703308 VO High-Performance Computing
Domain Decomposition and Load Balancing

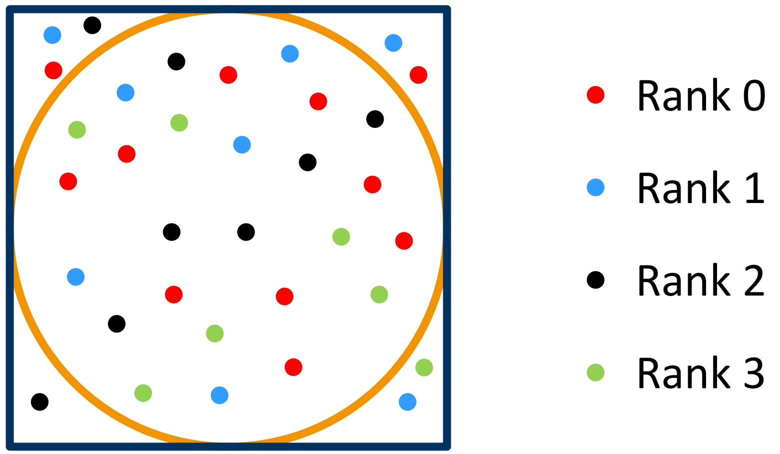
Philipp Gschwandtner

Overview

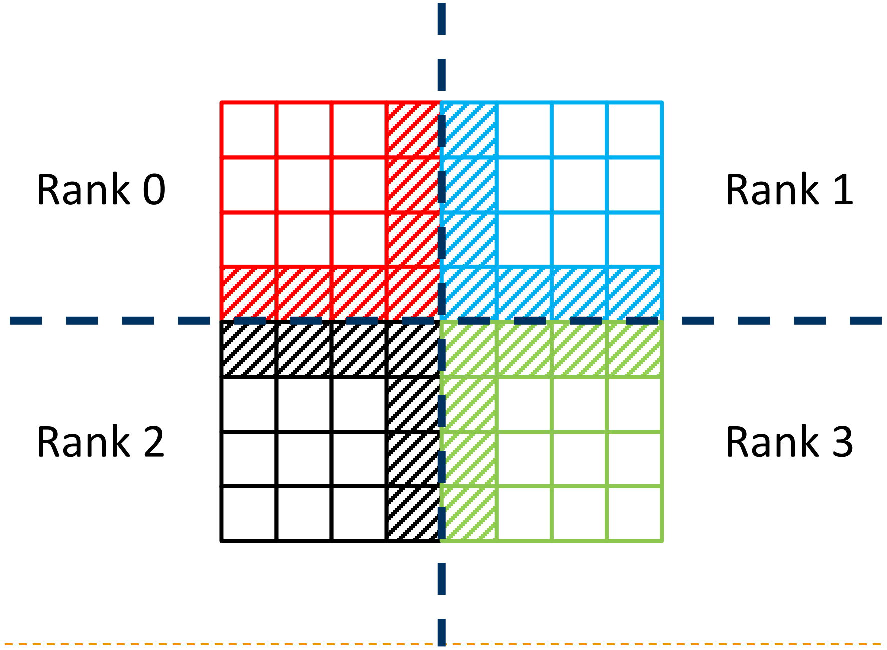
- domain decomposition
- load (im)balance
- “Tales from the Proseminar”

Motivation (domain decomposition)

- Monte Carlo π
  - distributes samples among ranks
  - no communication except at the end
  - also called “embarrassingly parallel”

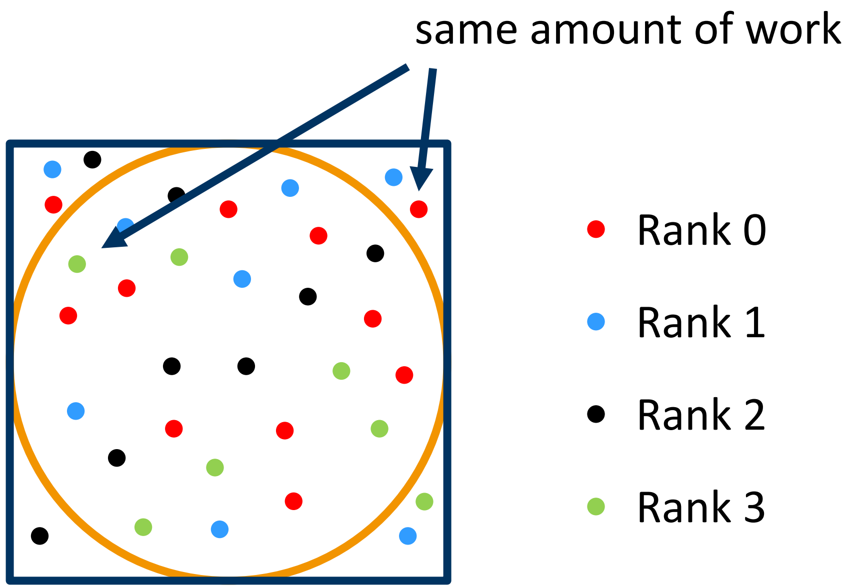


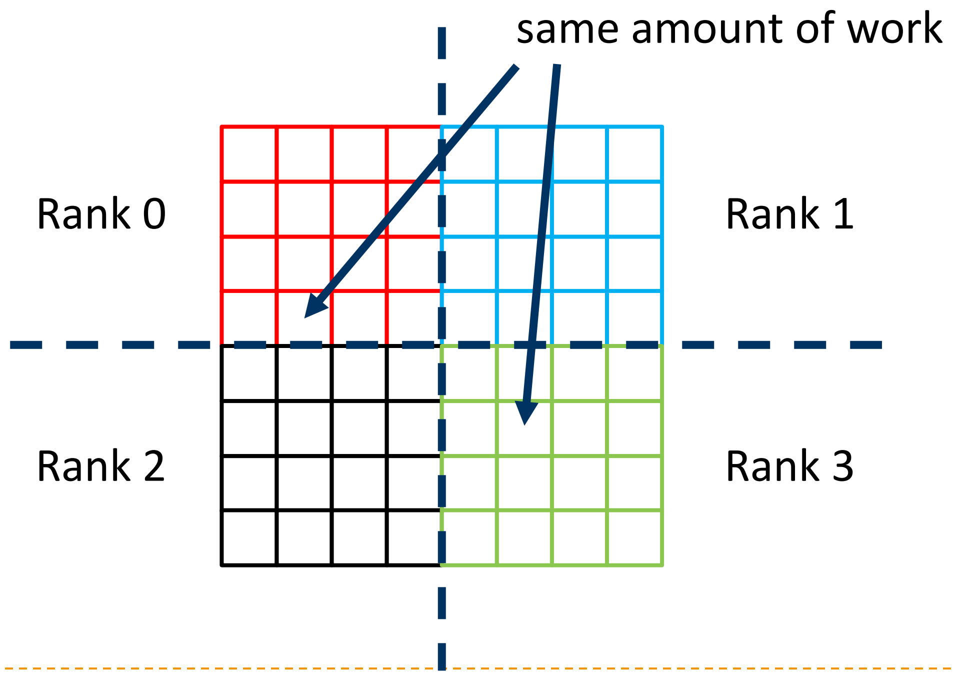
- 2D heat stencil
  - distributes grid cells among ranks
  - ghost cell exchange required at borders!



Motivation cont'd (load balancing)

- Monte Carlo π
  - amount of work per rank depends only on number of samples
- 2D heat stencil
  - amount of work per rank depends only on number of elements





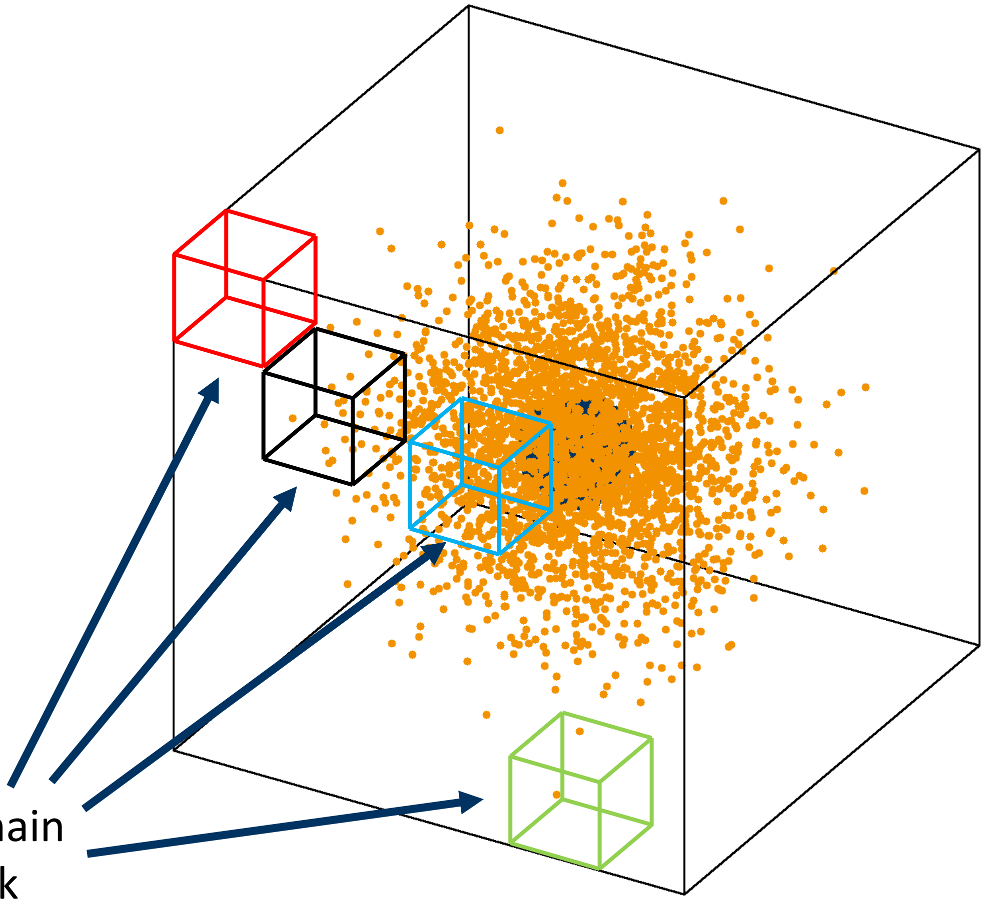
Motivation cont'd (load balancing)

- this is called "balanced load" or "load balance"
  - all the ranks have the same amount of work
  - easily achieved, as any sub-domains of equal size entail same amount of work, no matter in which part of your domain
    - only true for a subset of realistic applications

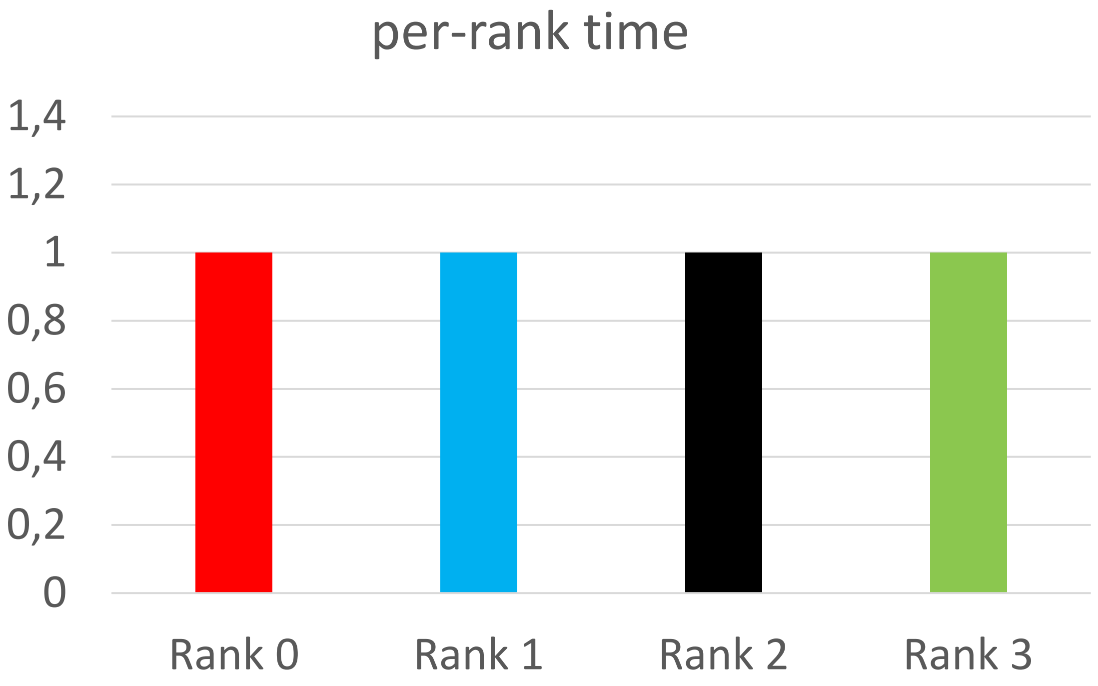
- there's also "unbalanced load" or "load imbalance"
  - ranks do not have the same amount of work
  - either the sizes of sub-domains per rank vary, or their entailed amount of work
    - happens all the time for realistic use cases

Motivation cont’d (load balancing)

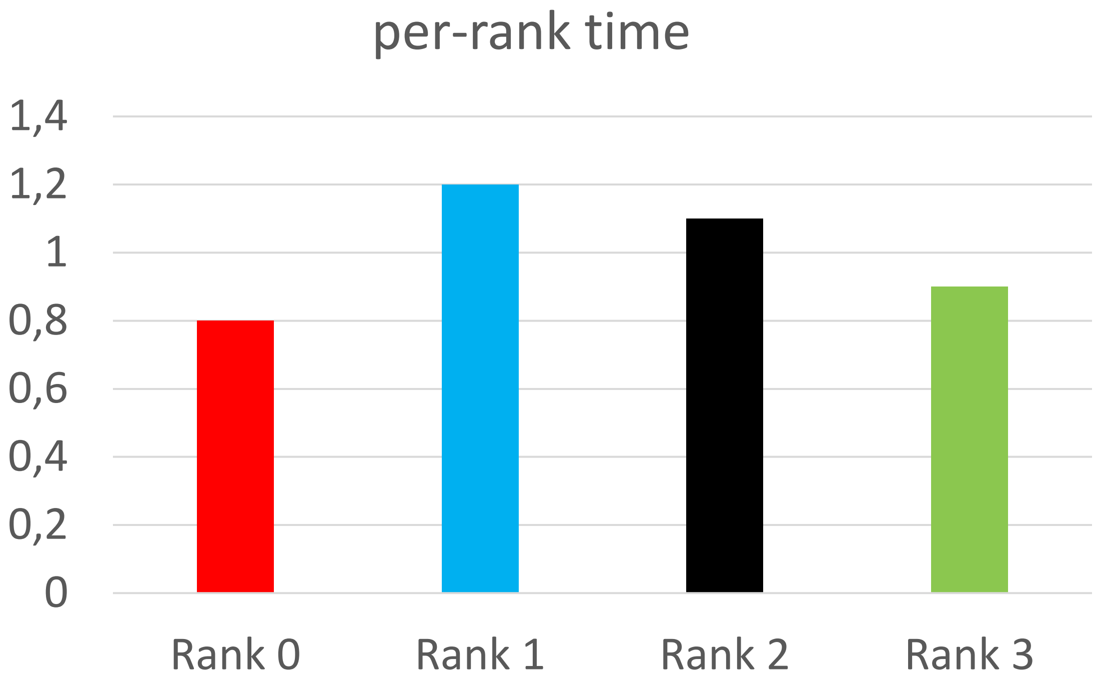
- particle (“lagrangian”) simulation
  - no fixed grid
  - particles can move inside the domain
  - particle distribution affects workload



Motivation cont'd (load balancing)



$t_{\mathrm{cpu}} = 4$
$t_{\mathrm{wall}} = 1$



$t_{\mathrm{cpu}} = 4$
$t_{\mathrm{wall}} = 1.2$

Domain Decomposition

Domain decomposition

- discretize the problem space
  - grid cells, particles, samples, …
- split the workload among the given number of ranks
  - usually also reduces the memory footprint
- goal: minimizing overhead, meaning:
  - minimize load imbalance (differences in workload per processing entity)
  - minimize amount of data to be transferred
  - minimize number of discrete communication steps (c.f. neighbor exchange!)

Cost of a point-to-point message

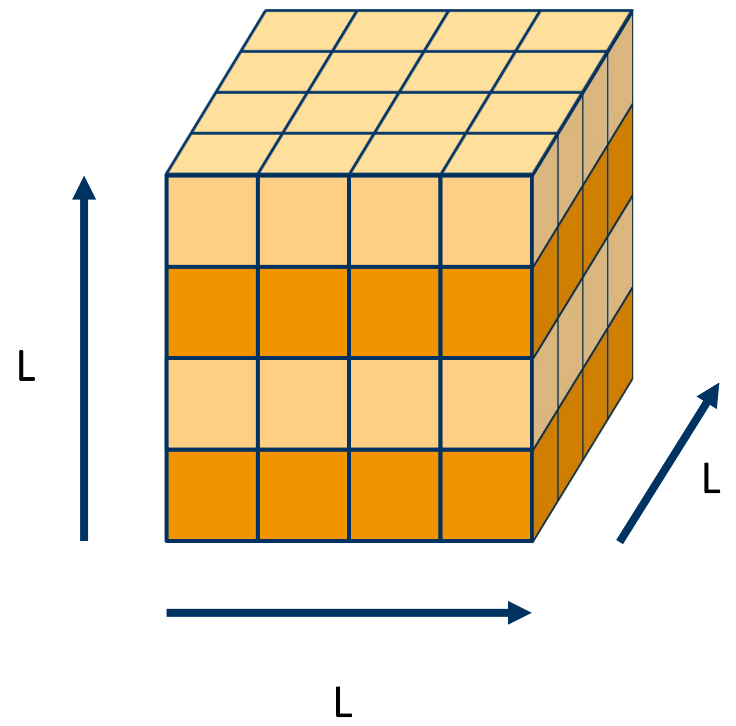
- $t = \text{latency} + \dfrac{\text{message size}}{\text{bandwidth}}$
- note: simplistic view, actual cost depends on
  - underlying protocols (eager, rendezvous, …), dependent on size of data, MPI send mode, repetition, etc.
  - additional data copies required
- latency and bandwidth are fixed properties of the hardware topology
  - note: rank-core mappings, link aggregation, etc…
- message size (and their number) is what we can influence
- can be easily computed for simple, regular decompositions

3D heat stencil example: slabs

$$c_{1D}(L, N) =$$
$$L \cdot L \cdot w \cdot 2 = 2wL^2$$

L: size per dimension
N: number of ranks
w: amount of data per element

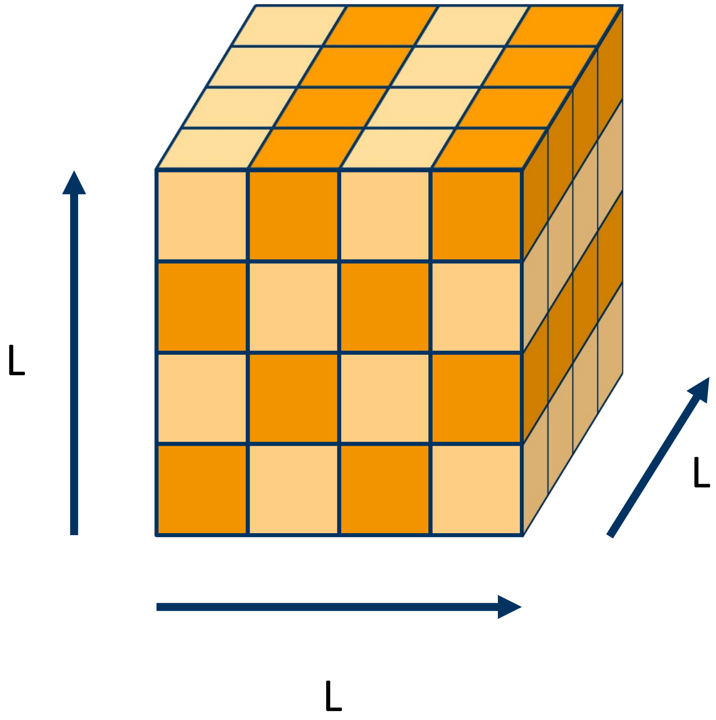
- pro: easy to implement
- con: communication volume does not decrease when increasing N



3D heat stencil example: poles

$$c_{2D}(L,N) = L \cdot \frac{L}{\sqrt{N}} \cdot w \cdot (2+2) = \frac{4wL^2}{\sqrt{N}}$$

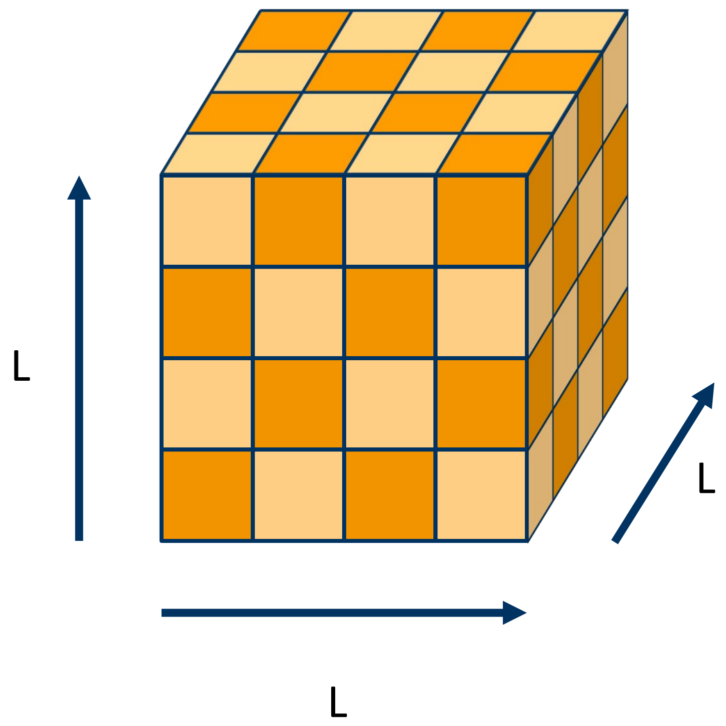
- pro: communication volume does decrease with increasing N
- con: surface-to-volume ratio also increases with N
  - communication grows disproportionally fast compared to computation



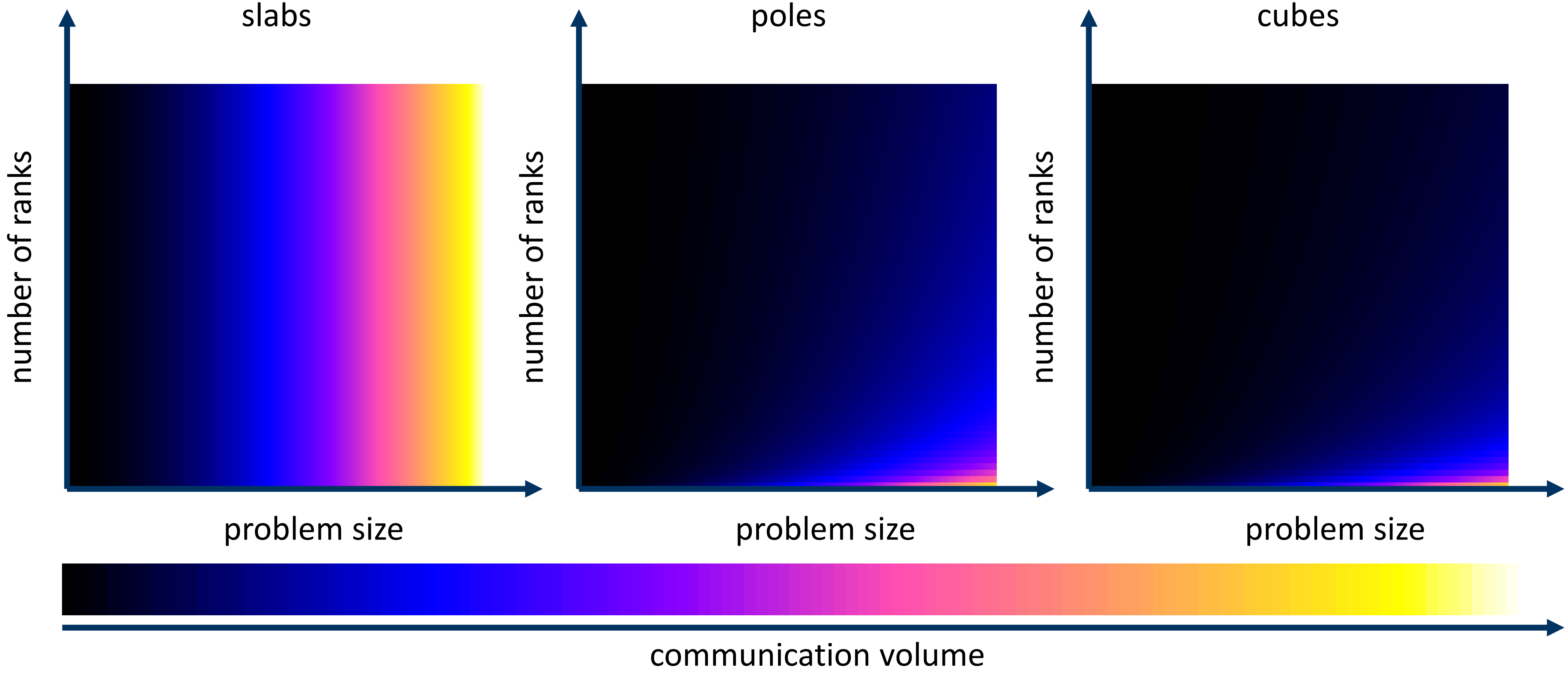
3D heat stencil example: cubes

$$c_{3D}(L, N) = \frac{L}{\sqrt[3]{N}} \cdot \frac{L}{\sqrt[3]{N}} \cdot w \cdot (2 + 2 + 2) = \frac{6wL^2}{\left(\sqrt[3]{N}\right)^2}$$

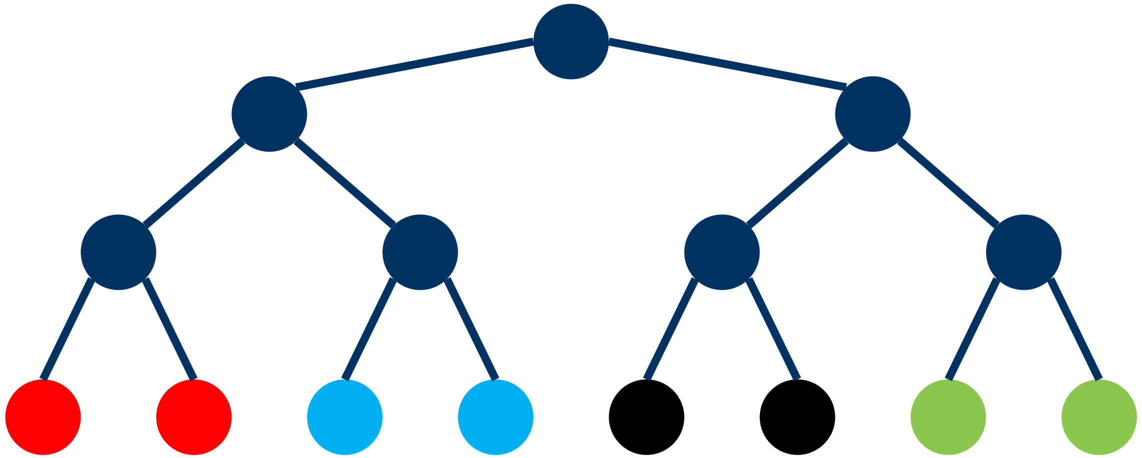
- pro: communication volume also decreases with increasing N
- con: still surface-to-volume-ratio issue
- but also further increase in number of messages, latency might be an issue

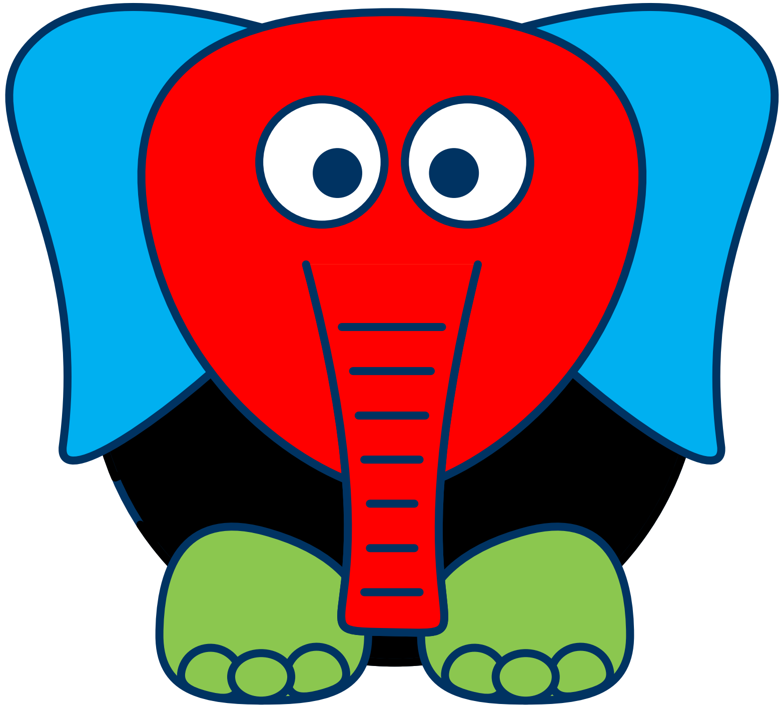


Comparing communication volumes



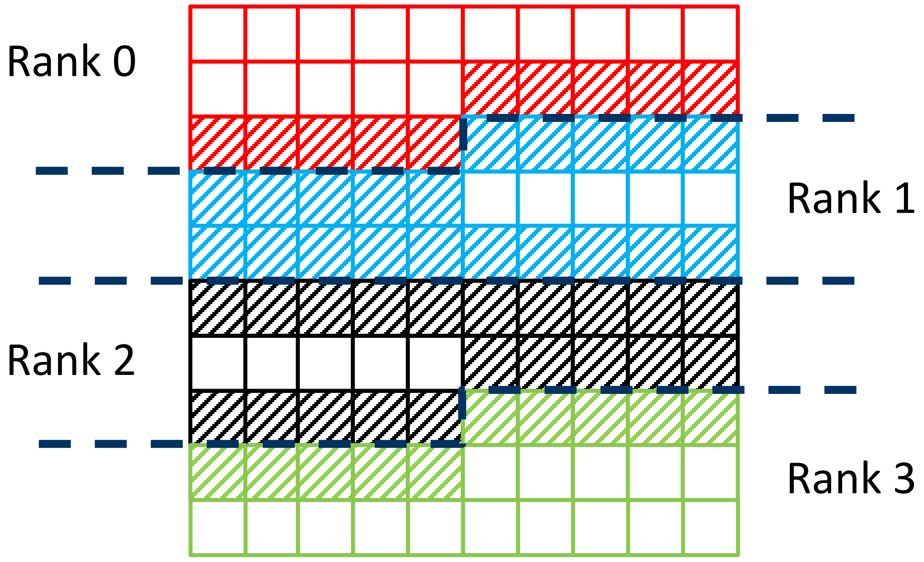
Non-rectangular domain decomposition



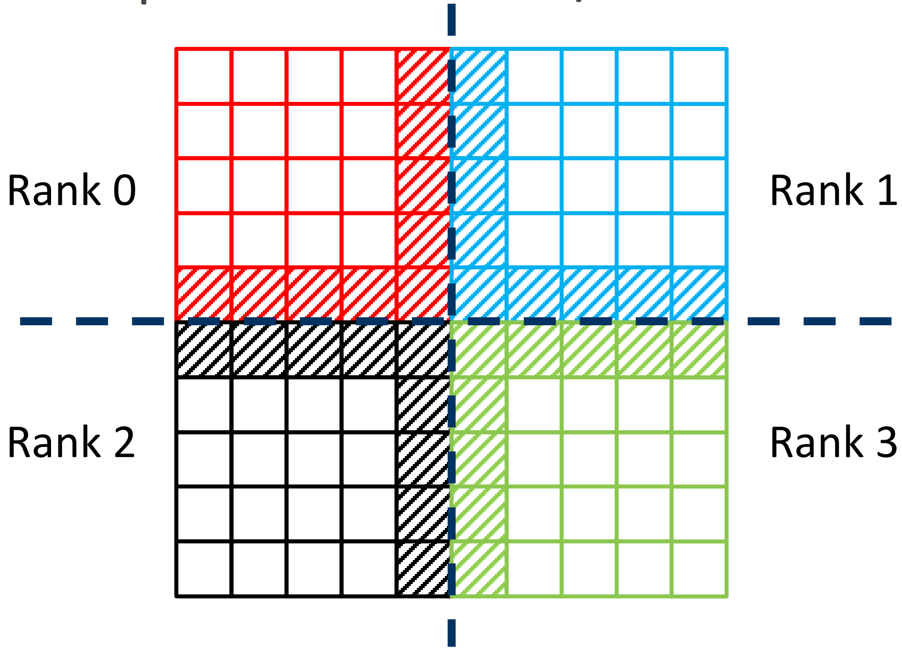


10x10 2D heat stencil decomposition variants

- 25 cells per rank, line-wise
  - 60 of 100 are at sub-domain border
  - worst per-rank communication vs. computation ratio is 20/25!



- 25 cells per rank, 2D blocks
  - 36 of 100 are at sub-domain border
  - worst per-rank communication vs. computation ratio is 9/25



Domain decomposition considerations

- How much data will have to be transferred?
  - more data requires more bandwidth & larger buffers
- In how many messages do I need to transfer the data?
  - additional messages means additional latency, buffer, and management overhead
  - one-sided communication might (!) help here
- What's the cache efficiency of the decomposition?
  - might only be an implementation issue, e.g. matrices can be transposed (rows vs. columns)
- How complicated is the implementation?
  - usual trade-off: e.g. 50% readability decrease for 2% performance increase?

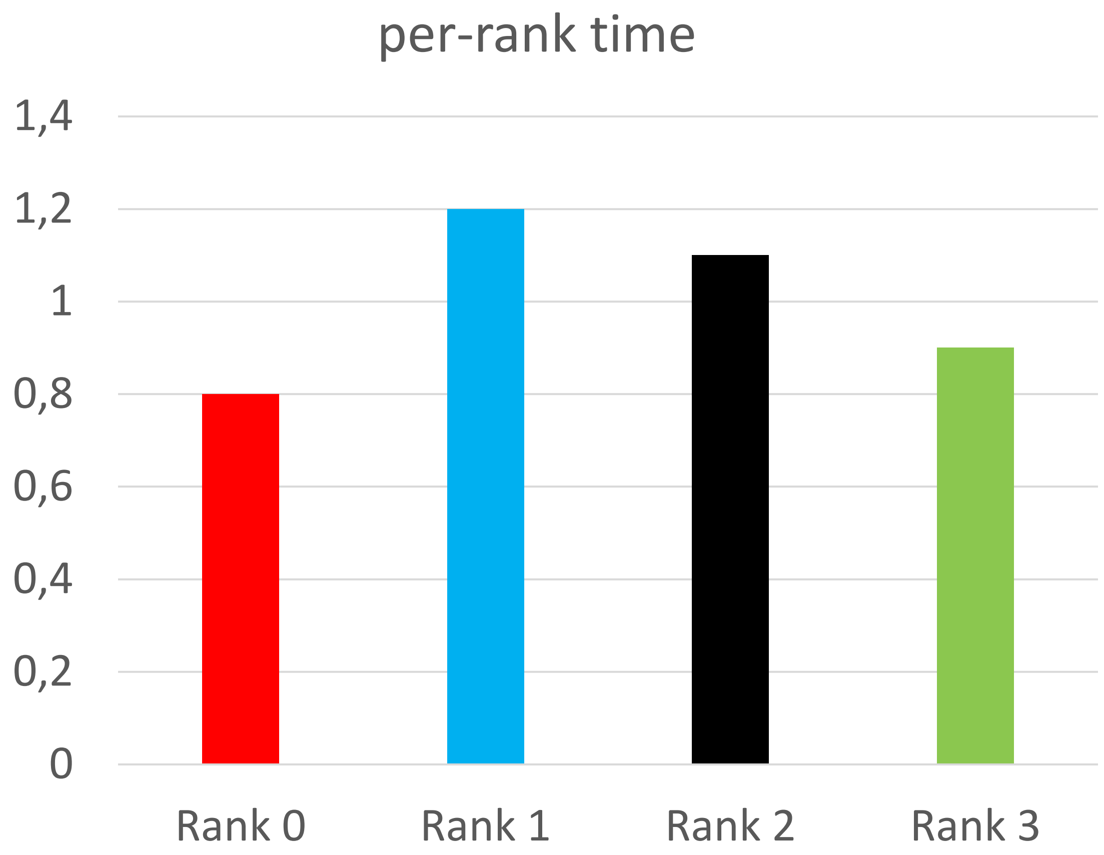
Possibly the two most important considerations

- Your domain cannot be decomposed efficiently?
  - change your algorithm design
- Your data structure cannot be decomposed efficiently?
  - change your implementation design
- Consider those before you start coding!

Load Imbalance

Load imbalance

- refers to the phenomenon of not all ranks finishing their work at the same time
  - leads to ranks waiting on others
  - unnecessarily increases wall time
  - …

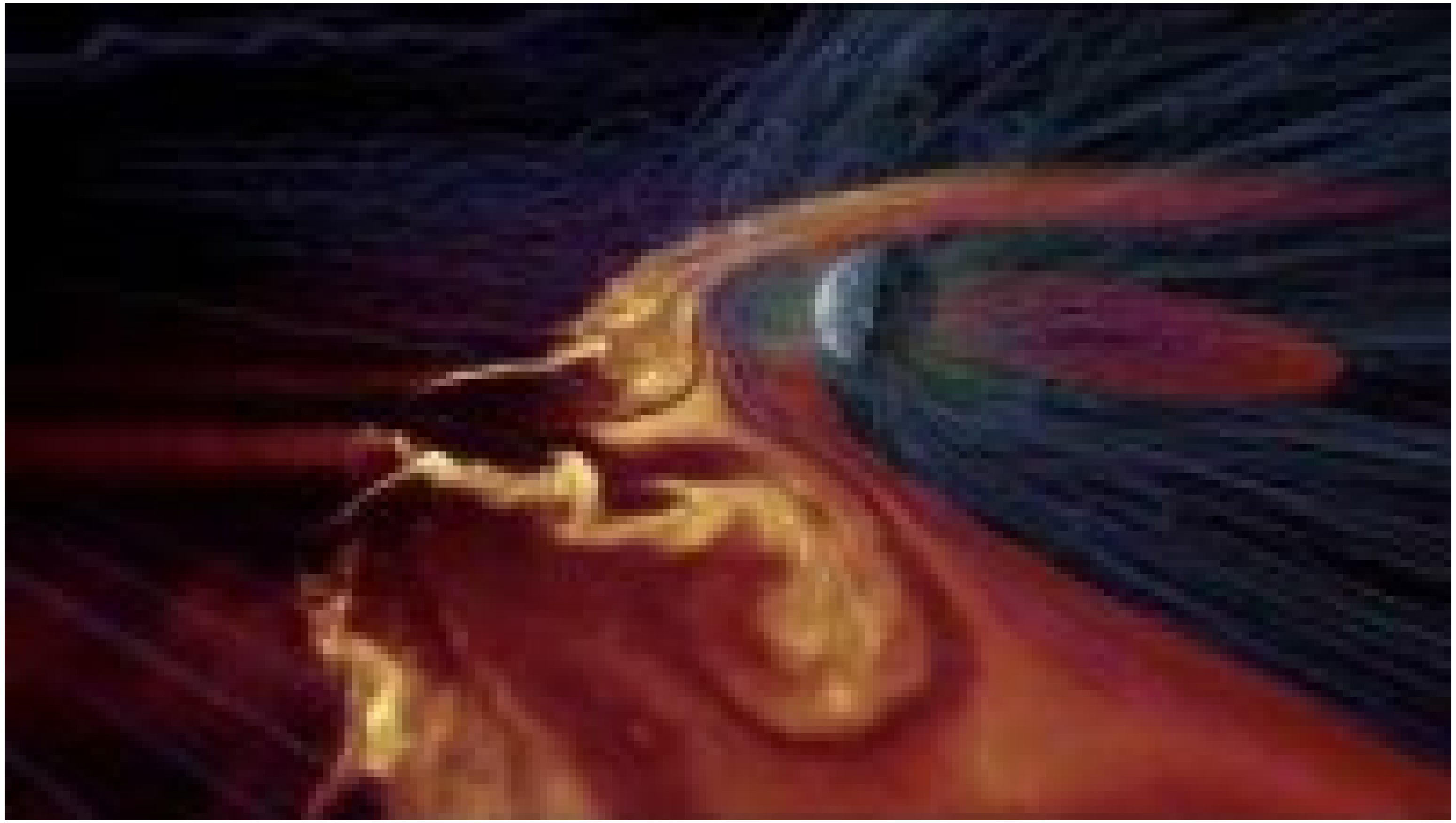


Static vs. dynamic load imbalance

- static load imbalance
  - caused by initial conditions, e.g.
    - mountains vs. plains
    - ocean shore vs. open sea
    - remainder in integer division
  - does not change during application execution
  - mitigation usually incurs no runtime overhead after initial setup
- dynamic load imbalance
  - caused by application data
    - moving particles (e.g. galaxy clusters)
    - partial availability of sensor data
    - convergence of iterative algorithms
  - does change during application execution
  - requires rebalancing (e.g. at fixed intervals, when reaching limits, …)
    - definitely can incur runtime overhead

Dynamic load imbalance: binary star system

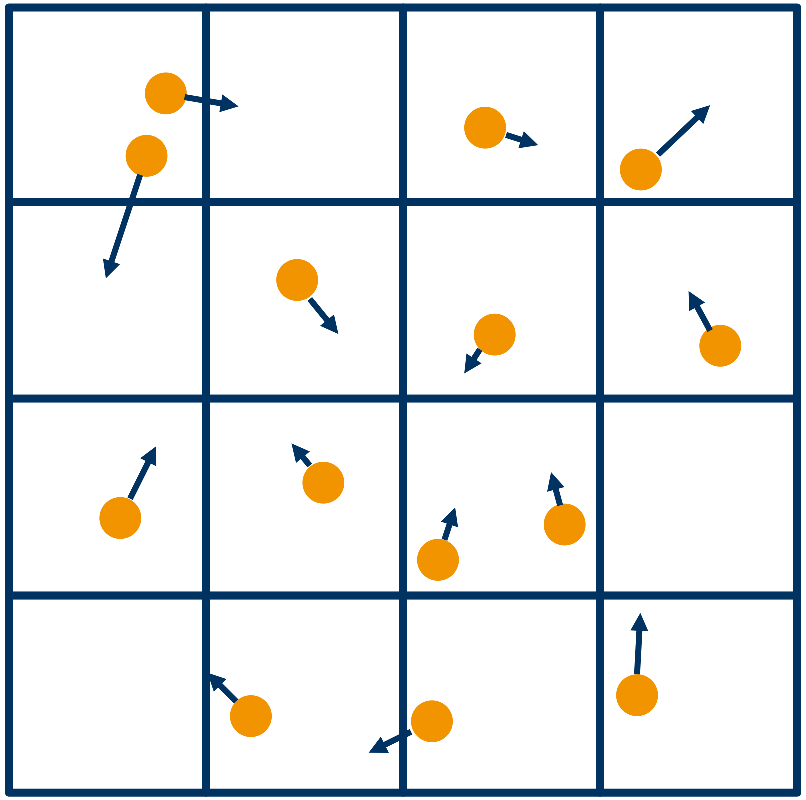
Space weather prediction cont'd



https://www.youtube.com/watch?v=piehWYdlOQA

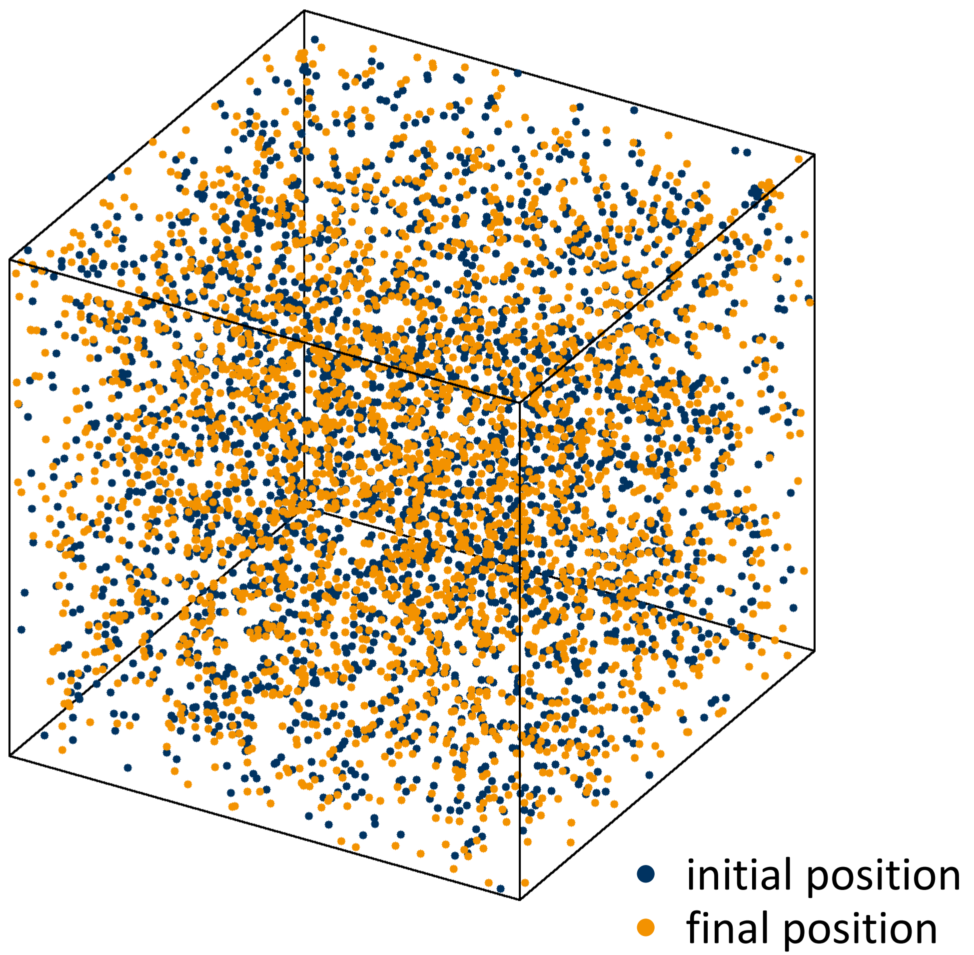
Particle-in-cell

- charged particles move through a grid of cells representing an electromagnetic field
  - the field exerts a force on the particles
  - the particle movement affects the field
  - e.g. electrons, protons, or alpha particles hitting Earth’s magnetosphere



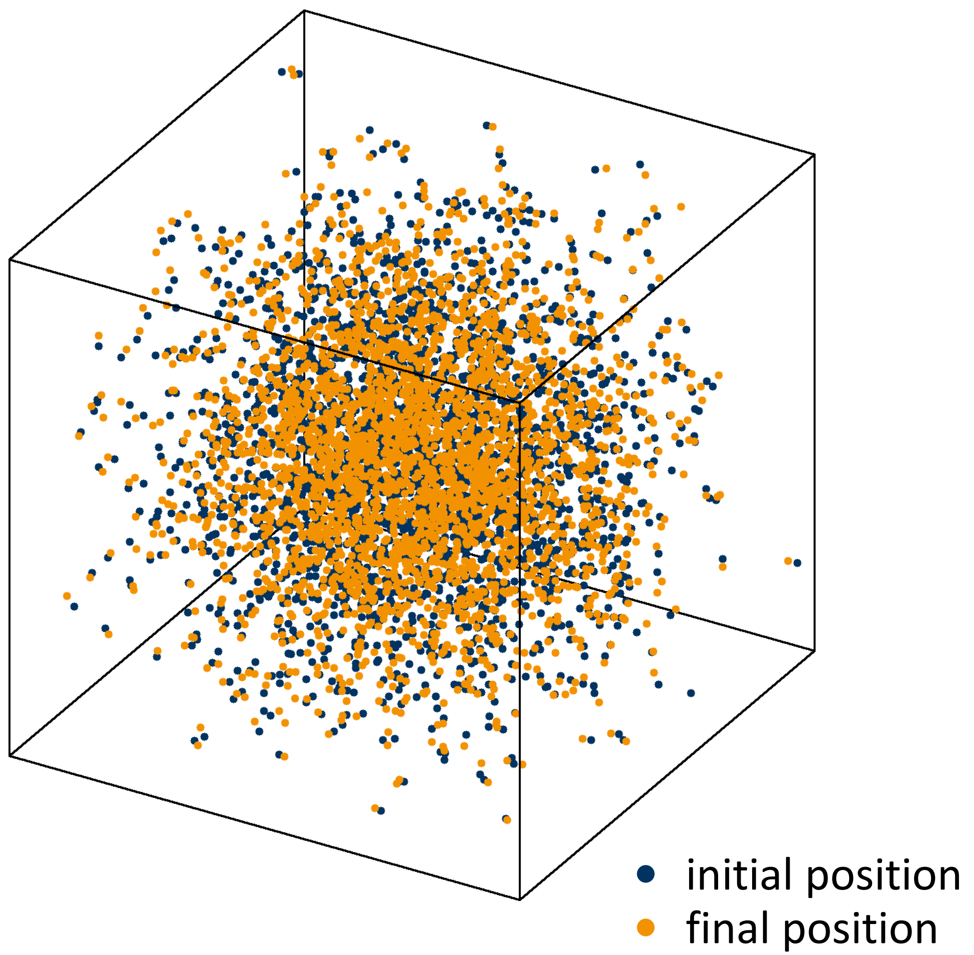
Particle-in-cell use case: uniform

- static load balance: particles uniformly distributed across domain
- dynamic load balance: particle positions almost constant



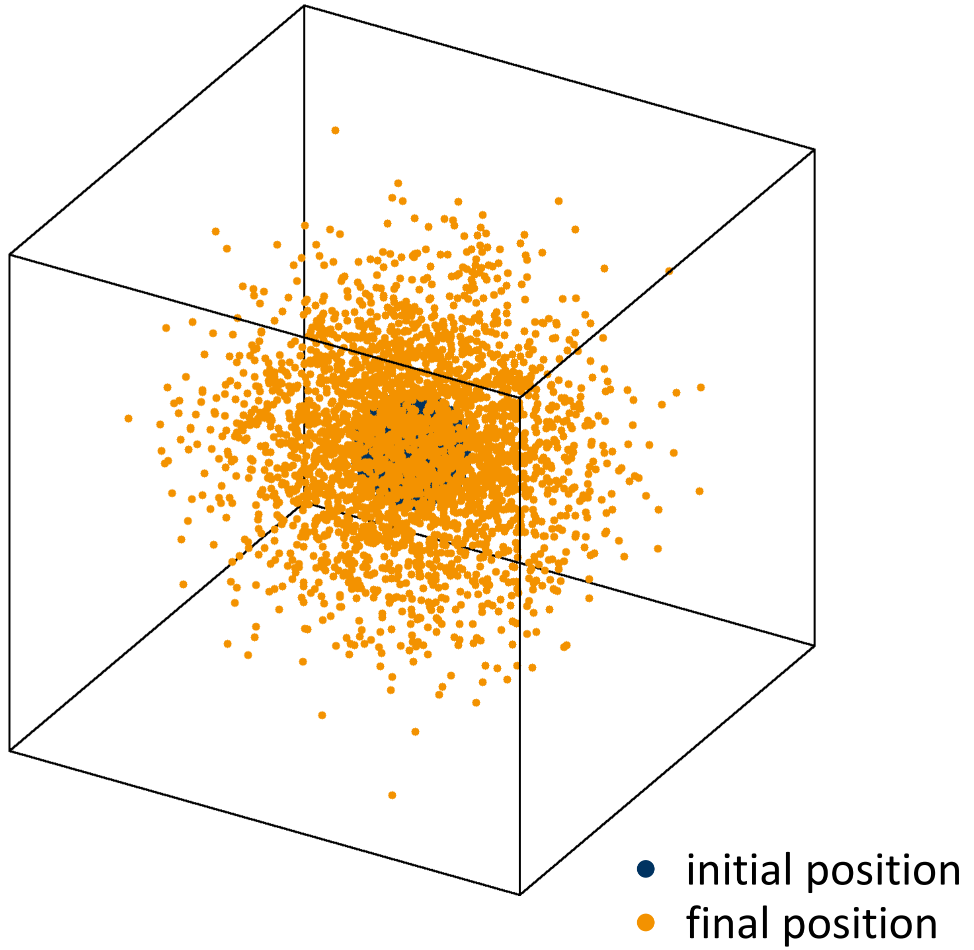
Particle-in-cell use case: cluster

- static load imbalance: particles non-uniformly distributed across domain
- dynamic load balance: particle positions almost constant



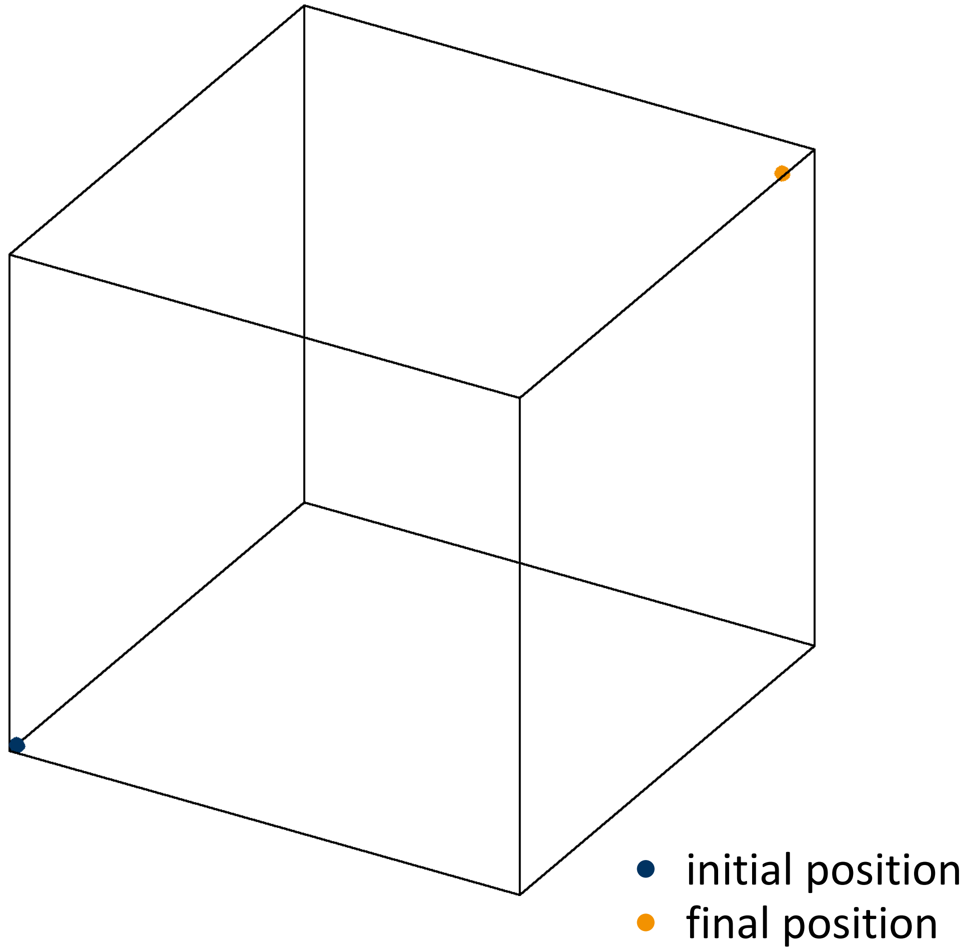
Particle-in-cell use case: explosion

- static load imbalance: particles non-uniformly distributed across domain
- dynamic load imbalance: particle positions changes drastically

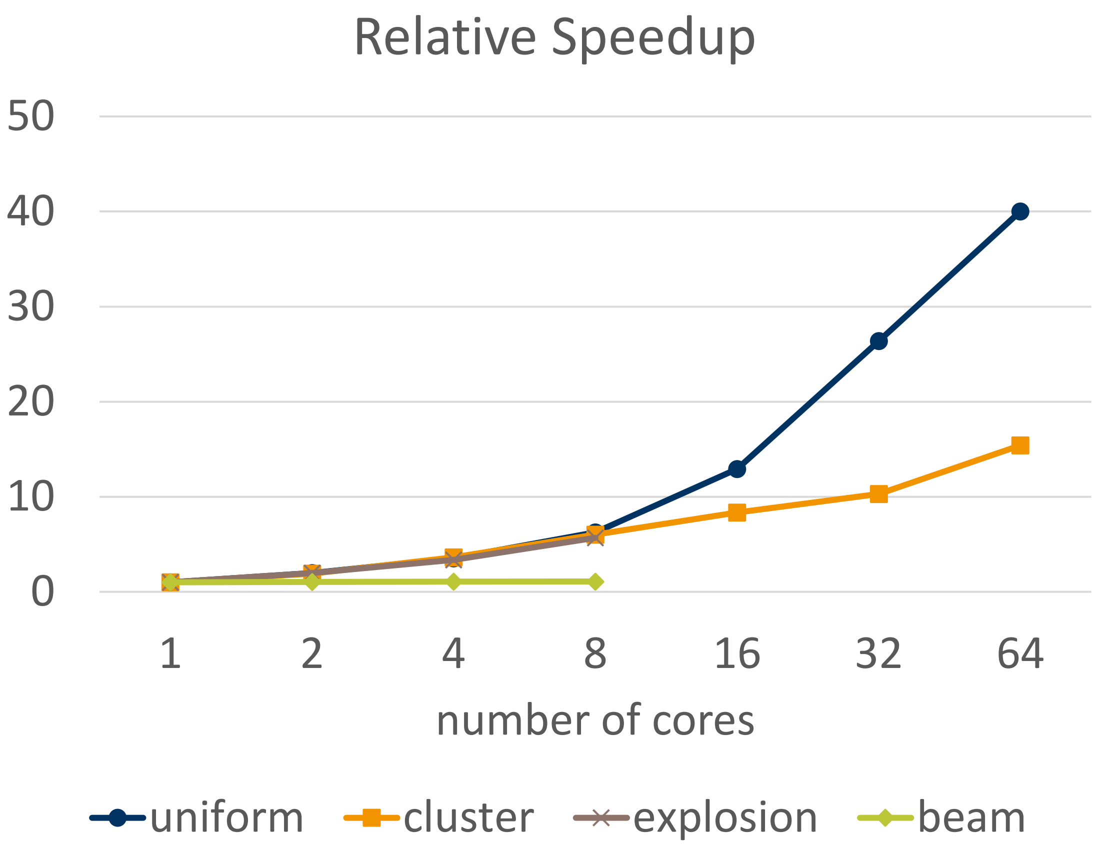


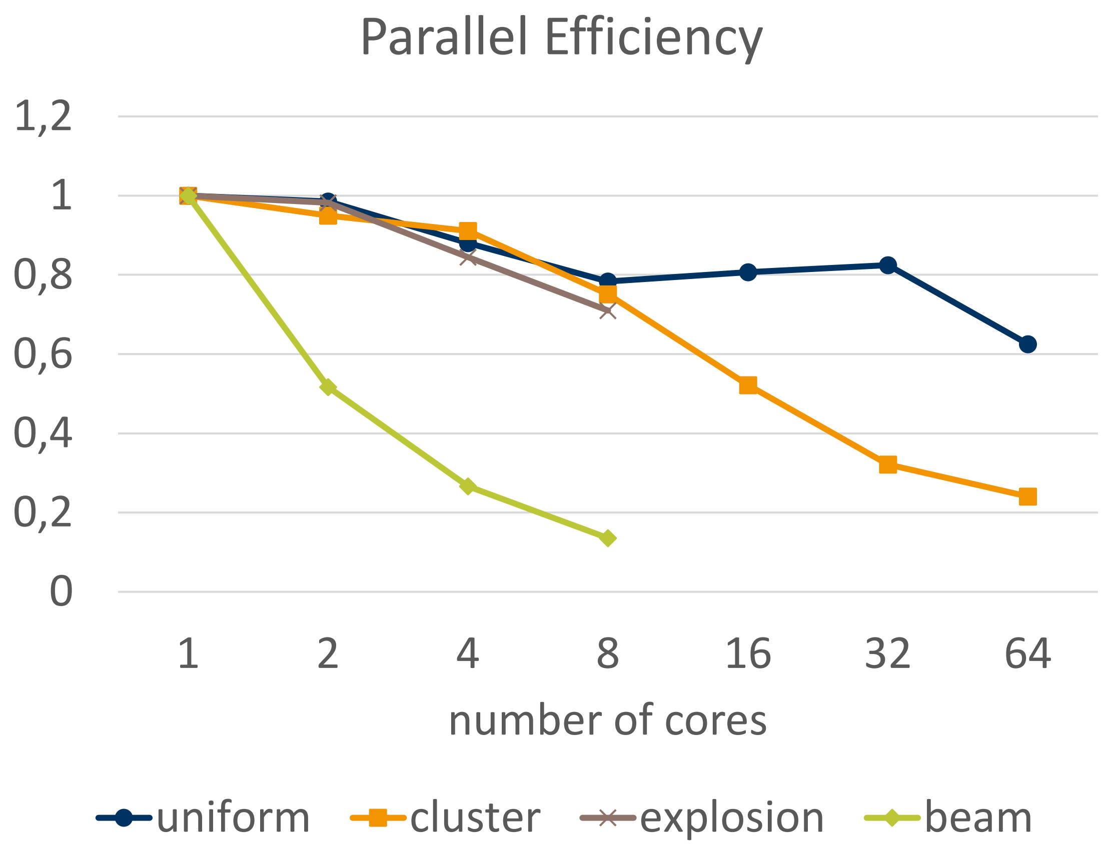
Particle-in-cell use case: beam

- static load imbalance: particles non-uniformly distributed across domain
- dynamic load imbalance: particle positions changes drastically
- extreme case for testing load-balancing algorithms
  - but could be real, e.g. beam of electrons



iPIC3D comparison, strong scaling, 1M particles, LCC2

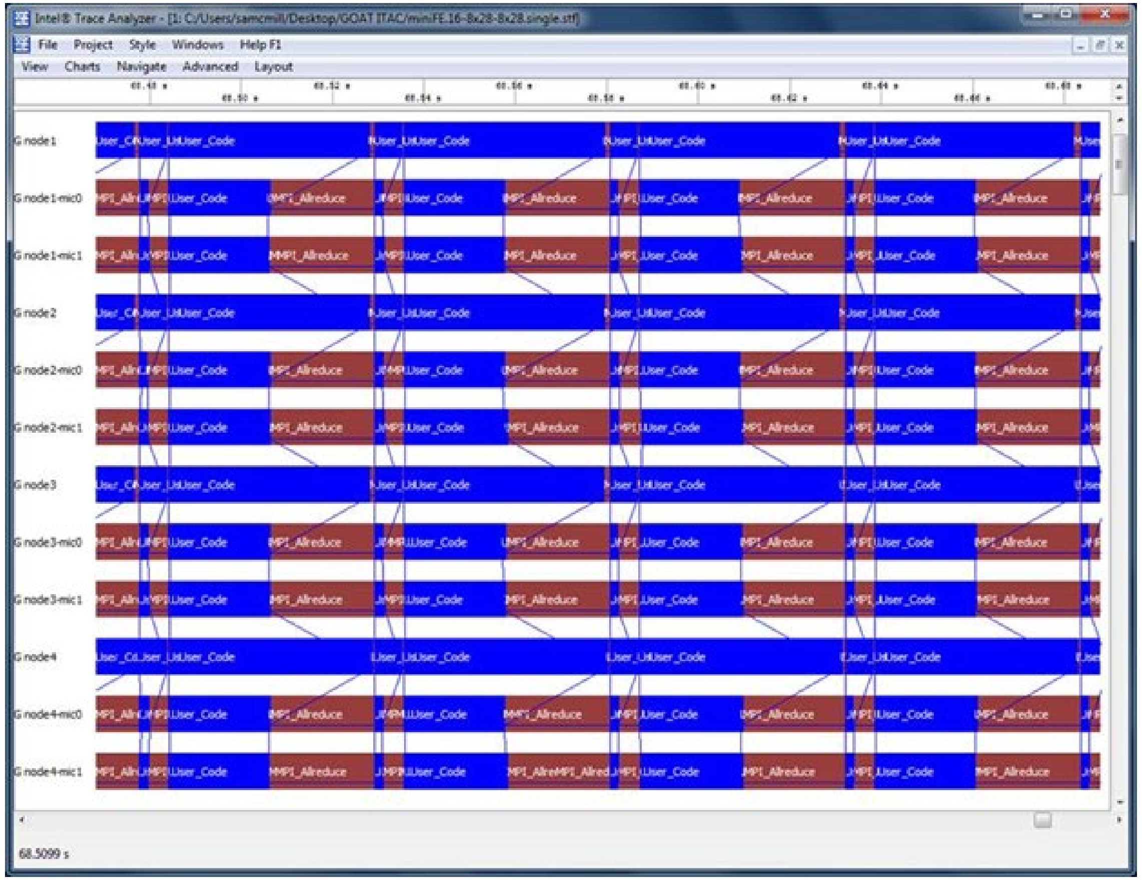




Detecting load imbalance with tools

- screenshot on the right shows Intel Trace Analyzer
  - miniFE benchmark of Sandia National Labs
  - work shared between CPU (lines 1, 4, 7 and 12) and Xeon Phi accelerators
  - blue bars are application work
  - red bars are MPI synchronization

- Xeon Phis are waiting 50% of the time for the CPUs!



Dealing with load imbalance

- quantify the amount of work of domain sub-ranges
  - e.g. heat stencil: `num_elements = end_index – start_index`
  - often requires meta data for more complex data structures

- choose a domain decomposition that allows
  - to split the workload between the given number of ranks in even shares
  - if required, rebalance the workload during runtime
    - often requires the ability to split and merge chunks of work/data

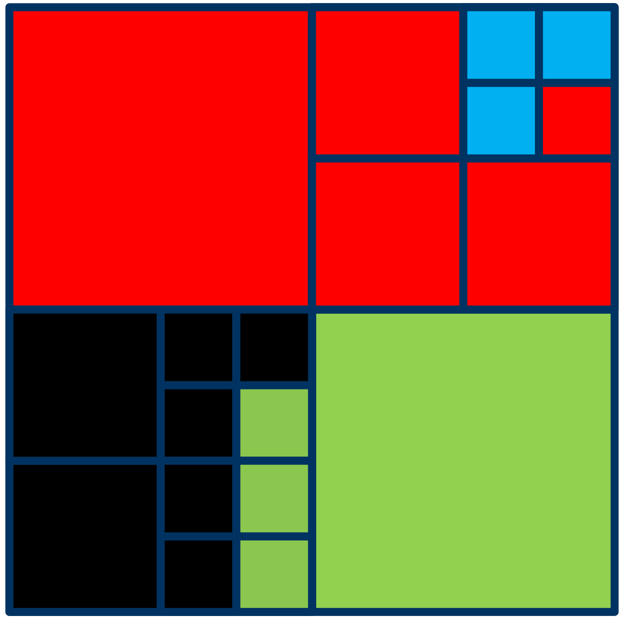
- trade-off in case of black box problems
  - many chunks: easy to balance load but increased management overhead
  - few chunks: little overhead, but difficult to balance load
  - note: decomposition itself is also overhead!

Dealing with load imbalance cont’d

- reactive
  - monitor system state (introduces overhead!)
  - when load imbalance is detected, try to mitigate
- predictive
  - build a load imbalance model
  - query the model for the state of the system in the near future
  - shift the workload before load imbalance occurs
- huge (!) amount of research dedicated to this field
  - a) find approaches of mitigating load imbalance
  - b) find ways of doing this automatically without user intervention (holy grail)
  - c) find models capable of predicting amount of work before execution (holy grail)

Dealing with load imbalance: static case

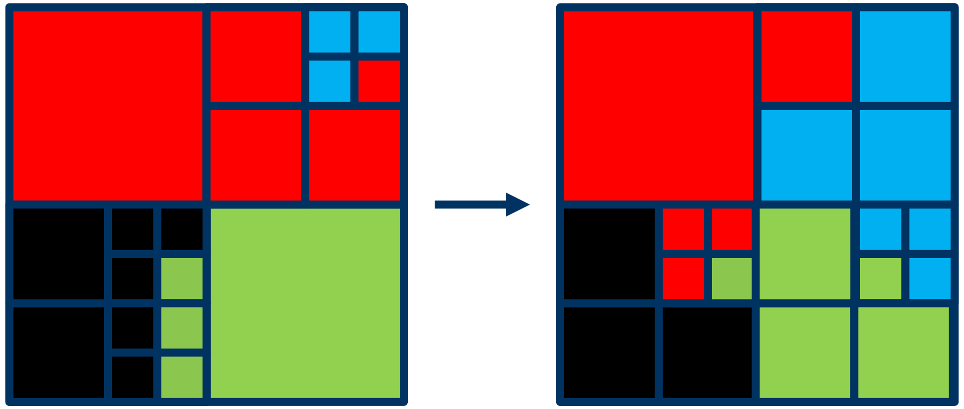
- static
  - first choose smart domain decomposition depending on predicted workload
  - then just execute as normal



- example on the right: quadtree
  - more work ➔ smaller subregions
  - 3D version: octree
  - called “Bounding Volume Hierarchies”

Dealing with load imbalance: dynamic case

- dynamic
  - some form of repeated balancing required, e.g.
    - at certain intervals
    - when reaching certain thresholds



t_x

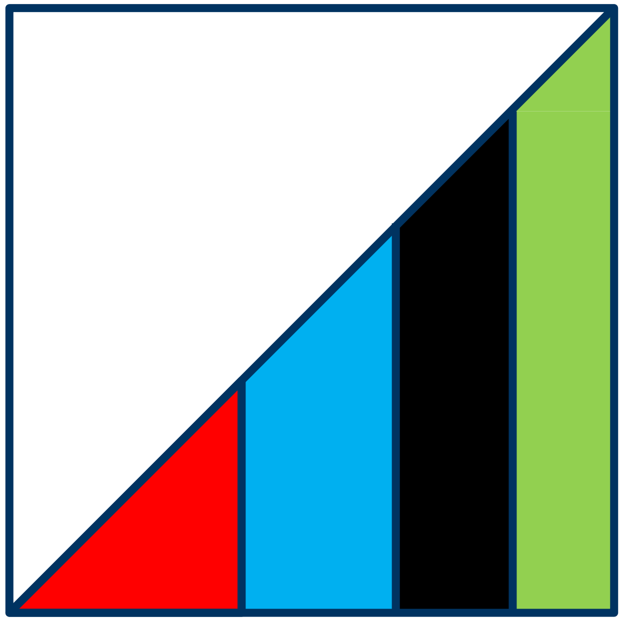
t_{x+1}

Insight into research: load balancing in AllScale

Dealing with load imbalance: domain-specific knowledge

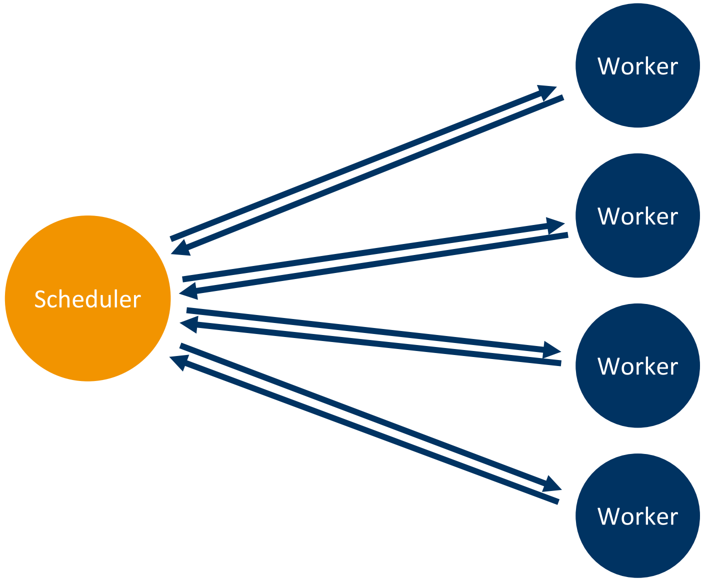
- if present, use domain-specific knowledge about the problem
  - e.g. structured problem with a workload gradient depending on an index

```
for(int i = 0; i < N; ++i) {
    for(int j = 0; j < i; ++j) {
        ...
    }
}
```

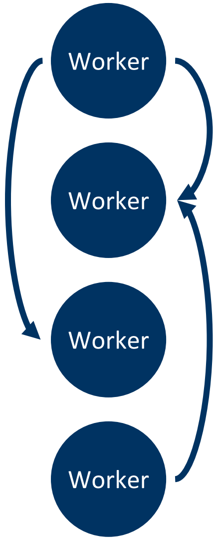


Centralized vs. decentralized

- Centralized
  - also master/worker, etc.
  - easy to implement
  - eventually poses a bottleneck



- Decentralized
  - usually some form of work-stealing
  - more difficult to implement
  - can scale to very large systems



Work sharing vs. work stealing

- Work sharing
  - push work to a local work queue
    - if queue gets too large, send work to other workers
  - entails communication during busy phases
  - hard to grasp load of other workers
  - work might be pushed around a lot before it is actually processed
- Work stealing
  - local worker pulls work to a local queue
    - if local queue is empty, pull more work from other workers
  - entails communication while idle
  - work will only be requested if local queue empty
  - work will be stolen at most once before being processed

Further means of load balancing

- use “smart” work assignment strategies instead of smart domain decomposition
  - assign small chunks of work in round robin fashion or in random order
    - c.f. OpenMP’s loop scheduling strategies
  - assign small chunks using space-filling curves, diffusion models, etc…
  - remember tradeoff between balanced load and overheads

- configure hardware
  - change CPU clock frequencies, e.g. slow down cores with little work
  - switch off hardware, e.g. cores that finished early
  - doesn’t reduce wall time but saves power and energy

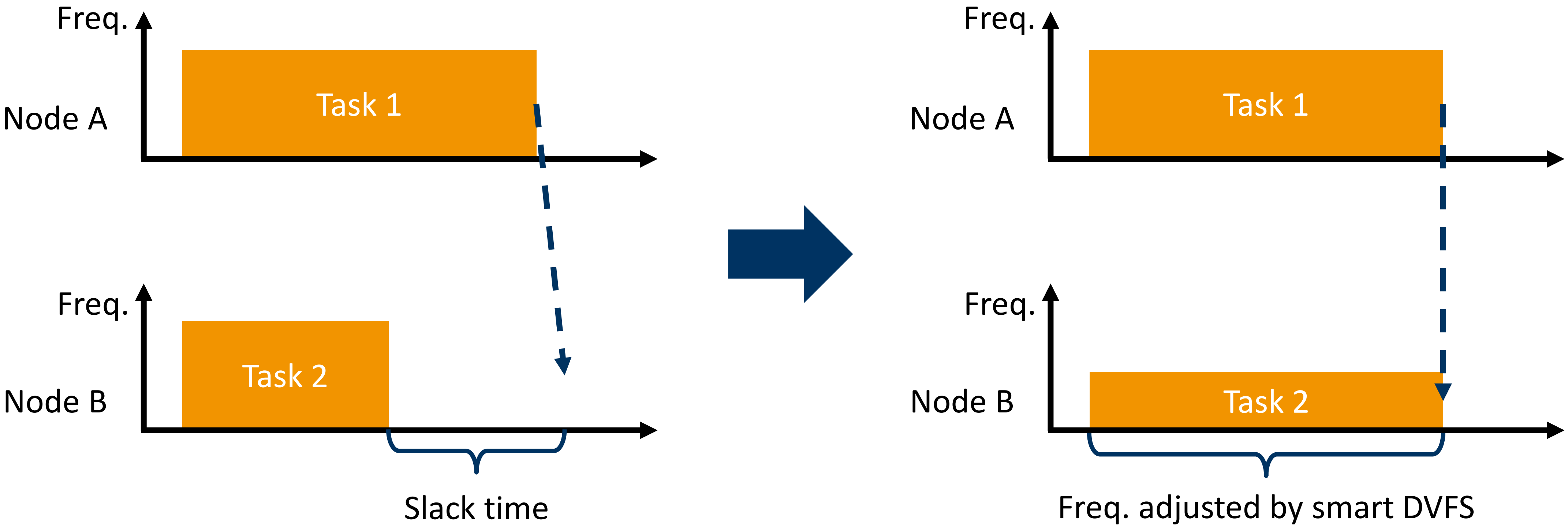
- biggest issue of all: which strategy to select for which problem…

MPI slack time optimization

- Recognize slack time in parallel applications
  - Wait states
  - Periods of extended memcopy operations or I/O
  - Even computation if not on the critical path
  - etc.

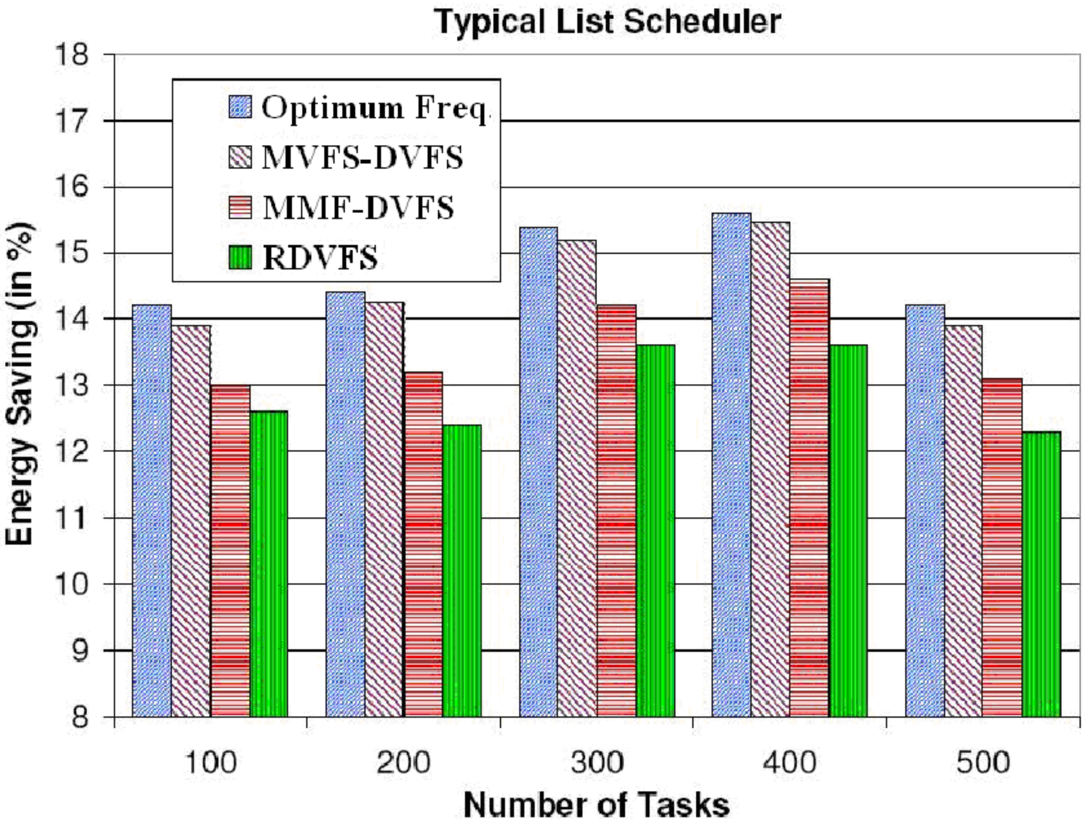
- Use DVFS to reduce energy footprint with minimal impact on wall time
  - Lots of work on that from 5-15 years ago

Slack time optimization example



Slack time optimization results

- Rizvandi et al. „Some Observations on Optimal Frequency Selection in DVFS–based Energy Consumption Minimization”
  - Simulations with 3000 randomly generated task graphs
  - Energy savings 10-20%



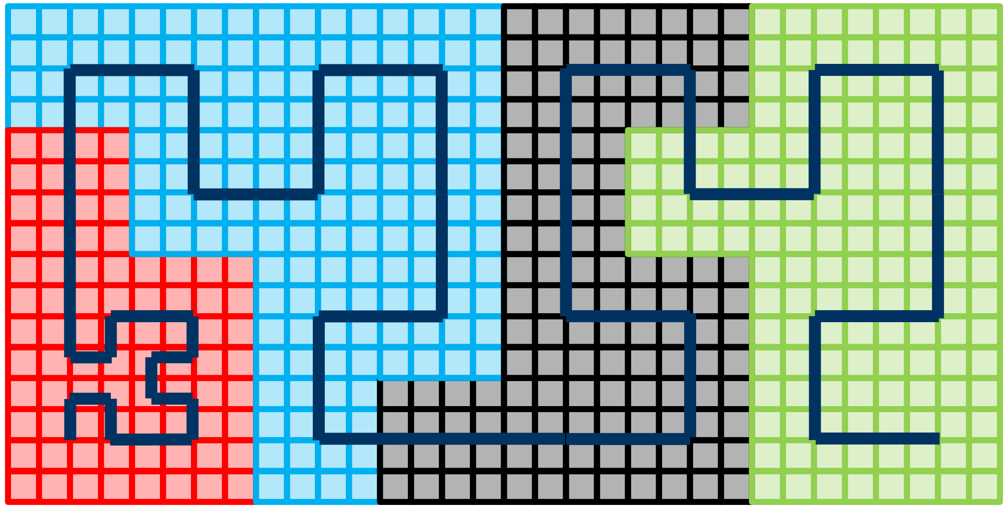
https://arxiv.org/ftp/arxiv/papers/1201/1201.1695.pdf

Load balancing method examples

- Diffusion
  - using e.g. a heat-stencil-like optimizer, but distribute workload instead of temperature
- Bidding
  - workload is advertised by worker, others can bid with varying “prices” (e.g. inverse idle metric), highest bid wins
- Random
  - randomly select N workers for each chunk of work, assign the one with the least load
- often inspired by real-life (im)balances
  - including spatial locality aspect
  - many often implementing in a hierarchical fashion

Space-filling curves (e.g. Hilbert)

- Map a single-dimensional index onto a n-dimensional problem space
  - self-avoiding, space-filling
  - preserve high spatial locality (contrary to e.g. Cartesian partitioning)
  - comparatively low overhead, efficient load balancing



- Downside: more complicated boundary definition

Additional reasons for load balancing

- external load caused by other users
  - not only on the CPUs, consider e.g. network or I/O
- heterogeneous systems
  - e.g. decide how to split work between CPUs and GPUs
- dynamic availability of additional resources
  - c.f. cloud computing

Tales from the Proseminar

Tales from the Proseminar: unrolling

- Which code version is faster?

```c
#define NDIMS 3

int foo(int a) {
    for(int i = 0; i < NDIMS; ++i) {
        a *= a;
    }
    return a;
}
```

```c
int foo(int a) {
    a *= a;
    a *= a;
    a *= a;
    return a;
}
```

Tales from the Proseminar: unrolling cont’d

- Both versions compiled with gcc 11.2 with –O0

Left: x86-64 gcc 11.2 -DVERSION=0 … Assembly

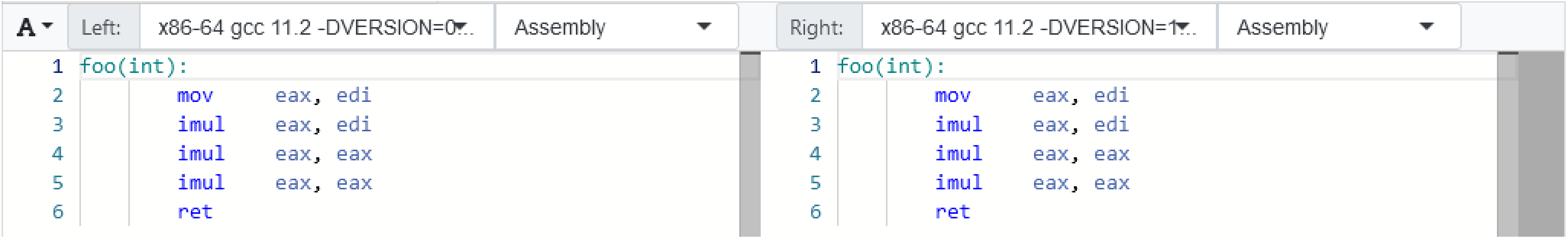
```
foo(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-4], 0
        jmp     .L2
.L3:
        mov     eax, DWORD PTR [rbp-20]
        imul    eax, eax
        mov     DWORD PTR [rbp-20], eax
        add     DWORD PTR [rbp-4], 1
.L2:
        cmp     DWORD PTR [rbp-4], 2
        jle     .L3
        mov     eax, DWORD PTR [rbp-20]
        pop     rbp
        ret
```

Right: x86-64 gcc 11.2 -DVERSION=1 … Assembly

```
foo(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

Tales from the Proseminar: unrolling cont’d

- Both versions compiled with gcc 11.2 with –O1
  - See for yourself: https://godbolt.org/z/s3b4Wv68d



Tales from the Proseminar: memory layout

- Compiled with both –O0 and –O3

```c
typedef struct Foo {
    float x;
    float y;
    float z;
} Foo;

#define SIZE 8

int main() {
    Foo foo[SIZE];
    float* bar = malloc(3 * SIZE * sizeof(float));
    ...
    printf("offset (array): %d\n", (void*)&(foo[1].z)-(void*)foo);
    printf("offset (struct): %d\n", (void*)&bar[5]-(void*)bar);

    return foo[1].x;
}
```

```
offset (struct): 20
offset (array): 20
```

Summary

- domain decomposition
  - means of controlling communication overhead
- load (im)balance
  - static: split the workload evenly among ranks
  - dynamic: continuously rebalance as required
- “Tales from the Proseminar”
  - compiler-based loop unrolling & memory layout

Image Sources

- Binary Star System video: courtesy of Ralf Kissmann & David Huber, University of Innsbruck
- Space Weather Prediction: https://twitter.com/maven2mars/status/984440044659159040
- Intel Trace Analyzer: https://software.intel.com/en-us/articles/understanding-mpi-load-imbalance-with-intel-trace-analyzer-and-collector