



703650 VO Parallel Systems WS2019/2020

Domain Decomposition and Load Balancing

Philipp Gschwandtner

Overview

- ▶ last week's feedback results
- ▶ domain decomposition
- ▶ load (im)balance
- ▶ “Tales from the Proseminar”

Last Week's Feedback Results

- ▶ 12 messages total
 - ▶ overall, you seem very happy
 - ▶ except for a few small complaints/requests/remarks
 - ▶ 6x fast speech...
 - ▶ 1x request for more code samples
 - ▶ 1x request for a break after 45 mins
 - ▶ 1x complaint regarding the time slot
 - ▶ 1x request for team presentations in the PS
- ▶ feel free to give further feedback anytime

Motivation (Load Balancing)

► Monte Carlo π

- amount of work per rank depends only on number of samples

```
...  
int localSamples = samples/numRanks  
...
```

► 2D heat stencil

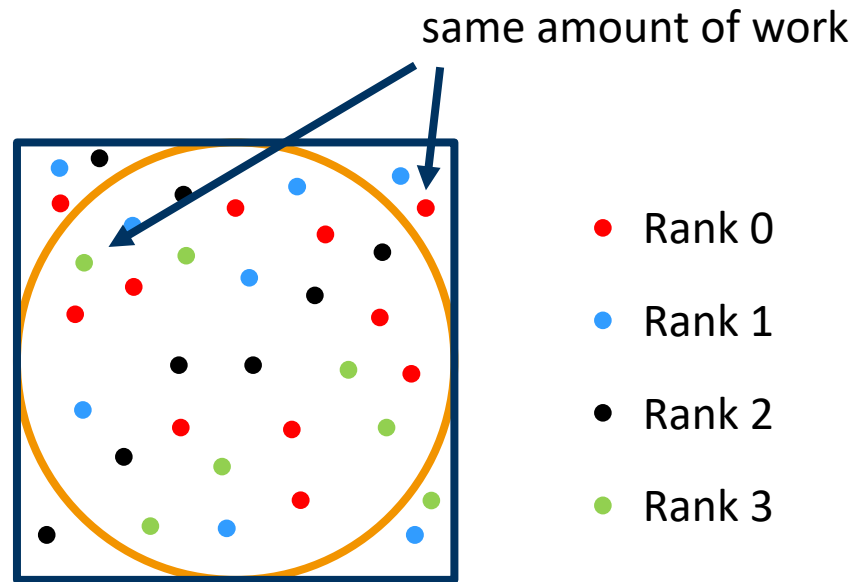
- amount of work per rank depends only on number of elements

```
...  
int localN = problemSize/numRanks  
...
```

Motivation cont'd (Load Balancing)

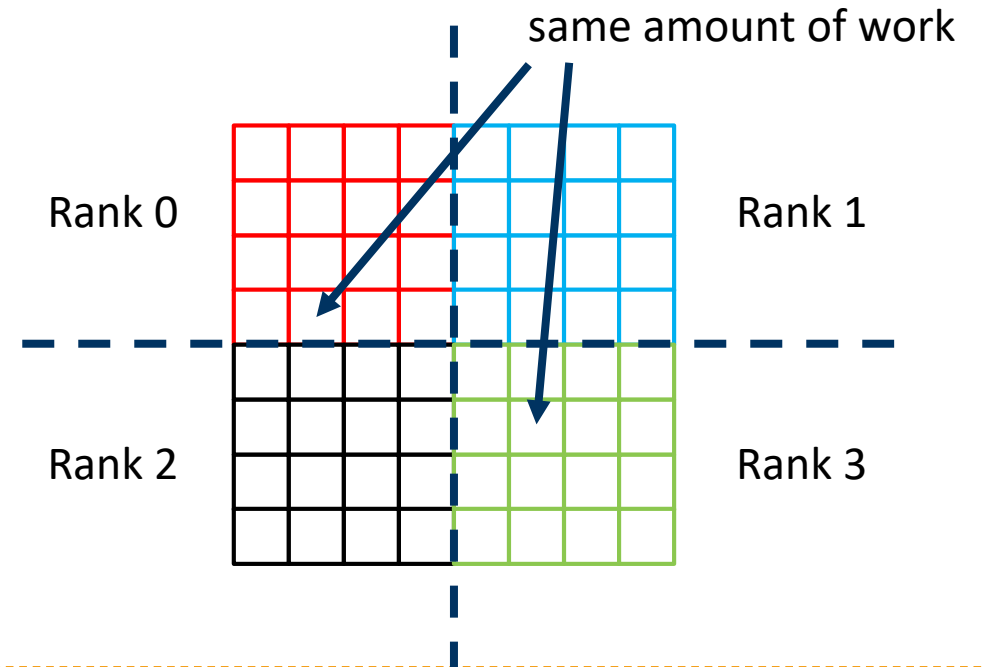
► Monte Carlo π

- 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

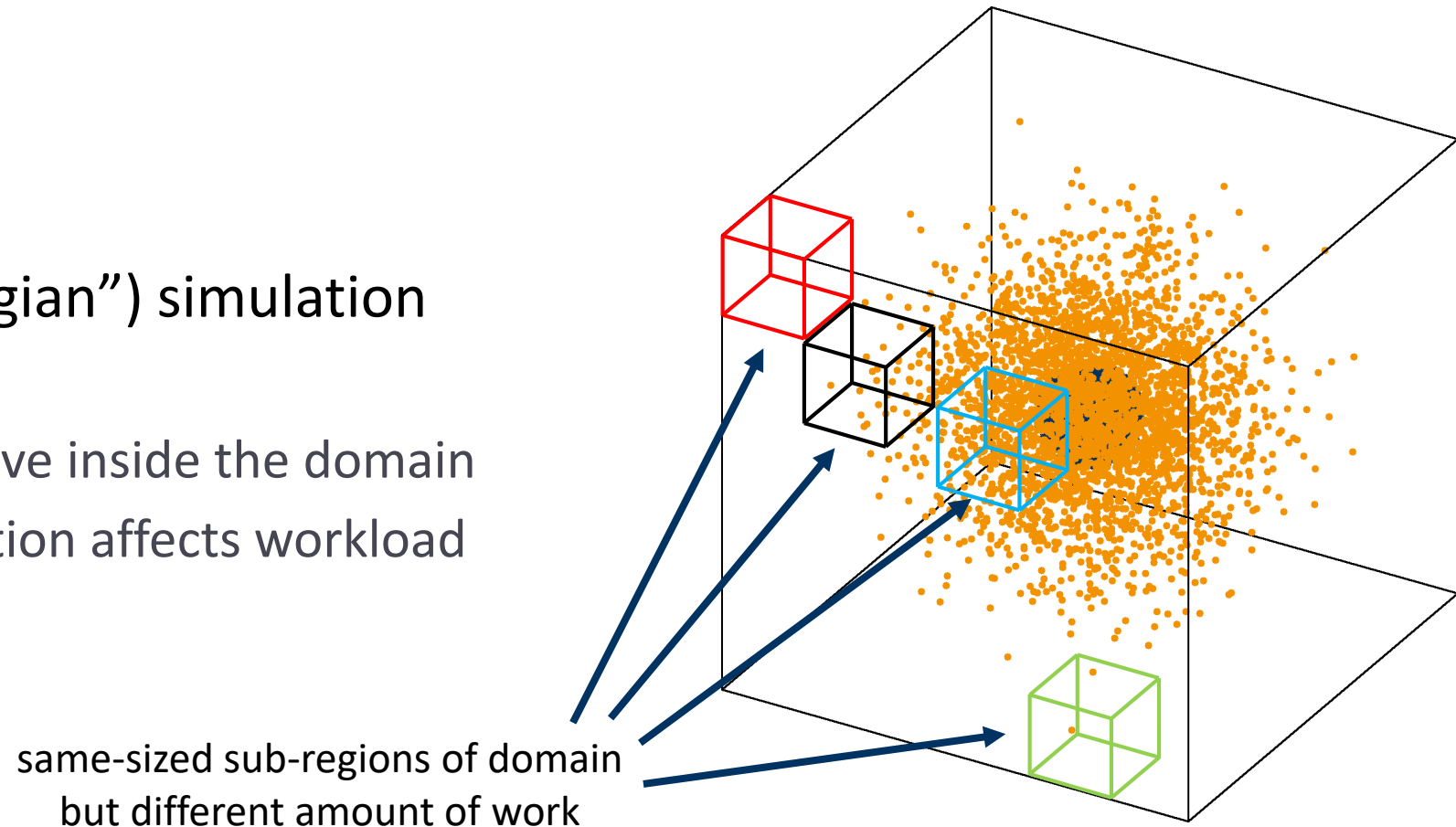


Motivation cont'd (Load Balancing)

- ▶ 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

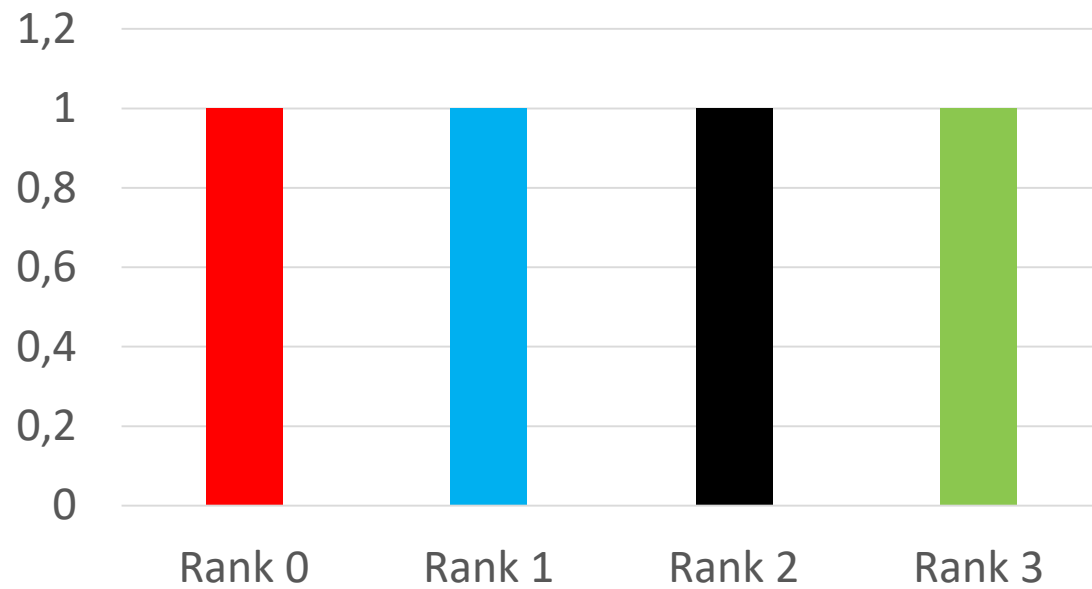
Motivation cont'd (Load Balancing)

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



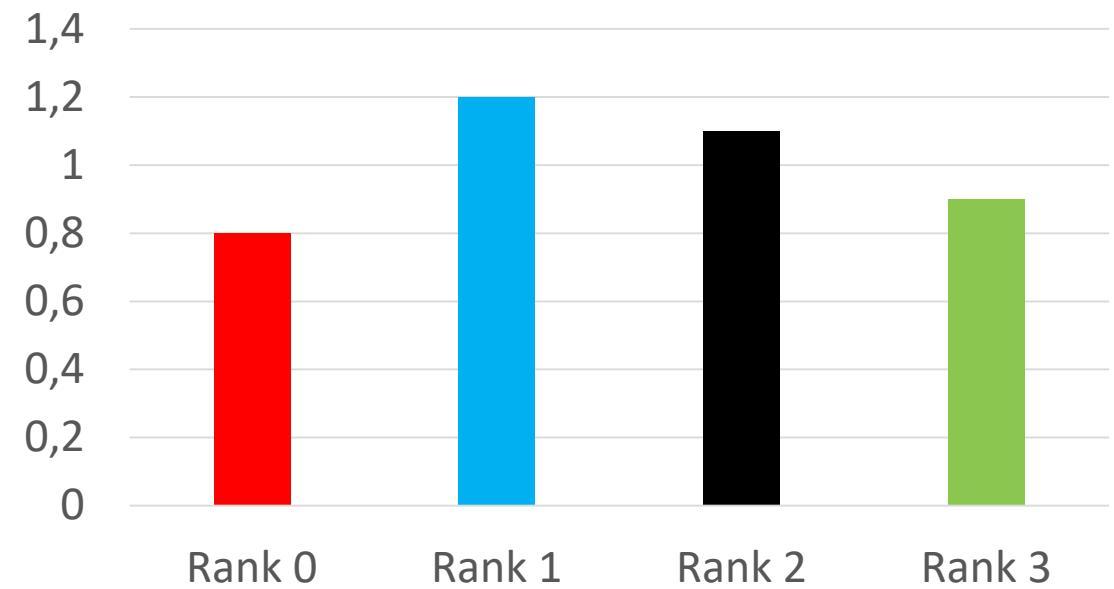
Motivation cont'd (Load Balancing)

per-rank time



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

per-rank time

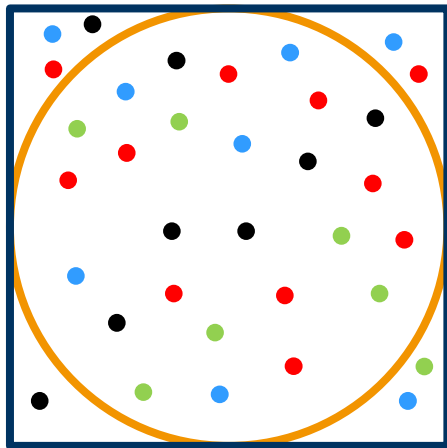


$$t_{\text{cpu}} = 4$$
$$t_{\text{wall}} = 1.2$$

Motivation cont'd (Domain Decomposition)

► Monte Carlo π

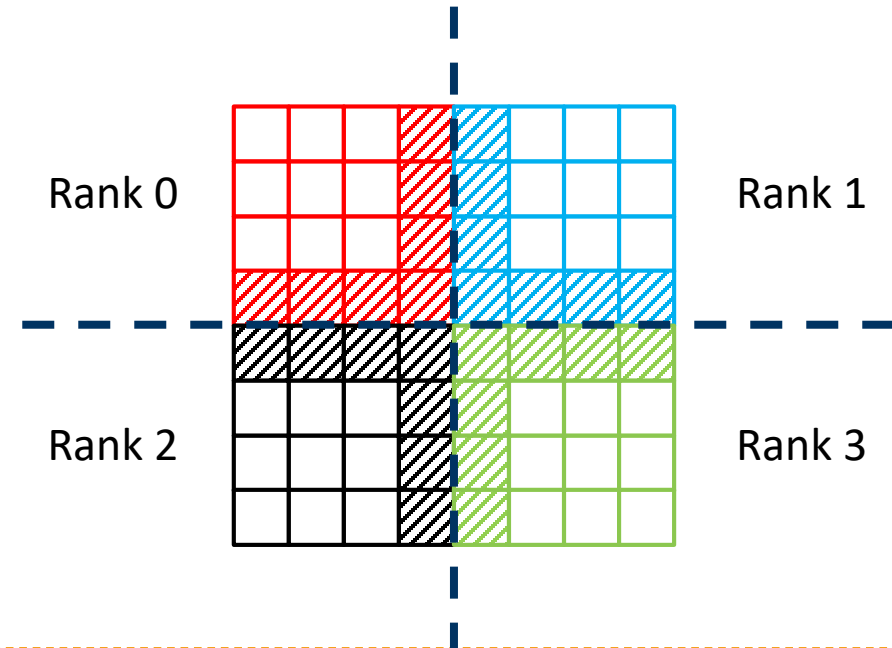
- distributes samples among ranks
- no communication except at the end
- also called “embarrassingly parallel”



- Rank 0
- Rank 1
- Rank 2
- Rank 3

► 2D heat stencil

- distributes grid cells among ranks
- ghost cell exchange required at borders!





Domain Decomposition



Domain Decomposition

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

The Bad News

- ▶ MPI does not offer domain decomposition
 - ▶ sure, there's `MPI_Scatter()` & alike
 - ▶ but it's only the underlying tool
- ▶ you need to take care of this yourself
 - ▶ or better yet, use a library

Cost of a Point-to-point Message

- ▶ $t = \text{latency} + \frac{\text{message size}}{\text{bandwidth}}$
- ▶ note: simplistic view, actual cost depends on
 - ▶ underlying protocols (eager, rendezvous, ...)
 - ▶ additional data copies required
- ▶ latency and bandwidth are fixed properties of the network
 - ▶ note: rank-core mappings!
- ▶ message size (and their number) is what we can influence
- ▶ can be easily computed for simple, regular decompositions

3D Heat Stencil Example: Slabs

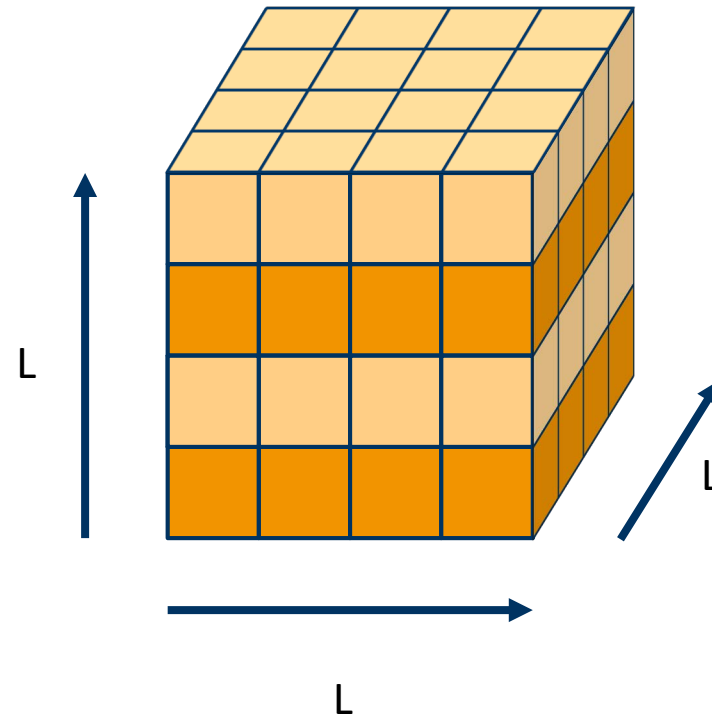
$$c_{1D}(L, N) = \\ L \cdot L \cdot w \cdot 2 = 2wL^2$$

L : size per dimension

N : number of ranks

w : amount of data per element

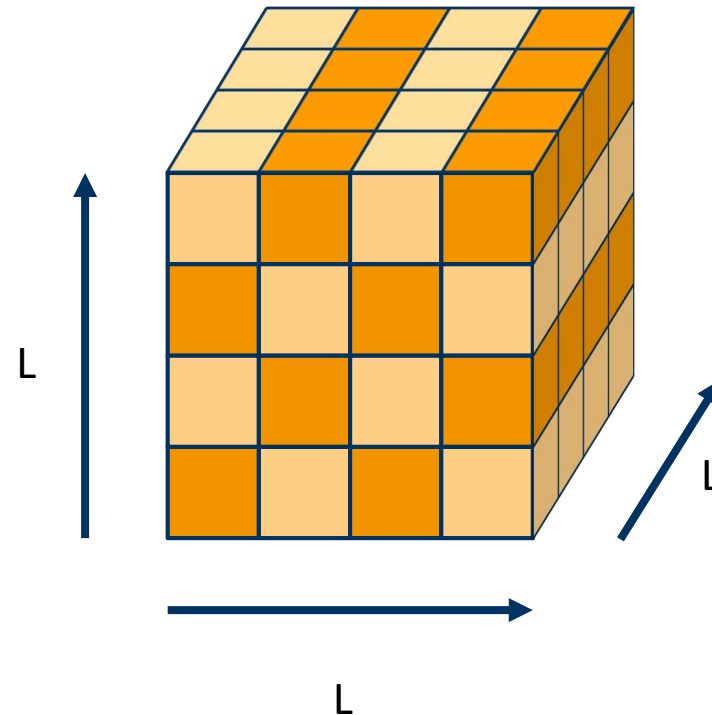
- ▶ pro: easy to implement
- ▶ con: communication volume does not decrease when increasing N



3D Heat Stencil Example: Poles

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

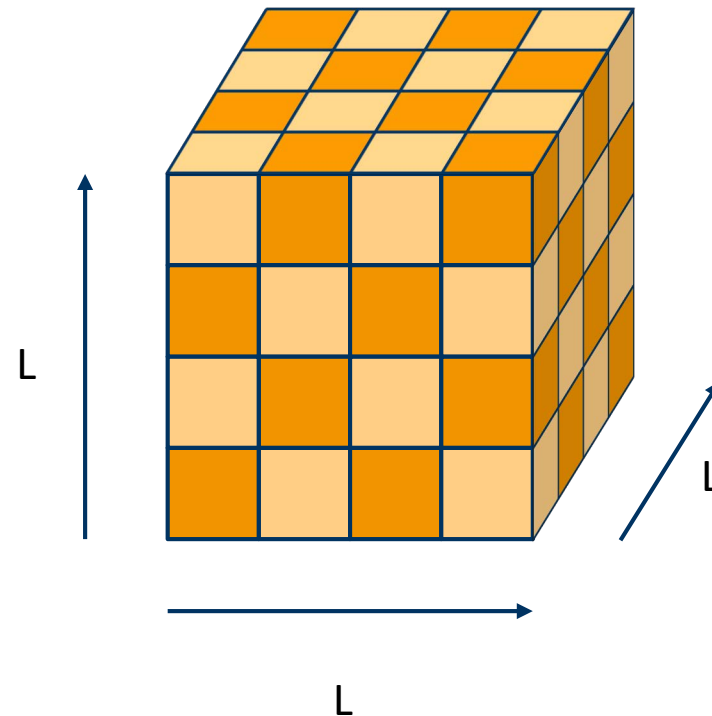
- ▶ pro: communication volume does decrease with increasing N
- ▶ con: surface-to-volume ratio also increases with N
 - ▶ i.e. communication grows disproportionately fast compared to computation



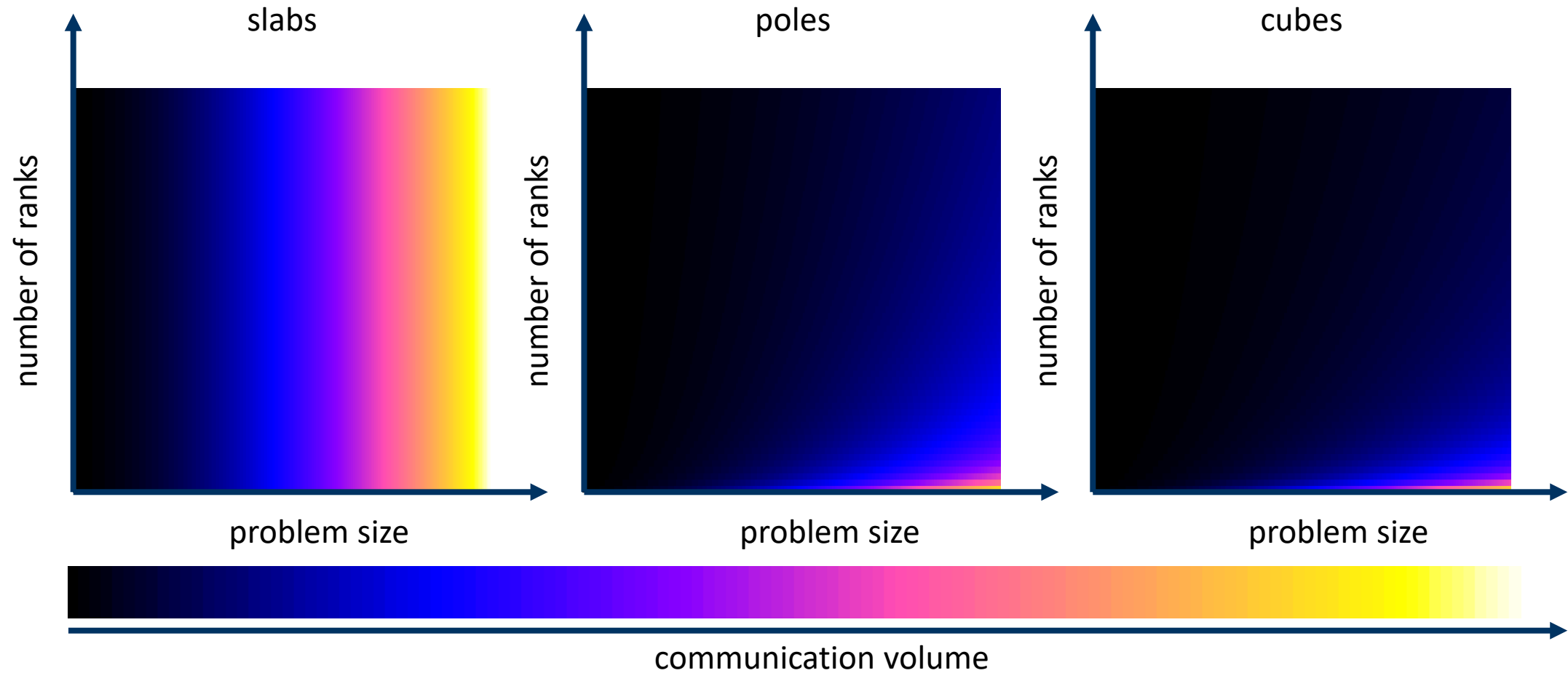
3D Heat Stencil Example: Cubes

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

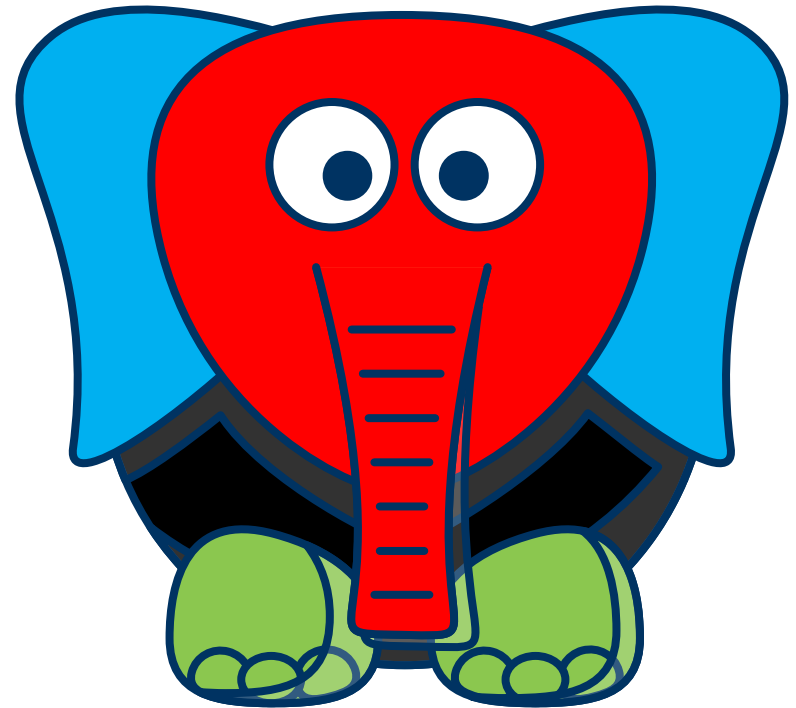
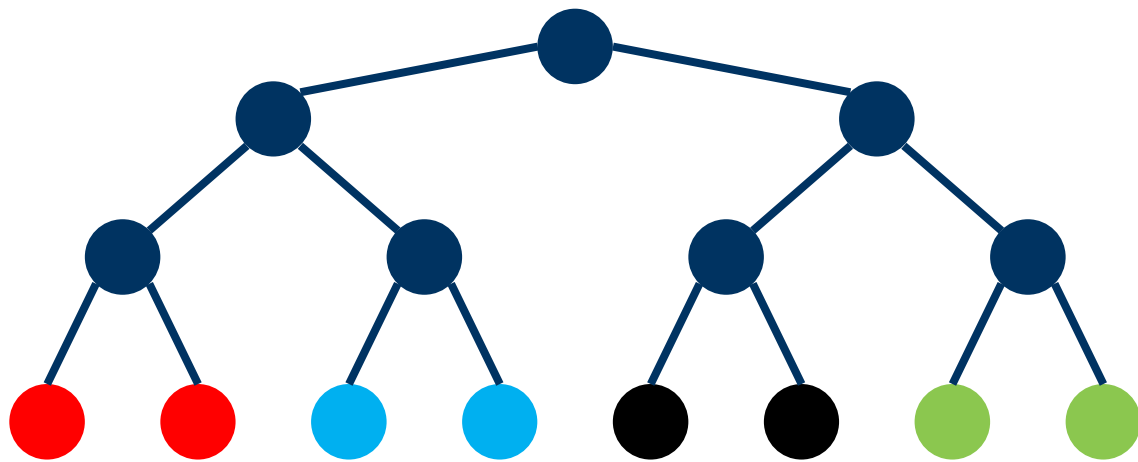
- ▶ pro: communication volume also decreases with increasing N
- ▶ con: still surface-to-volume-ratio issue
- ▶ but also further increase in number of messages, latency might be an issue



Comparing Communication Volumes



Non-rectangular Domain Decomposition



Domain Decomposition Considerations

- ▶ How much data will have to be transferred?
 - ▶ more data requires more bandwidth
- ▶ In how many messages do I need to transfer the data?
 - ▶ additional messages means additional latency and management overhead
- ▶ What's the cache efficiency of the decomposition?
 - ▶ might only be an implementation issue, e.g. columns can be transposed
- ▶ 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
- ▶ Your data structure cannot be decomposed efficiently?
 - ▶ change your implementation

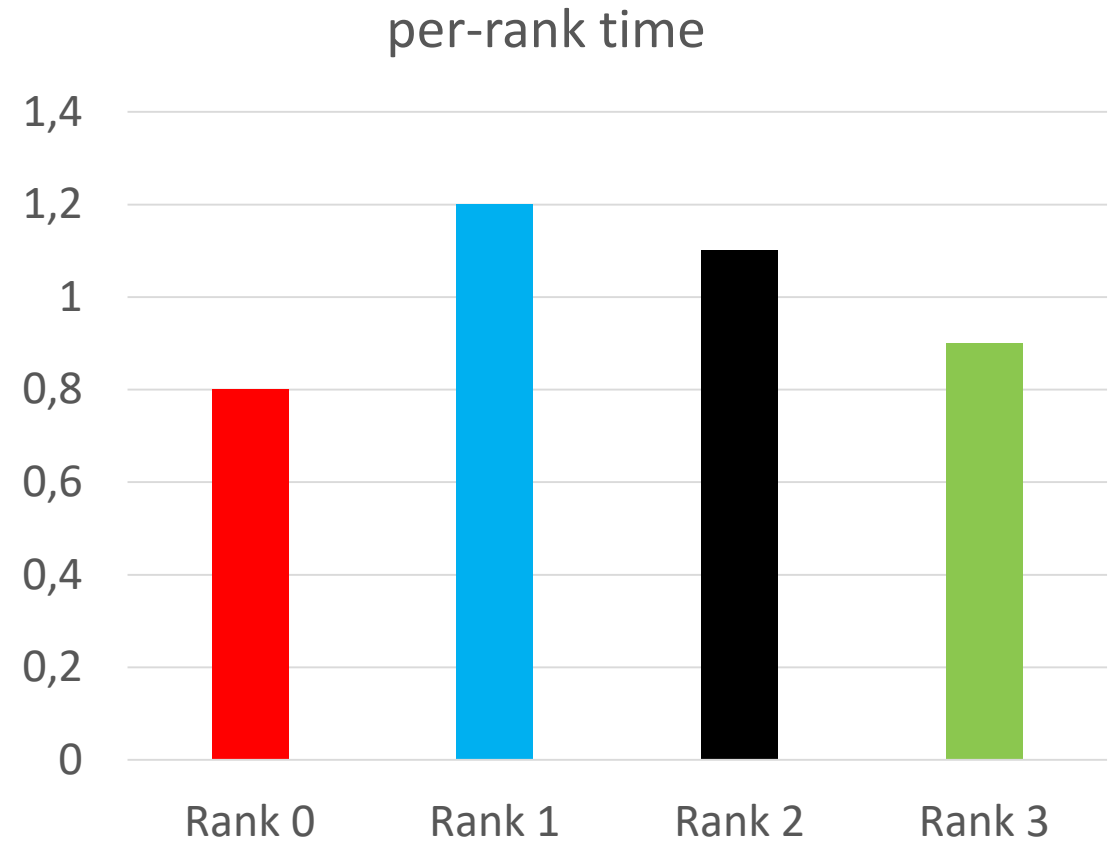


Load Imbalance



Load Imbalance

- ▶ refers to the phenomenon of not all ranks finishing their work at the same time
 - ▶ or ranks waiting on others
 - ▶ or ranks idling for longer periods
 - ▶ ...



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

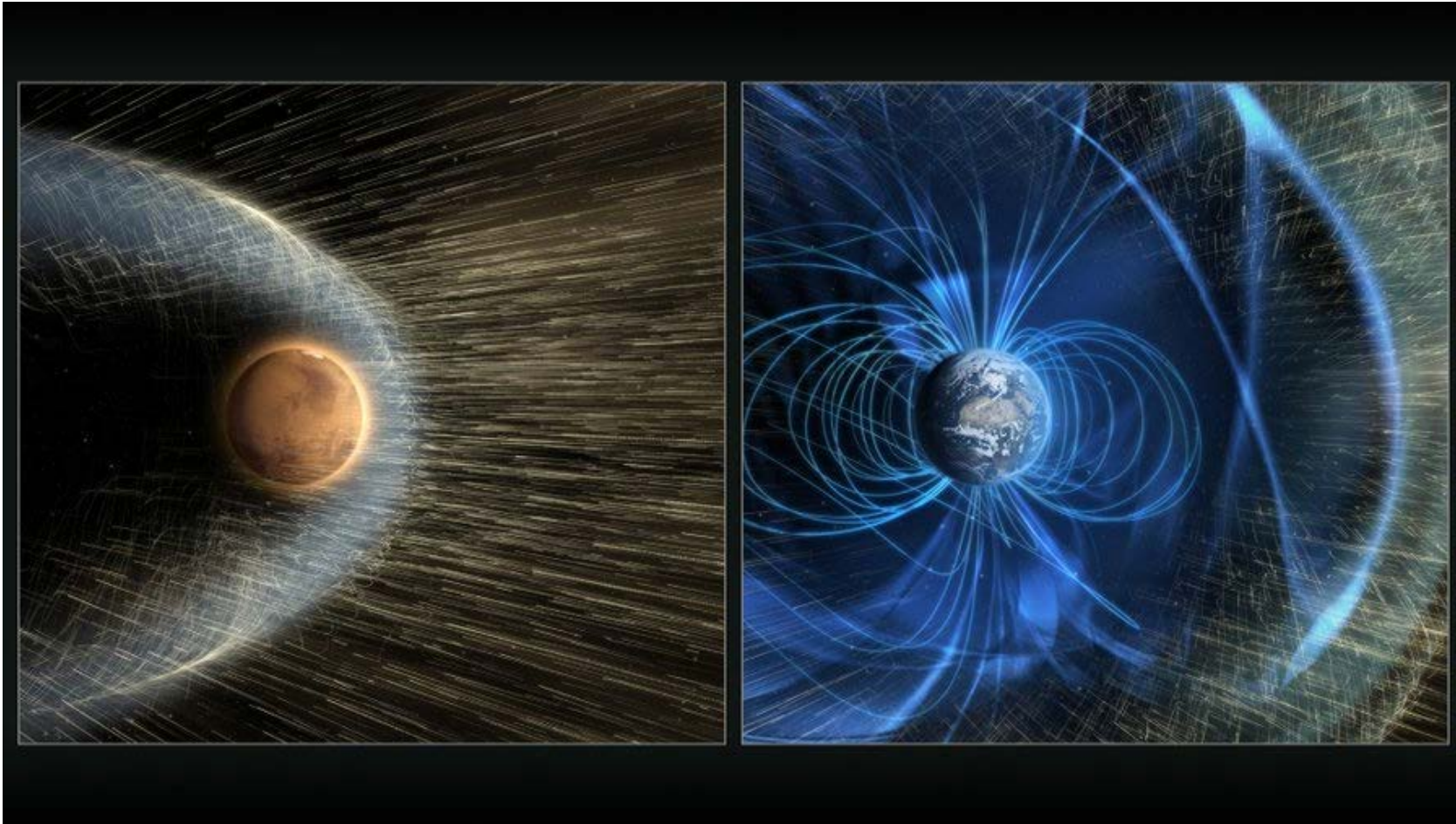
▶ dynamic load imbalance

- ▶ caused by application execution
 - ▶ 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 incurs runtime overhead

The Bad News Reloaded

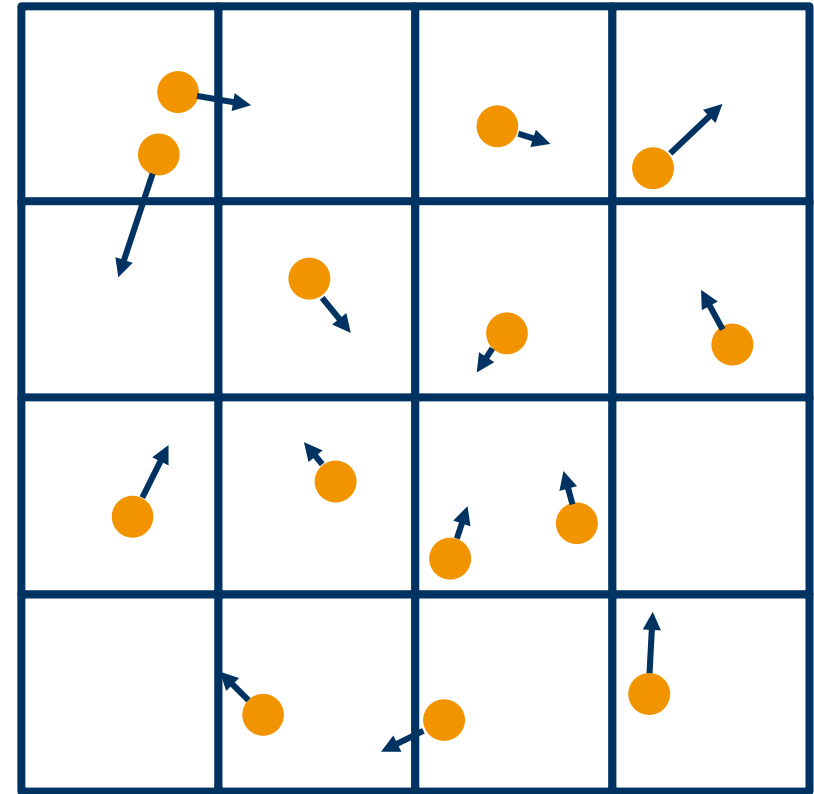
- ▶ MPI does not offer load balancing
 - ▶ yes, there's `MPI_Scatterv()` & alike
 - ▶ but it's only the underlying tool
- ▶ you need to take care of this yourself
 - ▶ or better yet, use a library

Recap: Space Weather Prediction (Particle-in-Cell Simulation)



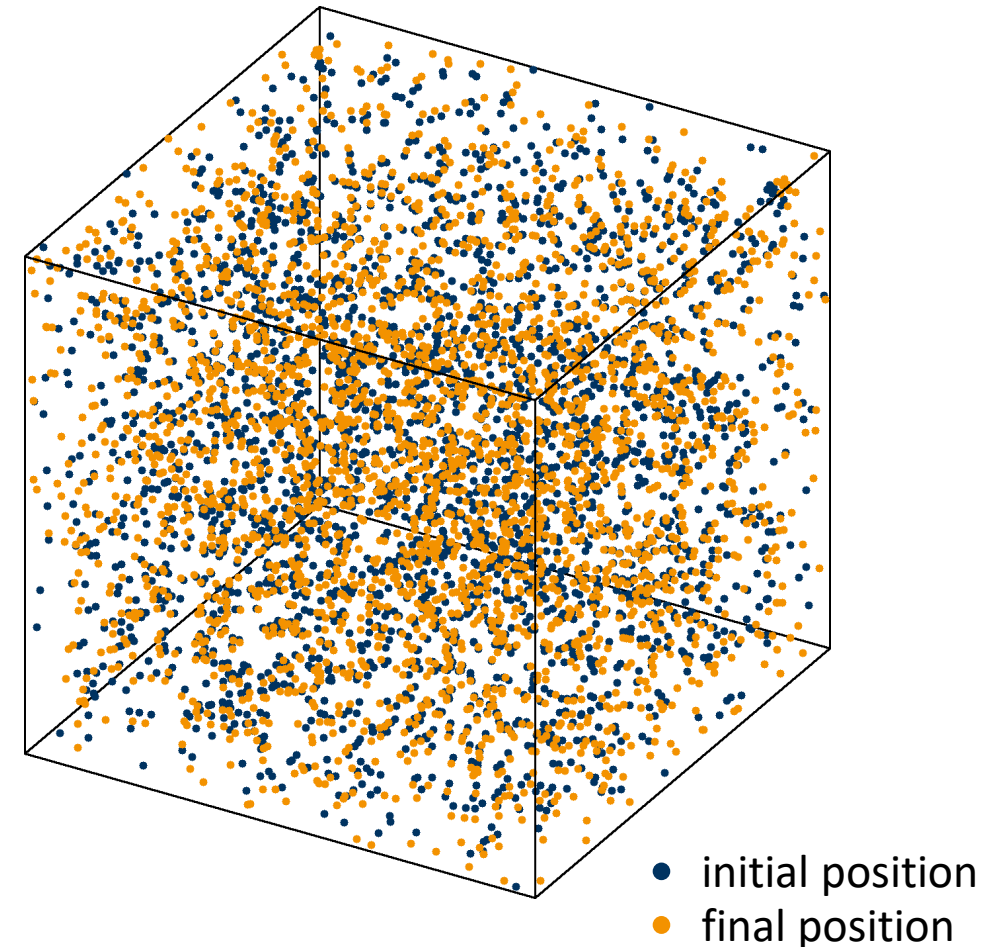
Particle-in-Cell

- ▶ moving, 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 hitting Earth's magnetosphere



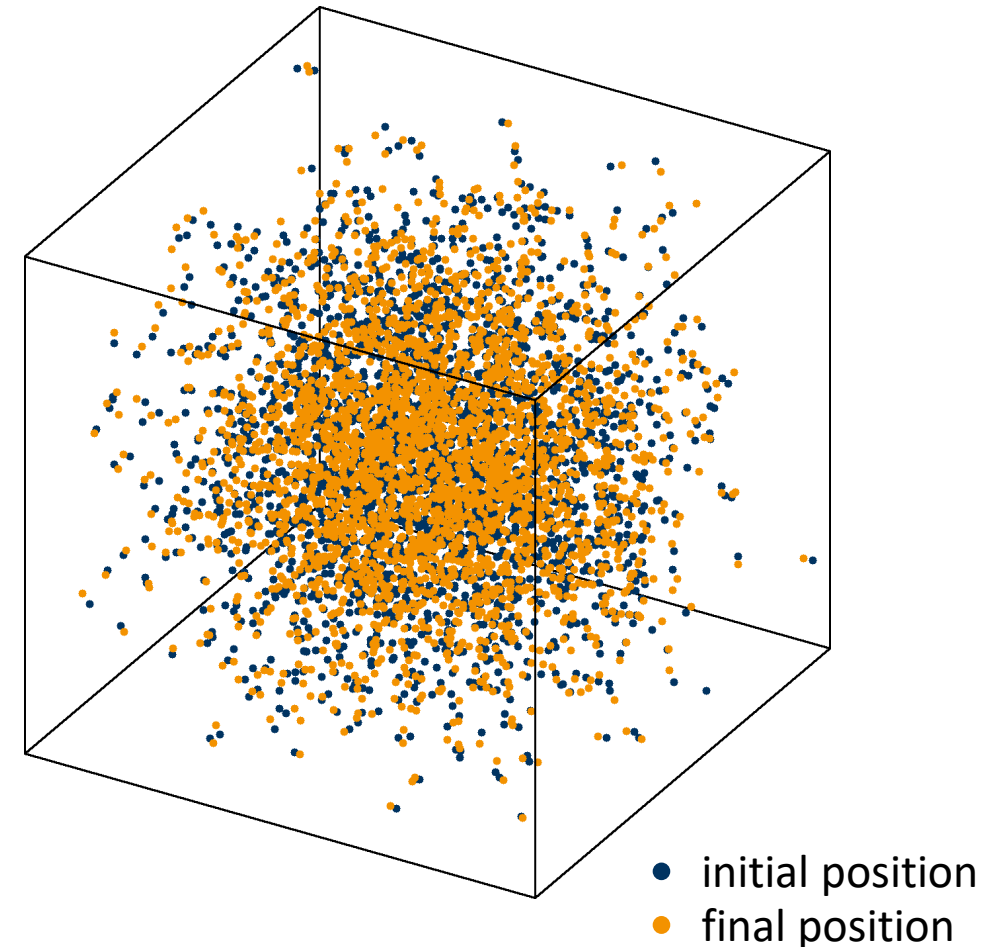
Particle-in-Cell Use Case: Uniform

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



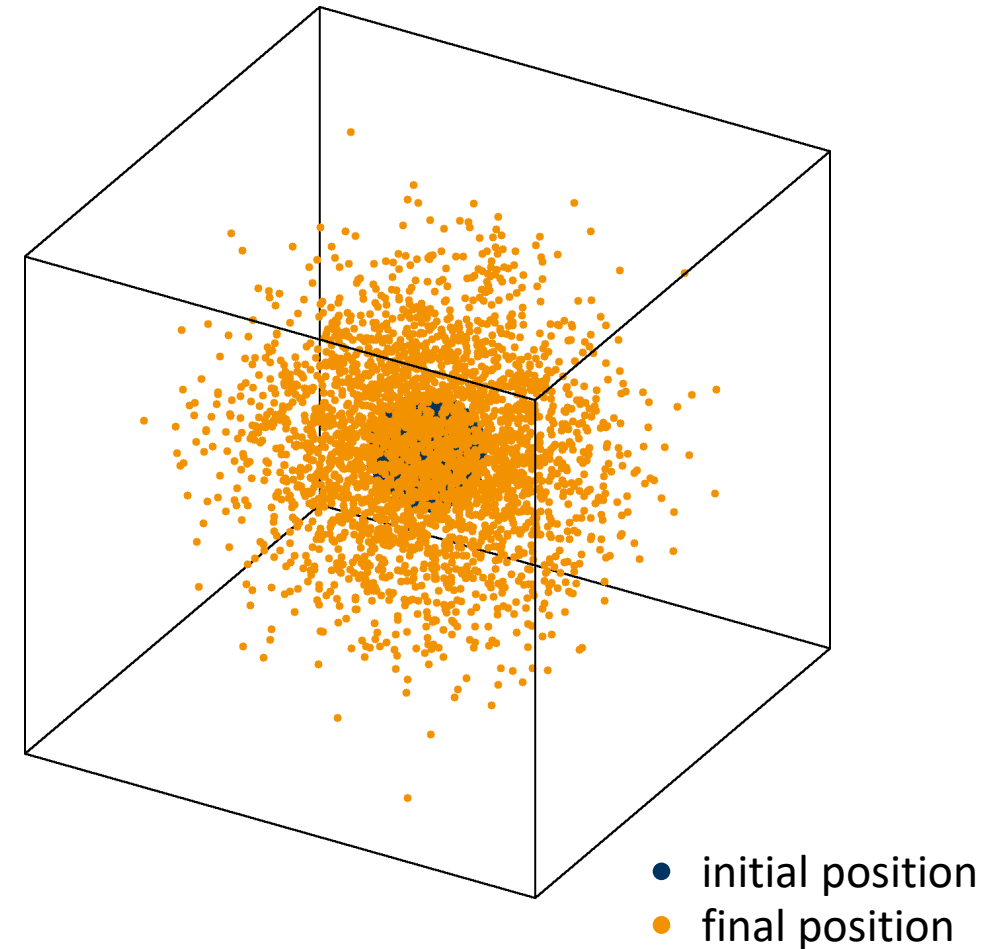
Particle-in-Cell Use Case: Cluster

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



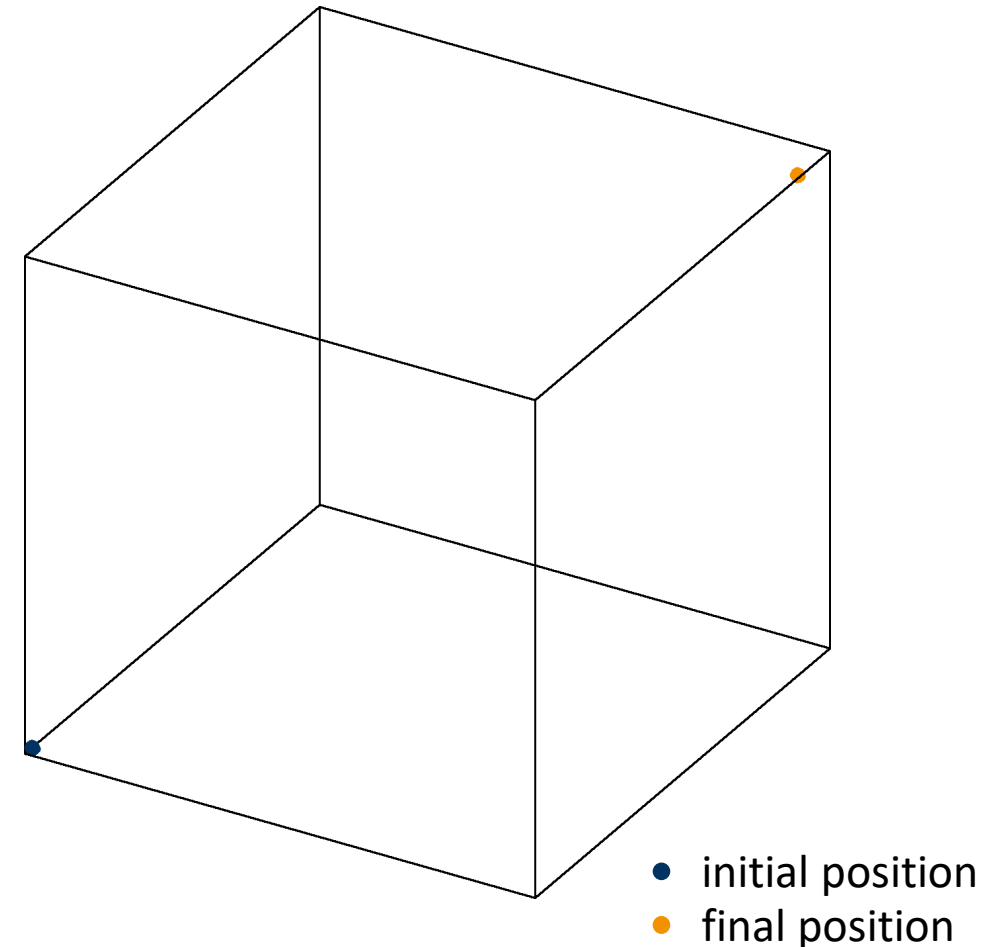
Particle-in-Cell Use Case: Explosion

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

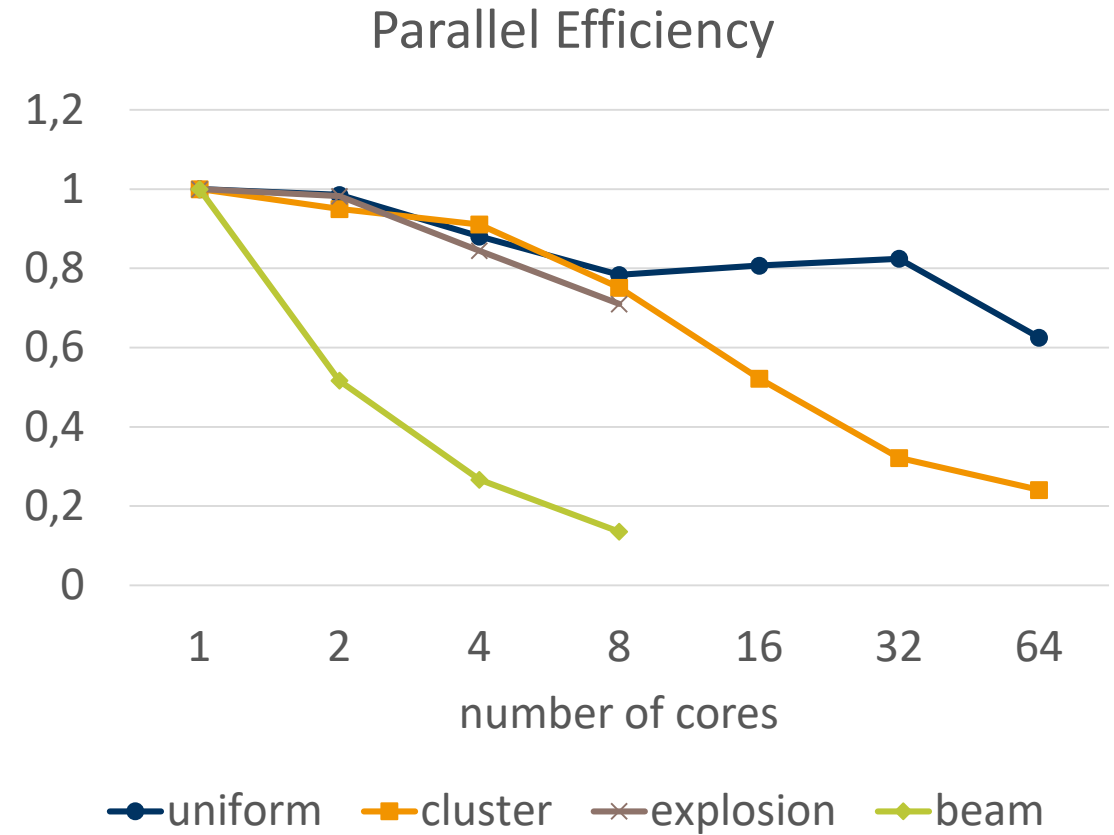
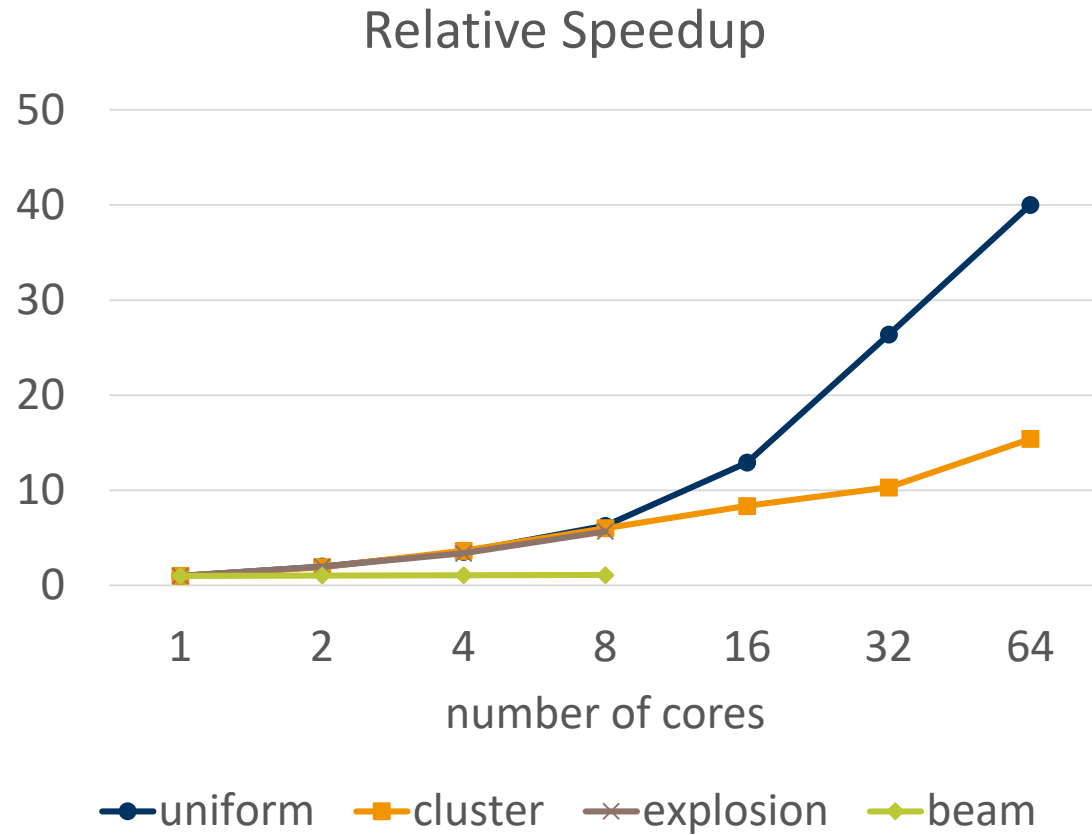


Particle-in-Cell Use Case: Beam

- ▶ 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

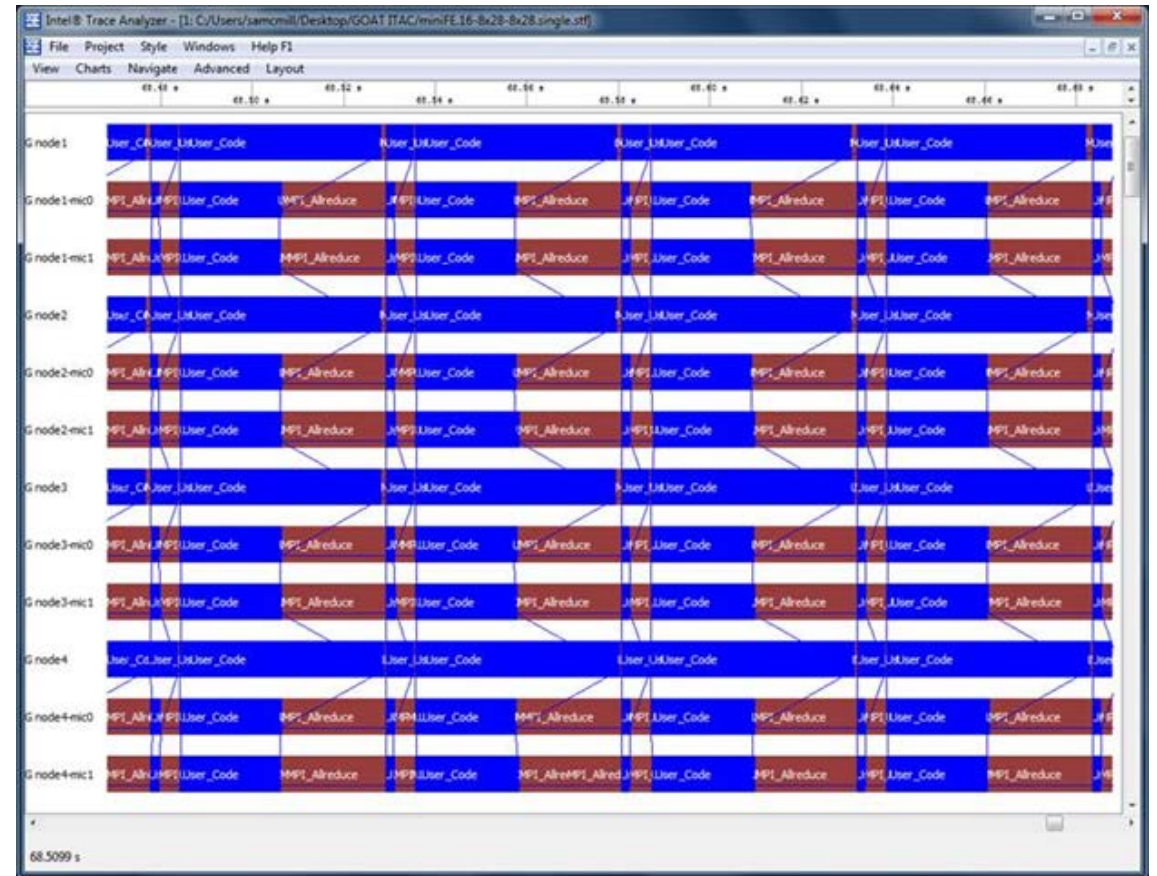


iPIC3D Use Case Comparison, Strong Scaling, 1M Particles, LCC2



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 ranges (e.g. elements)
- ▶ 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
 - ▶ also known as “adaptive refinement”
- ▶ trade-off with 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
 - ▶ 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)

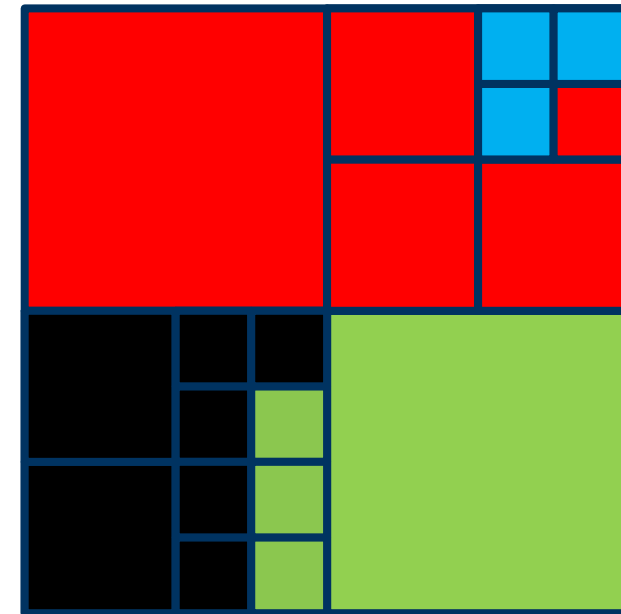
Dealing with Load Imbalance: Static Case

▶ static

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

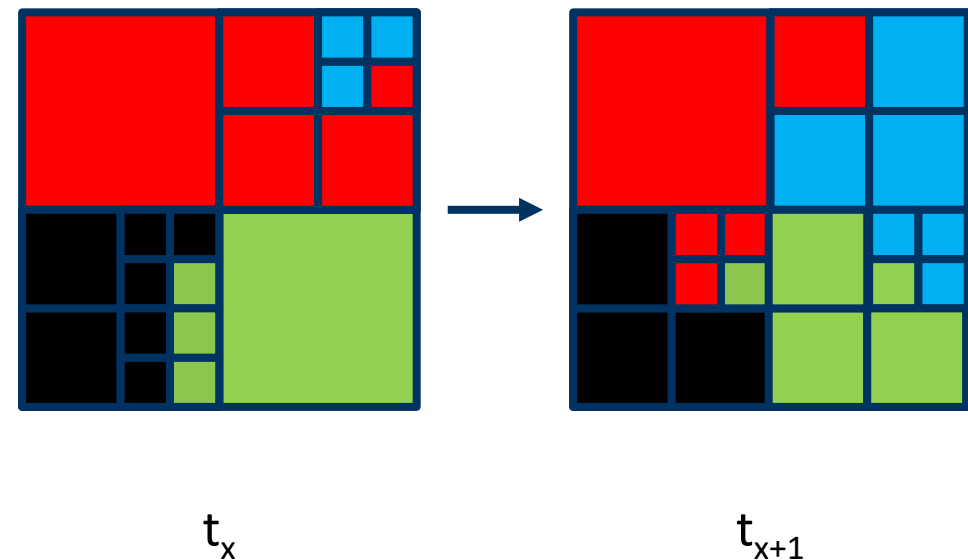
▶ example on the right: quadtree

- ▶ more work → smaller subregions
- ▶ 3D version: octtree



Dealing with Load Imbalance: Dynamic Case

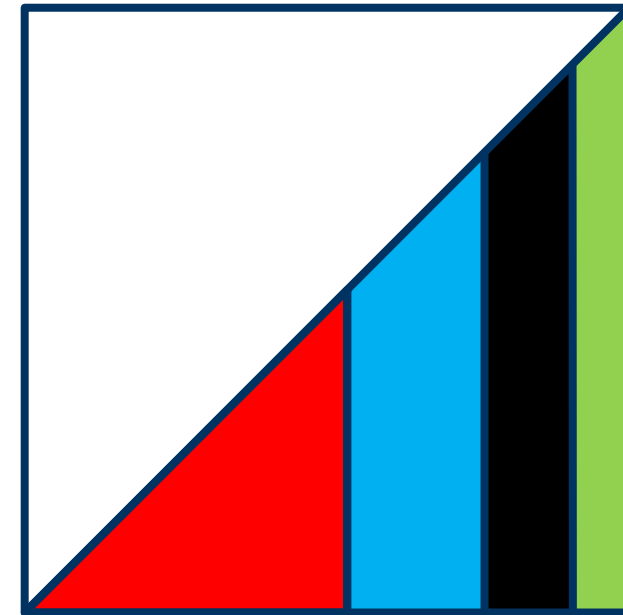
- ▶ **dynamic**
 - ▶ some form of repeated balancing required, e.g.
 - ▶ at certain intervals
 - ▶ when reaching certain thresholds
 - ▶ use e.g. worker queues and work stealing



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

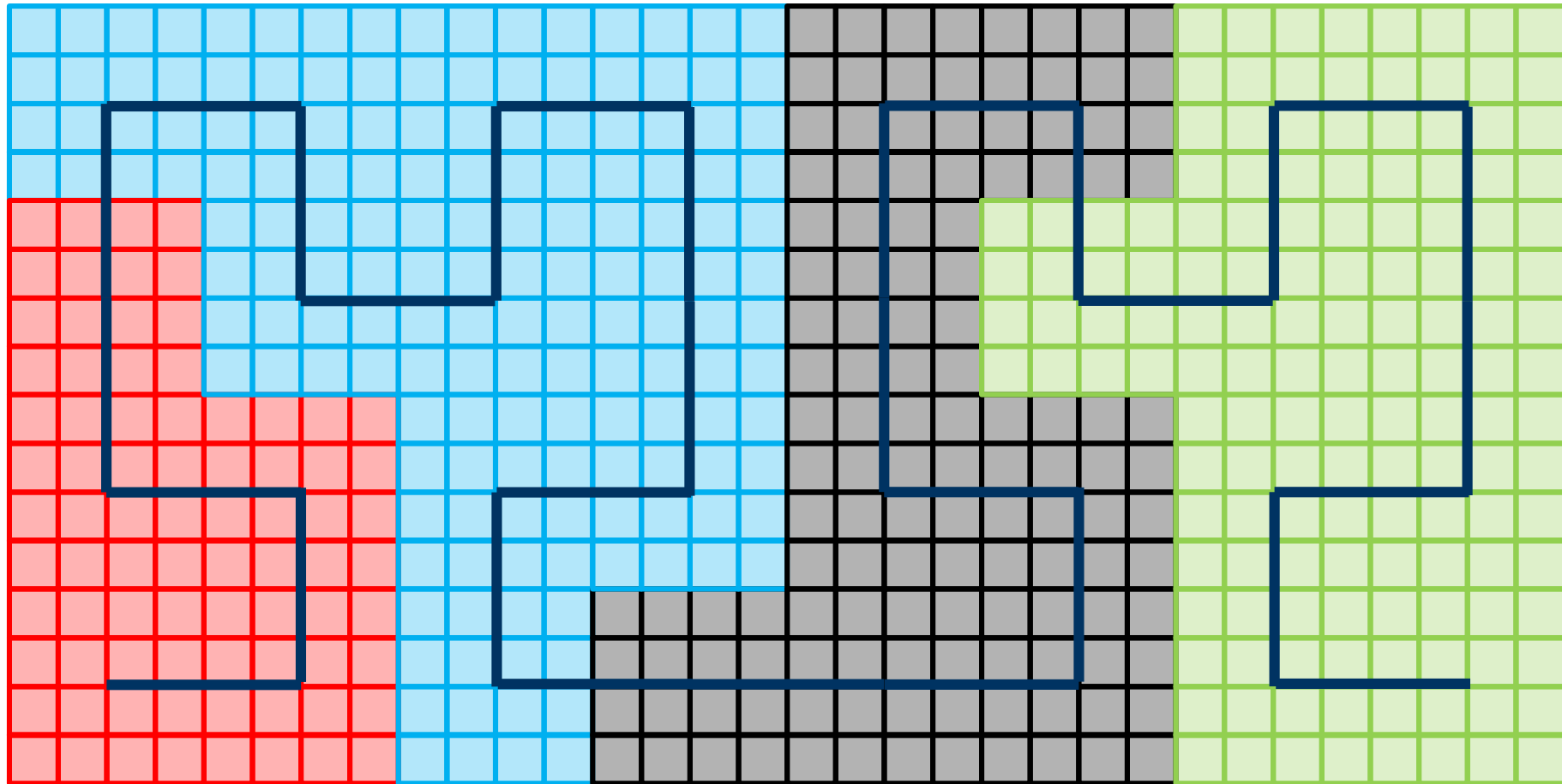
```
for(int i = 0; i < N; ++i) {  
    for(int j = 0; j < i; ++j) {  
        ...  
    }  
}c
```



Further Means of Load Balancing

- ▶ use “smart” work assignment strategies instead of smart domain decomposition
 - ▶ assign small chunks of work in round robin fashion
 - ▶ assign small chunks in random order
 - ▶ 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
 - ▶ doesn't reduce wall time but saves power and energy
- ▶ biggest issue of all: which strategy to select for which problem...

Space-filling Curves (Hilbert)



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: Efficient Ghost Cell Exchange

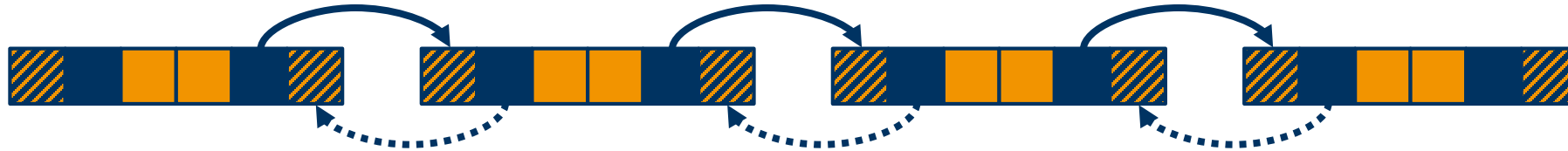


```
if(myRank > 0) {  
    // recv(left)  
}  
if(myRank < N-1) {  
    // send(right)  
}
```

time	rank 0	rank 1	rank 2	rank 3	...
0	send(right)	recv(left)	recv(left)	recv(left)	
1		send(right)	recv(left)	recv(left)	
2			send(right)	recv(left)	
3				send(right)	
...					

unnecessary serialization!

Tales from the Proseminar: Efficient Ghost Cell Exchange cont'd



```
if(myRank % 2) {  
    // recv, send  
} else {  
    // send, recv  
}  
// or  
MPI_Sendrecv(...)
```

time	rank 0	rank 1	rank 2	rank 3	...
0	send(right)	recv(left)	send(right)	recv(left)	
1		send(right)	recv(left)	send(right)	

done!

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”
 - ▶ efficient ghost cell exchange

Image Sources

- ▶ 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>