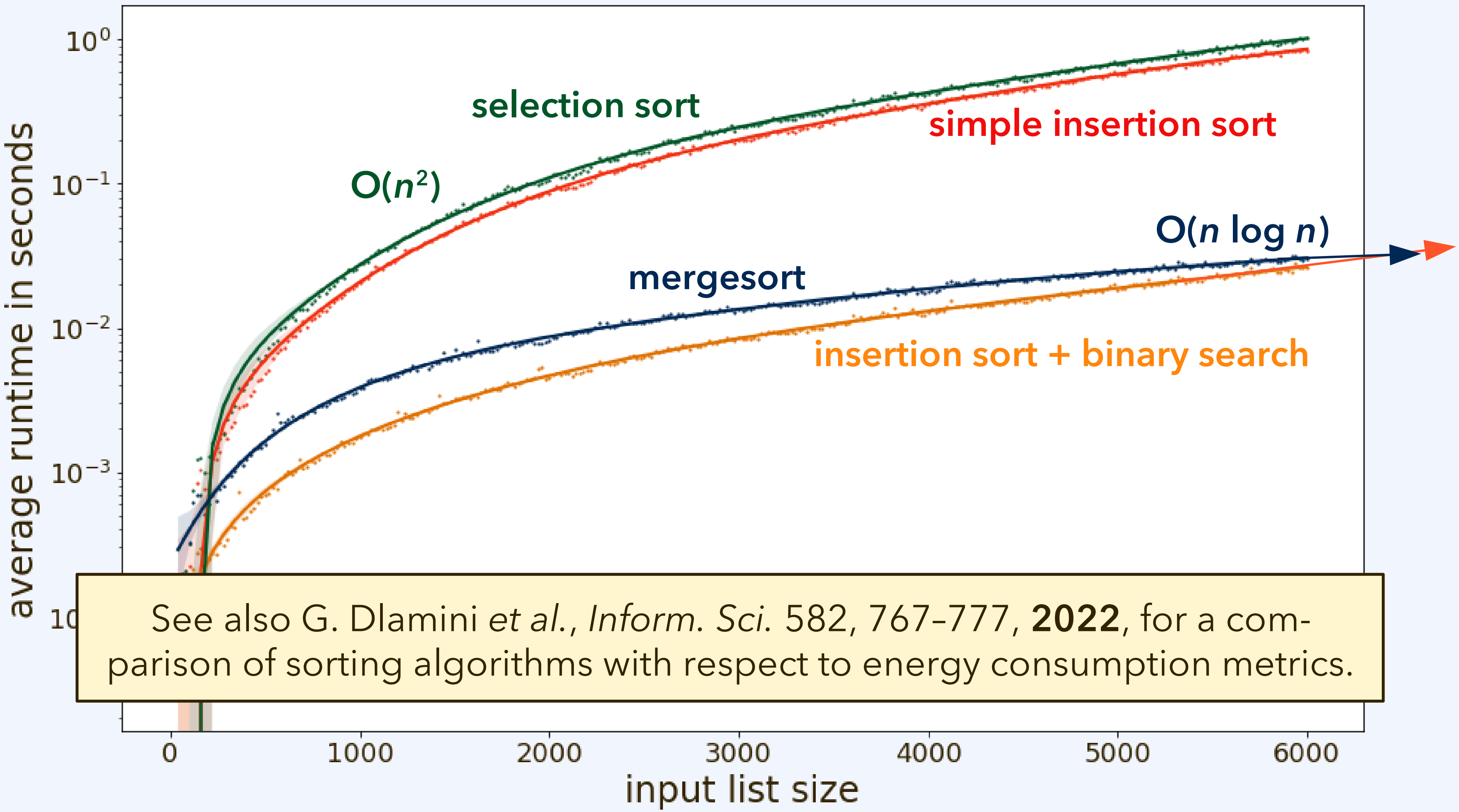
Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

5 Production

Requirements: The traditional view



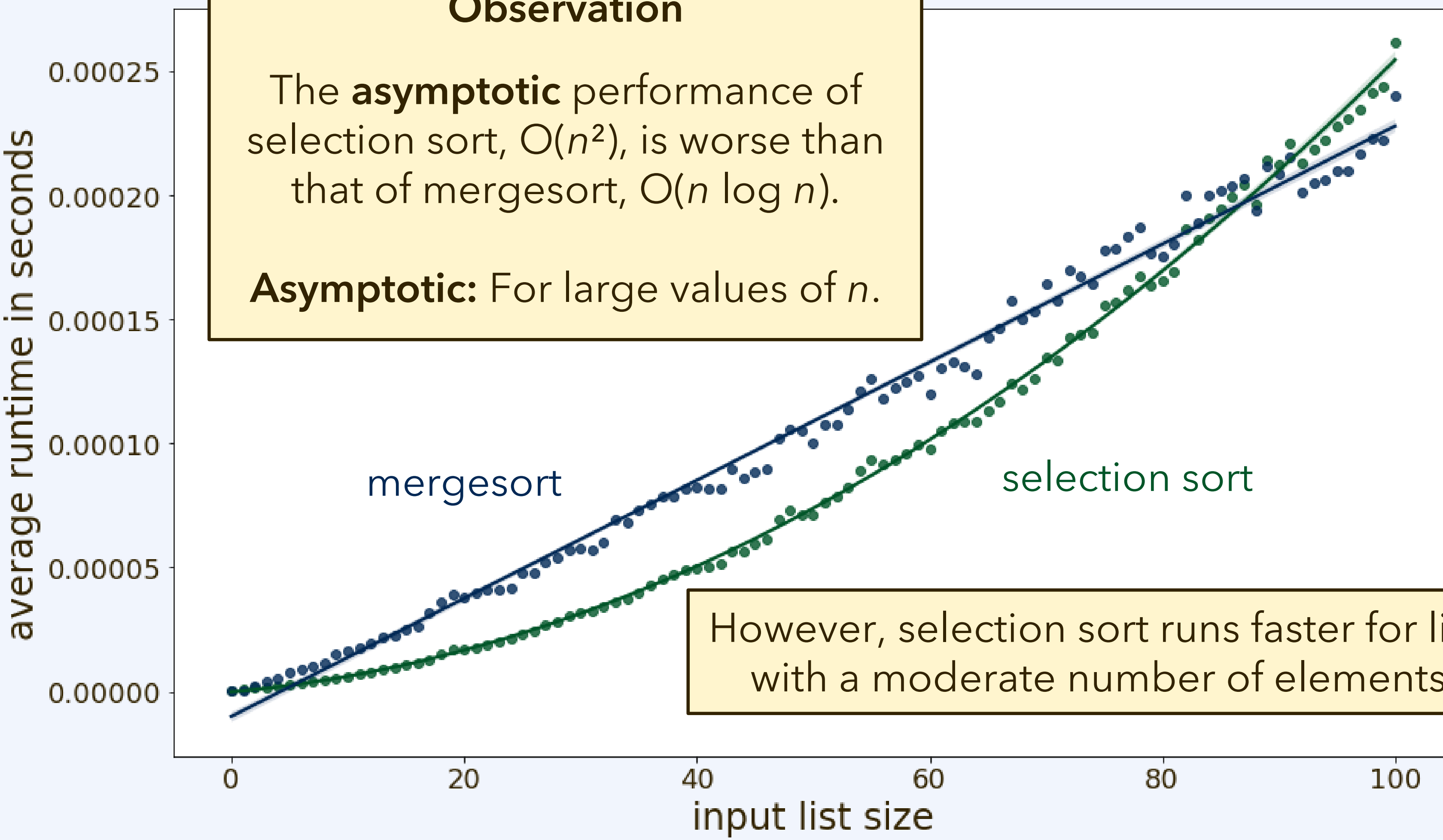
See also G. Dlamini *et al.*, *Inform. Sci.* 582, 767–777, **2022**, for a comparison of sorting algorithms with respect to energy consumption metrics.

Limitations of the traditional view

Observation

The **asymptotic** performance of selection sort, $O(n^2)$, is worse than that of mergesort, $O(n \log n)$.

Asymptotic: For large values of n.



However, selection sort runs faster for lists with a moderate number of elements.

Quantitative requirements modelling

Idea and prerequisites:

- The parameter space or **domain of the requirements model** is well defined, accounting for type and size of the input or use case and the execution conditions of the code (such as number of processes).
- We build a correlation or closed expression that serves as a model of the code, predicting its computational resource requirements. This can be:
  - Purely predictive, based on a theoretical analysis of the code. (Can always be done for a simplified model, if no data are available.)
  - Regression/parameterization of a model, known to be qualitatively right, to **performance data**. (Counts as supervised machine learning.)
  - Unsupervised machine learning from **performance data**.

> **Discussion:** For what purpose can it be helpful to have a quantitative requirements model? In what ways might we use it in practice?

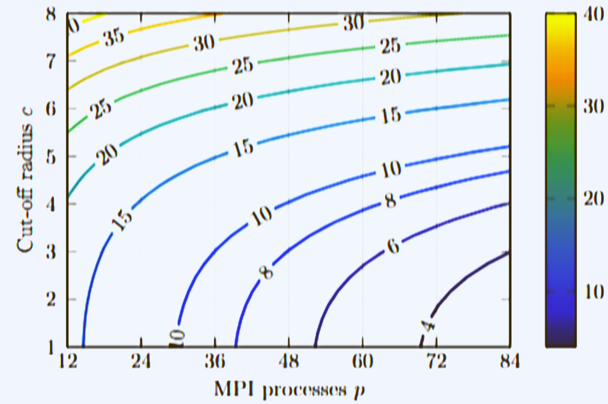
Quantitative requirements modelling

Table 2: 2-parameter models for the execution time of the *ms2* application.

Model	Fixed parameters	Model	$\bar{R}^2$
$T(n,m)$	$d=0.84,\ c=2.0,\ p=72$	$4.41+8.03\cdot10^{-5}\cdot m\cdot n\cdot\log n$	0.99
$T(p,m)$	$n=4{,}000,\ d=0.84,\ c=2.0$	$6.6+3.21\cdot m^2-0.42\cdot m^2\cdot\log p$	0.92
$T(p,d)$	$n=4{,}000,\ m=1,\ c=2.0$	$20.67-2.2\cdot\log p$	0.88
$T(p,c)$	$n=4{,}000,\ m=1,\ d=0.84$	$33.83+0.05\cdot c^3-4.89\cdot\log p$	0.79
$T(n,c)$	$m=1,\ d=0.84,\ p=36$	$-0.99+0.06\cdot c^3+1.81\cdot10^{-5}\cdot\log^2 n$	0.95
$T(m,c)$	$n=4{,}000,\ d=0.84,\ p=36$	$-23.49+10.09\cdot m+0.22\cdot c^3\cdot m$	0.95

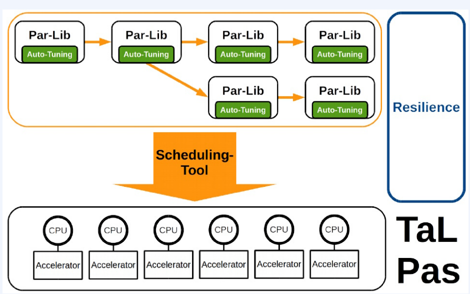
Table 3: 3-parameter models for the execution time of the *ms2* application.

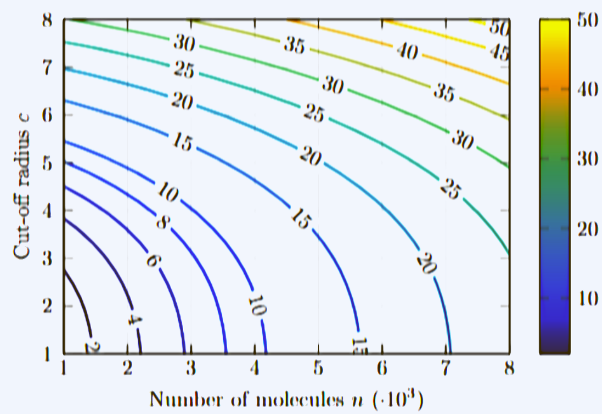
Model	Fixed parameters	Model	$\bar{R}^2$
$T(p,n,m)$	$d=0.84,\ c=2.0$	$62.28+2.03\cdot10^{-8}\cdot m^2\cdot n^{1.5}\cdot\log^2 n-9.63\cdot\log p$	0.83
$T(n,m,c)$	$d=0.84,\ p=72$	$9.24+5.71\cdot10^{-6}\cdot n\cdot\log n\cdot c^2\cdot\log c\cdot m$	0.88



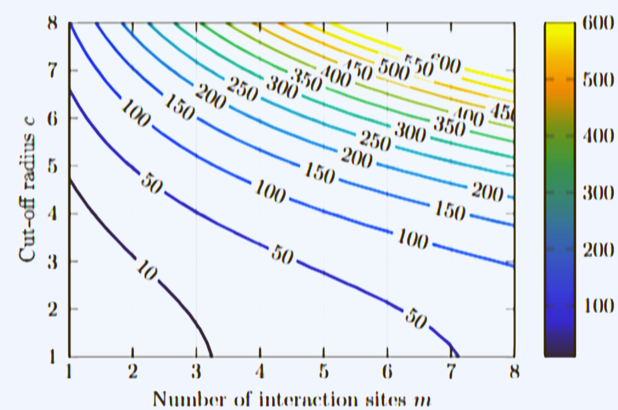
(d) $T(p,c)$

Federal Ministry
of Education
and Research





(e) $T(n,c)$



(f) $T(m,c)$

[1]S. Shudler *et al.*, in *Proc. ESPT-VPA 2017&18*, doi:10.1007/978-3-030-17872-7_8, **2019**.

Load balancing

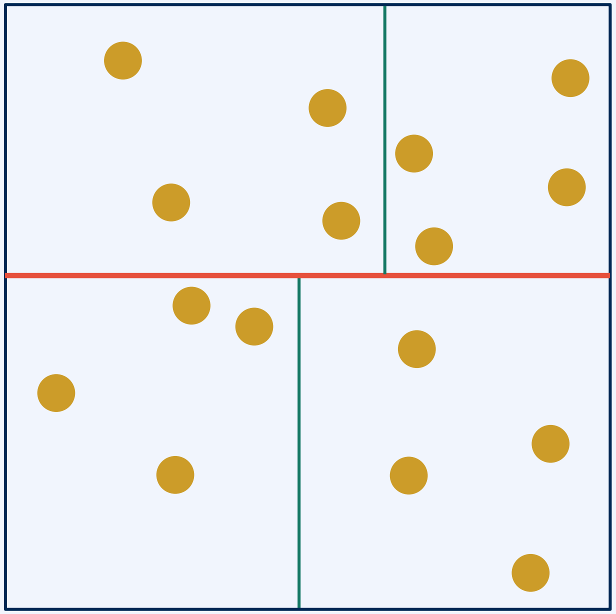
Requirements modelling can be used to predict how the way in which the domain is decomposed, for a given input case/scenario, influences the load of each of the parallel processes.

Usually, no quantitatively accurate requirements model exists. Even then, a rough approximation can be used as guidance for distributing the load.

Example scheme: **Recursive bisection** (with “k-dimensional tree,” k = 3).

From the top (whole domain) down to the bottom (single process), split the volume recursively into parts such that processes will receive a similar load.

Alternate between spatial dimensions.



Dynamic load balancing

Observations:

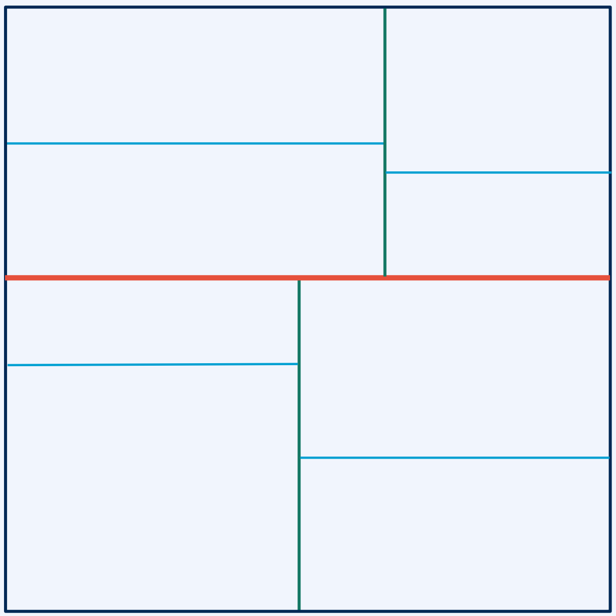
1) *Performance models* are not completely accurate. Moreover, they will usually neglect some of the parameters that influence the runtime.
2) The actual resource requirements will not always be the same for given parameter values. There can be a non-negligible *statistical uncertainty*.
3) Load can *change over runtime*, *e.g.*, from a changing density profile.
4) Anything can occur *on a node in the background, or at the hardware level* (poor cooling, needs to be clocked down, *etc.*). This cannot be reflected in the performance model, and it can change at runtime.

Execution times on HPC infrastructures are of the order of hours to days. The value of the consumed resources is substantial.

Therefore, it can be worth the effort to readjust the decomposition dynamically.

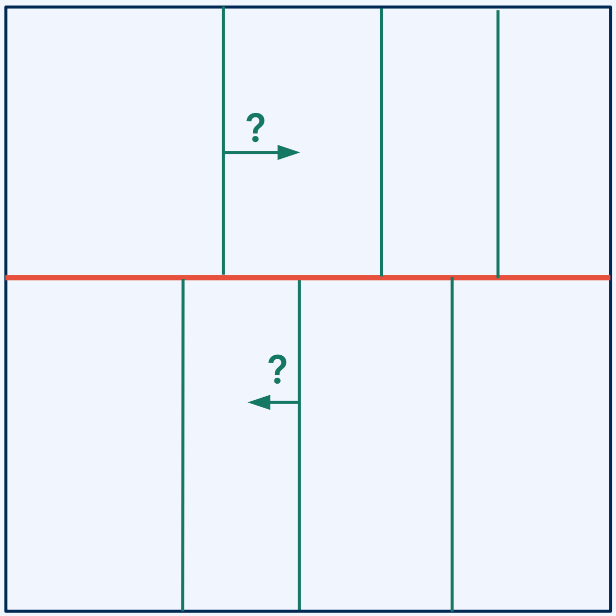
Dynamic load balancing algorithms

recursive bisection



Reconstruct decomposition, *e.g.*, every 10000 steps in a molecular dynamics simulation.

diffusive multisection[1, 2]



Gradual ("diffusive") changes can be implemented to adjust to configuration and performance.

[1]S. Seckler, *J. Computat. Sci.* **50**: 101296, doi:10.1016/j.jocs.2020.101296, **2021**.

[2]J. Sablić, E-CAM project deliverable 4.6, **2020**.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

Discussion and questions on the programming projects

INF205

22. april 2024

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

Conclusion

Weekly glossary concepts

What are essential concepts from this lecture?

Let us include them in the **INF205 glossary**.[1]

computational complexity

???

???

backtracking

load balancing

???

[1] https://home.bawue.de/~horsch/teaching/inf205/glossary-en.html

Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205
Resource-efficient programming

5 Production

5.1 Performance metrics
5.2 Load balancing
5.3 Greedy vs. backtracking

5.4 CMake
5.5 HPC deployment
5.6 MPI input/output